PGPG: Detecting and Mitigating Cache-based Microarchitectural Attacks Using Protean Code

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

Abstract—Achieving a reasonable trade-off between security and application performance is a challenging problem, owing to the fact that the two ideas are often inherently at odds with each other. Understanding the threat model for cache-based microarchitectural attacks requires a global knowledge of the memory access behavior of both the application and its co-runners. Previous schemes have focused primarily on security, and either impose significant performance penalties or require non-trivial alterations to the memory hierarchy or the run-time system environment.

In this paper, we present the Protean General Purpose Guard (PGPG) system, which leverages dynamic compilation techniques and hardware performance counters with the aim of overcoming the security-performance gap. When PGPG is tested against the FLUSH+RELOAD attack, we achieve conservative average performance gains of 19.7%, with a worst-case performance overhead of 0.03%, while maintaining application security.

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, yet also be resistant to attacks.

In this paper, we propose PGPG (Protean General Purpose Guard), a general-purpose framework for implementing *on-demand* security measures when side-channel attacks are detected. We take advantage of efficient, yet insecure, code when no attack is detected, but trade efficiency for security when an attack occurs. We build our detection and defense functionality on top of Protean Code [3], a unique dynamic compilation framework with minimal performance overhead itself. As our case study, we develop a microbenchmark that mimics GnuPG [9], a well-known cryptographic software suite.

To illustrate our system's potential, we present an on-demand defense against the FLUSH+RELOAD [4] cache side channel attack on an RSA algorithm microbenchmark. We show that the PGPG system not only detects the side-channel microarchitecture attack on the cache, but also mitigates the attack. Furthermore, we show the performance penalty of utilizing this system and compare it with the penalty of using the static defense. Our method results in an average speedup of 24.78% over the statically-defended version of our RSA microbenchmark when an attack is not present, and only a 0.32% slowdown when an attack is occurring.

Our main contributions are as follows:

- To the best of our knowledge, we are the first to develop a system that leverages dynamic compilation to combine the advantages of fast, but insecure, code, while maintaining equivalent security properties of slow, yet secure, code (Section 3).
- We extend the use of Protean code beyond code optimization and show how it can be used to make modifications to program semantics, such as including a dynamic defense (Section 3.3).
- We develop and evaluate an implementation encompassing a hardware event-based detector and a Protean-based dynamic defense component (Section 4).
- We reimplement the FLUSH+RELOAD attack (spy process) to test the attack detection (Section 2.2) and to ensure that the Defender successfully thwarts an attacker process.

2 Background

We provide an overview of the Protean dynamic compilation system, as well as the FLUSH+RELOAD attack and GnuPG.

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 [8] or PIN [5], Protean Code does not interpret the original application and so has negligible ($< 1\%$) performance overhead [3].

Protean code works by recompiling target functions at runtime to introduce performance optimizations—in this case, security measures—when needed. This just-in-time recompilation occurs in a separate process and can be farmed out to an unused hardware core, introducing negligible overhead in the original program's execution time. When the recompiled function is ready, all references to the original function (conveniently modified by Protean code to use a function look-up table) can be changed to refer to the new function. Thus, all invocations of the original function will invoke the new recompiled function.

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 being run by the victim process. It does this by repeatedly loading an instruction stored in memory that is 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's address space.

2.3 GnuPG

GnuPG uses the square-and-multiply exponentiation algorithm for its RSA implementation. Versions of GnuPG prior to 1.4.14 were vulnerable to

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

```
for(int i = 0; i < 32; i++)
{
  lsb = exp & 1;
  exp = exp >> 1;
  val = square(val);
  val = reduce(val);
  if (lsb == 1)
  {
    val = mul(val);
    val = reduce(val);
  }
}
```

Fig. 1. Vulnerable RSA Square-Reduce-Multiply-Reduce Exponentiation Algorithm.

```
for(int i = 0; i < 32; i++)
{
  lsb = exp & 1;
  exp = exp >> 1;
  val = square(val);
  val = reduce(val);
  val2 = mul(val);
  val2 = reduce(val2);
  if (lsb == 1)
  {
    val = val2;
  }
}
```

Fig. 2. Safe RSA Square-Reduce-Multiply-Reduce Exponentiation Algorithm.

