# GRIFFIN: Guarding Control Flows Using Intel Processor Trace

Xinyang Ge
Microsoft Research
xing@microsoft.com

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Trent Jaeger
The Pennsylvania State University
tjaeger@cse.psu.edu

## Abstract

Researchers are actively exploring techniques to enforce *control-flow integrity* (CFI), which restricts program execution to a predefined set of targets for each indirect control transfer to prevent code-reuse attacks. While hardware-assisted CFI enforcement may have the potential for advantages in performance and flexibility over software instrumentation, current hardware-assisted defenses are either incomplete (i.e., do not enforce all control transfers) or less efficient in comparison. We find that the recent introduction of hardware features to log complete control-flow traces, such as Intel *Processor Trace* (PT), provides an opportunity to explore how efficient and flexible a hardware-assisted CFI enforcement system may become. While Intel PT was designed to aid in offline debugging and failure diagnosis, we explore its effectiveness for *online* CFI enforcement over unmodified binaries by designing a parallelized method for enforcing various types of CFI policies. We have implemented a prototype called GRIFFIN in the Linux 4.2 kernel that enables complete CFI enforcement over a variety of software, including the Firefox browser and its jitted code. Our experiments show that GRIFFIN can enforce fine-grained CFI policies with shadow stack as recommended by researchers at a performance that is comparable to software-only instrumentation techniques. In addition, we find that alternative logging approaches yield significant performance improvements for trace processing, identifying opportunities for further hardware assistance.

***CCS Concepts*** • **Security and privacy** → *Operating systems security*

***Keywords*** Intel Processor Trace; Control-Flow Integrity

## 1. Introduction

Control-Flow Integrity [6] (CFI) is a runtime security defense that limits a program's indirect control transfers to a specified control-flow graph (CFG). CFI defenses were identified as the foundation of defenses against code-reuse attacks [40], as such attacks fundamentally aim to exploit compromised programs by executing adversary-chosen control flows. While attacks are still possible even when restricting program execution to a restrictive CFG [10], CFI can greatly reduce the attack surface available to adversaries.

To date, researchers have focused on software-only implementations of CFI. Such approaches can be divided into three categories: compile-time instrumentation, static binary instrumentation, and runtime instrumentation. The main advantage of compile-time instrumentation [8, 14, 18, 28, 31, 33, 35, 42, 45, 49] is that it has better performance than other approaches. However, it has two main limitations. First, such techniques can only instrument programs supported by that compiler. Second, they "fix" the CFI defense for resultant software, preventing systems from customizing their CFI defenses at runtime (e.g., to tighten security or balance performance). Static binary instrumentation [6, 44, 50, 52] could instrument programs written in a variety of languages and legacy binaries, but as yet cannot enforce as accurate CFGs as compile-time, cannot apply some compile-time optimizations (e.g., safe stack [30]), and also fixes the instrumentation. Runtime instrumentation [17, 39] has the flexibility of turning the protection on and off, but the overhead is far higher than for fixed instrumentation.

Hardware-assisted CFI enforcement avoids the limitations above by using hardware-generated logs to check indirect control transfers. But current methods either fail to provide a complete CFI defense or incur unacceptable overheads when applied completely. For example, researchers have applied the Last Branch Record (LBR) [5, Sec. 17.4] feature to enforce CFI defenses using its ability to store 8-16 control transfers [11, 36, 43] to improve performance. While researchers have shown that LBR traces may be augmented by heuristics [11, 36], others have shown that adversaries can evade these heuristic defenses [9]. To achieve complete enforcement, researchers have explored using the Branch Trace Store (BTS) [5, Sec. 17.4.5] to record control transfers [47] and the Performance Monitoring Unit (PMU) [5, Chap. 18] to trigger interrupts when the LBR fills [48]. Some have estimated that the BTS incurs a significant performance slow-



*ASPLOS '17* April 08–12, 2017, Xi'an, China



ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037716

down (20x-40x) [43], and the PMU will trigger interrupts after every 16 indirect control transfers.

In this paper, we explore the effectiveness of a CFI enforcement based on a new, commercially available hardware feature called *Processor Trace* (PT) on Intel CPUs [5, Chap. 36]. Intel PT was designed to aid in debugging and failure diagnosis by recording the minimal information necessary to reconstruct complete control-flow traces. For instance, researchers have previously developed offline techniques that are able to diagnose complex failures using Intel PT [29]. Given Intel PT's ability to record control flow traces, we investigate using Intel PT for *online* enforcement of CFI policies. We design and implement GRIFFIN, an operating system mechanism that leverages the Intel PT feature to enforce CFI policies over unmodified binaries. When a binary is run on GRIFFIN, GRIFFIN restricts the binary's execution by comparing each indirect control transfer in the binary's execution trace captured using Intel PT to a CFI policy for the binary. GRIFFIN may enforce CFI policies produced by executing the binary program (e.g., shadow stack) or leverage CFI policies produced elsewhere. In addition, GRIFFIN may change the CFI policies being enforced dynamically to manage performance and/or customize enforcement.

Our goal is to build a high performance CFI enforcement mechanism that is capable of enforcing a variety of CFI policies over unmodified binaries. Implementing online CFI enforcement using Intel PT to meet this goal presents several challenges. Intel PT logs the information necessary to reconstruct a complete control-flow trace, but leaves the reconstruction of the trace to post-processing. To enforce some CFI policies, we need to know the call sites executed at runtime, but Intel PT does not record call sites, as they can be inferred from other information. Thus, we must disassemble the binary and interpret the logged trace buffers to recover the program's control-flow trace necessary for CFI enforcement. We explore design choices that optimize the efficiency of trace processing and CFI enforcement. First, we design a data structure for fast lookup of basic block information for control-flow reconstruction that works efficiently for current programs of all sizes. Second, GRIFFIN leverages idle cores to run disassembly, trace processing, and CFI enforcement in parallel. The GRIFFIN design carefully prevents the need for synchronization delays among these parallel activities. While sequential processing may be necessary for enforcing some CFI policies, we retain the parallel processing of trace buffers and only perform CFI checks that require trace information from multiple buffers in a sequential manner.

We have implemented a prototype of GRIFFIN in the Linux 4.2 kernel that controls the execution of unmodified, user-space binaries and dynamically generated (i.e., jitted) code. We show that GRIFFIN is capable of enforcing three types of CFI policies, coarse-grained CFI, fine-grained CFI, and stateful CFI (see Section 4), on both forward edges (i.e., calls and jumps) and backward edges (i.e., returns). This allows GRIFFIN to enforce restrictive CFI policies recommended by researchers [6, 10] and to balance security with performance. GRIFFIN marks a significant improvement in hardware-assisted CFI enforcement mechanisms by providing complete and flexible CFI enforcement that performs comparably to software-only instrumentation methods. When enforcing a fine-grained CFG on forward edges and a shadow stack [6] on backward edges, GRIFFIN incurs an overhead of 11.9% on the SPECint benchmarks and 6.2% on SPECfp benchmarks, as compared to 11.6% and 6.0% overhead, respectively, for software-only shadow stack enforcement alone [16]. Since we utilize memory and idle core resources to achieve this performance, we evaluate the impact of such resource usage. Our experiments show that GRIFFIN's memory usage depends mainly on the program's size and processing backlog and its performance degrades gracefully with fewer cores.

