Co-location Detection on the Cloud

Mehmet Sinan İnci(✉), Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar

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

Abstract. In this work we focus on the problem of co-location as a first step of conducting Cross-VM attacks such as `Prime and Probe` or `Flush+Reload` in commercial clouds. We demonstrate and compare three co-location detection methods namely, cooperative Last-Level Cache (LLC) covert channel, software profiling on the LLC and memory bus locking. We conduct our experiments on three commercial clouds, Amazon EC2, Google Compute Engine and Microsoft Azure. Finally, we show that both cooperative and non-cooperative co-location to specific targets on cloud is still possible on major cloud services.

Keywords: Co-location on the cloud · Software profiling · Cache covert channel · Performance degradation attacks · Memory bus locking

1 Motivation

As the adoption of cloud computing continues to increase at a dizzying speed, so has the interest in cloud-specific security issues. A new security issue due to cloud computing is the potential impact of shared resources on security and privacy of information. An example is the use of caches to circumvent ASLR [11], one of the most common techniques to prevent control-flow hijacking attacks. Several other works target the exploitability of cryptography in co-located systems under increasingly generic assumptions. While early works such as [24] still required attacker and victim to co-reside on the same core within a processor, latest works [14,17] work across cores and managed even to drop the memory de-duplication requirement of `Flush+Reload` attacks [7,10,13,22]. Besides extracting cryptographic keys, there are plenty of other security issues explored in other related studies. Irazoqui et al. [16] study the potential of reviving the partially fixed Lucky 13 attack [8] by exploiting co-location.

All of the above attacks rely on the attacker's ability to co-locate with a potential victim. While co-location is an immediate consequence of the benefits of cloud computing (better utilization of resources, lower cost through shared infrastructure etc.), whether *exploitable* co-location is possible or easy has so far not been studied in detail. In his seminal work, Ristenpart et al. [18] studied the general feasibility of co-location in Amazon EC2, the most popular public cloud service provider (CSP) then and now, in detail. However, the cloud landscape has changed significantly since then: The EC2 has grown exponentially and operates data centers around the globe. A myriad of competitors have popped up,

© Springer International Publishing Switzerland 2016
F.-X. Standaert and E. Oswald (Eds.): COSADE 2016, LNCS 9689, pp. 19–34, 2016.
DOI: 10.1007/978-3-319-43283-0_2

all competing for the rapidly growing customer base [9]. CSPs are also more aware of the potential security vulnerabilities and have since worked on making their systems leak less information across VM boundaries. Furthermore, in their experiments, both co-located parties were colluding to achieve co-location. That is, both parties were willingly involved in communicating with the other to detect co-location. While being of high importance to show the feasibility in the first place, trying to co-locate with a specific and most likely unwilling target can be considerably harder. Since that initial work, until very recently only little work has dealt with a more detailed study on the difficulty of co-location. Therefore, we believe, the problem of co-location on cloud requires further in depth analysis examining different detection methods under diverse scenarios and access levels for the attacker.

1.1 Our Contribution

In this work we revisit the problem of co-location in public IaaS clouds. In particular we:

- study the co-location problem under two threat models in the Amazon EC2 Cloud, Google Cloud Engine and Microsoft Azure.
- develop a novel LLC software profiling tool that can detect an application or a library run by the non-cooperating co-located victim in the cloud, without the use of the memory de-duplication or any other memory sharing methods.
- demonstrate three co-location methods and compare their success rates on three popular public clouds.

2 Related Work

In the last few years several methods were proposed to detect co-location on commercial clouds [6,12,18,23,25]. These works use methods such as deducing co-location from instance and hypervisor IP address, hard disk drive performance degradation, network latency and L1 cache covert channel. However, in response to these works, most of the proposed techniques have been closed by public cloud administrators. Later Zhang et al. [23] were able to determine whether a particular user's VM had someone else co-residing in the same physical core. In particular, they utilized the well known `Prime and Probe` cache based side-channel technique to guess this information. However, the technique was applied in the upper level caches, thereby limiting its applicability to a physical core rather than the entire CPU or the machine. Furthermore, the technique was not tested in commercial clouds.