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 is a 0 or 1. Refer to Figures 1 and 2 for more detail.

In version 1.4.14 of GnuPG, the maintainer fixed this issue by performing the multiply-reduce operation in both cases, but simply throwing away the result in the event of a 0-bit. However, the maintainer noted that this would incur a "performance penalty."

3 Methodology

The PGPG framework consists of three parts: a Detector that monitors hardware LLC cache misses; a Decision-Making Engine, running as part of Protean code, that receives the miss counts makes decisions about potential attacks; and. a Defender that uses Protean code to implement the real-time defense.

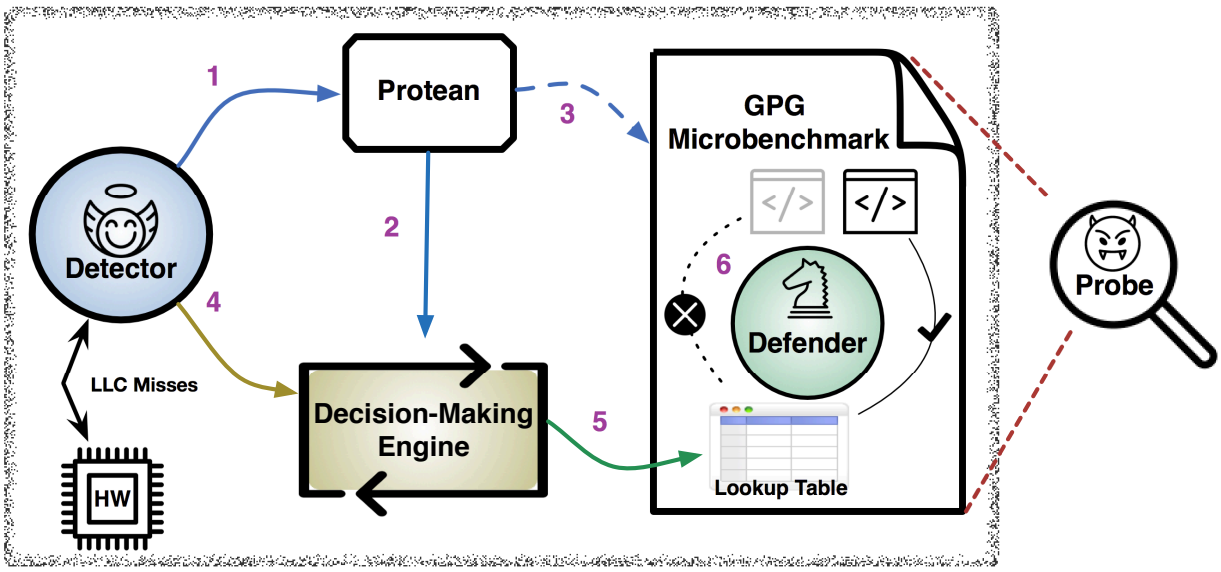

Fig. 3. The PGPG system consists of three major subsystems: a *Detector*, a *Decision-Making Engine*, and a *Defender*. When a spy process probes an application, such as the GPG microbenchmark, the PGPG system comes into play: 1) The Detector spawns a Protean process and begins obtaining the LLC misses from the hardware; 2) The Protean process spawns the Decision-Making Engine, which starts to listen for the LLC miss counts from the Detector; 3) The Protean process also "morphs" into the application binary. In our case, the GPG benchmark is Proteanized; 4) The Detector sends the LLC miss counts to the Decision-Making Engine; 5) The Decision-Making Engine continuously monitors for an attack; when an attack is occurring (based on a threshold value), it updates the Lookup Table entry; 6) The Defender ensures that the previously unsafe, but faster, function is nullified (the "X" path) and that the application uses the safer, but slower, function (the "checkmark" path). This ends up mitigating the attack.

3.1 The Detector

Because the underlying source of Cache-Based Microarchitectural Attacks (CBMA) is the spy's ability to repeatedly flush from and reload the shared value into the cache from memory, gaining visibility into run-time cache miss events is a direct mechanism for detecting such attacks.

