Microarchitectural side channel attacks (USRA)

Marcus Lai

July 2024

1 Introduction

The Cloud as a Service (CaaS) model has experienced significant growth in popularity in recent years. As we enter the age of big data, the need to store and analyze data at a scale that is unmanageable without the extensive resources provided by cloud infrastructure has become paramount. To optimize resource utilization, cloud providers enable clients to share hardware and achieve isolation through additional abstraction layers, such as virtual machines and containers. However, this resource sharing introduces the potential for side-channel attacks, particularly cache timing attacks like Flush+Reload and Prime+Probe. Although many cloud providers consider these attacks impractical, recent work by Zhao et al. has demonstrated their feasibility by executing a Prime+Probe attack on a vulnerable ECDSA implementation in Google Cloud Run, successfully extracting 81% of nonce bits from a victim container.

2 Background and Related Work

2.1 Cache

The temporal locality of programs, where the same memory locations are frequently reused, has led to the implementation of smaller, temporary storage closer to the CPU, known as cache, to facilitate faster access. Modern systems typically feature three levels of cache: L1 cache (split into data and instruction caches), L2 cache, and Last Level Cache (LLC or L3 cache). Each subsequent level of cache is larger than the previous one. L1 and L2 caches are private to each CPU, while the L3 cache is shared across the system. Cached memory is organized into units called cache lines, usually 64 bytes each.

To maintain cache coherency, mechanisms are in place to track whether a cache line has been modified, ensuring that outdated instructions or data are not used by the CPU for further processing.

2.2 Flush and Reload

What kind of attack and what data the attack recovers: The paper describes an attack that targets the shared Last-Level Cache (L3 cache) in modern processors. Specifically, it aims to exploit cache access patterns to extract sensitive information from other processes running on the same system. The paper specifically demonstrates the attack on OpenSSL, aiming to recover secret keys used in cryptographic operations like AES and RSA. By observing cache usage patterns, the attacker can infer the sequence of operations and the data being processed, leading to the recovery of these keys.

This is a side-channel attack, more specifically a cache-timing attack. Side-channel attacks exploit information that leaks from the physical implementation of a system, rather than weaknesses in the algorithm itself. In this case, the "Flush+Reload" technique observes differences in cache access times to gain insights into the operations being performed by a cryptographic algorithm.

Assumptions about the attacker: The paper assumes that the attacker and the victim share the same physical machine, such as in a virtualized environment or a shared cloud infrastructure. The attacker needs to have the ability to execute code on the same processor as the victim. The key assumption is that the attacker can flush specific cache lines and then reload them to observe if the victim has accessed those cache lines in the meantime. This implies that the attacker has knowledge of the memory layout and the ability to flush and reload cache lines efficiently.

Results: The results in the paper demonstrate that the Flush+Reload attack is highly effective at recovering cryptographic keys with a high success rate. Specifically, the paper shows that the attack can recover 98% of RSA keys from OpenSSL with high accuracy. The attack is particularly potent on platforms where processes share the same physical processor and L3 cache, such as on Intel processors with Hyper-Threading, or in virtualized environments like those found in cloud services.

Limitations:

Shared Environment Requirement: The attack requires that the attacker and victim share the same physical machine, which might not always be the case.

Knowledge of Memory Layout: The attacker needs to know the specific memory addresses or cache lines that the victim's code uses, which may not be straightforward to determine.

Defensive Measures: Modern systems may employ countermeasures such as cache partitioning or constant-time algorithms that can mitigate the effectiveness of such attacks.

Dependency on Specific Architectures: The attack relies heavily on certain CPU architectures, such as Intel's, which have shared L3 caches and specific cache management behaviors. Different architectures or more isolated environments might reduce the effectiveness of the attack.

2.3 Prime and Probe

Assumption of the Attacker vs the Victim: The attacker is assumed to have access to the same physical machine as the victim, but not necessarily the same core or memory. The attacker could be another VM in a cloud environment or a process with lower privileges. The critical assumption is that the attacker can run arbitrary code on the same machine but does not need direct access to the victim's memory space.