Shortly later, Bates et al. [6] demonstrated that a malicious VM can inject a watermark in the network flow of a potential victim. In fact, this watermark would then be able to broadcast co-residency information. Again, even though the technique proved to be extremely fast (less than 10 s), it was never tested in commercial clouds. Recently, Zhang et al. [25] demonstrated that Platform as a

Service (PaaS) clouds are also vulnerable to co-residency attacks. They used the `Flush+Reload` cache side-channel technique together with a non-deterministic finite automaton method to infer co-location with a particular server. The technique proved to be effective in commercial PaaS clouds like DotCloud or OpenShift, but would never work in IaaS clouds where the memory de-duplication is not implemented, as in most of the commercial IaaS clouds.

Finally, İnci et al. [12] demonstrated that many of the previously utilized techniques in [18] are no longer exploitable. Nevertheless, they prove to detect co-location *across cores* in Amazon EC2 by monitoring the usage of the LLC with the `Prime and Probe` technique. To enable the co-location test, the authors make use of hugepages commonly available in commercial clouds. This feature provides a large memory space for the attacker to move and hit necessary addresses to prime cache sets. Also in 2015, Varadarajan et al. [20] investigated co-location detection in public clouds by triggering and detecting performance degradations of a web server using the memory bus locking mechanism. Simultaneously Xu et al. [21] used the same memory bus locking mechanism to explore co-location threat in Virtual Private Cloud (VPC) enabled cloud systems.

3 Threat Models

Here we briefly outline two attacks scenarios for cross-VM attacks on public clouds. The main difference between the two scenarios is whether the target is predetermined or not. As we shall see, this makes a significant difference in terms of the requirements and cost of a successful attack. We provide concrete examples for both scenarios.

Random Victim
In this scenario there are four steps:

1. **Co-location:** The attacker spins instances on the cloud until it is determined that the instance is not *alone*; i.e. is co-located with another VM. Here the goal is to maximize the probability and thereby reduce the cost of co-locating with a viable target. Cheaper instances that use fewer CPU cores tend to share the same hardware in greater numbers. Therefore these instances have a better chance of co-location with other customers. Since we do not discriminate between targets, this step is rather easy to achieve.
2. **Vulnerable Software Identification:** The attacker detects a software package in the co-resident VM vulnerable to cross-VM attacks by monitoring corresponding LLC sets of libraries, e.g. an unpatched version of a cryptographic library. Cache access/performance and more broadly fingerprinting based techniques do exist in the literature to make successful attacks in the cloud environment [15,19,25]. Here, instances with lower number of tenants are less noisy therefore have higher success rate of library detection and the actual attack.
3. **Cross-VM Secret Extraction:** Here the attacker runs one of the cross-VM attacks [12,14] on the identified target. By exploiting cross-VM leakage

the attacker would be able to recover a sensitive information ranging from specialized pieces of information such as cryptographic keys, to higher level information such as browsing patterns, shopping cart, system load or any sensitive information of value. Noise plays a significant role in reliability of the extraction technique. Since co-location (first step) is easy to achieve, it is (almost) always advisable to opt for a less populated low noise instance to improve the chance of a successful attack in the later steps.
4. **Value Extraction:** The result is some sensitive information that can be turned into value with additional mild effort. For example, some information is valuable in its own right and can be converted into money with little or no effort, e.g., bitcoins, credit card information, credentials for online banking. Some others require further effort such as TLS session encryption key (secret key), e.g. for a Netflix streaming session. If the recovered secret is a private key of a public key encryption scheme (e.g. RSA secret key used a TLS handshake) the attacker needs the identity of the owner (website/company) to have further use for the secret key. In this case he may check the private key against public key repositories for noise correction and target identification.

Targeted Victim
This is the complementary scenario where we are given some identification information about the target.

1. **IP Extraction:** The attacker wants to focus its cycles on a server or a group servers that belong to an individual, cloud backed business, e.g. Dropbox or Netflix, or group/entity, e.g. dissidents of a political party. Here we assume that the attacker is capable of resolving the identification information to an IP or group of IPs of the target. In practice, this can be achieved rather easily by using public information and by using simple commonly available network tools such as `traceroute`/`tracepath`, `nmap` etc.
2. **Targeted Co-location:** The attacker creates instances on the cloud until one is co-located with the target instance on the same physical machine. The identification information of the victim, e.g. IP address, is used for co-location detection. For instance, using the IP the attacker can query the server creating CPU load and then run co-location tests. While co-location detection will be easier in this scenario due to the trigger; we will need many more trials to land on the same physical machine as the victim[1]. Nevertheless, we can accelerate targeted co-location by *searching*, for instance, only in the same region as the victim instance using the publicly available AWS IP lists [1]. Further, we can obtain finer grain information about the target's location simply by running `traceroute` or `tracepath` on the victim IP.
3. **Vulnerable Software Identification:** Since we know the identity of our target, it is safe to assume that we have some rudimentary understanding of

[1] Note that if the physical machine is already filled with the maximum number of allowed instances, then co-location may not be possible at all. In this case a clever albeit costly strategy would be to first mount a denial of service attack causing the target instance to be replicated and then try co-locating with the replicas.

the victim's setup including OS, communication and security protocols used etc. Even if this is not the case, it would be possible to run a discovery stage to survey the victim machine using its IP and by detecting process fingerprints through cross-VM leakage.