The detection mechanism uses the Linux `perf` API [2] 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 user-space. Root permissions are not required for a process to monitor its own LLC miss events. We use `perf` to continuously monitor the absolute counts of `LLC-misses` hardware event generated by the victim process. These hardware events arise when a data/instruction has to be fetched from the DRAM.

The Detector monitors the cumulative cache miss count and derives the rate at which misses occur to pass to the Decision-Making Engine in real-time.

3.2 Decision-Making Engine

The Decision-Making Engine receives a continuous stream of miss rates from the Detector and monitors them to determine when an attack is in progress. As shown in Figure 4, the miss rate (derivative of the LLC miss counts) tends to be very regular, and when a cache attack is in progress, the miss rate increases dramatically. Based upon this characterization of the miss rate, we opt for a simple threshold-based detection scheme. That is, when the miss rate rises above a given threshold $\mathbf{N}$, we invoke the Defender.

Of course, it is expected that occasionally the miss rate will fluctuate above the threshold. This may be due to context switching, other processes using the cache, or another reason. However, when an attack occurs, we observe that the miss rate will consistently remain above the threshold for the duration of the attack. Thus, we only invoke the Defender when the miss rate has exceeded the threshold $\mathbf{M}$ consecutive times.

Due to the large number of cache misses that occur when the Protean engine is starting up, the Detector and Decision-Making Engine coordinate to start monitoring the application only after Protean has initialized and when the vulnerable application starts to run. This prevents the Decision-Making Engine from acting on the initial cache misses and unnecessarily invoking the defense.

In real-world scenarios in which additional confounding factors may be present, this Decision-Making algorithm may not be effective. We discuss further approaches for this problem in Section 5.3.

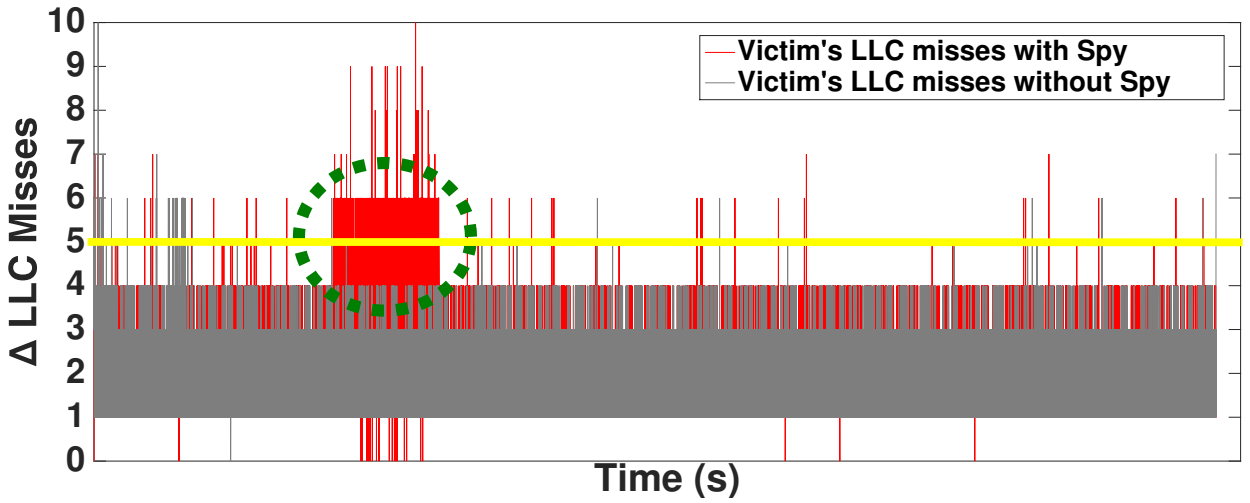

Fig. 4. LLC Miss Rates: When the LLC miss rate (delta of the LLC misses) is plotted for both the Victim and the Spy, the attack can clearly be distinguishable from the baseline rate. The attack is highlighted green area. The threshold for attack detection is shown by the horizontal yellow line.