Despite a highly-optimized GRIFFIN implementation that achieves performance comparable to software-only instrumentation techniques, such performance is generally considered insufficient for widespread adoption. We propose two modifications to Intel PT logging that have the potential to improve performance and reduce memory usage for CFI enforcement on forward edges. First, by logging the indirect call sites as well as indirect call targets, GRIFFIN can enforce fine-grained CFI policies without performing control-flow reconstruction, resulting in a 90% improvement in trace processing on average for SPEC benchmarks. We no longer need to log the conditional branches for this approach, resulting in a 60% reduction in trace size on average (although some traces may increase in size slightly). However, this first approach undermines our ability to enforce stateful CFI policies because we lack the information to reconstruct control flows to determine states. We propose a second approach that applies selective logging of conditional branches to collect the control-flow information necessary to evaluate stateful policies. For one stateful CFI policy [43], we find that the trace size for the worst case program increases from 0.9% of its original trace size after the first modification to 1.4% after enabling selective logging.

In particular, we make the following contributions:

- Repurpose commercially available Intel PT hardware for an online CFI defense to protect unmodified, user-space binaries, including jitted code.
- An operating system CFI mechanism for enforcing coarse-grained policies, fine-grained policies, and stateful policies, enabling enforcement of recommended CFI policies and balancing security and performance.
- A GRIFFIN implementation on Linux, finding that performance is comparable to software instrumentation for SPEC CPU2006 benchmarks.
- A study of two alternative logging approaches that improve trace processing performance for fine-grained and stateful CFI enforcement.

| Packet | Size | Usage |
|---|---|---|
| PGE | ≤8 | Packet Generation Enable packets provide the IP at which the tracing begins |
| PGD | ≤8 | Packet Generation Disable packets mark the end of tracing |
| TNT | 1 | Taken/Not-Taken packets indicate the direction of conditional branches |
| TIP | ≤8 | Target IP packets provide the target for some control-flow transfers |
| FUP | ≤8 | Flow Update packets provide the source address for asynchronous events |
| PSB | 16 | Packet Stream Boundary packets are unique patterns in the output log to serve as sync points for software decoders |

Table 1: Trace packets and their usage in Intel PT. For brevity, we omit the packets that are not related to this work.

## 2. Intel Processor Trace

In this section, we provide the background on Intel Processor Trace (PT) [5, Chap. 36]. Intel PT is a recent hardware feature that enables the recording of complete control flows with low overhead. When properly configured, Intel PT generates *trace packets* that encode control-flow information such as branch targets and branch taken indications. A software decoder can reconstruct the exact control flow when combining the recorded packets with program binaries.

To understand how Intel PT enables control-flow tracing, we show the main types of its generated data packets in Table 1. Intel PT records the beginning and the end of tracing through `PGE` and `PGD` packets, respectively. Throughout the program execution, Intel PT generates `TNT` packets to log whether conditional branches are taken (e.g., `jcc`) and `TIP` packets to log the targets of indirect branches (e.g., `call*` and `ret`). With such information and the disassembled binary, a software decoder can reconstruct the complete control flow. It is worth mentioning that direct branches (e.g., `call`) do not trigger any packets because of their deterministic effects on control flows.

Intel PT employs various techniques to minimize the size of generated packets. For instance, `TNT` uses one bit to indicate the direction of a conditional branch, and a one-byte `TNT` packet can log up to six executed branches. Additionally, Intel PT compresses a return into a bit in the `TNT` if the return target can be determined from a previous matching call. To reduce the size of `TIP` packets, Intel PT compresses the target address if the upper address bytes match the previous address logged, suppressing up to six bytes. All these features require a stateful processing of the logged trace packets, which presents a challenge for parallel processing.

Intel PT outputs packets directly to physical memory to avoid the cost of address translation. For flexibility, it can be configured to use multiple buffers that are not contiguous in the physical address space through a table-like data structure. An interrupt can be triggered when a buffer becomes full, but the interrupt is not precise. Writes to the next buffer may have occurred when the interrupt is signaled. So one must ensure that all writes are accounted for.

Intel PT supports both user-level and kernel-level tracing of selected processes and/or threads. In GRIFFIN, we employ user-level tracing of threads of selected processes.

## 3. Threat Model

When designing GRIFFIN, we focus on defending against user-space code-reuse attacks based on the following threat model. We run GRIFFIN in the kernel. We assume adversaries have no control over the operating system kernel, which prevents such adversaries from directly tampering with GRIFFIN. We assume that each protected program is benign but may contain memory safety errors (e.g., buffer overflow and/or use-after-free bugs) that would enable adversaries to write to arbitrary memory locations within the running program's address space. In particular, adversaries can corrupt control data (e.g., return addresses on the stack or function pointers on the heap) to subvert the program's expected control flow to launch code-reuse attacks. We assume the protected programs apply W⊕X defense [7, 37]. That is, programs are prevented from modifying their own code or mapping a code page as both writable and executable under legitimate execution.

## 4. Design Overview

In this section, we provide an overview of GRIFFIN's design. GRIFFIN is a hardware-assisted CFI enforcement system for defending against user-space code-reuse attacks. GRIFFIN leverages Intel PT to record the complete user-level execution of a monitored program and performs online control-flow checks based on the recorded execution trace. GRIFFIN checks indirect control transfers both for *forward edges* (i.e., indirect calls and jumps) and *backward edges* (i.e., returns). The GRIFFIN system design focuses on enforcement, assuming that CFI policies are given.

We design GRIFFIN to support multiple types of CFI policies to enable flexible tradeoffs between security and performance. The simplest but least secure CFI policies GRIFFIN supports are the *coarse-grained* policies. Under this class of policies, GRIFFIN checks only if the destination of an indirect control transfer is legitimate (i.e., is a legal target of any call or return). GRIFFIN also supports *fine-grained* policies which are more secure than the coarse-grained policies because GRIFFIN checks whether the destination is a legal target for the source for each source-destination pair. To achieve the best security offered by CFI, GRIFFIN supports *stateful* policies. Under such policies, GRIFFIN additionally uses execution state to restrict forward and/or backward edges. For example, we show how GRIFFIN enforces a stateful *shadow stack* [6] policy on backward edges, which restricts return targets to their corresponding call sites. In Section 7, we evaluate the performance of a type of policy we

call the *combination policy*. It enforces a fine-grained policy on forward edges and a shadow stack on backward edges, as recommended by researchers [6, 10]. Stateful forward-edge policies are a relatively new area of research. We design GRIFFIN to enforce a proposal to restrict indirect call sites to different sets of legal targets depending on the program's runtime control flow [43].

The goal of the GRIFFIN design is to optimize the performance of CFI checking when leveraging Intel PT traces. To do so, GRIFFIN performs both non-blocking and blocking control-flow checks to achieve better performance without sacrificing security. Non-blocking checks are triggered when an Intel PT trace buffer becomes full. During such checks, the program continues to execute. Blocking checks are done when the monitored program makes a security-sensitive system call. Since such a call may impact the integrity of the system, GRIFFIN intercepts those system calls and performs checks of all the control transfers that have not yet been checked before passing the calls to the kernel.

Today's computers rarely run at 100% CPU utilization on all cores. GRIFFIN leverages idle cores on a multi-core system for security checks by having multiple worker threads perform control-flow checks simultaneously. To enable worker threads process Intel PT trace buffers simultaneously, we force a Packet Stream Boundary (`PSB`) packet at the beginning of each Intel PT trace buffer.

# 5. System Design

In this section, we present GRIFFIN's design in detail. The design is motivated by the types of policies GRIFFIN can enforce. We begin with the simplest policies, the coarse-grained policies, then describe the extensions necessary to support fine-grained and stateful policies. We focus on the processing of a single user thread without jitted code. We will discuss the implementation details for multiple threads and jitted code in Section 6.