Results: The study successfully demonstrated the extraction of cryptographic keys across VMs with a significant degree of accuracy. The paper reports a covert timing channel bandwidth of up to 1.2 Mb/s, indicating the attack's efficiency.

Limitations: The attack's success relies on certain conditions, such as the ability to fill and probe the LLC accurately, which may require fine-tuned timing and an understanding of the cache architecture.

Additionally, countermeasures like cache partitioning, secure cache designs, or memory encryption could mitigate the effectiveness of the attack, although these are not yet widely implemented in commercial systems.

2.3.1 Eviction Set Generation

2.4 LLCFeasible

LLCFeasible extends previous works by demonstrating a Prime+Probe style attack on modern Intel Skylake architecture, which features a non-inclusive L3 cache. Below is a summary of their findings.

Challenges of Modern Cloud The LLCFeasible paper discusses two primary challenges of executing a cache timing attack in a modern cloud environment: 1) system noise and 2) a dynamic victim. Due to resource sharing, the LLCs of cloud machines are significantly noisier than those of controlled local servers. Additionally, victim processes in the cloud are often ephemeral. Consequently, a cache attack implementation, including eviction generation, must be both fast and stable.

Attacking a Non-inclusive Cache: the Snoop Filter A non-inclusive cache allows a cache line to persist in the victim's private cache without being present in the shared cache (LLC). This is problematic because previous cache attacks have relied on the inclusivity of caches to trigger eviction from the private cache by causing an eviction from the shared cache. However, recent works propose the existence of an inclusive and shared structure to ensure cache coherency, referred to by different groups as the *coherence directory* (CD) or the *snoopy filter* (SF). On systems with non-inclusive caches, the SF serves as a substitute target for cache contention, enabling Prime+Probe style cache attacks.

While the interactions of the SF, LLC, and private lines are not fully understood, the LLCFeasible paper suggests a few properties and behaviors of the SF based on empirical observations and previous works:

1. Exclusive or Modified cachelines, or private cachelines, are tracked by the SF
2. Shared cachelines in at least one private cache is tracked by the LLC
3. If an SF entry is evicted, the private cacheline is evicted ("inclusivity")
4. When a line in LLC needs to transition to state E or M due to access, it is removed from the LLC and an SF entry is allocated to track it.
5. When a private line transitions to a shared state, the SF entry is freed and the private line goes into the LLC.
6. The SF has the same hash function and number of sets and slices as the LLC, but has more ways (on Intel Skylake). Thus an SF eviction set is also an LLC eviction set.

2.4.1 Eviction Set Generation

Candidate Set Generation In virtual-to-physical address translation, the least significant 12 bits of the addresses are fixed and used as the page offset (since 4kB is the default page size). Since L2 set indices are bits 5-15 of the physical address on Skylake, generating a candidate set for a particular address with the first 12 bits fixed introduces four uncontrolled bits during virtual-to-physical address translation. Therefore, the paper describes the uncertainty of L2 cache as $2^4 = 16$, indicating that, on average, one in every 16 candidates with the same first 12 bits as the target will be congruent. L3 uncertainty is computed similarly, but with the additional consideration that L3 slices cannot be reliably predicted and controlled. Consequently, any candidate with the fixed first 12 bits would have an uncertainty of $n_{slices} \times 2^5$. This can be interpreted as: "an arbitrary cache line is unpredictably mapped to a slice (1 in every n_{slices} lines maps to the same slice). Given that a line maps to the same slice as the target, every $2^5 = 32$ lines with the same fixed first 12 bits would map to the same cache set."

Eviction generation begins by allocating a large enough buffer that contains an eviction set for a fixed or arbitrary cache line. LLCFeasible suggests empirically that allocating $3UW$, where U is the uncertainty of the cache level to be evicted from and W is the number of ways of that cache level, is sufficient to ensure an eviction set exists in the candidate buffer. Recall that U represents the number of lines needed on average to obtain a congruent address; thus, the formula aims to allocate three times as much memory as needed on average to obtain W congruent addresses to the target cache line.