4. **Value Extraction:** The attacker exploits cross-VM leakage to recover sensitive information. Further processing may allow to enhance quality of the recovered data using publicly available information. For instance, a noisy private key can be processed with the aid of the public key contained in the certificate belonging to the target to remove any imperfections.

4 Overview: Co-location Detection Methods

4.1 LLC Covert Channel

The LLC is shared across all cores in most modern CPUs and is semi-transparent to all VMs running on the same machine. By semi-transparent, we mean that all VMs can utilize the entire LLC but cannot read each other's data. We exploit this behavior to establish a covert channel between VMs in cloud. The covert channel works by two VMs writing to a specific set-slice pair in the LLC and detecting each other's accesses. LLC set address can easily be deduced from the virtual addresses available to VMs using hugepages as done in [12,14,17]. The cache slice on the other hand, cannot be determined with certainty unless the slice selection algorithm of the CPU is known. However, the covert channel can still work by priming more sets and accessing lines that go to the targeted set, regardless of its slice.

Prime and Probe: In the LLC, the number of lines required to fill a set is equal to the LLC associativity. However, when multiple users access the same set, one will notice that fewer than 20 lines are needed to observe evictions. By running the following test concurrently on multiple instances, we can verify co-location. The test works as follows:

- Calculate the set number by using the address bits that are not affected by the virtual to physical address translation. Prime a memory block M_0 in the set.
- Access more memory blocks $M_1, M_2, \ldots, M_n$ that go to the same set. Note that since the slice selection algorithm for the specific CPU is necessary to address a set/slice pair with certainty, the number of memory blocks n needs to be larger than the set associativity times the number of slices.
- Access the memory block M_0 and check for eviction from the LLC. If evicted, we know that the required b memory blocks that fill the set are among the accessed memory blocks $M_1, M_2, \ldots, M_n$.
- Starting from the last memory block accessed, remove one block and repeat the above protocol. If M_0 still has high access time, M_i does not reside in the same slice. If b_0 is now located in the cache, we know that b_i resides in the same cache slice as b_0 and therefore go to the same set.

– Once the b memory blocks that fill a slice are identified, we just access additional memory blocks and check whether one of the primed b memory blocks has been evicted, indicating that they collide in the same slice.

The covert channel works by continuously accessing data that goes to a specific cache set and measuring the access time to determine if a newly accessed data has evicted an older entry from the set. Due to this continuous cache line creation, when the second party makes accesses to the monitored set, they are detected. In general, if there is no noise present, the number of lines that can go to a set without triggering an eviction is equal to the associativity of the cache, assuming a first-in first-out (FIFO) cache replacement policy is employed.

When two VMs try to fill the same set, they have to access less number of data blocks to fill the specified cache hence detecting the co-location. Using the number of blocks necessary to fill a specific set with and without another instance interfering, we calculate a co-location confidence ratio.

4.2 Software Profiling on LLC

The software profiling method works in a realistic setting with minimal assumptions. The method works in a non-cooperative scenario where the target does not participate in a covert communication and continues its regular operation. The method does not require memory de-duplication or any form of shared libraries. It employs the `Prime and Probe` to monitor and profile a portion of the LLC while a targeted software is running. As for the memory addressing, we profile the targeted code address as a relative address to the page boundary. Since the targeted library will be page aligned, target code's relative address (the page offset) will remain the same between runs. Using this information, we can reduce our search space in the detection stage. Therefore, we need to monitor only 320 different set-slice pairs such as $X \mod 64 = Y$ where X is 320 different set numbers (since we have 10 cores and 32 different set numbers satisfying the equation) and Y is the first 6 bits (the first 6 bits of the LLC set number is directly converted to physical address) of the set number for the desired function.

For the RSA detection, the slice-selection algorithm of the CPU is required to locate the targeted multiplication code in the LLC in a reasonable time. Without the algorithm, it would take too much time to monitor potential cache sets. For our experiments, we have used the algorithm that was reverse engineered by İnci et al. in [12].

