Startup Project
Flush This: A replication of FLUSH+RELOAD

Erickson, Jeremy
University of Michigan, Ann Arbor
Advisor: Professor Kevin Fu

1 Summary of Results

- Verified cache timing and threshold information.
- Experimentally reproduced shared pages and cache timing side channel.
- Was unable to regenerate secret keys from cache timing side channel.

2 Introduction

In this paper, I attempt to replicate the work presented by Yuval Yarom and Katrina Falkner at Usenix 2014 entitled FLUSH+RELOAD: A High Resolution, Low Noise L3 Cache Side-Channel Attack[5]. This paper presents a new variant on the standard cache side channel attack leveraging artifacts of shared memory and cache timing to infer the instructions another process is executing. From that, the authors are able to accurately guess the component bits of the secret key used in the target Gnu Privacy Guard (GPG)[2] application.

Cache side-channel attacks are a well-known attack category. For one process (the Spy) to infer data about another process (the Victim), the Spy typically fills the processor cache by loading data from memory, then periodically checks to see which data is evicted from the cache as the Victim loads other data into memory. Depending on the cache architecture and which cache lines are evicted, the Spy may be able to infer which data the Victim has loaded into memory, thereby gaining insight into the operation of the Victim process. This is of particular interest in scenarios such as when the Victim process is encrypting or decrypting a message with a secret key. The ability to capture all or part of the bits of the secret key can allow an adversary to decrypt future (and past) secret communications, masquerade as a particular entity, or surreptitiously modify data in transit.

In the last decade, we've seen the shift from single-core processors to multi-core processors. In such an environment, the Spy faces an additional hurdle — the Victim process may be executing on a different processor core, and may therefore not share the same upper-level caches. In recent Intel processors, the L1 and L2 caches are replicated for each individual core and the L3 cache, or Last Level Cache (LLC) is the only shared cache. Recent cache side-channel attacks[4] have focused on this cache for reliability when the Spy and Victim processes do not share the same core.

3 Background

3.1 Shared Pages

When a process is executed, the operating system reads its code and data from disk and makes a copy in memory. So that two processes may not interfere with each other, each process is given its own virtual address space. Each 4096 byte (on today's hardware) segment (called a *page*) of virtual memory is mapped to a corresponding page in physical memory.

In several circumstances, the operating system will map two virtual pages from different processes to the same physical page. For instance, shared libraries, which are libraries loaded at runtime by multiple processes simultaneously, will map to the same read/execute-only physical pages because the library will be identical in both processes. Why should the operating system need to allocate twice the memory for duplicate code?

In another case, hypervisors will often enable explicit memory deduplication via the Kernel Same-page Merging functionality in modern systems. In this scenario, the operating system will periodically scan physical memory for duplicate pages, and if it finds them, it will update the page table to map all virtual pages with the same contents to the same single physical page. It will also mark that page "Copy on Write" such that if any process modifies the page, it will first create a new copy. This continues to preserve the important memory isolation between processes. It is important to note that this feature is only available on systems that have explicitly enabled it, and further only for processes that have also specifically enabled it. Hypervisors frequently enable this feature because the virtual machines (VMs) they run often contain very similar memory. A huge memory savings

can be realized by deduplicating the identical pages between multiple VMs.

As it turns out, there is apparently a third, largely undocumented way for processes to share the same virtual-to-physical page mapping: One process can use the *mmap* function to manually map a second process's instructions and data into its own virtual memory space. Presumably because the two sets of virtual pages originated from the same file and are otherwise identical, they appear to map to the same physical page. This technique does not appear to be widely known, and is only obliquely mentioned in [5] on page 6.

3.2 Cache Architecture

The modern Intel processor architecture uses a three-level physically-addressed inclusive cache hierarchy. Inclusive means that as instructions are retrieved from memory, they will populate all three levels of the cache. For the purposes of this attack, it is important that the LLC be populated with instructions as they are executed, as this attack is designed to work across cores and the LLC is shared between all cores. Additionally, the cache is physically-addressed, which means that if an instruction is cached by one process, it will be present in the cache when another process requests it. By measuring the time it takes to load an instruction, the Spy process can infer whether or not the Victim process has recently executed it.

