Capturing 0day Exploits with PERFectly Placed Hardware Traps

Cody Pierce, *Endgame* Matthew Spisak, *Endgame*
Kenneth Fitch, *Endgame*

Abstract

Advanced software exploitation is a rapidly changing field of study. In recent years, clever ways to bypass existing exploit defenses have become mainstream. Reactive defensive solutions based on known exploitation techniques have been proven ineffective, and easily circumvented. In this paper, we discuss a new system for early detection and prevention of *unknown* exploits. Our system uses Performance Monitoring Unit hardware to enforce coarse-grained Control Flow Integrity (CFI). By using hardware features that exist in modern processor architectures, and real-time CFI policy enforcement, we hope to prove that our approach is effective, suitable for practical use, while staying resistant to bypass.

1. Introduction

The PMU and CFI have been discussed and studied in academic literature for several years. Outside of academia, vendors are also hard at work to enforce the proper behavior of programs through control flow integrity. In most cases this enforcement is performed by validation logic inserted at compile time. While compile time instrumentation is powerful, it takes significant effort to leverage these features in commercial software.

Our research furthers the state-of-the-art in PMU-based defensive applications by introducing the concept of real-time control flow integrity, with the goal of developing a system that could be adapted to real-world scenarios such as web browsers and document readers. Additionally we demonstrate how the use of the Branch Prediction Unit and, more specifically branch mispredictions, is a strong candidate for implementing Control Flow Integrity policies. We intend to demonstrate that our hardware-assisted CFI (HA-CFI) system has a low performance impact and measurable prevention success against 0day exploits and previously unknown exploitation techniques.

1.1. Prior Art and Operational Constraints

Our work builds on previous research that identified the Performance Monitoring Unit of microprocessors as a good candidate for enforcing Control Flow Integrity. The Performance Monitoring Unit is a specialized unit in most microprocessor architectures that provides useful performance measuring facilities for developers. Most features of the unit are intended to count hardware level events during program execution to aid in program optimization and debugging.

In their paper, Yuan et al [YUAN11] introduced the novel application of these events to exploit detection for software security. Their researched focused on using PMU events along with the Branch Trace Store messages to correlate and detect code-injection and code-reuse attacks without source code. Xia et al explored the idea further in their paper CFIMon [XIA12]. By combining precise event context gathering with the BTS and PEBS to enforce real-time control-flow integrity. In addition to these foundational papers, others have pursued variations on the idea to specifically target exploit techniques such as Return-Oriented-Programming.

Alternatively, just-in-time CFI solutions have been proposed using dynamically instrumented frameworks such as PIN [PIN12] or DynamoRIO [DYN16]. These frameworks dynamically interpret code as it is executed while providing instrumentation functionality to developers. Applying control flow policies with a framework like PIN allows for the flexible and reliable checking of code. However, it often incurs a significant CPU overhead, in the area of 10 to 100x, making it unusable in the enterprise.

Our research into dynamic run-time CFI included parameters we feel would make this approach relevant to enterprise security, while also providing significant detection and prevention assurances. We established the following functional requirements:

1. Implementation must work on 32 and 64bit Operating Systems.
2. CFI policies must be applied without software recompilation or access to source code.
3. The system must not have prior knowledge of the program, or preprocessing of the program in any way.
4. Performance overhead of the solution should be minimal.

To maintain resiliency against bypass we included three additional parameters to scope the work:

1. Additional code will not be added to the running program in the form of "hooks" or validation logic.
2. It must work in the Kernel.
3. The system must be able to detect and prevent an exploit in real-time

2. Approach

HA-CFI uses PMU-based traps in order to apply coarse-grained CFI on indirect calls on the x86 architecture. The system uses the PMU to count and trap mispredicted indirect branches in order to validate branch destinations in real-time. In addition to gaining assistance from a carefully tuned PMU, a practical implementation of this approach requires support from Intel's Last Branch Record (LBR) feature, and a method for tracking thread context switching in a given OS. It also requires an algorithm for validating branch destination addresses, all while keeping performance overhead to a minimum. After more than a year of fine-tuning these hardware features, we have proven our model is capable of generically detecting control-flow hijacks in real-time with acceptable performance overhead on both Windows and Linux.

