PT-CFI: Practical Backward-Edge Control Flow Integrity Using Intel Processor Trace

Yufei Gu[†‡], Qingchuan Zhao[‡], Yinqian Zhang[⋆], Zhiqiang Lin[‡]
[†]Cloudera Inc, Palo Alto, California
[‡]Department of Computer Science, The University of Texas at Dallas
[⋆]Department of Computer Science and Engineering, The Ohio State University
firstname.lastname@utdallas.edu, yinqian@cse.ohio-state.edu

ABSTRACT

This paper presents PT-CFI, a new backward-edge control flow integrity (CFI) based on a recently introduced hardware feature in Intel x86 called Processor Trace (PT). Designed primarily for offline software debugging and performance analysis, PT offers the capability of tracing the entire control flow of a running program. In this paper, we explore the practicality of using PT for security applications, and propose a new CFI model that enforces a perfect backward-edge policy for native COTS binaries based on the traces from Intel PT. By exploring the intrinsic tracing properties inside PT with a system call based synchronization primitive and a deep inspection capability, we have addressed a number of technical challenges such as how to make sure the backward edge CFI policy is both sound and complete, how to make PT enforce our CFI policy, and how to balance the performance overhead. We have implemented PT-CFI and evaluated with a number of programs including SPEC2006 and HTTP daemon. Our experimental results show that PT-CFI enforces strong backward-edge CFI with no false positive or false negative, as well as minor performance overhead.

Keywords

Return oriented programming; Control flow integrity; Shadow stack; Intel PT

1. INTRODUCTION

Control flow hijacking has been one of the most severe cyber threats for over 40 years. Given an exploitable vulnerability such as a buffer overflow in a program, which accepts untrusty inputs, an attacker can directly compromise the program by taking control of the program's execution flows, and perform whatever malicious actions of his or her wishes. Over the past a few decades, we have witnessed numerous such attacks. Stack smashing [33], return-into-libc [48], return oriented programming (ROP) [39, 12] (and its variants such as BROP [5] and JIT-ROP [40]), jump-oriented programming (JOP) [7], and even call-oriented programming (e.g., COOP [36]) all belong to this category. It is likely that these attacks will continue to remain a major cyber threat for years to come.

CODASPY '17, March 22–24, 2017, Scottsdale, AZ, USA
© 2017 ACM. ISBN 978-1-4503-4523-1/17/03. . . $15.00
DOI: http://dx.doi.org/10.1145/3029806.3029830

Correspondingly, numerous defenses have been proposed to defend against control flow hijacking. Notable examples include stack canary [16] (which can defeat stack smashing), data execution prevention (DEP) [3] (which can defeat code injection), address space layout randomization (ASLR) [42] (which can make the hijack exploit code much harder to construct), and control flow integrity (CFI) [2] (which aims to ensure the integrity of control flow transfer always following legal program path). Canary, DEP, and ASLR are all practical defenses and they all have been adopted by industry in mainstream computing devices including even in the mobile platform. Therefore, simple stack smashing or code injection attack does not work anymore in modern computing platform, and the mainstream exploits have to use ROP or its variant (e.g., Q [38], return-to-signal [8], JIT-ROP [40], or BROP [5]). To really defeat ROP, it appears that CFI is the most promising technique since in theory it can fundamentally solve the control flow hijacking problem as all these attacks including ROP violate the intended program control flow. However, in practice CFI has not been widely adopted yet, at least in the case for protecting COTS binaries.

CFI essentially is a program path-level access control model. For any access control mechanism to work, it needs a policy and an enforcement. The first CFI model by Abadi *et al.* [2] uses an inlined reference monitoring (IRM) [20] in the program code to enforce the CFI policy. Specifically, at each indirect control flow transfer point (*i.e.*, indirect `call`, indirect `jmp`, and `ret`), the CFI enforcement code inlined with the original program will check with a CFI policy to detect whether there is a violation. The security policy in the CFI is quite simple: the execution of any control-flow transfer should not diverge from its legitimate path. To construct the CFI policy, the traditional form of CFI builds a control-flow graph (CFG) from the protected program, from which to get all the legal target(s) of each indirect control transfer. Then at the runtime, the inlined enforcement code will check the control flow target whether or not belongs to a set of white-listed ones. As such, CFI guarantees that the execution path of the program strictly follows an edge in its CFG.