Contribution 1 This paper introduces three optimizations to enable eviction set generation in the cloud environment. The first optimization involves further pruning the candidate set with fixed first 12 bits. This optimization leverages the fact that the L3 set index bits are a superset of the L2 set index bits, meaning any address that does not map to the same L2 set cannot map to the same L3 set. Therefore, after generating a candidate set (a set of addresses with the same lower 12 bits as the target address), an L2 eviction set is generated for the target address and used to prune any line that cannot be evicted by this set. Since the uncertainty of L2 is 16, this pruning reduces the candidate set size by a factor of 16.

Minimal Eviction Set After allocating a candidate buffer, it is used to generate the minimal eviction set. Two main classes of algorithms exist for eviction set generation:

1. *Group Testing*: This class of algorithms removes chunks of the buffer and rechecks for eviction. If eviction still occurs, a minimal eviction set must be within the remaining buffer, so the removed addresses are pruned. If eviction does not occur, the algorithm restores the addresses.

2. *Prime and Scope*: This class of algorithms accesses a growing set of addresses until the "breaking point" (the iteration where eviction first occurs) is found. The most recently added address, which is congruent to the target, is then saved into the eviction set. This process continues until W addresses are found, where W is the number of ways of the cache to be evicted from.

In essence, these algorithms are opposites: Group Testing prunes unnecessary addresses to find a minimal eviction set, while Prime and Scope constructs an eviction set by adding one congruent line at a time.

Contribution 2 Another contribution of the paper is an optimization to the Prime and Scope type eviction algorithm. Instead of incrementing the growing set one line at a time until the "breaking point" is found, LLCFeasible uses binary search to identify this point.

Contribution 3 The final contribution of the paper is the exploitation of "memory parallelism." While the paper does not elaborate in detail, this likely involves removing lfences between accesses during candidate or eviction set traversal, allowing memory lines to be read in a superlinear fashion. This also accelerates the prime and probe phases of the attack. In their experiments, this optimization reduces the probing time to only slightly slower (by around 20 cycles) than Prime and Scope (where only one cache line is probed) and significantly improves the latency of the prime phase.

Overall, these three contributions enabled the generation of eviction sets in the cloud environment and the successful execution of a Prime+Probe attack on a naive ECDSA implementation.

2.5 CoResident Evil: Covert Communication In The Cloud With Lambdas

The paper explores the feasibility of constructing a covert communication channel using serverless cloud services, specifically AWS Lambdas.

What kind of attack and what data the attack recovers: The paper explores the feasibility of constructing a covert communication channel using serverless cloud services, specifically AWS Lambdas. A covert channel allows for the transfer of data that is not detectable by traditional methods of monitoring or auditing. It achieves this by encoding data into something that is visible but not considered a communication medium. These channels are used when attackers have access to a victim's system and would like to retrieve the data on the system undetected.

Assumptions about the attacker: The paper assumes that the attacker is an insider, with access to the victim's lambda deployment system and knowledge of the cloud region where the victim is operating in. The attacker is also assumed to have the capability to launch lambdas from their own account.

Key Contributions:

Fast Co-residence Detection: A reliable, scalable, and fast (seconds for thousands of Lambdas) co-residence detector based on memory bus contention.

Dynamic Neighbor Discovery: A protocol for co-resident Lambdas to communicate their IDs using a hardware-based covert channel, enabling enumeration of co-resident neighbors and avoiding unwanted communication interference.

Covert Channel Demonstration: Demonstrated the establishment of Lambda covert channels on AWS, studying the feasibility and capacity of such channels.

Results It demonstrates a fast and scalable co-residence detection method using memory bus hardware. This method allows the dynamic discovery of co-resident Lambdas and enables data transfer at a rate of about 200 bits per second. The authors establish that covert communication via Lambdas is feasible.

Limitations: One limitation mentioned by the paper is the reliance on the lambda support for low-level languages by cloud providers since they enable the pointer arithmetic that is necessary for activating the memory bus channel. Also, since the channel relies heavily on the memory bus hardware locking, it is also possible to prevent the attack by fixing the underlying memory bus channel that it employs.