3.3 Defender

When the Defender is invoked, it uses Protean code to transform the vulnerable code into code that resists the FLUSH+RELOAD attack. This is fundamentally different from how Protean code has been used in the past (i.e., run-time optimizations). While the new, and safe, code delivers the same end result, its computational path is significantly altered such that the cache side channel is eliminated.

Doing this requires the source of both the vulnerable and safe versions of the code base. Requiring source code may restrict the use of this technique somewhat, but there are many cases in which it can be effective. For instance, GnuPG is an open source project and the code is freely available for use. It is trivial to compare the vulnerable GnuPG code against a patched version. Unfortunately, while we would have liked to demonstrate PGPG protecting the GnuPG project, building the GnuPG source with Protean code poses problems. We discuss these in Section 5.1.

As a result, we substitute a microbenchmark that reflects the code paths GnuPG takes when computing the RSA Square-Reduce-Multiply-Reduce Exponentiation Algorithm. In the microbenchmark, the main loop of `compute_val`, shown in Figure 1, is analogous to the main loop in `mpi_pow` in GnuPG.

Protean includes the LLVM [6] compiler to recompile target functions at runtime. When the Defender is invoked, we transform the LLVM bytecode for the `compute_val` function into that produced by a secure version (shown in Figure 2). In the current iteration of PGPG, we do this by directly moving the calls to `mul` and `reduce` above the conditional. However, we discuss more general approaches to generating secure code in Section 5.2.

4 EVALUATION

We evaluate the PGPG system against the FLUSH+RELOAD attack on a 16-physical core, dual socket Haswell server with 124 GB of memory. The processor is a 64-bit Intel Xeon(R) CPU E5-2630 v3 running at 2.40 GHz. Hyperthreading is enabled, allowing the processor to support 16 logical cores per socket. Furthermore, in our experimental setup, we set $\mathbf{N} = 5$ and $\mathbf{M} = 10$ after having performed application profiling.

4.1 Attack Detectability

We consider the loss of secret key material (or the end-result of another attack) to be far worse than can be addressed by any performance increase. Consequently, we consider any degradation in the practical security of the vulnerable PGPG binary when compared to a secure statically-compiled version to be unacceptable.

PGPG's Defender produces compiled code functionally equivalent to the safe version and our testing shows that the safe version takes effect quickly enough that the FLUSH+RELOAD attack is unable to infer the secret exponent. Thus, the primary concern is that our detector may be unable to detect an attack in progress.

In the FLUSH+RELOAD paper, the authors perform the attack on systems devoid of conflicting processes. In our testing of their attack on our own systems, the attack is strongly affected by additional processes running simultaneously causing cache interference. To get the attack to work, in practice, requires closing virtually all extraneous processes.

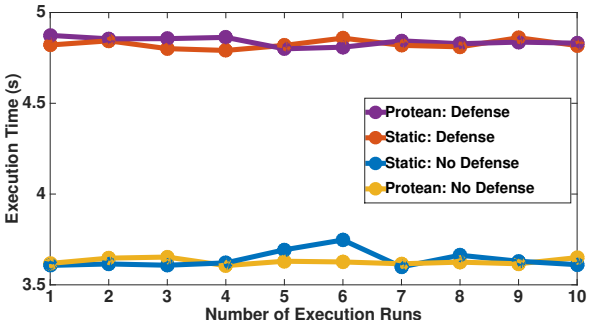

Fig. 5. Execution Times for Ten Trial Runs: As can be seen, the static and Protean versions of the defense/no defense are close to each other, with minor variances. This illustrates that the PGPG system introduces negligible overhead.

	Vulnerable	Safe
Static	3.64 s	4.82 s
Protean	3.63 s	4.84 s

TABLE 1
Average runtimes for the four cases of the GnuPG microbenchmark. The "Vulnerable" application binary is the faster, yet insecure, version, whereas the "Safe" binary is the slower, yet secure, version.