Unfortunately, there are two main challenges that hinder the practicality of CFI. First and foremost, how to make sure the CFI policy is both sound and complete. Often times, the statically extracted CFG (either from program source code or binary) is an over-approximation of the legitimate control flows of the program. This is because precise static extraction of CFG requires accurate pointer analysis of each indirect call or jump to estimate its targeted "points-to" set. This type of analysis, however, is challenge to get it accurate, primarily due to the dynamic nature (e.g., computed `jmp` and `call` target) of low-level programming languages like C. Meanwhile, the same function may have multiple legitimate call sites, corresponding to multiple edges in the CFG. However, at runtime, only one of the edges should be allowed at any state of execution. This issue cannot be addressed by CFI that relies on static code analysis alone; context-sensitive methods (*e.g.* using a shadow stack [2]) need to be used at runtime.

Second, how to enforce the CFI policy. When given program source code, one can use compilers to automatically insert an IRM at each indirect control flow transfer point to enforce the policy. However, it becomes much more challenging when only given application binaries. The first CFI model rewrites the protected binary, but it requires the corresponding debugging symbols. Without those debugging symbols, one has to solve both the binary disassembling and rewriting challenge. While dynamic binary instrumentation (DBI) does not face disassembling and rewriting issues since it rewrites the binary on-the-fly, it still faces other challenges such as the high performance overhead (as DBIs are usually slow).

Prior work on binary-level CFI have been striving to address these challenges. However, most studies fall short in addressing the first challenge. For instance, BinCFI [52], CCFIR [51] and BinCC [47] only enforces a coarse-grained CFI policy: indirect jumps or calls are restricted to a white-list of targets and function returns are constrained to call-preceded addresses. It has been shown, however, by follow-up studies that these coarse-grained CFI implementations can be bypassed by advanced ROP attacks [18]. Specially the weak backward-edge policy—returning to only call-preceded targets—can be easily circumvented by ROP attacks with only call-preceded gadgets [11, 21]. A more recent work, TypeArmer [46], advances the state-of-the-art of the forward-edge policy for binary-level CFI by statically analyzing binary code to match callers and callees. But still, no backward-edge policy is provided. Shadow stacks [2, 19] match return addresses to their call sites, and hence offer strong backward-edge policy. However, shadow stacks are difficult to implement correctly to offer reliable security guarantee [12]. Moreover, existing implementations of shadow stacks mostly rely on binary rewriting, facing the second aforementioned challenges.

To overcome the limitations of binary rewriting or instrumentations, recent studies resort to existing CPU hardware features to assist CFI policy enforcement. For instance, last branch records (LBR) was exploited by many notable works including kBouncer [34], ROPecker [15], CFIGuard [50], and PathArmer [45] to keep track of a short history (usually only upto 16 LBR entries) of indirect branches. The main issue of LBR-based solutions is that it is vulnerable to history-flushing attacks [37, 11], in which the malicious payload intentionally includes dummy branch instructions to flush LBR entries to hide suspecious indirect branches. Branch trace store (BTS), which in contrast to LBR records all prior indirect branches, was used by CFIMon [49], but it also comes along with higher performance overhead. As such, both LBR and BTS have limitations when used to enforce CFI policies, which therefore motivates us to seek alternative, more effective, approaches.

In this paper, we aim to fill this gap and propose PT-CFI, a practical backward-edge CFI (note that we leave forward-edge CFI for binary code in future works) that works for x86 COTS binaries by using a recent innovative hardware feature Intel Processor Trace (PT). While Intel had offered prior hardware-based tracing features such as LBR and BTS, PT provides many compelling features. Specifically, the path history recorded by LBR is limited to a few dozen instructions. Meanwhile, BTS has significant slowdown though it supports unlimited path history. Therefore, Intel recently introduced PT, which can log execution trace with extremely low performance impact (less than 5% performance overhead) and provide a complete control flow tracing with both cycle count and timestamp information.