How does this paper compares with our work: Our work is focused on a side channel attack that takes advantage of the cache and how certain parts of the cache hierarchy are shared between multiple cloud users. A side channel attack is a method used to extract sensitive information (information such as cryptographic keys) from a system by exploiting indirect data sources such as physical or implementation-specific characteristics of the system. In our work, the attacker does not have access to the victim's system.

3 Experiments and Results

3.1 Latency Profiling of Cache Hierarchy on leapx Servers

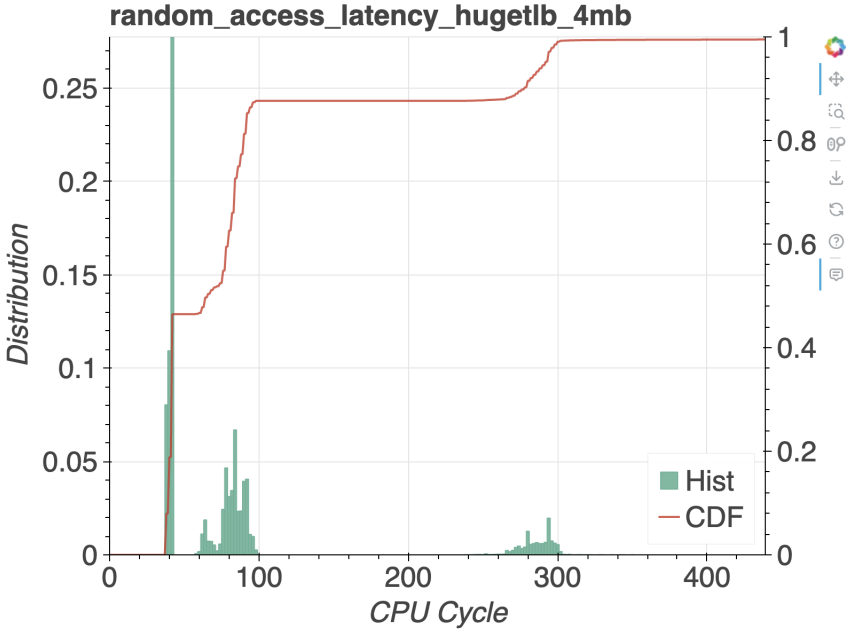
The ability to reliably map cache latencies to accesses on different levels of cache is critical for cache timing attacks. The following outlines our experiments to understand the cache latencies of our local server.

3.1.1 Singlecore and Multicore Random Access

A straight-forward and effective way to obtain the different types of cache latencies is to simply allocate a large buffer and access it multiple times. In theory, accessing a large enough buffer enables us to observe cache hits from all parts of our cache hierarchy and should allow us to piece together a distribution for the access overhead from all levels of our cache. It is worth noting that the first access of a flushed cache line must yield DRAM latency and that a subsequent access must yield L1 latency.

Figure 1 demonstrates the result of a single-core experiment that accesses a 4Mb buffer randomly (as opposed to sequentially which may trigger prefetching). An immediate observation to note is the presence of only 3 peaks. This is unintuitive because it is well known that cache hierarchies have 3 levels, so this brute force experiment should yield 4 cache peaks with the addition of a DRAM peak. Two initial hypotheses were:

Figure 1: Single Core Random Access — 4 Mb Buffer



1. **the peaks of two of the cache levels are merged since they are not distinct enough**

