Co-Residency Detection and Memory Bus Locking

By: Jacob Nelson and Matthew Buell

CSE350 - Cloud Security

Spring 2017

1. Introduction

In today's technological landscape cloud computing has become a valuable resource for many. From large enterprises that store massive amounts of data, to small start-ups that cannot afford to host their own servers, it has provided a unique opportunity to limit the physical infrastructure needed for almost any venture. However, a key aspect of cloud computing is that the hardware is usually dedicated to more than one customer, leaving avenues for exploitation that would not be possible otherwise. Beyond traditional methods to secure a machine, cloud security must also consider the possible attack strategies that arise due to co-location. This paper outlines one such strategy that harnesses the innate behavior of the memory bus during specific operations. By hogging the memory bus resource, it is possible for a process to degrade the performance of another by a significant amount. Other possible attacks have also been shown by previous research, proving that a malicious co-resident instance is capable of monitoring fluctuations in memory and execution times to extract secret keys[1]. Both of these strategies rely on the attacker recognizing that they are co-located with the target, which has also proved to be possible[2]. This project aims to recreate the cooperative detection scheme presented in Varadarajan et al. to study its effectiveness and possible uses. We continue on to propose that cloud security companies could make use of this example to detect threats in the wild and mitigate the security concerns in section 6.

2. Background

The paper we modeled our work after presents a cooperative co-resident detection algorithm that provided a trustworthy way to determine if two cloud instances were co-located. Based on a simple lock and probe mechanism, the *lock instance* freezes up the memory bus using atomic operations designed to go to memory with each invocation, while the *probe instance* performs calculated reads designed to hit DRAM every time. While attempting reads, the probe instance also measures the runtime needed to complete its actions. Based on this information is is possible to determine that there is a degradation in performance, leading to the conclusion that the lock instance is present. This strategy relies on a number of assumptions. First, we assume that the instances are co-located. This cooperative approach allows us to prove that locking the memory bus slows down the probe instance. Second, we assume that the machine in question is based on the Intel x86 architecture. The locking mechanism relies on the fact that when atomic operations are performed on unaligned memory spanning two cache lines, the memory bus is always locked[3]. Finally, we assume that the file system is Linux based such that cache information can be determined dynamically through system information files. In our experiment, we use Ubuntu 14.04.

3. Motivation

Our motivation for this project stemmed mainly from our trip with Lehigh University to Silicon Valley, and particularly the additional time spent with tech companies during LSV++. After meeting with Bracket Computing, a cloud security company focused on creating a hypervisor capable of monitoring behavior, we were inspired to continue learning about the subject once back at school. Thus the experimental cloud security course was born. The beginning of the semester was rooted in grasping basic components of cloud security. Once we had a foundation, a research paper published by Varadarajan et. al caught our attention. It focused on co-residency detection and the application of co-location based attacks. Our goal was to replicate their lock and probing mechanism to detect co-residency. It is a case study that can be further applied to applications such as Bracket's cloud security platform, which will be discussed in section 6 of this paper. Ultimately, our interest in cloud security led us to mimic research that we found to be interesting.

4. Implementation

The following sections describe the implementation of the co-residency detection strategy using locking and probing.

[1] Zhang T, Zhang Y, Lee R. 2016. Memory DoS Attacks in Multi-tenant Clouds: Severity and Mitigation.

[2] Varadajan V, Zhang Y, Ristenpart T, Swift M. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds.

[3] Admin. "Bus Lock." Intel® Software. Intel, 25 Apr. 2017. Web. 14 May 2017. <https://software.intel.com/en-us/node/544402>.

4.1 Overview

As illustrated in *Figure 1*, two virtual machines running Ubuntu 14.04 through VirtualBox are allocated one of the CPU's cores including that core's cache memory. Despite each instance having been allocated its own hardware, the two instances still share the host computer's memory bus and L3 cache. Our hypothesis was that locking the memory bus on one instance using atomic operations would result in performance degradation in the probing instance. Overall we wrote 154 lines of code to accomplish the locking and probing mechanisms.

Figure 1 provides a visual representation of the environment used in this experiment. VirtualBox acted as a hypervisor for the two Ubuntu virtual machines. It runs atop a 1.6 GHz Intel i5 processor in Lehigh University's Sandbox. Each VM is provided a dedicated CPU with associated caches. The L3 cache is shared between all cores. That along with the computer's main memory becomes an important component of this project. By targeting the memory bus, co-residency and co-location attacks are possible.