## 5.1 Coarse-Grained Policies

To enforce coarse-grained policies, we do not need to reconstruct the control flow because the destinations of indirect control transfers are given in `TIP` packets. To quickly check if an indirect control transfer is legitimate, GRIFFIN maps a page at a constant offset from each code page to store whether a code location is legitimate. We refer to it as a *coarse-grained policy page*. The distance between a code page and a policy page is a constant for fast lookup. We explain how this constant is chosen in Section 6.1. If a code location is legitimate for indirect control transfers, then the corresponding location on the policy page is set to 1; otherwise, it is 0. To support dynamic libraries, GRIFFIN monitors library loading in each monitored process. When a library is loaded, GRIFFIN sets up a coarse-grained policy page for each code page in the library. It is worth noting that we do not need to disable return compression since a compressed return is guaranteed to be legitimate as it matches with its corresponding call. The returns that are not compressed are recorded with `TIP` packets.

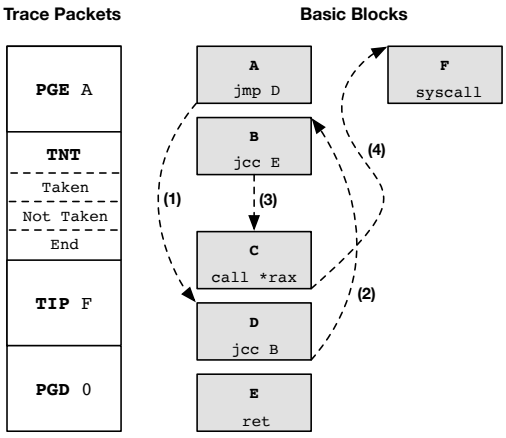


Figure 1: Following control flows using trace packets.

## 5.2 Fine-Grained Policies

Unlike the support for coarse-grained policies, we encode fine-grained policies as a bitwise matrix (referred to as the *policy matrix*). Each row of this matrix corresponds to a possible source for indirect control transfers, and each column corresponds to a possible destination for indirect control transfers. An entry in the matrix is set to 1 if the source-destination pair is legitimate; otherwise, it is set to 0. The matrix is stored at a constant virtual address for a monitored process. We grow this matrix dynamically by adding rows and columns when a library is loaded.

To enforce fine-grained CFI policies, we need to know the source and destination of each indirect control transfer. However, the source address is not directly available in Intel PT's trace. To recover it, we need to reconstruct the control flow so that we know the source address when an indirect control transfer occurs. Next, we first use an example to explain how to reconstruct the control flow based on Intel PT's trace packets and program binaries. Then, we describe how we make the procedure efficient.

To reconstruct the control flow, we take as input Intel PT's trace packets and program binaries. The basic idea is to disassemble the binary and follow the execution by tracking the trace packets. We illustrate this process with an example shown in Figure 1. The initial `PGE` packet identifies that the execution begins at block `A`. Then, the direct `jmp` instruction at the end of block `A` leads the control to block `D`. Note that direct branches do not generate any trace packet due to their deterministic effects. The first "Taken" bit in the `TNT` packet indicates that the next conditional branch (i.e., the one at the end of block `D`) is taken, thus the control is transferred to block `B`. The conditional branch at the end of block `B` is not taken according to the next entry ("Not Taken") in the same

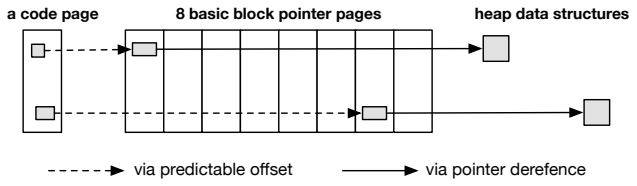


Figure 2: Relationship among basic blocks, pointers on the basic block pointer pages, and their heap data structures.

TNT packet, and the control falls through to the next block C. It ends with an indirect call to block F as logged by the TIP packet. When a system call is invoked at the end of block F, the trace is marked as disabled by a PGD packet since we only trace user-level execution.

The process for reconstructing the control flow is straightforward, but the design challenge is how to make it efficient in GRIFFIN's online processing. Instructions, blocks, or functions tend to execute many times during a program's execution. Disassembling the same instructions over and over again is *not* efficient since we only need the information about the basic blocks for control-flow reconstruction. The key design question is how to store and look up such information in an efficient way.

Previous researchers have proposed several approaches to enable fast store and lookups. MCFI uses an array to store information for each code address [33]. However, in practice, program code can be scattered in the address space (e.g., dynamic libraries), and MCFI leverages sandboxing techniques to restrict the program to use the first 4GB memory region. This requires modifying the system loader and potentially undermines other defenses such as address space layout randomization (ASLR). fastBT uses a simple hash table to map an original basic block to the translated block and reports a low conflict rate for SPEC benchmarks [38]. However, hash conflicts are inevitable when the program binaries are large (over 1MB). Furthermore, the hash table schema requires heavyweight locking for certain operations (e.g., code unloading), which slows both store and lookup operations that may happen in parallel.

To tackle these problems, we trade memory efficiency for lookup performance. Specifically, for each basic block, we allocate a heap data structure to store its information. Then, we allocate eight pages at predictable offsets from each code page to store pointers to their heap data structures. We refer to such pages as the *basic block pointer pages*. We use eight basic block pointer pages because every byte in the code page can lead a basic block and the pointer to its heap data structures takes eight bytes. We show the conceptual layout in Figure 2 and explain how we pick the offset in Section 6.1.

When a worker thread looks up the information of a basic block, it reads the pointer from the basic block pointer page. If the pointer is NULL, then the basic block has not been disassembled yet. In this case, the worker thread disassembles the block, allocates a heap data structure to store the disassembled information, and updates the basic block pointer page with the new pointer. Since multiple worker threads may perform read and write to the same pointer on a basic block pointer page, we use the `compare-and-swap` primitive to make sure the pointer write is atomic and occurs only when the present pointer is NULL.

In a basic block's heap data structure, we also store a row and column index if it may be the source and/or destination of an indirect control transfer. This enables GRIFFIN to quickly locate an entry in the policy matrix.

### 5.3 Stateful Policies

A stateful policy constrains the targets of indirect control transfers based on the program execution state (e.g., a call stack). Consequently, to enforce the stateful policy, we need to *sequentially* process the control-flow trace. This is seemingly in conflict with the goal of parallel processing. A key observation is that most computation in GRIFFIN is on parsing Intel PT trace buffers and reconstructing the control flow. The amount of time spent on checking CFI policies is actually small. This observation motivates us to split our trace processing into two phases: the *parallel* phase and the *sequential* phase. In the parallel phase, multiple trace buffers of a single user thread can be processed by multiple worker threads simultaneously. For shadow stack enforcement, the output of each worker thread is a list of calls and returns encountered in a trace buffer. In the sequential phase, the list of calls and returns is processed in the order of execution. This two-phase design provides the desired in-order processing while keeping GRIFFIN highly parallelized.

To enforce the shadow stack check on returns (i.e., backward edges), we simply check if a return matches the call on the top of the call stack. Exceptional cases in the shadow stack enforcement are described in Section 6.5. To enforce stateful checks on indirect calls (i.e., forward edges), we adapt the design for fine-grained policies with two main changes. First, we extend the policy matrix to store a stateful policy. Specifically, we allocate additional rows in the matrix for sources that have multiple states to store acceptable destinations. Each new row is associated with a source and one of its states. Second, we store the row indices in heap data structure of the source for fast lookup.