There is a special instruction, *clflush*, that allows any user process to flush a particular address from all cache layers. When used in conjunction with a shared memory address, it will evict the shared memory from the cache.

3.3 RSA Algorithm

The RSA algorithm is complex and a discussion of how it works will not be presented here. However, for the purposes of this attack it is important to describe the sequence of mathematical operations that will occur during the process of encrypting or decrypting a message with a secret key.

The implementation of GPG version 4.1.13 uses the square-and-multiply exponentiation algorithm which scans the bits of the binary representation for the sub-keys d_p = *d mod (p-1)* and d_q = *d mod (q-1)*. With these keys, it is possible to factor n and retrieve d.

For each bit of the subkeys scanned, GPG will either perform a square operation followed by a modulo reduce operation, if 0, or a square operation followed by a modulo reduce operation followed by a multiply followed by another modulo reduce operation, if 1.

4 Attack Description

To perform this attack, the Spy process first *mmap*s the GPG binary into its address space. This maps the GPG pages in the Spy process to the GPG pages in the Victim process. Next, the Spy process repeatedly loads target memory addresses for the square, reduce, and multiply functions, times how long it takes to retrieve the instructions, and uses the *clflush* instruction to flush the instruction from the cache. Since the instruction is flushed every cycle, the expectation is that on the next memory load, it will take a relatively long time to retrieve the instruction from main memory. However, if it takes a short time to retrieve the instruction, the Spy can infer that the instruction was cached, and was therefore executed by the Victim process.

Using the ability to infer which instructions the Victim process is executing, the Spy can produce a sequence of square-reduce-multiply-reduce (1 bit) and square-reduce (0 bit) operations and recreate the RSA subkeys with relatively high accuracy.

5 Replication

In replicating this work, I attempted to follow the authors' process as closely as possible. I used similar Intel hardware and used the provided assembly code for flushing cache instructions and timing instruction retrieval.

5.1 Cache Timing Thresholds

First, as done in the paper, I experimentally determined the length of time it takes to perform a memory read when the line is cached and not cached by repeatedly loading a memory address both with and without flushing it from the cache. When the line is cached in the L1 cache, it takes approximately 37 ticks, or processor cycles, to retrieve the instruction. When the instruction is not cached at all, it takes approximately 210 ticks, although with a high degree of variance. When running the later experiment, I observed that many instruction retrievals took approximately 96 ticks, while some took approximately 44.

[3] asserts that the instruction retrieval time from the L1 cache is 4 ticks from the L2 cache is 10 ticks, and 65 ticks from the L3 cache when the line is shared in another core. Assuming the fixed cost of the code to perform the memory read is approximately 33 ticks, it follows then that my experimental cache access times, minus overhead, were approximately 4 ticks, 11 ticks, 66 ticks, and 177 ticks for L1, L2, L3, and main memory, respectively. This agrees with [3].

While access times to main memory differ significantly, in only very rare circumstances did my main memory accesses (including overhead) dip below 200 ticks, and never below 150 ticks. Therefore, I believe a threshold value of 120 ticks to determine whether an instruction is cached or not, as chosen in the paper, is reasonable.

5.2 Finding Instructions to Time

Contrary to how the square-reduce-multiply-reduce routine is presented in the paper, the GPG code base is com-

plex and poorly documented. The main square- reduce-multiply-reduce routine is found in mpi-pow.c, and the calls to the square, multiply, and reduce functions are the mpih_sqr_n_basecase, mul_n_basecase, and mpih_divrem functions, respectively. In the mpih_sqr_n_basecase and mul_n_basecase functions, I selected an instruction inside a loop and not near the beginning of the function, to avoid spurious retrievals due to speculative execution. However, the mpih_divrem function is a large switch statement. So as to select an instruction that is executed on every invocation of the function, I selected the return instruction at the end of the function.

