(AIN'T NUTHIN' BUT A) PC THANG: Detecting and Mitigating Cache-based Microarchitectural Attacks Using Protean Code

(Working Title!)☺

Jeremy Erickson *Sai Gouravajhala* *Akshitha Sriraman*
EECS 583, Fall 2015
University of Michigan, Ann Arbor

1 Introduction

Security fixes often come at the expense of performance. For instance, intentionally introducing spurious operations is a common defense against side-channel attacks. Such operations may include introducing noise or forcing running time uniformity across algorithmic operations. However, this is fundamentally at odds with the goal of running efficient and fast programs. In an ideal world, we want code to run quickly, but also be resistant to attacks.

In this paper, we propose PC THANG, a general-purpose framework for implementing *on-demand* security measures when side-channel attacks are detected. We hypothesize that we can take advantage of efficient, yet insecure, code when no attack is detected, but trade efficiency for security when an attack occurs. We build this functionality on top of Protean Code [5], a unique dynamic compilation framework with minimal performance overhead itself.

To illustrate the system's potential, we present an on-demand defense against the FLUSH+RELOAD [6] cache side channel attack on GnuPG [2]. We aim to show that the PC THANG system not only detects the side-channel microarchitecture attack on the cache, but also mitigates the attack. Furthermore, we aim to show the performance penalty of utilizing this system and compare it with the penalty of using the defense static fix.

2 Background

In this section, we provide background information regarding Protean Code and the FLUSH+RELOAD attack.

2.1 Protean Code

Dynamic compilation enables our platform to adapt to threats that may or may not be present at runtime. In contrast to other dynamic compilers such as DynamoRIO [1] or PIN[7], Protean Code does not interpret the original application and so has negligible ($< 1\%$) performance overhead.

2.2 FLUSH+RELOAD

In the FLUSH+RELOAD attack, a spy process that is co-located with a victim process on the same CPU, but not necessarily the same core, attempts to infer the instructions the victim process is running. It does this by repeatedly loading a memory value shared with the victim process, timing the load, and flushing it from the cache with the `clflush` instruction. If the load returns slowly, it was likely loaded from main memory. If the load returns quickly, it was likely loaded from the shared last-level cache (LLC), indicating that the victim process recently loaded the instruction. In practice, these cases are clearly distinguishable.

This technique relies on two requirements: *a*) that the cache architecture is an inclusive cache[1], and *b*) that the spy process can manage its virtual memory such that it can share access to a physical page with the victim process. In practice, the latter requirement can be achieved on modern Intel processors by using `mmap` to load the victim executable into the spy process' address space.

GnuPG uses the square-and-multiply exponentiation algorithm for its RSA implementation. Versions of GnuPG prior to 1.4.14 were vulnerable to FLUSH+RELOAD: for a given bit of the secret exponent, either a square-reduce-multiply-reduce operation would occur (if the bit were a 1), or a square-reduce operation would take place (if the bit were a 0). By inferring whether the multiply-reduce operations occurred between consecutive square-reduce operations, the spy process would be able to determine whether the next bit of the secret key were a 0 or 1. In version 1.4.14 of GnuPG, the maintainer fixed this issue by performing the

[1] That is, when the victim loads the instruction of interest, it will be present in all cache levels, and most importantly, the shared cache.

multiply-reduce operation in both cases, simply throwing away the result in the event of a 0-bit. However, the maintainer noted that this would incure a "performance penalty."

3 Methodology

Our general-purpose framework consists of two parts: SNOOPDETECT, which detects ongoing side channel attacks, and DREPROTECT, which implements the real-time mitigation measures.

3.1 Using SNOOPDETECT

Because the underlying source of Cache-Based Microarchitectural Attacks (CBMA) is the spy's need to repeatedly flush from and reload the shared value into the cache from memory, gaining visibility into runtime cache miss events is a direct mechanism for detecting such attacks. Recent Intel microprocessors offer the ability to log detailed information about certain architectural events via the Precise Event-Based Sampling (PEBS) performance counter mechanism [3]. When an instruction *i* triggers an event of interest, the PEBS mechanism generates a PEBS record, which the hardware then logs to an in-memory buffer. Each PEBS record contains *i*'s Program Counter (PC), the memory address accessed by *i*, and the values of the general-purpose registers as of *i*'s commit. When the buffer is full, an interrupt notifies the OS' PEBS driver to process the records and provide a new buffer for the hardware to use.