## 6. Implementation

We implemented a prototype of GRIFFIN on Debian 8 with a 64-bit 4.2 Linux kernel running on an Intel i7-6700K quad-core processor (a 6th-generation Skylake processor). To implement online disassembling, we ported an open-source disassembling library for x86 called distorm [15] (12,497 SLoC). We made a few changes in distorm, such as adding support for Intel TSX instruction set. We also merged our code for TSX support into the mainstream version 3.3 of distorm. When disassembling a binary, we leverage MODE

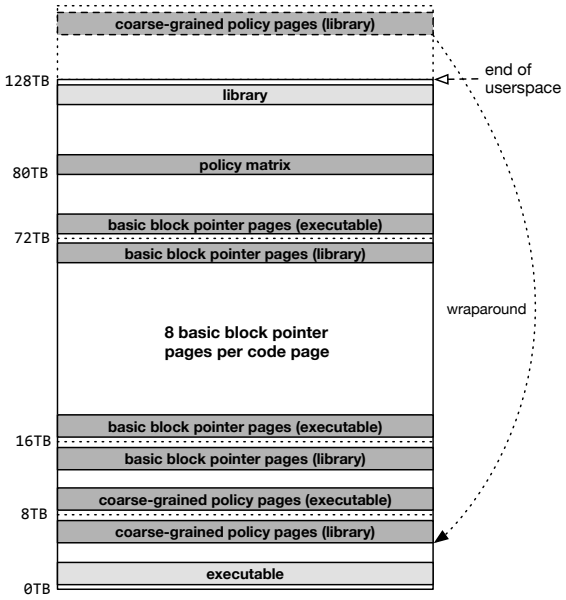


Figure 3: The user-level address space layout in GRIFFIN.

packets in an Intel PT trace to decide if it is x86 or x86-64. This allows GRIFFIN to support 32-bit programs running on a 64-bit kernel. Excluding the distorm code, our prototype consists of 1,625 SLoC in C.

In our prototype, we leverage prior work to identify the set of security-sensitive system calls [11, 43]: `mmap`, `mremap`, `remap_file_pages`, `mprotect`, `execve`, `execveat`, `sendmsg`, `sendmmsg`, `sendto`, and `write`. This list is motivated by the observation that many exploits disable DEP to run injected code after hijacking the control. The last four calls are used to confine network daemons from sending information after being compromised. The list is configurable and can be extended to include other system calls. Below, we describe the implementation choices we made to enable GRIFFIN to support multiple, multi-threaded processes, including jitted code, for the binaries we ran.

### 6.1 Memory Management

GRIFFIN runs inside the Linux kernel. When it parses the trace buffers and performs control-flow checks for a monitored process, it runs in that process's context. We use both kernel and user memory pages in GRIFFIN. We use kernel pages for two purposes. First, the buffers given to Intel PT for tracing are kernel pages. Second, the heap data structures for storing basic block information are also allocated from kernel memory.

We store the coarse-grained policy pages, the basic block pointer pages, and the policy matrix in the user-level address space. The motivation here is that these pages are so frequently accessed that a fast lookup mechanism is the key to GRIFFIN's runtime efficiency. Therefore, we map these pages either at some constant offset from the corresponding code page (i.e., the coarse-grained policy pages and the basic block pointer pages), or at a fixed location (i.e., policy matrix). The user-level address space is private to each process, enabling GRIFFIN to use the same offset and the fixed location for different processes.

To store these pages in the user-level address space, we use the memory layout as shown in Figure 3. Note that the Linux kernel currently locates the main executable of a process at the bottom of the user-level address space, and locates other libraries at the top. The goal is then to find a way to map these pages so that no two pages would overlap each other. Conceptually we divide the 128TB user-level address space into 16 identical ranges of 8TB. We choose the constant offset to be 8TB. So, for each code page, we map its coarse-grained policy page and 8 basic block pointer pages in the subsequent 9 ranges. For libraries, we let the address of these pages wrap around the user-level address space as shown in Figure 3. Thus, for a basic block at `addr`, we store its basic block pointer at `((addr&~0x7)+((addr&0x7)+2)*8TB)&(128TB-1)`. One concern is that the coarse-grained policy page and the basic block pointer pages of the executable could overlap those of a library. Fortunately, the executable and the libraries are mapped at different portions of a range, hence those pages will not overlap. It is worth noting that the coarse-grained policy pages and the basic block pointer pages are position-independent. This allows us to share them between monitored processes. We only need one physical copy for each binary, and they are lazily populated when a code page is accessed.

We store the policy matrix at 80TB. We map a $2^{16} \times 2^{16}$ bitwise matrix at this location. The matrix uses up to 512MB of the process's virtual address space but is lazily populated when dynamic libraries are loaded.

To prevent the monitored program from modifying these user-level memory pages, we map them as read-only user pages in our current prototype. To enable our kernel worker threads to write to these pages without triggering page faults, we set the `AC` bit in the `EFLAGS` register to override the SMAP protection, and clear the `WP` bit in the `CR0` register. This enables memory writes to read-only pages from the kernel. We acknowledge that a more principled approach is to map these pages as *supervisor* pages in the user address space. This requires modification to the memory manager and the page tables. We leave it to future work.

### 6.2 Context Switch

In GRIFFIN, we trace each thread separately. This requires changing the Intel PT configuration state during context switches. We follow the suggestion made in the Intel manual [5, Sec. 36.3.5.2] to use `XSAVES` and `XRSTORS` instructions to save and restore the configuration state at context switch. We enabled these two instructions in Linux 4.2 kernel by adding 30 SLoC to its context switch code.

### 6.3 Fork

The `clone` system call is a UNIX primitive to create a new task (i.e., thread or process). We hook the `clone` system call to notify GRIFFIN on task creations. GRIFFIN allocates a trace buffer for each new task and initializes its Intel PT configuration state accordingly. We directly write to the new task's `XSAVES` area so that the initialized state will be loaded into the processor's registers when its context is switched in.

To support stateful checking for a new process, GRIFFIN follows the semantics of `fork` and makes the child process inherit the call stack state from its parent process. Specifically, GRIFFIN flushes the current trace buffer of the parent process and informs the sequential phase to duplicate the call stack state after processing the buffer. The sequential phase of the child process is blocked until the duplication is done. However, the execution of the child process and its parallel-phase trace processing are not.

### 6.4 Just-In-Time Compilation

Managed languages such as JavaScript often leverage a Just-In-Time (JIT) compiler to transform the byte code into native code at runtime for faster execution. Protecting programs that have JIT engines such as a browser is non-trivial for GRIFFIN. The key challenge is that jitted code can change over time. This may cause GRIFFIN to use obsolete basic block information when reconstructing the control flow of the changed code, which leads to undefined behaviors. A simple flush of the basic block information upon changes to jitted code does not work in practice because pending trace buffers may rely on the old basic block information. Keeping a history of code changes and precisely matching trace packets with the right code version can be both difficult and expensive.

To tackle this challenge, we propose to refactor the JIT engine so that it never modifies existing code in place. As a proof-of-concept, we refactored Firefox's baseline JIT engine to eliminate in-place code updates that alter the original control flows. Specifically, we made the following changes.

**Code Retirement.** When the jitted code is no longer needed, Firefox will poison the code by converting it to no-ops, and unmap the pages via the `munmap` system call. We modified the JIT engine to skip poisoning. When Firefox unmaps an executable page, we mark the page and defer the actual page reclaim until there is no pending trace buffer that was generated before the `munmap` system call.

**Incremental Garbage Collection.** Firefox uses an incremental garbage collector that divides the mark-and-sweep process into time slices for better responsiveness. The incremental garbage collector has to account for new object allocations between slices of the mark-and-sweep process. Thus, the JIT engine dynamically inserts jump instructions in object allocators to execute a piece of code that marks newly allocated objects during the mark-and-sweep process. We made these jump instructions persistent in object allocators and changed the piece of code to mark newly allocated objects only if the mark-and-sweep process is under way.

### 6.5 Shadow Stack

When implementing the shadow stack, we have to handle a few corner cases. We omit the discussion of setjmp/longjmp, C++ exceptions and UNIX signals since we handle them in a way similar to previous work [16, 17].