In summary, there are two stages to the software profiling on LLC;

– **Profiling Stage:** The first step of the profiling is to monitor the targeted LLC sets while the profiled code, the software is not running. The purpose of this stage is to measure the idle access time of 20 lines for each set to have a threshold to detect whether there is a cache miss or not in the next stage.
– **Detection Stage:** We send RSA decryption requests to candidate IPs in order to discover the IP address of the victim. After triggering the decryption we

begin to monitor the portion of LLC to detect accesses due to the decryption. If we detect accesses in targeted set-slice pairs then we know that the correct IP address is found. As a double check, in addition to the RSA detection, we also detect AES encryption. In order to so we monitor another portion of the LLC where the AES T-tables potentially reside. And if the victim is co-located with the attacker, we can detect and monitor these T-table accesses.

4.3 Memory Bus Locking

The memory bus locking method exploits atomic instructions therefore we explain these special instructions shortly in the following.

Atomic Operations: Atomic operations are defined as indivisible, uninterrupted operations that appear to the rest of the system as instant. When operating directly on memory or cache, an atomic operation prevents any other processor or I/O device from reading or writing to the operated address. This isolation ensures computational correctness and prevents data races. While all instructions on single thread systems are automatically atomic, there is no guarantee of atomicity for regular instructions in multi-thread systems as used in almost all modern systems. In these systems, an instruction can be interrupted or postponed in favor of another task. The rescheduling, interruption and operating on the same data can cause pipeline and cache coherency hazards. Therefore the atomic operations are especially useful on multi-thread systems and parallel processing.

In older x86 systems, processor locks the memory bus completely until the atomic operation finishes, whether the data resides in the cache or in the memory. While ensuring atomicity, the process results in a significant performance hit. In newer systems - prior to Intel Nehalem and AMD K8 - memory bus locking was modified to reduce this penalty. In these systems, if the data resides in cache, only the cache line that holds the data is locked. This lock results in a very insignificant system overhead compared to the performance penalty of memory bus locking. However, when the operated data surpasses cache line boundary and resides in two cache lines, more than a single cache line has to be locked. In order to do so, memory bus locking is again employed. After Intel Nehalem and AMD K8, shared memory bus was replaced with multiple buses with non-uniform memory access bridge between them. While getting rid of the memory bottleneck for multiprocessor systems, this also invalidated the memory bus locking. Now, when a multi-line atomic cache operation has to be performed, all CPUs has to coordinate and flush their ongoing memory transactions. This emulation of memory bus locking results in a significant performance hit.

In x86 architecture, there are many instructions that can be executed atomically with a lock prefix are ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, DEC, FADDL, INC, NEG, NOT, OR, SBB, SUB, XADD, XOR. Also, XCHG instruction executes atomically when operating on a memory location, regardless of the LOCK use. In order to maximize the flushing penalty, we tested all atomic instructions available to the platforms and measured how long each instruction takes to execute. Since the flushing is succeeded with the atomic operation itself, longer the instruction executes, stronger the performance hit becomes. Therefore we have used the XADDL instruction that resulted in the strongest penalty. In short, we employ this mechanism to slow down a server process running in the cloud and detect co-location **without cooperation** from the victim side.

Cache Line Profiling Stage: Our attack is CPU-agnostic and employs a short, preliminary cache profiling stage. This stage eliminates the need for the information like the cache line size and the cache access time. Our purpose here is to obtain data addresses that span multiple cache lines hence triggers a bus lock. First, we allocate a block of small, page-aligned memory using *malloc*. After the allocation, we start performing atomic operations on this block in a loop of 256 since no modern cache line is expected to be larger than 256 bytes. In each loop, we move our access pointer by one and record atomic operation execution times. When we observe a time larger than the pre-calculated average, we record the address. After all 256 addresses are tested, we obtain a list of addresses that span across multiple cache lines. Later during the locking stage, we operate only on these addresses rather than a continuous array, making the attack more efficient.

Dual Socket Problem: Memory bus locking works on systems with multiple CPU sockets. Even further, our tests reveal that the bus locking penalty clearly reveals whether the target and the attacker run in the same socket or not. As seen in Fig. 1, the memory access time is clearly distinguishable between same socket and different socket locks. On a dual socket system with two Intel Xeon E5-2609 v2 CPUs with 2 cores each. Note that this information is significant to the attacker since an architectural attack using the LLC requires the attacker and the target to be running in the same socket.