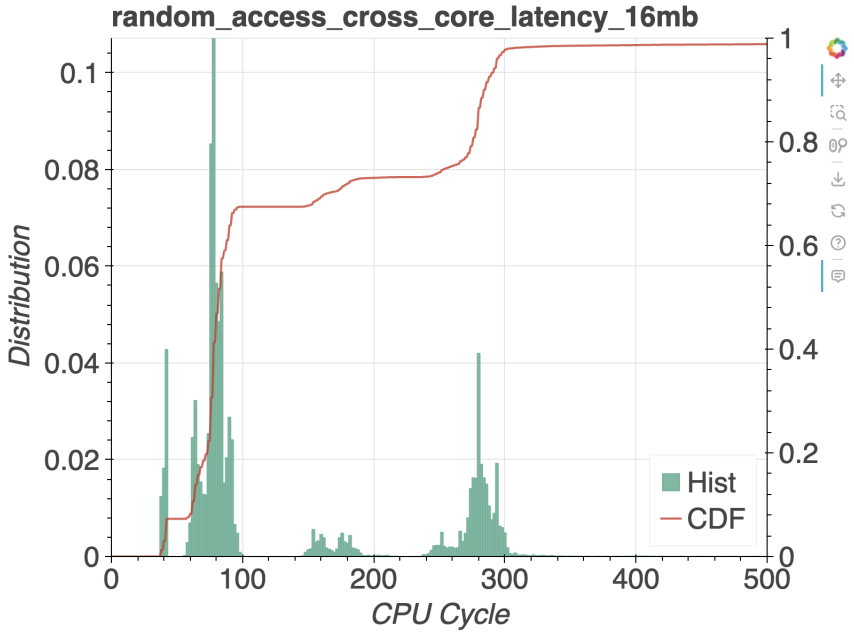
2. **single core access doesn't occupy L3 cache**

This motivated a second test where two cores are used to access the same set of cache lines, displayed in Figure 2. We observe that the rest of the peaks align and an additional peak emerges between the peak with the highest and second highest latency. We had initially believed that this new peak was L3 latency, and that single-core programs don't actually occupy the L3 cache. However, we now understand this extra peak as "L2-Remote" or "L1-Remote".

3.1.2 The 3-core Experiment

As part of our attempt to construct an eviction set to perform Prime and Probe style attacks, we tried to construct an L1-to-L2 eviction set for a fixed, arbitrary cache line x (achieved by accessing cachelines with the same first $n >= 12$ bits since L1 is virtually mapped). However, we observed that accessing more conflicting lines of x than the documented ways of our L1 did not evict x. This remained the case when we accessed over two magnitudes more conflicting lines than necessary for eviction and fixed more bits of the eviction set to ensure cache conflicts ($n = 15$ instead of $n = 12$, for example). Moreover, an extended experiment where we measured a reaccess of the entire eviction set demonstrates

Figure 2: Multi Core Random Access — 16 Mb Buffer



that every line was in L1. Not only did this rule out the possibility that a non-LRU eviction policy was interfering with x's eviction, but it also suggests that we did not evict anything from L1 at all. This suggests three possibilities:

1. **Our eviction set doesn't map to the same set**
2. **L1 and L2 share the** 38 **to** 42 **cycle peak**
3. **One of the L1 or L2 peak is missing**

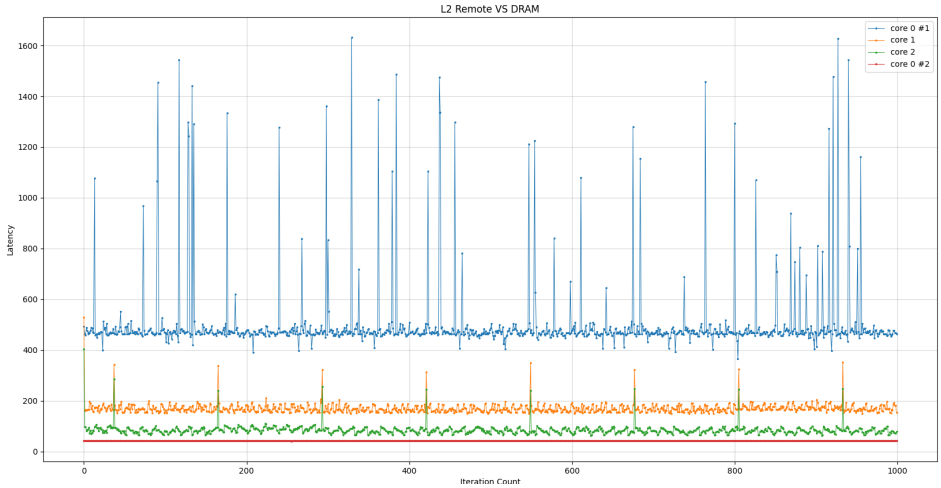
This result brings our previous latency results into question, since what should have been L2 accesses gave L1 latency.