Modern Intel processors support several PEBS events. Of particular interest to us is the ability to track LLC load miss events, which arise when a data/instruction has to be fetched from the DRAM. In this work, we use the `MEM_LOAD_UOPS_RETIRED_L3_MISS` PEBS event.

The SNOOPDETECT system relies on existing hardware and operating system support for the advanced PEBS performance counters available in Intel processors. The detection mechanism uses the Linux `perf` API [4] to configure the hardware to record LLC miss events. `perf` support is a standard part of recent versions of Linux, and allows LLC miss events to be recorded entirely from userspace. Root permissions are not required for a process to monitor its own LLC miss events.

LLC miss records received from the hardware enter the SNOOPDETECT processing pipeline where SNOOPDETECT maps the misses to source code locations and invokes DREPROTECT to provide real-time protection as the victim process continues to run.

3.2 Invoking DREPROTECT

SNOOPDETECT periodically checks the LLC miss rate, triggering DREPROTECT if the rate of LLC miss events on the key PCs exceeds a given threshold. This threshold is directly determined by the rate at which a spy process will have to perform the flush operation in order to launch a successful attack.

DREPROTECT then uses dynamic compilation techniques to thwart the spy's attack, while the victim continues to execute. This is where the tradeoff between efficiency and protection takes place.

3.3 Defense Mechanism

SNOOPDETECT's ability to quickly and precisely detect LLC misses at runtime allows a real-time defense mechanism with minimal application interference. For instance, the FLUSH+RELOAD paper focuses on performance inefficient mechanisms that are always in place to thwart side-channel attacks if and when they occur.

DREPROTECT, however, is capable of achieving a reasonable trade-off between security and performance, by making cryptographic operations secure (thereby, compromising performance), only when SNOOPDETECT reports the existence of a spy.

When DREPROTECT is invoked, it uses the Protean Code infrastructure to dynamically modify the executed code in order to thwart the CBMA side-channel attack. The secure version of the binary dynamically introduced by Protean code contains instructions that establish uniformity or randomness in terms of access time or space utilization.

In order to thwart the FLUSH+RELOAD attack, we adopt the proposed solution of using Protean code to dynamically inject a Multiply Reduce operation along with every Square Reduce operation, thereby creating uniformity across accesses to both 1's and 0's. However, we will mitigate the corresponding performance overhead by only applying it when necessary.

4 Evaluation

4.1 Accuracy

Fundamentally, if the supplied secure code mitigates the vulnerability, our approach should be able to apply the same defense mechanism and mitigate it an equivalent amount. However, there is the possibility that, either through poor detection, or a coding error, our approach will be less effective than the original mitigation. This is unacceptable.

To demonstrate that our approach provides no loss of security, we will develop and test it against a work-

ing implementation of the FLUSH+RELOAD attack and show that the spy process is unable to infer which cryptographic operations the victim is executing, thereby invalidating the attack.

Since the mitigation is not guaranteed to be in effect at any given time, it is important to demonstrate that SNOOPDETECT is able to detect the presence of an attack and DREPROTECT is able to implement the mitigation quickly and under a variety of different operating conditions. Thus, we will evaluate our solution under a range of different loads, from completely idle to 100% CPU load and perform the requisite analysis (e.g., specificity and sensitivity).

4.2 Performance Overhead

As our solution ultimately provides no benefit to security over a static approach, it is important that it maintains the same security benefits while negating a substantial part of the performance overhead for implementing them. The efficiency of our approach is ultimately lower-bounded by the runtime performance of the fast, insecure version of the program, and must by upper-bounded by the runtime performance of the slow, secure version.

In the common case, we intend for the fast, insecure code to be viable, so we plan to demonstrate that when no attack is detected, the protean binary runs with minimal overhead, strictly less than the upper-bound of the slow, secure static binary.

References

[1] *DynamoRIO: Dynamic Instrumentation Tool Platform*. URL: dynamorio.org.

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

[3] Intel(R). *Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*. June 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[4] Linux Programmer's Manual. *perf_event_open(2) Linux Programmer's Manual*. July 2015.

[5] Michael A Laurenzano et al. "Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 558–570. URL: http://dl.acm.org/citation.cfm?id=2742212.

[6] 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.

[7] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005. URL: http://web.stanford.edu/class/cs343/resources/pin.pdf.