**Intel TSX.** Intel Transactional Synchronization Extension (TSX) is a hardware mechanism that exposes and exploits hidden concurrency in multi-threaded applications. Intel PT logs TSX events when a transaction begins, commits or aborts. To support TSX, we store these events together with calls and returns. If a transaction aborts, the calls and returns in the aborted transaction are skipped.

**On-Stack Replacement.** On-Stack Replacement (OSR) is a runtime technique commonly used by JIT engines to switch between different implementations of the same function [26]. It works by trapping the execution and replacing the stack frame with a new one as if the process was running in a different function. We modified Firefox to inform GRIFFIN of the OSR entries, and forgive an unmatched return if it targets a valid OSR entry.

**Switch Table.** Due to the dynamic nature of object comparisons for a `switch` statement in JavaScript, Firefox implements the object comparion code in functions different from the one that contains the `switch` statement. For better performance, the object comparison code modifies its return address to directly return to the code of the matching case in the `switch` statement. This leads to a mismatching call and return pair. We modified Firefox's JIT compiler to replace the return instruction in the object comparison code with a semantically equivalent indirect jump instruction.

### 6.6 Stateful Forward-Edge Policy

GRIFFIN is capable of restricting indirect calls based on the control-flow paths led to them. We implemented the stateful policy approach proposed in PathArmor [43] that restricts invocation of *constant callbacks* passed from callers. This effectively limits those indirect calls to a single target based on the current call stack. We develop an LLVM pass and apply the analysis to the program source to identify those indirect calls as well as the passed constant callbacks from different callers. Since GRIFFIN already maintains a shadow stack, it can easily identify the caller and enforce stateful forward-edge checks accordingly.

## 7. Evaluation

In this section, we evaluate the effectiveness and performance of GRIFFIN. We run the RIPE benchmark [46], a collection of exploits, as a sanity check of the effectiveness of GRIFFIN for CFI enforcement. For the performance evaluation, we test GRIFFIN when enforcing coarse-grained and combination policies on the SPEC CPU2006 benchmarks

used in most CFI research [16, 33, 35, 39, 42, 43, 51]. We also examine how GRIFFIN performs on real-world applications including a browser (Firefox 45), a web server (nginx 1.6.2), an FTP server (vsftpd 3.0.2), and an email server (exim 4.84). When evaluating coarse-grained policies, we disassembled the binaries based on their debug information to uncover all legitimate targets for indirect control transfers. We compute fine-grained policies by matching function signatures as previous works do [33, 42, 44]. Additionally, we evaluate a stateful forward-edge policy on SPEC benchmarks [43]. Finally, we explore how more targeted logging can impact the efficiency of control-flow checking.

### 7.1 Effectiveness Evaluation

We run the RIPE benchmark [46] as a sanity check of GRIFFIN's implementation as a working CFI enforcement mechanism rather than a comprehensive security evaluation. The RIPE benchmark consists of a vulnerable program and a set of 850 exploits using various techniques, which can be categorized by the type of code pointers (e.g., function pointer or return address), the location of memory corruption (e.g., stack or heap). The RIPE benchmark was originally developed on Ubuntu 6.06. In our experiment, many exploits failed because of built-in system protection mechanisms, such as DEP and ASLR, changes in the runtime layout, as well as compatibility issues due to the usage of newer-version libraries.

To make more exploits succeed on the vanilla Debian 8.2, we disabled the ASLR and compiled the vulnerable program without stack protection. We ended up with 82 working exploits. GRIFFIN can deterministically detect and prevent all 82 attacks under both coarse-grained and combination policies. This experiment shows that our prototype of GRIFFIN works as expected. It does not mean that GRIFFIN can stop all control-flow attacks. The security of GRIFFIN is determined by the control-flow policies deployed. We show that GRIFFIN is capable of enforcing many known CFI policies.

### 7.2 Performance Evaluation

In this section, we evaluate GRIFFIN's runtime performance and memory overhead by running SPEC CPU2006 benchmarks and a set of real-world applications. We allocate a 64KB buffer for Intel PT to trace each user thread, and an additional 4KB buffer to prevent trace packet loss from the interrupt skid issue (see Section 2). We discuss the performance impact of different buffer sizes using SPEC CPU2006 benchmarks. We use six worker threads to parallelize trace buffer processing. We evaluate the impact of applying different numbers of worker threads on the nginx web server.

#### 7.2.1 SPEC CPU2006

We run SPEC CPU2006 benchmarks compiled with GCC 4.9 under -O2 optimization level to evaluate GRIFFIN's runtime performance on CPU-intensive benchmarks. We run all SPEC CPU 2006 benchmarks except for 447.dealII and 481.wrf because they cannot be compiled on current systems [39]. We used the "train" workload in all benchmarks. The experimental results on SPEC CPU2006 benchmarks are shown in Figure 4. For each benchmark, we make four measurements: (1) the Intel PT hardware overhead measured by taking interrupts without processing trace buffers; (2) total overhead for enforcing coarse-grained policies; (3) control-flow reconstruction overhead for fine-grained and stateful policies; and (4) total overhead for enforcing combination policies. Cases (2-4) all include hardware overhead.

On average, Intel PT tracing introduces a 4.7% slowdown. The average slowdown under the coarse-grained policy is 5.6%. Control-flow reconstruction incurs a 8.3% slowdown on average. The average slowdown under the combination policy is 9.5%. The average overhead is sensitive to outliers like perlbench. For medians, the slowdown becomes 2.4% under the coarse-grained policy, and 5.6% under the combination policy. We also checked the performance impact of different Intel PT trace buffer sizes. We found that using 64KB buffers is slightly better than using either 4KB or 1MB buffers.

Finally, we examined the performance impact of stateful forward-edge policies. We did not observe noticeable performance differences because both the number of stateful indirect calls and the number of associated states are limited, as acknowledged by [43].

We compared GRIFFIN's performance with prior solutions in Table 2. The prior solutions are from three categories: compiler-based, binary-instrumentation-based, or hardware-based. We chose these solutions because they are representative in each category and they measured SPEC CPU2006 benchmarks on Linux. To have direct comparisons, we further divided SPEC CPU2006 benchmark into SPECint and SPECfp. In Table 2, four prior solutions, MCFI [33], $\pi$CFI [35], ROPecker [11] and PathArmor [43], have much better performance than GRIFFIN. However, MCFI and $\pi$CFI do not support the shadow stack check, and ROPecker and PathArmor are not complete (i.e., they do *not* check all indirect control transfers). binCFI [51] only supports a coarse-grained policy but its performance is worse than GRIFFIN mainly because it uses static binary instrumentation. Lockdown [39] supports a policy similar to the combination policy, but its performance is much worse than GRIFFIN because of its dynamic binary instrumentation. Finally, the shadow stack work [16] evaluated two compiler-based shadow stack schemes: traditional shadow stack and parallel shadow stack (i.e., maintain one stack pointer for both the normal stack and the shadow stack). The performance of the parallel shadow stack scheme is better than the traditional shadow stack scheme, but it is vulnerable to a recent attack [13]. GRIFFIN implements both the traditional shadow stack scheme and the stateless forward-edge check in the combination policy. GRIFFIN's performance under this