In contrast, when we test the PGPG Detector, we use it on a shared multi-user server running dozens of processes. The amount of extraneous cache evictions, and therefore misses, on our system should be higher than, or at least similar to, the worst environment on which the FLUSH+RELOAD attack can run. In Figure 4, we can clearly detect the attack, even with the system noise.

Intuitively, it stands to reason that a sophisticated adversary could modify its pattern of generating cache misses to avoid the static threshold or even more sophisticated heuristic-based approaches to detecting an attack in progress (discussed in Section 5.3. However, the FLUSH+RELOAD attack is critically dependent on how quickly it can probe memory locations for recent use and often misses instruction retrieval in the best of cases due to context switching or innocuous cache evictions. In practice, if the attack does not run continuously and at a fast rate, it will be completely ineffectual. Thus, we do not believe an adversary has effective means to avoid even our relatively simple detection scheme.

We therefore posit that the PGPG system provides an equivalent defense to the statically-compiled safe version.

4.2 Performance Overhead

As our solution ultimately provides no additional 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.

In the common case, we assume no ongoing attack, and that the binary is free to run at its maximum speed in a fast, insecure manner. In the event an attack occurs, the binary will adapt to its slower, secure runtime and defend against the attack. Intuitively, the overall runtime of our PGPG binary can be modeled by the equation:

$$TE = (1 - AP) * IE + (AP * SE) \begin{cases} TE = \text{total execution time} \\ AP = \text{attack percentage} \\ IE = \text{insecure execution time} \\ SE = \text{secure execution time} \end{cases}$$

That is, the total execution time (TE) is a weighted average of the time spent executing insecure code (IE) while the attack does not occur, along with the time spent executing secure code (SE) while the attack occurs.

As seen in Table 1, our PGPG binary runs in an average of 3.63 seconds when no attack is detected, compared to the statically-compiled vulnerable binary average of 3.64 seconds. As our PGPG binary makes no intentional optimizations to the vulnerable code and in fact introduces (minimal) overhead, we consider these execution times equivalent and attribute the difference to general execution variance. Such variance can be seen in Figure 5.

Similarly, when we force an attack "detection" immediately after starting the PGPG binary such that it spends its entire execution time running the safe encryption routine, we see an average execution time of 4.84 seconds, compared to the statically-compiled safe binary average of 4.82 seconds. This corresponds to only a 0.41% slowdown if the PGPG binary spends all its time running the secure code.

If we conservatively assume that 20% of the time, an attack will occur (in practice, we believe an attack will occur less than 1% of the time), then the overall average runtime of our PGPG binary can be calculated as follows:

$$\begin{aligned} E[TE] &= (0.8 * 3.63s) + (0.2 * 4.84s) \\ &= 3.87s \end{aligned}$$

While under conservative attack loads, the expected total execution time, $E[TE] = 3.87s$, is still 19.7% faster than the secure statically-compiled code with an equivalent security guarantee.

5 Discussion

5.1 Why a Microbenchmark?

Ideally, PGPG would be able to provide a defense for GnuPG. Unfortunately, as a Gnu project, GnuPG uses compiler optimizations inherent to GCC, the Gnu C Compiler. Protean code uses the Clang [7] compiler,

which is unable to handle these optimizations. This is also discussed in the Protean Code paper [3].

Consequently, to demonstrate the validity of our approach, we build a microbenchmark that is representative of GnuPG's main cryptographic algorithm, which we postulate comprises the majority of GnuPG's execution time.

GnuPG's main cryptographic routine can be found in the `mpi_pow` function, in a loop that iterates over the bits of a secret exponent. A particular branch is only taken if the next bit of the secret exponent is a 1. Within this branch, long-running *multiply* and *reduce* functions are called. By monitoring the cache timing of instructions within these functions, the adversary can infer when they are executed. By inferring when the corresponding *square* and *reduce* functions outside the branch are executed, the adversary can differentiate between successive 1s and 0s in the secret exponent.