Because control-flow hijack attacks often stem from a corrupted or modified VTable, many CFI designs focus on validating all indirect branches. For example, once an attacker is able to layout memory appropriately with a fake VTable with custom virtual function pointer(s), a use-after-free or type confusion bug then exercises code that invokes the virtual function in this hijacked VTable providing the initial code execution. These call sites are indirect branches. Because these call sites have never before jumped to the attacker controlled address, this indirect call is almost always mispredicted by the branch prediction unit. Therefore, by only focusing on mispredicted indirect call sites we greatly limit the number of places that a CFI check is necessary.

HA-CFI configures the Intel PMU on each core to count and generate an interrupt on every mispredicted indirect branch. The PMU is capable of delivering an interrupt any time an event counter overflows, and thus HA-CFI sets the initial counter value to -1 and resets the counter to -1 from the interrupt service routine to generate a trap for every occurrence of the event. In this way, the HA-CFI interrupt service routine becomes our CFI component capable of validating each mispredicted call and determining whether it is the result of malicious behavior. To validate target indirect branch addresses, HA-CFI builds a comprehensive whitelist of valid code pointer addresses as each .dll/.so is loaded into protected processes. When a counter overflows, the Interrupt Service Routine (ISR) called is then able to compare the mispredicted branch to a whitelist, and determine if the branch is anomalous.

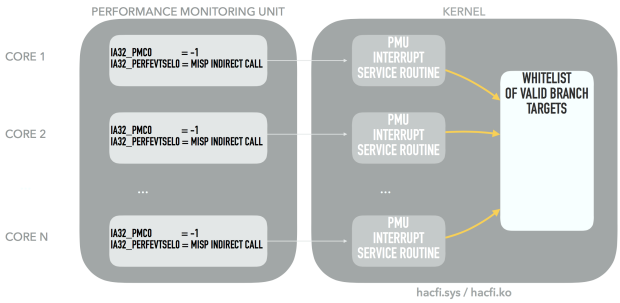

Figure 1: High level design of HA-CFI using the PMU to validate mispredicted branches

3. Design

To ensure we minimized the overhead of HA-CFI while maintaining an extremely low false-positive rate, several key design decisions had to be made, and are described below.

3.1. The Indirect Branch

On the Intel x86 architecture, an indirect branch can occur at both a CALL or JMP instruction. This paper focuses exclusively on the CALL instruction for several reasons. First, indirect JMP branch locations were found most often to be utilized for switch statements, in which the jump table data would rarely if ever be corrupted or utilized in a control-flow hijack attack. In our experimentation on Linux, we found roughly 12% of hijacked indirect branches occurred as part of an indirect JMP, but occurred even less frequently on Windows. In fact, in one ActiveX Flash binary on Windows, 98% of all indirect branches were found to be CALL instructions. Secondly, ignoring mispredicted JMP instructions further reduces the overhead of HA-CFI. Therefore, we opted to omit mispredicted JMP branches during this research, which can be achieved with settings on the PMU and LBR.

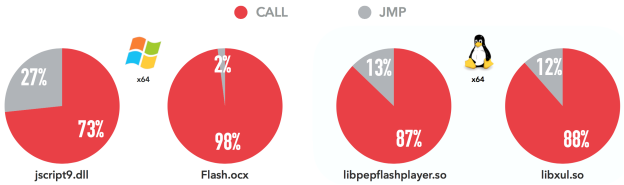

Figure 2: A breakdown of hijackable indirect JMP vs CALL instructions found in Windows and Linux x64 binaries

The Intel PMU supports hundreds of events for counting, many of which provide conditional filters and "unit-masks", offering numerous additional combinations and finer granularity for these events. We evaluated two similar events for counting mispredicted indirect CALL instructions: BR_MISP_RETIRED.NEAR_CALL and BR_MISP_EXEC.TAKEN_INDIRECT_NEAR_CALL. The main difference between the two events is that the BR_MISP_RETIRED event can be utilized as a precise event for PEBS. We verified that some speculative mispredicts counted by the BR_MISP_EXEC event occasionally included branches that were not actually mispredicted. While the BR_MISP_RETIRED.NEAR_CALL technically includes both direct and indirect CALLs, rarely, if ever, is a direct call actually mispredicted. As a result we chose this event as it causes fewer interrupts due to the exclusion of speculative events.