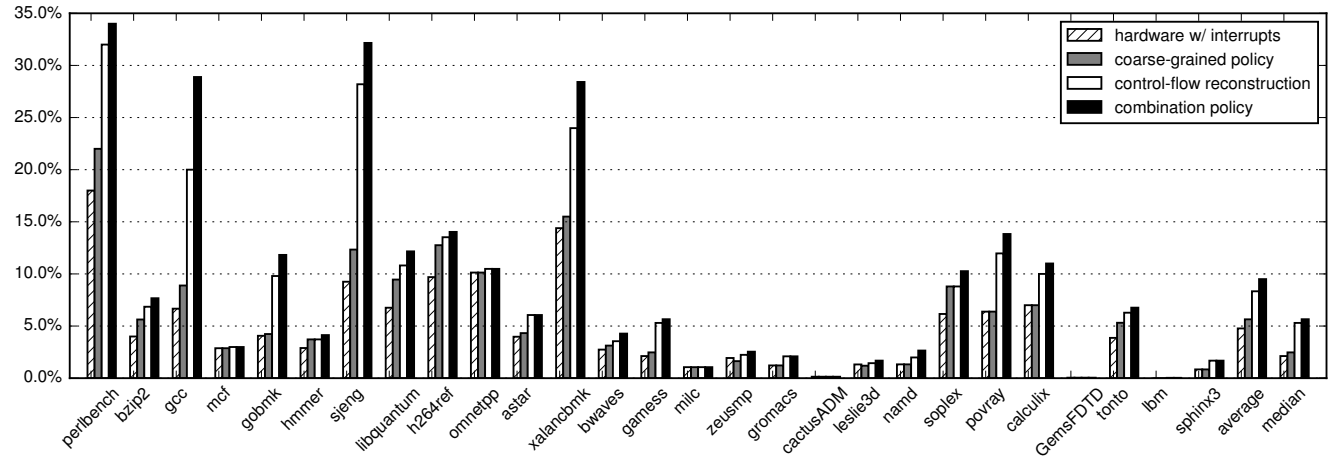


Figure 4: The performance overhead for running SPEC CPU2006 benchmarks with different configurations.

| | Legacy Binary | No Instru. | Complete | Policy | SPEC CPU2006 (%) | |
|---|---|---|---|---|---|---|
| | | | | | SPECint | SPECfp |
| MCFI [33] | × | × | ✓ | fine-grained | 4.0 | 1.6 |
| $\pi$CFI [35] | × | × | ✓ | fine-grained (incrementally growing CFG) | 4.0 | 1.75 |
| shadow stack [16] | × | × | ✓ | shadow stack | 11.6* | 6.0 |
| | × | × | ✓ | parallel shadow stack | 4.7 | 3.0 |
| binCFI [51] | ✓ | × | ✓ | coarse-grained | 9.6 | 6.5 |
| Lockdown [39] | ✓ | × | ✓ | combination | 47.18 | 20.55 |
| ROPecker [11] | ✓ | ✓ | × | heuristics | 2.3 | |
| PathArmor [43] | ✓ | × | × | combination (restrict constant callback) | 3.3 | |
| GRIFFIN | ✓ | ✓ | ✓ | coarse-grained | 9.3 | 3.0 |
| | ✓ | ✓ | ✓ | combination | 16.0/11.9* | 6.2 |

Table 2: The comparison between different CFI techniques. The performance overhead (%) with an asterisk exclude perlbench and gcc in SPECint. We also exclude Fortran benchmarks evaluated in Lockdown and GRIFFIN from SPECfp.

more restricted policy is on par with the compiler-based implementation of the traditional shadow stack scheme.

### 7.2.2 Applications

We evaluate GRIFFIN's performance on real-world applications including three server programs and a client program under both the coarse-grained and combination policy. There are two main differences between these applications and the SPEC CPU2006 benchmarks. First, they are less CPU-intensive. The server programs are I/O-bound and the browser is user-oriented. Second, they all use multiple processes/threads. This could potentially impact GRIFFIN's performance because of the competition on CPU resources.

To benchmark the nginx web server, we create 32 concurrent connections and send 10,000 HTTP requests for files of different sizes using ApacheBench [1]. To benchmark the vsftpd server, we use pyftpbench [3] to request 10MB files in 10 concurrent connections. To benchmark the exim email server, we run the sendemail script [4] to repeatedly send 1KB emails. The chosen workloads are consistent with prior work [43, 48]. We evaluate their throughput reduction.

To evaluate GRIFFIN's performance on Firefox, we use the SunSpider 1.0.2 benchmark. In our current prototype, we do not enforce forward-edge policies for indirect control transfers within jitted code but simply measure the overhead while allowing all targets. We note that an actual policy may be derived from the JIT compiler with additional engineering effort [34]. For non-jitted code, we enforce checks on both forward and backward edges as in other experiments.

We show the results in Figure 5. On average, GRIFFIN incurs modest performance overhead for server programs – 1.8% under the coarse-grained policy and 2.7% under the combination policy. This is because server programs are often I/O-bound. When they wait on I/O, no Intel PT trace is generated, and GRIFFIN has more time to process the control-flow trace. For Firefox, GRIFFIN incurs 7.5% overhead under the coarse-grained policy and 13% under the combination policy.

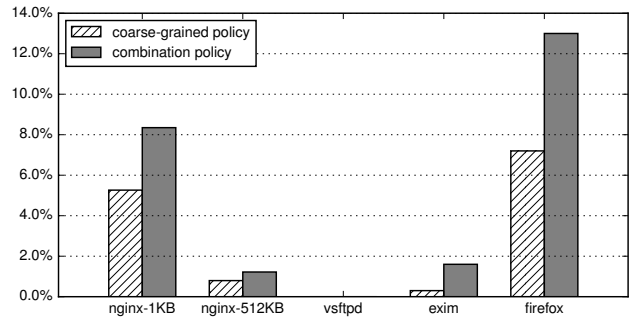


Figure 5: Performance overhead of real-world applications.

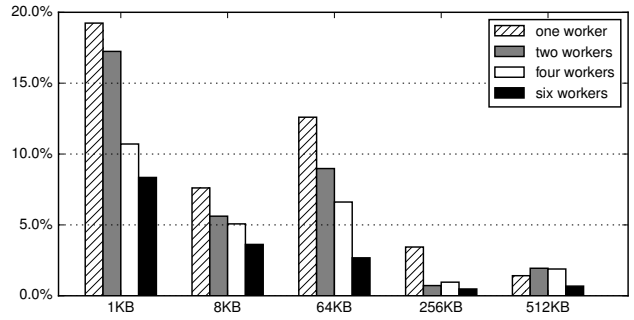


Figure 6: The impacts of different numbers of worker threads on GRIFFIN's performance for the nginx web server.

### 7.2.3 Worker Threads

GRIFFIN leverages multiple worker threads to speed up the processing of trace buffers and CFI enforcement. To understand how different numbers of worker threads impact the performance of real-world applications, we run the nginx web server under the combination policy with the same workload as discussed in Section 7.2.2.

From Figure 6 we can see that using fewer worker threads affects GRIFFIN's performance because of the decreased parallelization. However, the difference decreases as the file size increases. This is aligned with our expectations on I/O bound programs. When the program spends more time on waiting for I/O, the performance slowdown will decrease.