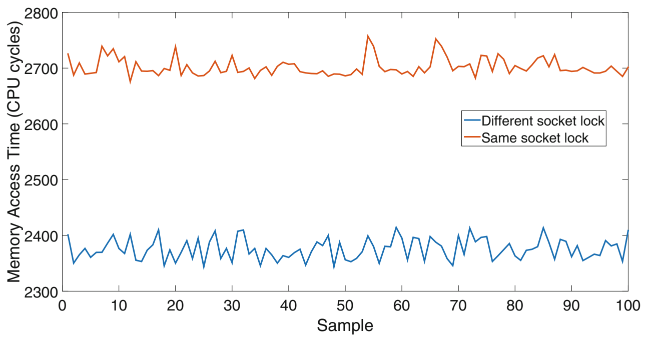


Fig. 1. The memory access times during a bus lock triggered with the XADDL instruction. Red and blue lines respectively represent access times when the attacker resides in the same socket (different core) and different sockets. (Color figure online)

5 Experimental Approach and Results

5.1 Co-location Results in Commercial Clouds

In all three aforementioned commercial clouds, we have launched 4 accounts with 20 instances per account, achieving co-location in each cloud. Also note that, we only classify the instances running in the same CPU socket as co-located and ignore the ones running on different sockets.

Amazon EC2: In Amazon EC2 we used **m3.medium** instance types that have balanced CPU, memory and network performance. This instance type holds 1 vCPU, 3.75 GB of RAM and 4 GB of SSD storage. According to Amazon EC2 Instance Types web page [4], these instances use 10 core Intel Xeon E5-2670 v2 (Ivy Bridge) processors.

Out of 80 instances launched, we have obtained 7 co-located pairs and one triplet verified by the tests. Moreover, we have tried to co-locate with instances that have launched previously. Surprisingly, we have been able to co-locate with instances that have launched 6 months prior.

Google Compute Engine: In GCE, we used **n1-standard-1** type instances running on 2.6 GHz Intel Xeon E5 (Sandy Bridge), 2.5 GHz Intel Xeon E5 v2 (Ivy Bridge), or 2.3 GHz Intel Xeon E5 v3 (Haswell) processors according to [5]. Out of 80 instances launched, we have obtained only 4 co-located pairs.

Microsoft Azure: In Azure, we used **extra small A0** instance types with 1 virtual core, 750 MB RAM, maximum 500 IOPS and 20 GB disk storage that is not specified as neither SSD nor HDD [2]. Out of 80 instances launched, we have obtained only 4 instances that were co-located. However, this was partly due to the highly heterogeneous CPU pool that Azure employs. Our first account had instances with AMD Opteron CPUs while the second had Intel E5-2660 v1 and the last two had Intel E5-2673 v3. Naturally, we could only achieve co-location among instances that have the same CPU model. Out of 40 Intel E5-2673 v3 instances, we detected 4 co-located instances.

5.2 LLC Covert Channel

In the following, we present the results in GCE. The confidence ratio is highest at 1 as seen in Fig. 2. There are 8 instances (meaning 4 pairs) that have higher than 50 % confidence ratio among 80 and the co-located pairs are found by binary search at the end. Hence, it is confirmed that they are indeed co-located with each other.

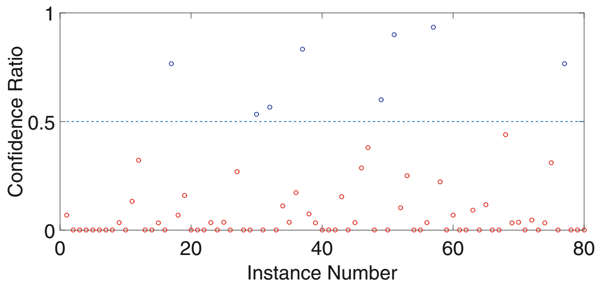


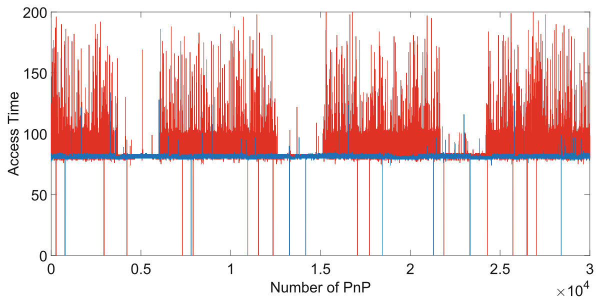
Fig. 2. GCE LLC Test Confidence Ratio Comparison