However, PT is not designed for online security protection but rather for offline software debugging or performance analysis. As such, there are a number of technical challenges in order to make a PT-CFI. These challenges include how to derive the CFI policy based on the PT trace and the monitored binary, how to enforce the CFI policy, and how to make sure the control flow monitoring would not introduce large overhead. We have addressed these challenges by exploring the intrinsic tracing property inside PT with a system synchronization primitive and a deep inspection capability. We have implemented PT-CFI and evaluated with both the SPEC2006 CPUINT benchmark suite and Nginx HTTP daemon. Experimental results show that PT-CFI only introduces very small overhead for the running binary.

Contribution. The main contribution of this paper can be summarized as follows:

- We make the first attempt of exploring PT for real-time monitoring of control flow violation and propose PT-CFI, a new practical backward-edge CFI model for x86 COTS binaries.
- We devise a number of enabling techniques including system call based enforcement and synchronization which enforces the CFI policy at the entry point of each system call, and a deep inspection primitive, which is invoked like exception handling when a CFI policy is incomplete.
- We have implemented PT-CFI, and applied it to detect ROP attacks, which overcomes several limitations of prior work such as Kbouncer and ROPecker. Meanwhile, the performance overhead of PT-CFI is quite small (around 20% on average for a set of tested SPEC2006 benchmarks).

2. BACKGROUND AND RELATED WORK

2.1 Control-Flow Hijacking and ROP

Memory corruptions are one of the most commonly exploited vulnerabilities in programs written in native programming languages such as C/C++. By allowing unsanitized input to overwrite data or code in the victim program's memory space, these vulnerabilities enable a wide range of attacks, such as information leakage, arbitrary code execution and privileges escalation [41]. While non-control-data attacks [14, 22, 9, 23] have been demonstrated in previous work (especially in recent years), control-flow hijacking still remains the most commonly used attack method. In control-flow hijacking attacks, control data that are used to direct the program's control flows are corrupted by the attacker. When these data (*e.g.*, return addresses, indirect jump targets) are loaded into the program counter the program's execution will be diverted from its designed target.

There are several instances of control flow hijacking attacks such as code injection [33], code-reuse [48], return-oriented programming (ROP) [39, 12], jump-oriented programming (JOP) [7], and the recently introduced call-oriented programming (COP) [36]. Among them, ROP attacks are increasingly becoming the mainstream: they are more advantageous than code injection attacks, because they defeat the widely used Data Execution Prevention (DEP) protection; ROP attacks are also resilient to defense against simple return-to-libc attacks because they can reuse library code without explicit function calls [39].

In ROP attacks, short sequences of code, dubbed *gadgets*, that already exist in the victim program are chained together and reused for purposes other than their designed logic. In conventional ROP

Table 1: Binary-level control-flow integrity enforcement

	backward-edge policy	CFI enforcement
CFI [2] (2005)	shadow stack	binary rewriting
DROP [13] (2009)	heuristic	dynamic instrumentation
ROPDefender [19] (2011)	shadow stack	dynamic instrumentation
CFIMon [49] (2012)	call-proceded targets	critical function + async
MoCFI [17] (2012)	shadow stack	runtime hooking
CCFIR [51] (2013)	whitelist targets	binary rewriting
BinCFI [52] (2013)	call-proceded targets	dynamic instrumentation
kBouncer [34] (2013)	heuristic + call-proceded	critical function
ROPecker [15] (2014)	heuristic	non-exec page
CFIGuard [50] (2015)	whitelist targets	PMU interrupts
PathArmer [45] (2015)	call/return matching	dynamic instrumentation
BinCC [47] (2015)	bounds-check	static rewriting
O-CFI [29] (2015)	bounds-check	static rewriting
TypeArmer [46] (2016)	None	dynamic instrumentation

attacks, these gadgets all end with `ret` instruction. Hence attackers can prepare a sequence of return addresses on the stack to "return" to these gadgets in orders that will fulfill specific functionality. Later work shown that indirect jumps or calls can also be used to construct "return" gadgets without `ret` instructions [12, 7].