### 7.2.4 Memory Usage

We measured GRIFFIN's memory usage under coarse-grained and combination policies for both SPEC CPU2006 benchmarks and real-world applications. GRIFFIN's memory usage has two parts: control pages for policies and basic blocks, and Intel PT trace buffers. Control pages are dynamically populated at runtime and can be shared between processes. We measured the actually populated control pages. The size of Intel PT trace buffers changes dynamically depending on the rates of Intel PT generating new trace packets and GRIFFIN processing existing trace packets. We measured the peak size of Intel PT trace buffers.

| | coarse-grained | | combination | |
|---|---|---|---|---|
| | control pages | buffers | control pages | buffers |
| perlbench | 1.2 | 5.2 | 7.3 | 356.5 |
| bzip2 | 0.4 | 2.0 | 1.4 | 7.6 |
| gcc | 2.2 | 2.8 | 16.4 | 11.6 |
| mcf | 0.4 | 2.3 | 1.4 | 3.0 |
| gobmk | 1.0 | 1.0 | 6.0 | 445.2 |
| hmmer | 0.7 | 1.1 | 2.3 | 1.0 |
| sjeng | 0.5 | 1.8 | 1.7 | 113.4 |
| libquantum | 0.5 | 2.0 | 1.5 | 21.1 |
| h264ref | 0.8 | 2.5 | 4.2 | 3.9 |
| omnetpp | 1.2 | 2.3 | 4.9 | 2.7 |
| astar | 0.5 | 1.4 | 1.7 | 1.0 |
| xalancbmk | 3.3 | 4.7 | 13.7 | 202.6 |
| bwaves | 0.1 | 1.4 | 3.5 | 32.8 |
| gamess | 4.5 | 2.7 | 55.2 | 3.1 |
| milc | 0.5 | 1.0 | 2.2 | 0.8 |
| zeusmp | 0.1 | 1.0 | 5.3 | 1.2 |
| gromacs | 0.7 | 5.3 | 5.7 | 6.6 |
| cactusADM | 0.3 | 2.0 | 5.1 | 2.1 |
| leslie3d | 0.1 | 0.4 | 4.4 | 0.5 |
| namd | 0.7 | 2.6 | 2.6 | 7.3 |
| soplex | 1.1 | 2.5 | 4.6 | 1.9 |
| povray | 1.3 | 1.6 | 4.6 | 2.1 |
| calculix | 0.9 | 3.8 | 8.6 | 8.9 |
| GemsFDTD | 0.1 | 3.8 | 6.7 | 9.7 |
| tonto | 2.7 | 3.6 | 15.5 | 8.7 |
| lbm | 0.4 | 2.5 | 1.4 | 2.1 |
| sphinx3 | 0.6 | 1.1 | 3.1 | 1.7 |
| nginx | 0.6 | 1.0 | 10.0 | 1.0 |
| exim | 0.1 | 0.8 | 8.6 | 0.8 |
| vsftpd | 0.1 | 0.5 | 6.0 | 0.6 |
| firefox | 103.7 | 9.8 | 377.4 | 84.1 |

Table 3: Peak memory usage (MB) for control pages and trace buffers for the coarse-grained and combination policy.

The results are shown in Table 3. The memory usage for control pages is determined by the enforced policy and program code size. The combination policy uses more control pages than the coarse-grained policy because of the need for control-flow reconstruction. Firefox has a much larger code base than other programs (>100MB) and incurs the highest use of control pages. The peak memory usage for Intel PT trace buffers is determined by two factors: the difference between trace generation and consumption rate and the time a program executes.

We explored using one control page per code page by compressing the basic block information into a one-byte data structure. However, the reduction in control pages was less than the increase in Intel PT trace buffers incurred by the slowdown in our trace processing. Thus, we speculate that further performance improvements may significantly reduce Intel PT trace buffer sizes for programs of higher overhead.

### 7.2.5 Runtime Policy Change

By the design of separating the CFI enforcement from the policy, GRIFFIN naturally supports changing the enforced policy at runtime. We demonstrate one approach that strikes a balance between the performance and security using coarse-grained policies and combination policies.

Specifically, GRIFFIN monitors the memory usage for each program by the number of outstanding trace buffers to process. If the number of outstanding trace buffers surpasses a threshold, GRIFFIN switches to a coarse-grained policy. Once it catches up, GRIFFIN switches back to a fine-grained policy. Note that enforcement cannot be resumed for some stateful policies, such as the shadow stack. In this case, we fall back to a fine-grained policy instead.

Given that perlbench incurs the highest runtime overhead in our evaluation for SPEC CPU2006 benchmarks, we use it to evaluate the effectiveness of dynamic policy change. We use three different thresholds – 256MB, 128MB and 64MB. A lower threshold will have a better performance but weaker security because GRIFFIN will enforce the coarse-grained policy for a longer period of time. In our experiments, we observed 1.0%, 1.7% and 3.0% of the trace is checked by the coarse-grained policy under the three thresholds, respectively. Compared to enforcing the combination policy for the whole execution of perlbench, the incurred overhead are reduced by 8.8%, 17.6% and 23.5%, respectively.

#### 7.2.6 Hardware Enhancements

Current Intel PT hardware does not encode branch source addresses, but instead requires GRIFFIN to reconstruct the control flow to determine such information. As shown in Figure 4, GRIFFIN spends a significant amount of time on control-flow reconstruction while CFI checking is done with little additional cost. Thus, a natural optimization is to augment the trace with addresses of indirect branches to eliminate the requirement for control-flow reconstruction when enforcing stateless policies.

We manufacture traces that include branch addresses before every indirect control transfer and evaluate the impact on the trace size and performance for fine-grained CFI enforcement. Specifically, we insert a `FUP` packet before every `TIP` packet and remove all `TNT` packets. Our experiment shows an average of 90% performance improvement on trace processing and 60% trace size reduction when enforcing a fine-grained stateless policy for SPEC CPU2006 benchmarks. In the worst case, the performance still improves by 62% and the trace size increases by 19%.

However, the proposed enhancement undermines GRIFFIN's potential to enforce stateful policies because of the lack of detailed control-flow information, such as conditional and direct branches. We propose that future hardware designs should consider adding the support for selectively toggling the generation of the complete control-flow trace (i.e., `TNT` packets) for the minimal execution necessary to enforce stateful policies.

For forward edges, reconstructing only necessary control flows enables GRIFFIN to enforce stateful checks based on partial (but sufficient) execution state with little extra processing. For example, to support PathArmor's stateful policy, we modified the manufactured trace to include the complete control-flow information for the period of execution between when a constant callback is passed and the callback returns. For the program with the most stateful forward edges (433.milc) in the SPEC benchmarks, its trace size increases from 0.9% of its original trace size after removal of all `TNT` packets to 1.4% after enabling selective logging.

For backward edges, enforcing a shadow stack requires GRIFFIN to reconstruct the entire control flow to track all direct calls. To improve the efficiency of the shadow stack enforcement, we likely need a different approach. Fortunately, a recently introduced feature called Intel Control-flow Enforcement Technology (CET) [2] enables shadow stack enforcement for unmodified binaries directly from the hardware. We envision that GRIFFIN can leverage CET to do efficient shadow stack checking in the future.

## 8. Related Work

**Control-Flow Integrity.** In 2005, Abadi *et al.* introduced Control-Flow Integrity (CFI) [6], which restricts a program's execution to its Control-Flow Graph (CFG). The initial implementation of CFI labels indirect branches and target instructions, and allows an indirect control transfer at runtime if the source and destination have the same label. Researchers have applied the technique to a variety of programs including privileged software [14, 22, 31, 45]. However, many early implementations focus on reducing the runtime overhead, enforcing coarse CFGs in practice (e.g., use one or two labels) [8, 14, 18, 28, 45, 49–51].

Researchers proposed attacks that exploit the overapproximation inherent in coarse-grained CFI policies [9, 10, 19, 23, 24, 41]. Therefore, researchers started developing fine-grained CFI [33, 42] and context-sensitive CFI implementations [35, 43] to mitigate these attacks. However, Control-Flow Bending attacks demonstrate techniques to bypass all CFI implementations without a shadow stack [10].

**Binary Rewriting.** Researchers have also developed approaches to enforce CFI for legacy binaries without requiring the source code. For instance, binCFI and CCFIR derive the CFI policy directly from binaries and insert checks for enforcement [50, 51]. TypeArmor improves the precision of the computed CFI policy by taking high-level program semantics into account [44]. However, like compiler-based approaches, static binary rewriting "fixes" CFI checks into the program and does not allow adjusting the desired protection to balance security and performance at runtime. In addition, static binary rewriting needs to dynamically translate code addresses of the original programs to the rewritten ones for compatibility, incurring extra performance overheads [51].