Event Name	Code	Mask	Description
BR_MISP_RETIRED.NEAR_CALL	0xC5	0x02	Direct and indirect mispredicted near call instructions retired.
BR_MISP_EXEC.TAKEN_INDIRECT_NEAR_CALL	0x89	0xA0	Taken speculative and retired mispredicted indirect calls

Intel branch misprediction event codes

3.2. Added Precision with the LBR

CFIMon [XIA12] chose to validate branches in a delayed manner using the Intel Branch Trace Store (BTS), a completely separate hardware feature from the PMU. One of the reasons they focused on Intel BTS is because branch traces are recorded in a precise manner with the tradeoff that analysis of the trace data cannot be performed in real-time, thus only offering a detection solution. HA-CFI offers both detection *and* prevention of control-flow hijacks since it's operating in real-time, validating each mispredicted branch as it occurs.

The interrupt instruction skid is one of the greatest challenges to our approach. This term represents the number of instructions that execute past the instruction that actually triggered the counter overflow in the PMU. Since our ISR is responsible for validating each mispredicted branch destination address, we can't rely on the instruction pointer within the context of the interrupted thread to actually represent the precise branch destination. In fact, [AMD07] states that this skid can be up to 72 instructions. Frequently the skid is a single instruction or just a handful of instructions. Regardless, to precisely resolve the exact branch that caused the PMU to generate an interrupt, we make use of Intel's Last Branch Record (LBR) stack.

The Intel LBR stack size varies across different Intel microarchitectures: 16 entries for Haswell, and 32 entries for Skylake. A powerful feature of the LBR is the ability to filter the types of branches that are recorded. For example, returns, indirect calls, indirect jumps, and conditional branches can all be included or excluded. With this in mind, we can configure the LBR to only record indirect call branches occurring in user mode. With this filter in place, our ISR queries the top of the LBR stack knowing that it was the likely source for the PMC overflow. In this way, the LBR offers a level of precision in determining the exact branch destination address, and is thus a requirement for our system to operate.

Additionally, the most significant bit of the LBR branch FROM address indicates whether the branch was actually mispredicted. As a result, this provides a quick filter for the ISR to ignore the branch if it was predicted correctly. It's important to note that we are not iterating over the entire LBR stack, only the most recently inserted branch.

3.3. On-Demand PMU-Assisted CFI

HA-CFI is focused on protecting commonly exploited applications such as browsers, mail clients, and Flash. As such, the PMU and LBR are both configured to only operate on mispredicted indirect calls occurring in user mode, ignoring those that occur in ring-0. Moreover, by monitoring thread context switches in both Windows and Linux, we can turn the entire PMU on and off depending upon which applications are being protected. For example, when a thread as part of Internet Explorer is schedule in, we enable the PMU to resume trapping mispredicted indirect call instructions. However, as soon as a thread not being monitored, such as a thread in Calc.exe, is scheduled in we can immediately disable the PMU for that core. This design decision is perhaps the most critical element in keeping our performance overhead at an acceptable level.

3.4. Runtime Whitelist Generation

The final component to the HA-CFI system is the actual integrity check that involves querying a whitelist data structure containing valid destination addresses for indirect calls. Whitelist generation is performed at run-time for each image loaded into a protected process.

We first focused on post-processing captured branch data. In this setup, we had our kernel driver and ISR in data collection mode only, buffering each mispredicted indirect call and loaded image base addresses, and sending it to a user-space client. After running numerous benchmarking tools we started with a working data set of over 150 million mispredicted branches. We used this data to iteratively explore and add new static analysis algorithms for the identification of valid indirect call destinations. Eventually, we generated a whitelist such that all branches from our dataset could be verified in a hashtable leaving zero unknown captured branches.