Through the CD paper, we learned about the *Remote-L2* access which is triggered when a requested cacheline exists in the private cache of another CPU, resulting in a fetch from that CPU instead of DRAM. The CD paper also suggests that this Remote-L2 access would put the requested line into L3.

Using this, we designed a 3-core experiment where 3 cores would access a fixed memory line in succession. If Remote-L2 is possible and does indeed put the accessed share line into L3, we should observe a DRAM access by the first core, a Remote-L2 access by the second core and a L3 access by the third core. The results of our experiment are shown in Figure 3.

In Figure 3, Core 0 first accesses x twice to confirm that x is cached properly (the first access yielding DRAM latency and the second yielding L1 latency).

Figure 3: 3-core experiment



Then, Core 1 accesses x, yielding Remote-L2 latency. Next Core 2 accesses x, yielding L3 latency.

Figure 3 reinforces our new interpretation that L1 and L2 latency share the same peak, or otherwise that we are missing one of them in our latency measurements. This is because suppose the second peak from the single-core random access experiment result (around 90 cycles) is indeed L2. Then it would imply that in our 3-core experiment, the 3rd core has x stored in its L2 private cache before it has even seen or accessed x. Instead, a better explanation should be that the 90 cycles peak is L3 latency and that the new peak we see in our multi-core random access experiment that we identified as L3 is in fact Remote-L2 accesses.

We were able to confirm this theory more systematically by running the 3-core experiment using intel-pcm. Using the cache hit counters, we confirmed that without the 3rd core's accesses, which correspond to the 90 cycle peak, we did not get L3 hits (suggesting indeed that the 90 cycle peak is L3). Furthermore, pcm confirms that we were getting L1 and L2 hits even before we observe the second peak. Although this could be noise from the system causing extra L1 and L2 hits, we believe it is highly likely that they share the first latency peak.

3.1.3 L1 Latency and L2 Latency

L1 and L2 hits sharing a distribution, and one with very low variance at that, suggests potential issues with our measurement function itself. On a high level, our measurement function is a memory load wrapped before and after by a serializing macro-instruction in assembly. A small experiment where we remove the memory load from our measurement function and simply run the serializing

instructions confirms our suspicion that our measurements cannot be taken at face value, since we still get the exact same latency of 38 to 42 that we previously identified as L1. One natural theory we have is that the serializing instructions only ensure our reads finish before the serializing instructions, but do not prevent **concurrent execution** between the serialization and the load. We further hypothesize that since the load is executed faster than the serializing instructions, the latency of the load is masked by the serialization; thus we only see the overhead of the measurement.

To test this theory and reduce the effect of concurrent execution, we designed an experiment with an averaging approach to load a fixed cacheline a large number of times between the serializing instructions, then divide the total latency by the number of loads executed. This consistently gave us 7-9 cycles per access even when we varied the number of loads (as long as we had a *large* number of loads $n >= 10000$), which could potentially be interpreted as the *true* L1 access. However, just as reads can execute in parallel to the serializing instructions, nothing prevents the reads from potentially executing in parallel either. Thus, we were not able to guarantee that 7-9 cycles is the true L1 latency.

This averaging approach would not work on L2, since we would need to first know that a cacheline is in L2 before can measure it, which we cannot guarantee.

A final experiment we conducted was to try other serializing instructions other than the default rdtscp such as cpuid, rdtsc, but we did not achieve better results.

3.1.4 Other Findings

One important thing to note is that our measurements aren't returning in units of CPU cycles. This was shown by a shifted result when we reduced the frequency of our CPUs (from the default 2.4GHz to 800Mhz). The result of the 3-core experiment on a lower clock frequency is shown in Figure 4. This suggests that we cannot reliably compare our measurements with previous works, since we may not be comparing the same timestamp or counter.

3.2 Eviction Set Generation Experiment for Prime+Probe Attack

3.2.1 Single Core Eviction Set

3.2.2 Multi Core Eviction Set

4 Current Status and Next Steps

Figure 4: 3-core experiment (800 Mhz)