4.2 Locking

Locking the memory bus is actually quite simple. As described above, all it took was an atomic operation on memory that spanned two cache lines. The following is the pseudo code that we modeled our algorithm of, provided by Varadarajan et al. In all, this piece required only 32 lines of C code to be completed.

Pseudo-code for lock

```
-->
// allocate memory multiples of 64 bits
char_ptr = allocate_memory((N+1)*8)
//move half word up
unaligned_addr = char_ptr + 2

loop forever:
        loop i from (1..N):
                atomic_op(unaligned_addr + i,
                                    some_value)
        end loop
end loop
<--
```

The first step is to allocate some size memory buffer that is a multiple of 8 bytes and long enough that it spans at least two cache lines. In our implementation we chose the length to be the same as the number of sets in the cache, for no particular reason other than it was a large stretch of memory. Next, we offset a *long long int* pointer to that memory address such that the unaligned pointer will span two cache lines. By looping forever over the buffer, we continuously lock up the memory, thus degrading the performance of all other accesses to memory. For the actual implementation of this algorithm take a look into our GitHub repository[4].

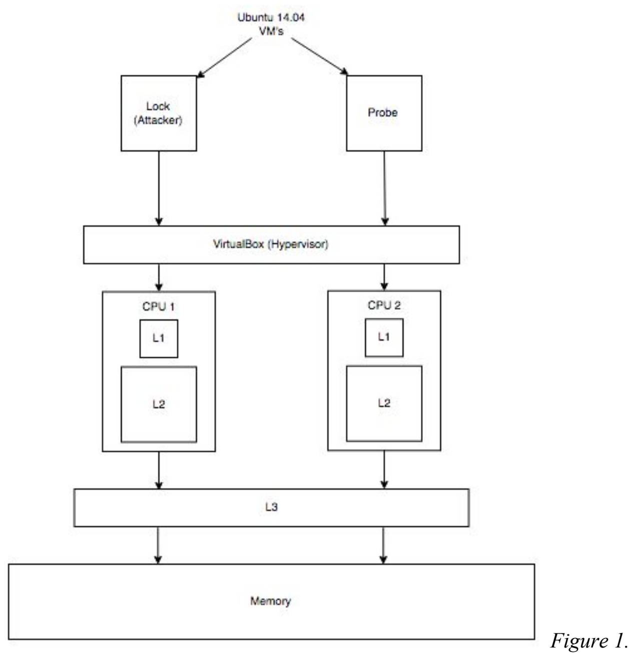

Figure 1.
A schematic for the environment used in this project. Ubuntu instances running on VirtualBox simulate a cloud user and hypervisor. The physical hardware is shared by both VMs and is targeted in co-residency detection.

4.3 Probing

While the lock was straightforward to implement, the probe took much more consideration to complete. Each memory access is required to hit data that was not already in the L3 cache, forcing retrieval from main memory. The following pseudo code is borrowed from Varadarajan et al. and was used as a basis for our own implementation. In total, the probing code took up 122 lines of C code.

Pseudo-code for probe

```
-->
size = LLC_size * (LLC_ways + 1)
stride = LLC_sets * cacheline_sz
buffer = alloc_ptr_chasing_buff(size, stride)
```

4 https://github.com/jacnel/co-res

```
loop sample from (1..100): //number of
                              samples
        start_rdtsc = rdtsc()
        loop probes from (1..10000):
                probe(buffer); //always hits
                                   memory
        end loop
        time_taken[sample] = (rdtsc() -
                              start_rdtsc)
end loop
<--
```

First, a piece of memory that is large enough to fill the cache more times than there are sets is set aside to probe. This was accomplished using a *pointer chasing buffer*. Each element in a pointer chasing buffer is a pointer to somewhere else in the buffer, which again contains another pointer to some location within the buffer. It is closed, so that by following each pointer it is possible to touch every element in memory. A large enough space in memory is needed such that it cannot all fit in the L3 cache, and cannot successfully preload any cache lines. The stride is large enough so that just one access occurs per cache line. A bit reverse permutation provide a pseudo random access of indices into a cache line length of memory, whose value then points to the memory one stride away. When the value of the next position in memory plus a stride will be out of the bounds of the allocated memory, it will instead point to a new index into the first cache line length of memory. The process then repeats until every pointer in memory is followed and it wraps back to the origin. *Figure 2* provides a visual representation of the buffer.. The open source code by afborchert's GitHub page was essential for writing the probe[5].