Dynamic binary instrumentation has also been explored to protect legacy binaries from control-flow hijacking attacks. It has the flexibility to choose the protection at runtime as GRIFFIN does. ROPdefender implements shadow stacks and Payer *et al.* complements the protection by enforcing fine-grained CFI policies for forward edges [17, 39]. How-

ever, the performance overhead is inherently high (up to 4x) due to the cost of dynamic binary instrumentation.

**Hardware-Assisted CFI.** Researchers have proposed various CFI defenses with hardware assistance, and most of them do not require any instrumentation. CFIMon [47] leverages Branch Trace Store (BTS) to record control transfers and implement CFI checks. However, BTS incurs significant performance slowdown (20x-40x) [43]. CFIMon has only been evaluated on I/O-bound applications.

Researchers have explored using the Last Branch Record (LBR) feature to build CFI defenses. LBR records a *small* number of the most recent control transfers with minimal overheads. Despite the performance benefits, defenses built on LBR are fundamentally limited by the capacity of the LBR stack. For instance, both kBouncer [36] and ROPecker [11] rely on heuristics to detect attacks. A recent proposal shows that an adversary can break these heuristics by using long gadgets and/or launching history-flushing attacks [9]. PathArmor [43] aims to enforce shadow stacks on backward edges. However, PathArmor only checks a small fraction of executed returns at runtime due to the LBR limitation. According to a trace we collected from nginx, it checks less than 0.1% of total returns.

To overcome the limited size of LBR, CFIGuard [48] proposes to combine the LBR feature with the Performance Monitoring Unit (PMU). The key idea is to program the PMU to trigger an interrupt when the LBR stack fills. By properly handling such interrupts, CFIGuard checks all executed indirect branches. Though CFIGuard proposes to improve the performance by only recording indirect control transfers, triggering an interrupt on every 16 of them can slow down programs significantly, particularly for CPU-intensive programs. CFIGuard has only been evaluated for I/O-bound server applications.

LMP [27] uses Intel Memory Protection Extension (MPX) to implement the shadow stack. Compared to shadow stack implementations through randomization, it does not rely on information hiding and is immune to side-channel attacks [21]. However, unlike GRIFFIN, LMP only protects backward edges, requires program source for recompilation and also fixes the instrumentation.

Several hardware CFI systems are proposed to check control flows directly by the hardware. Intel CET [2] protects the forward edges using a coarse-grained policy and the backward edges using a shadow stack. We propose how to integrate CET into GRIFFIN in Section 7.2.6. HCFI [12] and HAFIX [20] modified the ISA to enforce CFI policies (e.g., shadow stack) directly from the hardware. Despite the performance benefits, they are not as practical as GRIFFIN because GRIFFIN can run on unmodified commodity hardware.

Besides our work, there are two concurrent and independent efforts that leverage Intel PT for CFI enforcement. FlowGuard [32] and PT-CFI [25] both aim to accelerate their trace processing and policy checking by avoiding reconstruction of complete control flows in their fast processing paths. Without the complete control flows, the source addresses of indirect branches are not available, so FlowGuard and PT-CFI cannot enforce fine-grained, forward-edge CFI policies like GRIFFIN. FlowGuard refines CFI policies using offline training on common paths, resorting to control-flow reconstruction only for less frequent paths. PT-CFI restricts control flows on backward edges by applying shadow stack checking. We cannot directly compare GRIFFIN's performance with FlowGuard and PT-CFI because they did not use the "train" workload in their SPEC CPU2006 benchmarks. FlowGuard used the "test" workload, and PT-CFI used a customized workload.

## 9. Conclusion

In this paper, we presented GRIFFIN, a hardware-assisted, operating system CFI enforcement mechanism. GRIFFIN uses Intel Processor Trace (PT) to protect user-space processes (e.g., web browsers) from control-flow hijacking attacks. GRIFFIN is capable of enforcing stateful CFI policies on both forward and backward edges, enabling the strong prevention for attacks on control flow. GRIFFIN also enables tradeoffs between security and performance by supporting a variety of control-flow policies. GRIFFIN is designed to leverage Intel PT traces efficiently. When enforcing coarse-grained CFI policies, it uses `TIP` packets in the trace directly. GRIFFIN reconstructs the complete control flow from Intel PT traces in parallel to enforce fine-grained CFI policies. It also produces stateful information in parallel before sequential checking for low overhead. As a result, GRIFFIN can achieve comparable performance with software-only instrumentation defenses on the SPEC CPU2006 benchmarks, showing that strong, hardware-assisted CFI enforcement can be a viable alternative.

## Acknowledgements

We thank the anonymous reviewers for their constructive feedback to this work. Specially, we thank Mathias Payer and Gang Tan for providing invaluable insights and suggestions at various times of this project. We would like to thank Graham McIntyre and Pedro Teixeira for their tremendous help at the early stage of this project. We are also grateful to Beeman Strong for his quick and detailed answers to our questions about Intel Processor Trace. The work was supported by the National Science Foundation under grant number CNS-1408880. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

## References


[1] ApacheBench: a complete benchmarking and regression testing suite. `https://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Intel control-flow enforcement technology (CET) preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.

[3] pyftpdlib. `https://github.com/giampaolo/pyftpdlib`.

[4] sendemail. `http://caspian.dotconf.net/menu/Software/SendEmail`.

[5] Intel 64 and IA-32 architectures software developer's manual. *Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, 2016.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 340–353. ACM, 2005.

[7] S. Andersen and V. Abella. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, 2004.

[8] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pages 353–362. ACM, 2011.

[9] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2014.

[10] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2015.

[11] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21th Network and Distributed System Security Symposium (NDSS)*. ISOC, 2014.

[12] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2016.

[13] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 952–963. ACM, 2015.

[14] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pages 292–307. IEEE, 2014.

[15] G. Dabah. diStorm - Powerful Disassembler Library for x86/AMD64. `https://github.com/gdabah/distorm`.

[16] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 555–566. ACM, 2015.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 40–51. ACM, 2011.

[18] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. ISOC, 2012.

[19] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pages 401–416. USENIX Association, 2014.

[20] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*. ACM, 2015.

[21] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[22] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016.

[23] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014.

[24] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pages 417–432. USENIX Association, 2014.

[25] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2017.

[26] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43. ACM, 1992.

[27] W. Huang, Z. Huang, D. Miyani, and D. Lie. LMP: light-weighted memory protection with hardware assistance. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. ACM, 2016.

[28] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*, pages 383–398. USENIX Association, 2009.

[29] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: a technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 344–360. ACM, 2015.

[30] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014.

[31] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, pages 195–208. ACM, 2010.

[32] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. Transparent and efficient cfi enforcement with intel processor trace. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017.

[33] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014.

[34] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1317–1328. ACM, 2014.

[35] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 914–926. ACM, 2015.

[36] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, pages 447–462. USENIX Association, 2013.

[37] PaX Team. Documentation for the PaX project - overall description. `https://pax.grsecurity.net/docs/pax.txt`, 2008.

[38] M. Payer and T. R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR)*. ACM, 2010.

[39] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 144–164. Springer, 2015.

[40] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[41] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, pages 745–762. IEEE, 2015.

[42] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[43] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 927–940. ACM, 2015.

[44] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[45] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, pages 380–395. IEEE, 2010.

[46] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

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

[48] P. Yuan, Q. Zeng, and X. Ding. Hardware-assisted fine-grained code-reuse attack detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 66–85. Springer, 2015.

[49] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 29–40. ACM, 2011.

[50] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, pages 559–573. IEEE, 2013.

[51] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2013.

[52] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN International Conference on Virtual Execution Environments (VEE)*. ACM, 2014.