The key to proper whitelist generation for indirect call addresses is to exhaustively find all code pointer addresses present in each loaded image. While the specific implementation is slightly different between PE and ELF binaries, the approach is the same. When loaded into memory, PE/ELF binaries are scanned in search of code pointer addresses, addresses that point into the .text section, using the following checks:

- Exports: All function symbols are enumerated and addresses retrieved
- Relocations: Relocations section is scanned searching for code pointers and recording addresses that point to .text
- Callbacks: The .text section is scanned using simple pattern matching to identify code pointer addresses that we label callback functions

4. Implementation Challenges

Throughout the course of our research, we encountered numerous hurdles to meet our original goal of low-overhead, high detection stats. Some of these challenges such as registering for PMU interrupts on Windows, and properly tracking thread context switches are highlighted in this section.

4.1. Windows PMU and APIC Programming

We first configured the performance management unit (PMU) and the programmable interrupt controller (APIC) to use a supplied function pointer as the interrupt handler for counter overflows. Our initial prototype was developed under Linux. Programming the interrupt controller to handle PMU overflows was relatively straightforward, but porting the same techniques to Windows proved problematic. We were aware that directly modifying the interrupt descriptor table (IDT) would violate Windows' Kernel Patch Protection [MIC16], aka PatchGuard, causing a system halt when the operating system detected our changes. After significant research, we discovered an undocumented option in the Windows Hardware Abstraction Layer (HAL) that registers a driver supplied interrupt handler for PMU interrupts. Calling the `HalpSetSystemInformation` routine with the `HalProfileSourceInterruptHandler` information class allowed us to register a callback for performance interrupts supported by Windows.

```
NTSTATUS status;
PVOID buffer[1];
buffer[0] = profileSourceInterruptHandler;
status = HalpSetSystemInformation(HalProfileSourceInterruptHandler,
                                  sizeof(PVOID),
                                  buffer);
```

Registering a PMU Interrupt Handler on Microsoft Windows

4.2. Thread Tracking

The largest issue with implementing CFI on Windows was restricting PMU monitoring to a single process or thread. The PMU isn't natively aware of thread context and will continuously count and generate interrupts. In our Linux prototype, we leveraged a kernel API for registering callbacks whenever a thread was swapped out on a given processor. To our knowledge, Windows has no such functionality.

The technique we ultimately arrived at makes use of the Asynchronous Procedure Call (APC) mechanism. Windows allows developers to register APC routines for a given thread, which are then added to a queue to be executed at certain points. This can be applied to scenarios requiring a thread to be interrupted when an asynchronous event occurs, such as the fulfillment of an IO request. However, we leveraged the fact that the APC routine list is drained opportunistically when a thread resumes execution after being swapped out. By maintaining an APC registered on all threads that we seek to monitor at all times, we are notified that a thread has resumed execution when our routine executes. The routine re-enables the PMU counter if necessary and updates various tracking metrics. We detect when a processor swaps out a thread and begins executing another when we receive an interrupt in a different thread context. We can then disable the PMU counters, if needed.

5. Results

To evaluate our system, we measured success both in terms of performance overhead added by HA-CFI as well as detection statistics when tested against various exploits in common client applications, including the most common web browsers, as well as Microsoft Office and Adobe Flash. We sourced exploits from Metasploit modules for testing, as well as numerous live samples from popular Exploit Kits found in the wild.