2.2 Control-Flow Integrity

CFI is a widely studied technique to prevent control-flow hijacking attacks. It was first proposed by Abadi *et al.* [2] in 2005, which aims to enforce policies on the control-flow transfers of a software program so that the execution of the program does not diverge from the legitimate path. The traditional form of CFI first constructs a control-flow graph (CFG) from a program and then checks the target of each indirect control transfer (*e.g.*, indirect jump, indirect call, and return) at runtime so that only a set of white-listed targets for each indirect control transfer is allowed. In this way, CFI guarantees that the execution of the program strictly follows an edge in its CFG. A CFI implementation can be either *fine-grained* or *coarse-grained*. In a fine-grained CFI, each indirect control transfer has its own set of target addresses that can be allowed to take at runtime. This is usually achieved through program analysis because the sets of targets are program-dependent. In contrast, a coarse-grained CFI partitions the indirect control transfers and their target addresses into several *equivalence classes*, and the total number of equivalence classes is usually no more than three [32].

While many efforts have achieved fine-grained CFI via complex source code analysis, such as CFLocking [6], Forward-edge CFI [44], RockJIT [31], MCFI [30],CPI [25], CCFI [28], πCFI [32], *etc.*, fine-grained binary-level CFI remains very challenging. Table 1 summarizes prior research on binary-level control-flow integrity. Particularly, CFIMon [49], BinCFI [52], and kBouncer [34] only enforce a coarse-grained backward-edge policy: returns are only allowed to addresses preceded by a call-site. CCFIR [51] and CFIGuard [50] enforce slightly finer-grained policy by allowing a smaller set of white-listed return targets. BinCC [47] and O-CFI [29] restrict returns across a specified boundary, greatly reducing the usable gadgets. The strongest backward-edge CFI policy is enforced by shadow stacks (implemented in the original CFI, ROPDefender [19], MoCFI [17] and PathArmer [45] through static binary rewriting or dynamic binary instrumentation), which strictly matches call/return pairs. In contrast to these prior efforts, PT-CFI aims to enforce a perfect backward-edge CFI policy using shadow stacks, without static binary rewriting or dynamic instrumentation.

2.3 Hardware-Assisted ROP Detection

Besides our work, there has been a few studies exploring hardware-assisted approaches for ROP detection. Most notable results among them are CFIMon [49], kBouncer [34], ROPecker [15] and CFIGuard [50]. ROPecker, kBouncer and CFIGuard are studied the use of last Branch Record (LBR) that are available on Intel processors for ROP detection. Note that LBR provides a hardware mechanism to record the source address and target address of most recently used branches. These approaches statically scan the program binary to construct a database of ROP gadgets. Once the detection is triggered (the time of ROP check differs in these approaches), the most-recent branches are compared with the gadget database, and according to a specific security policy (*e.g.*, number of instructions in a gadget, consecutive gadget numbers detected in the LBR), the LBR data may indicate an ROP attack. Unfortunately, recent studies [37, 11] have shown that these LBR-based approaches are vulnerable to several attack methods. The most noteworthy attacks among them are LBR-flushing attacks, in which ROP code intentionally induce unimportant branches to fill the limited number of entries in LBR (usually less than 16).

Most close to our work is CFIMon [49], which exploited Branch Trace Store (BTS) on Intel processors. Intel BTS is a processor component that provides program tracing mechanisms to software layers, which captures all types of control flow information, including direct and indirect jump and call, and also function return. Both the source address and target address are stored in a specific memory region for batch processing. CFIMon is triggered to detect control flow violation when the memory buffer is full or sensitive functions are accessed. CFIMon detects control flow violation by monitoring if backward CFG edges return to a call-preceded target, and if indirect calls transfer control flows actually to the first instruction of a function. Indirect jumps are marked as suspicious if not seen before. But the policy to treat these suspicious indirect jumps is not clearly defined, leaving CFIMon potentially vulnerable to carefully crafted ROP attacks [49]

In addition to exploiting existing hardware features in commodity processors, some other work designed new hardware components to detect ROP attacks. Relevant to our work is due to Lee *et al.* [26] that implemented an FPGA-based ROP detection system for ARM devices that executes asynchronously with the protected program. Unlike our work, they do not maintain any synchronization between the monitoring program and monitored program, and therefore, ROP detection cannot effectively prevent the attacks from damaging the system. Moreover, their approach has a common issue as all other studies that detect ROP attack by developing new hardware components: while they all present interesting ideas, nevertheless, because they require additional hardware supports, the likelihood of real world adoption is low.