To do this successfully, the instructions within the *square*, *multiply*, and *reduce* functions must be chosen carefully. Since it takes approximately 100 or 250 clock cycles (for cache hits or misses, respectively, assuming the kernel does not perform a context switch on the core) to infer whether a given instruction is present in the cache, checking a single instruction in each of these functions takes between 300 and 750 clock cycles. It is important that the spy choose instructions contained within a loop that gets executed many times, as this increases the chance that the instruction will still be cached when the spy process gets a chance to check it. GnuPG exhibits loops within the relevant functions, and thus so does our microbenchmark.

Another important consideration for instruction caching is that even if the instruction is not executed, it may end up being loaded into the cache. Unlike data caching, the hardware instruction prefetcher will attempt to prefetch instructions that are likely to be executed. This means that instructions at the beginning of a function are likely to be prefetched whether the function is called or not. Similarly, when instructions are cached, entire cache lines are filled from memory. The last few instructions of one function can be fetched when the following function in memory is prefetched or executed. Consequently, our microbenchmark functions contain a number of *NOP* instructions at the beginning and end of each function, increasing the length of the functions in memory to model those in GnuPG.

5.2 Generic Defending

In the current iteration of PGPG, we directly modify the `compute_val` function. However, for the overall PGPG technique to see wide adoption and be used in other software packages, the modification of a vulnerable function must be automated.

This can be done by precompiling the *safe* source code of the target function into LLVM bytecode before the vulnerable Protean binary is compiled and storing it in a special data section of the Protean binary (as opposed to the Protean runtime shared library). It can later be retrieved when the Defender gets invoked and the Defender can construct a replacement target function from scratch, as opposed to modifying an existing function. This approach can scale to support multiple function replacement in a generic way. However, due to time constraints, we leave this to future work.

5.3 Advanced Detection Techniques

The existing detector could be made smarter by implementing certain modifications that collect and process more information from the victim process at run-time.

Our original proposal—and our former detector implementation—uses the Precise Event-Based Sampling (PEBS) performance counter mechanism [1] to track LLC-miss events and their source code origin. Identification of the most-evicted PCs paves way for multiple discrete defense mechanisms. The Protean Defense class could be developed such that it uses this information for an appropriate dynamic function swapping defense. 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. Our previous implementation uses the `MEM_LOAD_UOPS_RETIRED_L3_MISS` PEBS event.

We used the built-in Linux `perf` [2] kernel support to gather LLC miss events from the hardware. A small memory mapped region was allocated for each thread of the application, and this region was used for asynchronous communication with the kernel's `perf` mechanism. The kernel logs LLC miss events into this region until they are processed by the detector.

Our prior detector then builds a map from the key PCs to the number of LLC miss records received for those PCs, and reports the rate at which LLC miss events occur for these "hot" source code lines.

We modified this implementation to our current detection mechanism of collecting absolute counts alone, because the simpler approach worked reliably to detect an attack, and even with an generic defense approach, we would have no need to identify the

specific cache miss locations unless we had multiple discrete defenses in place simultaneously.

Furthermore, our future work proposals include using machine learning techniques to develop a more robust detector. The detection algorithm can be trained to recognize the miss pattern created by other processes, including a spy process. This pattern can be used as a signature to differentiate between innocent large-working set co-runners and a spy.

The caveat to this approach, however, lies in the fact that the algorithm would succumb to a more intelligent spy that mirrors the eviction pattern of an innocent application. The use of PEBS to identify the highest source of cache misses could be a potential defense for such an attack.

6 Conclusion

In this paper, we introduce the system, with the aim of overcoming the security-performance gap. Our system takes advantage of Protean code's dynamic compilation techniques to swap out fast, yet insecure, code for slower, yet secure, code. The novelty of our system lies in the fact that this swapping process can take place dynamically, leading to an overall speedup in execution time over always running the statically-compiled "Safe" version of the application binary.

We test our program against a microbenchmark that acts as a proxy for the GnuPG cryptographic suite, all while using the FLUSH+RELOAD cache side-channel attack. Based on a threshold value, our PGPG system is not only able to successfully detect the attack, but also mitigates the attack's efficacy. We model execution times as an equation and show that we achieve conservative average gains of 19.7%, with a worst-case overhead of 0.03%.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[7] *clang: a C language family frontend for LLVM*. URL: http://clang.llvm.org/.

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

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