5.1. Performance

After completing our prototype, we were concerned about the overhead of monitoring with HA-CFI and its impact on system performance and usability. Since each mispredicted branch in a monitored process would cause an interrupt, there was the potential for a very high number of interrupts to be generated. We subjected our prototype implementations to several tests to measure overhead. Monitoring Internet Explorer 11 while running a JavaScript performance test suite, the driver detected approximately 83,000 interrupts per second on average. In contrast, monitoring an "idle" IE resulted in roughly 1,000 interrupts per second. Our performance analysis revealed that overhead is highly dependent upon the process being protected. For example, with Firefox we saw around 10% overhead while running Dromaeo [DRO16] JavaScript benchmarks, with PassMark benchmarking tool we saw 8-10% overhead, but with Internet Explorer under heavy usage this number was above 10% We have deployed HA-CFI on systems in daily use monitoring web browsing, and observed little impact on performance or usability.

5.2. Exploit Detection

We extensively tested HA-CFI against a variety of exploits to determine its efficacy against as many bug classes and exploitation techniques as possible, with an emphasis on recent samples using approaches intended to bypass other mitigation measures. We ran one set of tests against more than 15 Metasploit exploits targeting Adobe Flash Player, Internet Explorer, and Microsoft Word. HA-CFI detected and prevented exploitation for each of the tested modules, with an overall detection rate greater than 98%. We attribute the 2% false negative rate to instruction skid, as described above.

We found the Metasploit results to be encouraging, but came to the conclusion that they did not provide sufficient diversity in exploitation techniques needed to comprehensively test HA-CFI. We used the VirusTotal service to download a set of samples used in real-world exploit kit campaigns from several widespread kits [KAF16]. In total, we tested forty-eight samples comprising twenty unique CVE vulnerabilities. We analyzed the samples to verify that they employed a varied set of both Return-Oriented Programming (ROP) and "ROPless" techniques. HA-CFI succeeded in detecting all 48 samples, with an overall detection rate of 96% in a multiple trial consistency test.

CODE EXECUTION TECHNIQUE	# SAMPLES	HA-CFI DETECTION RATE	EMET DETECTION RATE
ROP	37	95%	100%
ROPless Technique A	1	100%	0%
ROPless Technique B	10	100%	0%

Results of VirusTotal Sample Testing, by Exploitation Technique

BUG CLASS	# CVEs	# SAMPLES	HA-CFI DETECTION RATE
Out-Of-Bounds Write	3	6	83%
Buffer Overflow	3	6	83%
Integer Overflow	2	6	100%
Use-After-Free	4	14	100%
Double Free	2	4	100%
Type Confusion	3	6	100%
Race Condition	1	4	100%
Uninitialized Memory	1	1	100%

Results of VirusTotal Sample Testing, by Bug Class

5.3. Case Studies

The first case study is CVE-2015-2419, a double-free in jscript9 and this particular vulnerability is being exploited in most of the popular exploit kits. This specific example depicted below was taken from the Magnitude exploit kit. We chose this sample since the exploit authors used the traditional ROP approach to code-execution, where the initial control-flow hijack jumps directly to stack pivot gadget, and then eventually a gadget that returns into a call to VirtualProtect. Our HA-CFI system was able to reliably detect and prevent this initial control-flow hijack branch jumping to the stack pivot gadget. In comparison, Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [EME16] is only able to detect the stack pivot once the VirtualProtect gadget is invoked. The figure below shows where HA-CFI and EMET detect this particular exploit found in the wild.