2.4 Intel Processor Trace

Intel Processor Trace (PT) is a new hardware feature for software program debugging and performance profiling, which is available in Intel Broadwell or later processors. It traces the control flow of software programs with minimum performance overhead that is sufficient low for PT to be used in production systems.

More specifically, the control flow information is collected by PT in *data packets* in real-time, which are then sent to memory buffers or other output methods for processing. While several types of PT packets are defined by Intel and collected at runtime, three types of packets are particularly useful in control flow tracing: Taken Not-Taken (TNT) packets, Target IP (TIP) packets, Flow Update Packets (FUP). The TNT packets collect taken and not-taken indication for conditional direct branches; the TIP packets collect target addresses for indirect calls, indirect jumps and returns; asynchronous events such as exceptions and interrupts will generate FUP packets together with TIPs. Unconditional direct branches are excluded from the PT packets. PT also compresses conditional branches and use only one bit to indicate branch taken or not-taken in TNT packets.

Intel PT supports filtering packets based on the Current Privilege Level (CPL) or CR3. Therefore, it is possible to trace all user-space programs or selectively trace only one program. Context switch can also be supported so that multiple programs can be traced sequentially. Moreover, the precise timing of each data packet are also optionally recorded. Therefore, with the knowledge of binary information, one can reconstruct the entire control flow of the original software program, together with the precise timing of each branch.

Given the capability of Intel PT in tracing program control flows, it is attempting to use PT for control-flow violation monitoring and detect ROP attacks. However, Intel has designed PT particularly to reduce overhead with the cost of increased decoding overhead. According to Intel web site, the decoding of the traces is "several orders of magnitude slower than tracing". A typical use case defined by Intel is to execute a software program and capture the trace data asynchronously in memory regions that can be processed after the execution of the program. Therefore, our design of PT-CFI faces several technical challenges that we will elaborate in later sections.

Prior to ours, only a few work has explored the use of Intel PT in practical applications. Balakrishnan *et al.* [4] and Thalheim *et al.* [43] studied the use of Intel PT to implement fine-grained provenance systems. Kasikci *et al.* [24] developed a software failure diagnose system using PT. However, in all these existing work, PT packets are collected for offline analysis. Therefore, our study presents the first attempt of online uses of Intel PT technology.

3. PT-CFI **OVERVIEW**

In this section, we present an overview of PT-CFI. We first describe a simplified running example in Section 3.1, which will be used throughout the paper to discuss various technical challenges we have to solve in Section 3.2. Then, we present our key insights of how to solve the challenges in Section 3.3. Next, we discuss our CFI policy in Section 3.4. Finally, we give an overview of PT-CFI in Section 3.5.

3.1 A Running Example

Fig. 1(a) illustrates the source code of a very simple program, which accepts command line inputs and then executes one of the three functions accordingly: `foo`, `bar`, and `overflow`. Among them, `overflow` function contains a stack overflow, and an attacker can compromise this program to execute a shell for instance. We compile this program using `gcc` without canary protection. The partial binary code is illustrated in Fig. 1(b).

We run this program with four different inputs: The first three just triggered the three different function pointers but the fourth one triggered the `overflow` along with a ROP payload to execute `/bin/sh`. The system call (syscall for short henceforth) traces for the first two were the same (both have 29 syscalls in total, and they both trigger a `write` syscall by `printf`), as show in Fig. 1(c). The third one has only 28 syscalls without the `write` system call compared to the first two, but the forth one triggered the unexpected `execve` syscall.