5.3 LLC Software Profiling

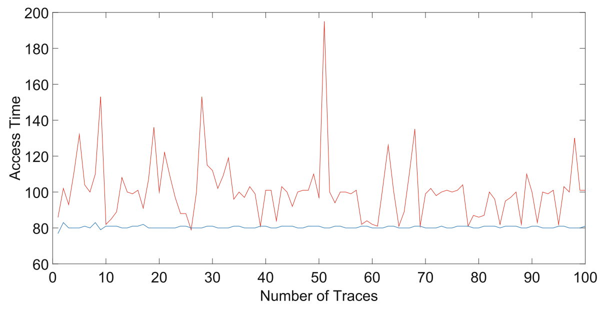
We conducted the LLC Software Profiling experiments on the co-located Amazon EC2 instances with 10 core E5-2670 v2 processors. As for the software target, in order to demonstrate the versatility of the attack, we chose the RSA (Libgcrypt version 1.6.2) that uses sliding window exponentiation and the AES (OpenSSL version 1.0.1g, C implementation) that uses T-tables. Note that the detection method is not limited to these targets since the attacker can run and profile any software which uses shared library in his instance and perform the attack.

For the RSA detection, the slice-selection algorithm of the CPU is required to locate the targeted multiplication code in the LLC within reasonable time. In our experiments, we have used the algorithm that was reverse engineered by İnci et al. in [12]. The first step of the profiling is to monitor the targeted LLC sets while the profiled code, RSA is not running. After the regular operation of sets are observed, the RSA request is sent to several IP addresses, starting from attacker's own subnet. As soon as the request is sent, the profiling starts and traces are recorded by the `Prime and Probe`. If the RSA decryption is running on the other VM, the pattern of multiplication can be observed as in Fig. 3. In general, the multiplication is performed between 2000–8000 traces. In these traces, we look for the delta of two profiles for each set-slice pair. In Fig. 4, the difference between two profiles is illustrated for two co-located instances. Both figures show that there are two set-slice pairs with significantly higher access times (4–8 cycles) in average of 10 experiments. Hence, it can be concluded that these two sets are used by RSA decryption and this candidate instance is probably co-located with the attacker.

After we obtain IP addresses of several co-location candidates, we trigger AES encryption by sending random ciphertexts and at the same time monitor the LLC. For this part of the detection stage, since AES encryption is much faster than RSA decryption we can only catch one access to monitored T-table position. Hence, we send 100 AES encryption requests to each instance in the IP list. If we observe 90 % cache miss for one of the set-slice pairs, it can be concluded that the AES encryption is performed by the co-located instance, as seen in Fig. 3(b).



(a) RSA Pattern



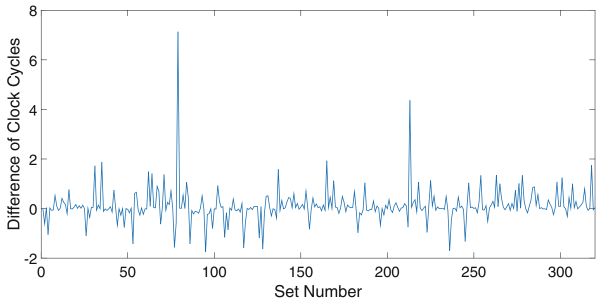
(b) AES Pattern

Fig. 3. Red and blue lines represent idle and RSA decryption/AES encryption access times respectively (Color figure online)

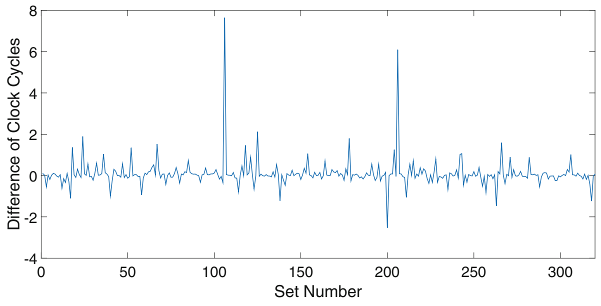
5.4 Memory Bus Locking

The performance degradation due to the memory bus locking is application specific. Therefore we tested various applications as seen in Table 1 to see how each one is affected. As expected, the applications with frequent memory accesses are more affected by the locking. For example, the GnuPG which mostly uses the ALU and does seldom memory accesses slowed down only by 29 %. An Apache web server that frequently loads content from memory on the other hand has a slowdown by the factor of 4.28.