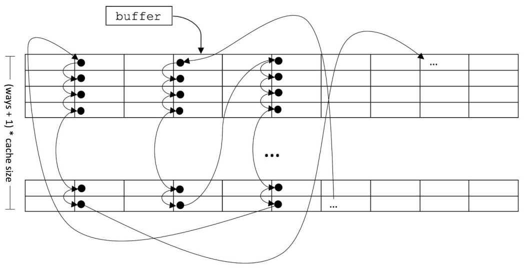

Figure 2. This shows how the pointer chasing buffer works. An array of void pointers pointing to memory locations within the array itself. The value of every element in the array points to the memory location of another element. A single void pointer can therefore walk along each element and probe every location in memory. The pointers are arranged in a way that the next pointer will always cause a cache miss.

Ten iterations of sequentially probing the memory space are used to sample the execution time. This information was collected and will be discussed in the Analysis section (5). Again, for the actual implementation of this algorithm take a look at the GitHub repository we created.

In order to measure the latency, we recorded the time stamp counter reading at the beginning and end of each iteration. We found the difference between those two values and recorded the number of CPU clock cycles it took for each iteration of the sample loop to complete. One hundred samples were recorded for the probe while it was not subject to memory bus locking and another one hundred for while it was subject to memory bus locking. Each series of samples were taken on the same machine to mitigate hardware inconsistencies.

5. Analysis

This section discusses the results and analyzes our work. We show that the co-residency detection strategy is functional and we are able to degrade the performance of two instances that are co-located.

Figure 3 depicts the results of one hundred samples of the probe under normal conditions (blue) as well as when subject to memory locking (orange). After one hundred trials of both scenarios, the probe took on average 273 million CPU cycles to complete while not subject to memory-bus locking from a collocated instance, and 517 million CPU cycles while subject to memory-bus locking from a co-located instance. This results in an 89.5% increase in latency. At almost 2x slower, locking the memory bus is clearly a viable QoS attack.

From this plot we also see an interesting pattern arise. Under normal conditions the probe execution times remain stable, but increase for one run at regular intervals. Similarly, the plot of probing times under locking conditions shows a regular increase in the number of cycles needed. We hypothesize that this result is due to the operating systems scheduling algorithm regularly swapping out the running process to perform other tasks.

Figure 4 displays the range of both the first and third Quartile as well as a range of error for each scenario. For both scenarios, the range the beginning of the first quartile and the

[5] https://github.com/afborchert/pointer-chasing

end of the third quartile are very small, however, the range of the data itself for both scenarios are very large.

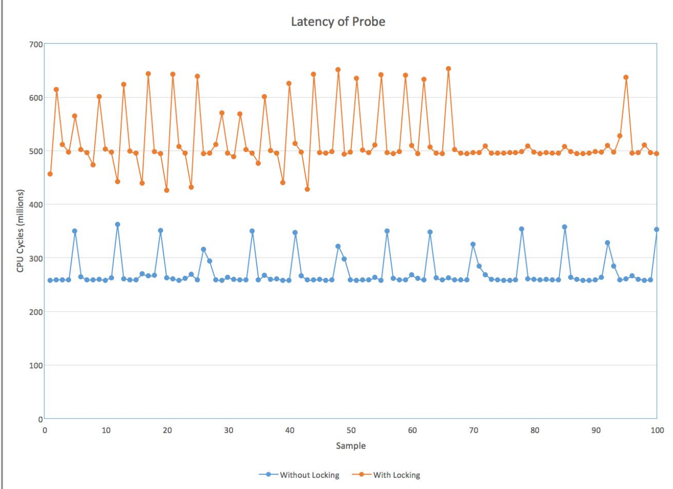

Figure 3. A scatter plot displaying the results of the probe while subject to both memory bus locking (orange) and no memory bus locking (blue).