Note that PT is primarily designed for debugging and performance analysis. A typical use case of PT is the following: programmers compile the target program, and execute it atop a PT enabled platform. During the execution, the hardware will generate a large volume of PT packets, which is often stored in a log file for offline analysis. There are already available tools such as `perf` that is able to parse the PT packets and reconstruct the entire execution path history of a program based on the trace. While PT packets has already been compressed, usually it will still generate up to hundreds of megabytes of trace data per second per core. As an example, we also illustrate partially decoded PT packets in Fig. 1(d). More specifically, we can notice that there are various types of PT packets, including:

- **Target IP (TIP) packets**: if a control flow transition is triggered by an *indirect* control flow transfer, the hardware will generate a TIP packet, which is particular useful when building our CFI model. Usually, a TIP packet contains a virtual address of the target or just an offset whose base address is shared by prior TIP packets. An instance of TIP packet is `TIP 0x4004d0`, as illustrated in Fig. 1(d), which is actually an indirect `jmp` to the starting address of `_start`. The next TIP packet TIP 0x4a6 is actually also a `jmp` target address, caused by the first instruction in the PLT entry of `__libc_start_main`. Note that direct call does not have a TIP packet. That is why there is no TIP pointing to the first instruction in the PLT entry of `__libc_start_main`.

- **Taken Not-Taken (TNT) packets**: if there is a conditional control flow transfer (i.e., all of the `jcc` instructions such as `je/jne`), then the hardware will generate a bit in a TNT packet and this bit represents taken or not-taken for that particular branch. A TNT packet can at most encode six TNT bits. There are also several TNT packets in Fig. 1(d), such as TNT TTN (3) and TNT TNTNTN (6). Combined with the original binary code, TNT packets can be used to capture the exact execution path of a program. Since attacker cannot alter the destination address of conditional branches, TNT packets are out of CFI interest. In addition, similar to `jcc`, unconditional direct branches (e.g., *direct* `jmp/call`) are excluded from the PT packets since they can be also directly recovered with program code.

- **Flow Update Packets (FUP)**: if there is an asynchronous event such as exceptions and interrupts, the hardware will generate a FUP packet together with TIPs. As an example, FUP 0x7f569383a1e0 means the control flow will transfer from instruction at 0x7f569383a1e0. Followed, there is also a `TIP.PGD` and `TIP.PGE`, which denotes Packet Generation Disable (PGD), and Packet Generation Enable (PGE). These two TIP sequences are usually from the interrupt handler execution. FUP packets are also out of our current CFI interest.

3.2 Technical Challenges

If we directly analyze the recorded PT packets offline without any additional effort, we can certainly use PT for control flow diagnosis or forensics since we can rebuild the entire control flow trace and any deviation from normal control flow will be detected. However, such an offline usage cannot be used for online CFI. Therefore, we have to solve a number of technical challenges.

- **How to define the CFI policy**. To design a CFI model, we have to first define the CFI policy and extract them from the binary code. However, what we have is merely the PT trace and also the binary code of the corresponding program we aim to protect. While we can build a run-time control flow graph based on PT traces and then compare with the CFG extracted from the static binary code, we may be able to form a CFI policy to detect most attacks. However, such an approach would be too slow. Meanwhile, statically we still do not know all of the legal target for indirect call and indirect

```
#include <stdio.h>
void foo(char *str){
    printf("foo:%s\n", str);
}
void bar(char *str){
    printf("bar:%s\n", str);
}
void overflow(char *str){
    char buf[32];
    strcpy(buf, str);
}
void main(int argc, char **argv){
    void (*fptr) (char *);
    int choice;
    choice = (atoi(argv[1]) % 3);
    switch (choice) {
    case 0:
        fptr = foo; break;
    case 1:
        fptr = bar; break;
    case 2:
        fptr = overflow;
    }
    fptr(argv[2]);
}
```

(a) Source Code

```
0000000000400490 <printf@plt>:
400490:       jmpq   *0x200b8a(%rip)
400496:       pushq  $0x1
...
00000000004004a0 <__libc_start_main@plt>:
4004a0:       jmpq   *0x200b82(%rip)
4004a6:       pushq  $0x2
...
00000000004004c0 <atoi@plt>:
4004c0:       jmpq   *0x200b72(%rip)
4004c6:       pushq  $0x4
...
0000000000400590 <frame_dummy>:
400590:       cmpq   $0x0,0x200888(%rip)
...
00000000004005bd <foo>:
4005bd:       push   %rbp
...
00000000004006c0 <__libc_csu_init>:
4006c0:       push   %r15
4006c2:       mov    %edi,%r15d
...
00000000004004d0 <_start>:
4004d0:       xor    %ebp,%ebp
...
0000000000400570 <__do_global_dtors_aux>:
400570:       cmpb   $0x0,0x200ad9(%rip)
...
0000000000400626 <main>:
400626:       push   %rbp
...
400648:       callq  4004c0 <atoi@plt>
40064d:       mov    %eax,%ecx
...
4006b3:       callq  *%rax
4006b5:       leaveq
...
00000000004006c0 <__libc_csu_init>:
...
0000000000400734 <_fini>:
```