In addition to specific software performance degradation, we also measured the effect of multiple locks executed in parallel. To do so, we have used the `openmp` parallel programming API [3] and ran the lock in multiple threads. Figure 5(d) shows the memory access times when 0 to 8 locks run in parallel. As the figure shows, the first lock does slowdown the memory accesses by 100 % while the second and third locks do not further degrade the memory performance. However, after fourth and fifth locks, we observe an even stronger degradations.



(a) RSA Analysis for the first co-located instance



(b) RSA Analysis for the second co-located instance

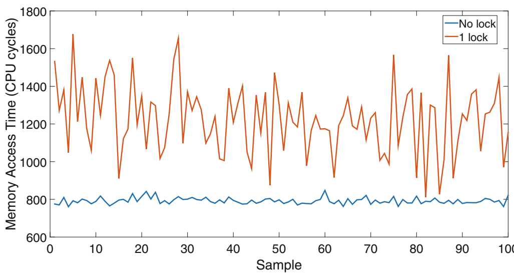
Fig. 4. The difference of clock cycles between base and RSA decryption profiling for each set-slice pairs over 10 experiments

Table 1. Application slowdown on an Intel Xeon 2640 v3 due to memory bus locking triggered on a single core.

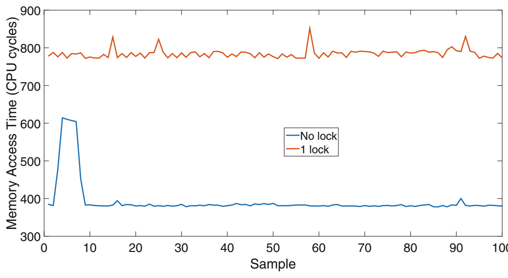
Process	Normalized execution time
Apache	4.28×
PHP	0.1×
GnuPG	0.29×
HTTPerf	0.29×
Memory access	5.38×
RAMSpeed int	5.01×
RAMSpeed fp	4.88×
Media stream	2.36×

5.5 Comparison of Detection Methods

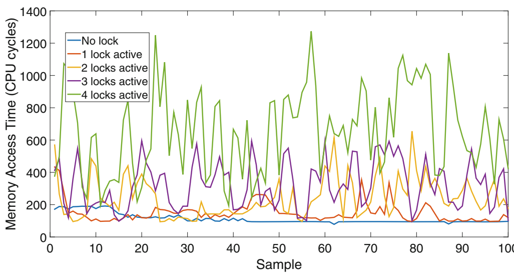
As explained in Sect. 3, co-location can be exploited in both random and targeted victim scenarios. Malicious Eve can directly look for attack vectors to steal information from her neighbors or she can go after a specific target and spin up



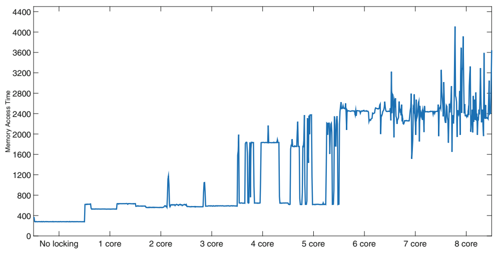
(a) Amazon EC2



(b) GCE



(c) Microsoft Azure



(d) Lab setup using Intel Xeon E5-2640 v3

Fig. 5. Memory access times with and without an active memory bus lock of (a) Amazon EC2 m3.medium instance (b) GCE n1-standard1 instance (c) Microsoft Azure A0 instance (d) Lab setup (Intel E5-2640 v3) (Color figure online)

Table 2. Comparison of co-location detection methods.

Detection method	Worst case	Average	Best case
Memory bus locking OPD[a]	0.1×	3.28×	6.1×
LLC covert channel	53 %	73.5 %	93 %
LLC software profiling	50 %	70 %	90 %

[a]OPD: Observed Performance Degradation

instances until she is co-located. However, if the detection method does not provide reliable results, the attacker can discard the co-located instances or even have false positives due to noise. Therefore a useful and efficient co-location detection method is essential.

Table 2 shows that all three methods inspected in this study work with high accuracy in a real commercial cloud setting. All methods work with minimalistic requirements, no hypervisor access or specific hardware. In comparison, while the memory bus locking has the least clear co-location signal in the worst case, other two methods are more prone to the LLC noise. Also, as seen in Table 1 the memory bus locking gives more reliable results with applications with frequent memory accesses. So for the uncooperative co-location scenario, depending on the workload of the target instance, one can use either the memory bus locking or the software profiling to detect co-location with high accuracy.