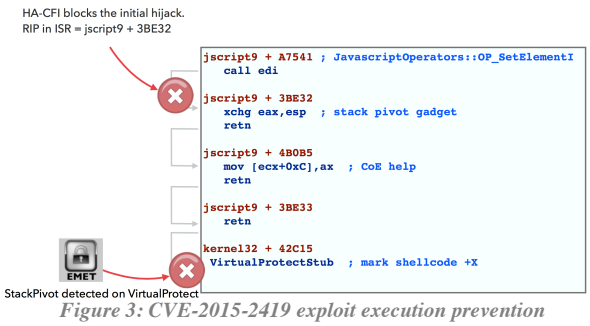

Figure 3: CVE-2015-2419 exploit execution prevention

A second case study to highlight is CVE-2014-0515, a heap overflow in Adobe Flash. This bug is also commonly found in popular exploit kits, and while the bug and sample itself are almost two years old, the significance of this bug is that the exploit authors implemented a technique that is "ROP-less". In other words, a DEP bypass is implemented without using ROP gadgets. Instead, the authors hijack a virtual function call for FileReference.cancel() in ActionScript, but re-use a function within the Flash binary that invokes VirtualProtect marking a region of memory as executable. The exploit then searches for this VirtualProtect wrapper function, and points the virtual function pointer for cancel() to the wrapper function. ActionScript code is then able to mark the region of memory containing shellcode as executable, the process is repeated to hijack the same call to then jump straight to the shellcode address. This control-flow hijack effectively bypasses anti-ROP mitigations such as Microsoft EMET, while HA-CFI is able to capture and prevent the initial hijack branch since the wrapper function is not a normal branch for indirect call sites. The figure below depicts this example found in the SweetOrange exploit kit.

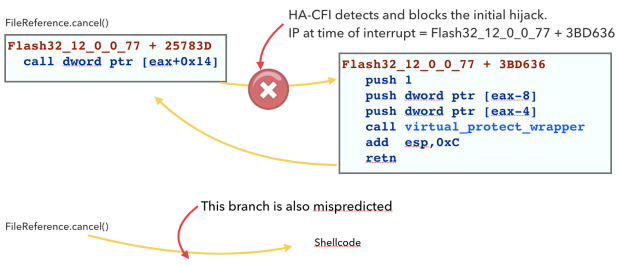

Figure 4: CVE-2015-0515 exploit execution prevention

5.4. Future Work

While the PMU is largely supported in current virtualization products, most hypervisors don't emulate Intel's Last Branch Record (LBR) feature. Since HA-CFI uses the LBR for precise lookups of the offending mispredicted branch, HA-CFI is currently unable to run in virtualized environments. We have performed some initial investigations into modifying Xen [XEN16] to support the required MSR's and determined it would be reasonable to modify the hypervisor to add the needed LBR support and filtering, making our approach potentially viable on Xen guest operating systems.

Additionally, our current whitelist approach encounters challenges identifying legitimate branches into just-in-time (JIT) compiled code. We are continuing our research in this area with hopes of properly identifying legitimate just-in-time code pages.

6. Conclusion

Modern exploitation techniques are rapidly changing. A new approach to exploit detection that can work with complex applications is needed. In this paper we demonstrated a practical and novel approach to exploit detection that uses the Performance Monitoring Unit to enforce control flow integrity on branch mispredictions. Using a run-time generated whitelist we can determine the validity of indirect calls to locations classified as malicious. This approach greatly reduces the overhead of the instrumentation by moving the policy enforcement to a "coarse-grained" verifier on mispredicted indirect branch targets. The data provided also shows the efficacy of such a system on samples captured in-the-wild. These samples, from popular exploit kits, allow us to measure against unknown threats further validating its application. As exploits advance, we also need advanced exploit detection. Using HA-CFI we have advanced the state-of-the-art to give enterprise-scale security software an upper hand in the detection of exploitation.

7. References

[YUAN11] L. Yuan, W. Xing, H. Chen, B. Zang, "Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters", APSys'11, July 11-12, 2011.

[PIN12] PIN: A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[DYN16] DynamoRIO: Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/

[XIA12] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, IEEE Computer Society, 2012.

[LI14] X. Li and M. Crouse, "Transparent ROP Detection using CPU Performance Counters." https: / / www.trailofbits.com / threads / 2014 / transparent rop detection using cpu perfcounters.pdf. Threads 2014.

[AMD07] P. Drongowski, "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors", AMD, Nov 2007.

[DRO16] Dromaeo JavaScript Benchmark. http://www.dromaeo.com

[EME16] The Enhanced Mitigation Experience Toolkit. https://support.microsoft.com/en-us/kb/2458544

[KAF16] Kafeine. Exploit Kit Samples. http://malware.dontneedcoffee.com/

[MIC16] Kernel patch protection for x64-based operating systems.
https://technet.microsoft.com/en-us/library/cc759759(v=ws.10).aspx
[XEN16] Xen Project
http://www.xenproject.org/