5.3 Inferring Instruction Execution

Using a separate Spy process, I divided time into a series of time slices of 2500 ticks, as done in the paper. I repeatedly flushed and reloaded the instructions at the specified memory locations for the square, multiply, and reduce functions. During normal computer operation with no Victim process running, all instruction retrievals came from memory, as expected. When the Victim process began to run, immediately the spy process detected a sequence of instructions retrievals that took less time to retrieve than the threshold. Each of these instruction retrievals indicates that the Victim retrieved and executed the instruction. From this, I generated a set of operations that the Victim executed during each time slice, some combination of square, multiply, reduce.

5.4 Reproducing Secret Keys

As described in Section 3.3, the secret subkeys can be reproduced by watching the sequences of square-reduce and square-reduce-multiply-reduce, which indicate bit values of 0 and 1, respectively. Using this model, I identified sequences of square-reduce and square-reduce-multiply-reduce, and built a sequence of 0's and 1's.

I retrieved the GPG private key, stripped out the GPG header information, and translated the hex keys for d_p and d_q to binary. I then manually compared them against the sequence of 0's and 1's generated by the Spy process. Unfortunately, they did not appear to match or even be similar. Section 6.3 discusses this problem.

6 Discussion

6.1 GPG and Static Linking

As mentioned in Section 3.1, [5] describes several ways in which two processes may share the same physical page. Unfortunately, from the paper's Introduction section, it is implied that GPG's mathematical operations are part of a shared library. Because of this, I originally tested this side channel on the *libc* shared library, for which I was able to clearly determine that the virtual addresses for both Spy and Victim processes mapped to the same physical pages. Upon discovery that GPG is entirely statically linked, I scoured the paper for an explanation of how the Spy process may have shared physical pages with the Victim process and found a reference to *mmap* on page 6.

However, after using *mmap* to load the GPG code into the Spy process's address space, I used the */proc/<pid>/pagemap* files for both the Spy and Victim processes to determine to which physical pages their virtual GPG pages mapped, and regardless of the options I used to do the mapping, the page frames always differed.

I was therefore surprised to discover that the Spy process was still able to retrieve cache timing information from the Victim process executing specific instructions. It is possible that this discrepancy is due to a coding or other experimental error. It is also possible that this indicates a mystery worth investigating further, such as a bug in Linux.

6.2 Syscalls

In my initial approach, I had the Spy process detect uses of the *printf* function from the *libc* shared library. This produced inconsistent instruction retrieval times. I believe this is because invocations of the *printf* function that write to the terminal require a syscall, which forces a context switch. When calling printf frequently, I suspect the resulting frequent context switching caused the irregular and much larger retrieval times between 400 and 800 ticks that I observed.

6.3 Trace Resolution

As mentioned in Section 5.4, my regeneration of the secret subkeys did not result in subkeys matching the originals. In an effort to narrow down the scope of the error, I manually reviewed several segments of ordered inferred operations (square, multiply, and reduce) and confirmed that my code to turn these traces into individual bits was behaving as I intuited the bits to be. Given that this code appears to be operating correctly, I suspect that the error comes from insufficient accuracy of the side channel itself in my experimental setup. There are many factors that could contribute to this, such as picking unsuitable instructions to watch for execution or operating while the system was performing other operations that may have interfered.

In an effort to determine if I had made a fundamental error, I cloned and built a GitHub-hosted flush-reload project[1]. This was a replication of the FLUSH+RELOAD project in 2014, soon after the original paper was published. Unfortunately, this code did not work as advertised, presumably because when I compiled GPG, all of the static addresses they had used were slightly different than the compiled binary on my system. Using their annotated instruction addresses, I regenerated new addresses that referred to approximately the same assembly instructions and ran their code. Unfortunately, despite detecting large numbers of square, multiply, and reduce operations, their code was unable to generate secret subkeys either.

References

[1] Daniel Ge, David Mally, and Nick Meyer. *Implementation of the FLUSH+RELOAD side channel attack*. URL: https://github.com/DanGe42/flush-reload.

[2] *GnuPG*. URL: https://www.gnupg.org/.

[3] David Levinthal. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. URL: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[4] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *IEEE Symposium on Security and Privacy 2015*. May 2015. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7163050.

[5] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.