6 Conclusion

In conclusion, we represent three co-location detection methods working in three most popular commercial clouds (Amazon EC2, Google Compute Engine, Microsoft Azure) and compare their efficiencies. In addition, for the first time we have achieved targeted co-locations in Amazon EC2 Cloud by applying the LLC software profiling for AES and RSA processes. For the memory bus locking method, we have observed that frequent memory accesses lead to more significant degradation. As for the cache covert channel, we show that the method works in a cooperative scenario with high accuracy. And finally we presented the LLC software profiling technique that can be used for variety of purposes including co-location detection without the help of memory de-duplication or cooperation from the victim side.

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

References

1. AWS IP Address Ranges. http://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html
2. Microsoft Azure Sizes for virtual machines. https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-size-specs/

3. The OpenMP API specification for parallel programming
4. Amazon EC2 Instances (2016). http://aws.amazon.com/ec2/instance-types/
5. Google Compute Engine Instance Types (2016). https://cloud.google.com/compute/docs/machine-types
6. Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., Butler, K.: On detecting co-resident cloud instances using network flow watermarking techniques. Int. J. Inf. Secur. **13**(2), 171–189 (2014). http://dx.doi.org/10.1007/s10207-013-0210-0
7. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: "Ooh Aah.. Just a Little Bit": a small amount of side channel can go a long way. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 75–92. Springer, Heidelberg (2014)
8. Fardan, N.J.A., Paterson, K.G.: Lucky thirteen: breaking the TLS and DTLS record protocols. In: Security and Privacy, pp. 526–540 (2013)
9. Gaudin, S.: Public cloud market ready for 'hypergrowth' period. Computerworld Article, April 2014. http://www.computerworld.com/article/2488572/cloud-computing/public-cloud-market-ready-for-hypergrowth-period.html
10. Gülmezoglu, B., İnci, M.S., Apecechea, G.I., Eisenbarth, T., Sunar, B.: A faster and more realistic flush+reload attack on AES. In: COSADE, pp. 111–126 (2015)
11. Hund, R., Willems, C., 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 (2013). http://dx.doi.org/10.1109/SP.2013.23
12. İnci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. Technical report. http://eprint.iacr.org/
13. Irazoqui, G., İnci, M.S., Eisenbarth, T., Sunar, B.: Fine grain Cross-VM attacks on Xen and VMware. In: 2014 IEEE Fourth International Conference on Big Data and Cloud Computing (BdCloud), pp. 737–744, December 2014
14. Irazoqui, G., Eisenbarth, T., Sunar, B.: S$A: a shared cache attack that works across cores and defies VM sandboxing? And its application to AES. In: IEEE S&P (2015)
15. Irazoqui, G., İnci, M.S., Eisenbarth, T., Sunar, B.: Know thy neighbor: crypto library detection in cloud. In: Proceedings on Privacy Enhancing Technologies, vol. 1, no. 1, pp. 25–40 (2015)
16. Irazoqui, G., İnci, M.S., Eisenbarth, T., Sunar, B.: Lucky 13 Strikes Back. In: ASIA CCS 2015, pp. 85–96 (2015)
17. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: IEEE S&P, pp. 605–622 (2015)
18. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: CCS 2009, pp. 199–212 (2009)
19. Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Memory deduplication as a threat to the guest OS. In: Proceedings of the Fourth European Workshop on System Security, p. 1. ACM (2011)
20. Varadarajan, V., Zhang, Y., Ristenpart, T., Swift, M.: A placement vulnerability study in multi-tenant public clouds. In: 24th USENIX Security Symposium, USENIX Security 2015, Washington, D.C., pp. 913–928 (2015)
21. Xu, Z., Wang, H., Wu, Z.: A measurement study on co-residence threat inside the cloud. In: 24th USENIX Security, pp. 929–944 (2015)
22. Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: USENIX Security 2014, pp. 719–732 (2014)
23. Zhang, Y., Juels, A., Oprea, A., Reiter, M.K.: HomeAlone: co-residency detection in the cloud via side-channel analysis. In: IEEE S&P (2011)

24. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: CCS 2012, pp. 305–316 (2012)
25. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-tenant side-channel attacks in PaaS clouds. In: CCS, pp. 990–1003 (2014)