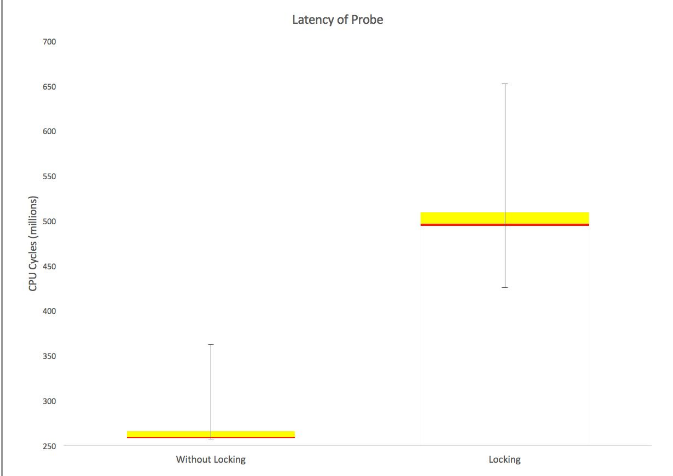

Figure 4. A box and whiskers plot displaying Quartile 1 (Red), Quartile 3 (Yellow), and the maximum and minimum values of both locking and nonlocking samples.

6. Further Discussion

Our experiment took place under the major assumption that the two instances in question were colocated. However this is not the case in the wild. In order to mitigate attacks against specific clients, cloud service providers use placement algorithms to locate instances among their server hardware. By creatively placing instances they can prevent attackers from actively co-locating with a target. While this would solve the problem of attacking a particular client of the cloud service, this would not stop an attack on the cloud service as a whole.

Mitigating a QoS attack against a cloud service provider via hardware exploitation poses the difficult problem of how to differentiate between an attack on the server and what is a legitimate instance. A hypothetical solution would be to flag clients whose instances have high memory costs. With this solution, flagging clients may result in illegitimate flags resulting in wasted time and money in reviewing client's usage. On the other hand, not flagging clients enough may result in this attack being successful and resulting in normal users suffering.

In our implementation of a QoS attack, the locking mechanism locks the memory bus consistently upon every iteration of the for loop, making a pattern that would be easy to recognize by cloud service providers. The decrease in speed was consistent upon every run of the memory locking probe. In order to avoid detection by a cloud service provider, an attacker may attempt to hide the malicious activity of their probe via varying the rate at which it locks the memory bus.

Machine learning and pattern recognition could also be implemented to recognize attacks. This process would use access patterns classified as either benign or malicious. The system would then learn what to look for. However, one drawback of this approach is the lack of training data. While there is an endless source of benign data, there is not so much for malicious behavior. An acute lack of examples of attacks makes this strategy less feasible, but still an interesting thought. Perhaps the access patterns of attackers is so well defined in the case of co-location QoS exploits that it would be possible.

While this project was able to demonstrate the capability for one virtual machine to detect the location of another through measuring latency and that a resident instance can have a significant impact on the performance of other co-located VMs, there are some enhancements that could be made with this procedure. First, we used VirtualBox to simulate a hypervisor. Although it worked fairly well, it is not applicable to the wild, where hypervisors choreograph between many physical machines. Second, only two virtual machines were used in the tests with no other programs running. In the public cloud there may be much more just two instances using a machine. Lastly, our implementation relied on cooperation between the two VMs to detect the attack. In the real world an attacker would need to come up with a creative way to force the target instance to lock the memory bus so it can be probed. In spite of the obvious drawbacks of our strategy, it demonstrated in an understandable fashion the underlying mechanisms involved in co-residency detection and co-location base quality of service attacks.

Finally, for Bracket, this presents an opportunity to develop efficient tools that saves the cloud provider complexity in their hypervisor, reduces overhead when a certain level of security is not necessary and also provides safety for Bracket clients. The *metavisor* sits between the hypervisor and the VM instances to monitor the behavior and check for inconsistencies. It is responsible for maintaining the integrity of communication between cooperating servers, compute nodes and storage nodes. Our intuition is that Bracket's product is in a unique position to monitor for possible QoS side channel attacks like the memory locking example demonstrated by our project. However, there are difficulties in both detecting and mitigating the attacks. If an exploit in progress is discovered, then the recourse is not simple. Instances must be migrated, or user processes killed. Migration can be costly in terms of computation and killing processes falsely labeled as malicious is bad for business. Further research into both co-resident exploits as well as possible responses to such exploits is necessary, but we believe that Bracket's positioning provides a special ability to provide higher security without putting the burden on cloud providers.