(b) Partial Disassembly

```
execve(..) = 0
brk(0) = 0x243b000
access(..) = -1 ENOENT
mmap(..) = 0x7fcaddfec000
access(..) = -1 ENOENT
open(..) = 3
fstat(..) = 0
mmap(..) = 0x7fcaddfd3000
close(3) = 0
access(..) = -1 ENOENT
open(..) = 3
read(..) = 832
fstat(..) = 0
mmap(..) = 0x7fcadda07000
mprotect(..) = 0
mmap(..) = 0x7fcadddc1000
mmap(..) = 0x7fcadddc7000
close(3) = 0
mmap(..) = 0x7fcaddfd2000
mmap(..) = 0x7fcaddfd0000
arch_prctl(..) = 0
mprotect(..) = 0
mprotect(..)     = 0
mprotect(..) = 0
munmap(..)          = 0
fstat(..) = 0
mmap(..) = 0x7fcaddfeb000
write(1, "foo:bb\n", 7) = 7
exit_group(7)
```

(c) Output of `strace` when run `foo`

```
...
00001f4f:  TIP 0x4004d0
00001f5a:  TIP 0x4a6
...
00001fff:  TIP 0x4006c0
0000200a:  TNT TTN (3)
0000200d:  TIP 0x590
00002011:  TNT TNTNTN (6)
00002014:  TNT NTTN (4)
0000201a:  TIP 0x626
00002021:  TIP 0x4c6
00002027:  TIP 0x7f5693b2a4e0
...
0000208f:  TIP 0x40064d
00002099:  TNT NNN (3)
0000209a:  TIP 0x5bd
000020a1:  TIP 0x496
000020a7:  TIP 0x7f5693b2a4e0
...
0000219f:  FUP 0x7f569383a1e0
000021aa:  TIP.PGD no ip
000021ad:  TIP.PGE 0xa1e0
...
000022f7:  TIP 0x4005df
00002302:  TIP 0x6b5
00002307:  TIP 0x7f5693770ec5
00002312:  TIP 0x9376e426
0000231a:  TIP 0x93b2a4e0
00002322:  TNT NNNNNT (6)
...
0000239a:  TIP 0x5340
000023a7:  TIP 0x400570
000023b2:  TNT NNTTNN (6)
000023b5:  TIP 0x734
...
```

(d) Partial PT Trace when run `foo`

Figure 1: A Running Example Used to Illustrate Our PT-CFI Approach.

jumps, a grand challenge today for static binary code analysis (due to the need of the sophisticated point-to analysis).

- **How to enforce the CFI policy**. PT packets are directly generated by the underlying hardware in an asynchronized way. However, the policy enforcement and program execution should be synchronized. Otherwise, a control flow hijack attack might have already caused damages before it is detected. While rewriting the binary code and insert our CFI enforcement code might work, we would like to avoid using any binary rewriting especially for x86 COTS binary, since we aim for a practical CFI.

- **How to minimize the overhead**. Performance is often a critical factor while designing CFI. For instance, one can easily design a CFI by using dynamic binary instrumentation (e.g., PIN [27]). However, such an approach often has high overhead. While PT packet generation has less than 5% overhead, PT packet consumption as well as our CFI enforcement must be designed in an efficient way.

3.3 Key Insights

Having analyzed the internals of various PT packets and understood how software interacts with the PT hardware, we have obtained the following key insights to address the above challenges.

- **Using TIP sequence graph for the CFI policy**. Ideally, we should have used the entire PT trace to rebuild a complete CFG and compared with the statically extracted CFG from binary code for the policy. While purely static analysis of the binary code cannot resolve many indirect call and indirect jump edges, we can use the runtime traces to connect them. However, this is an expensive approach because parsing each

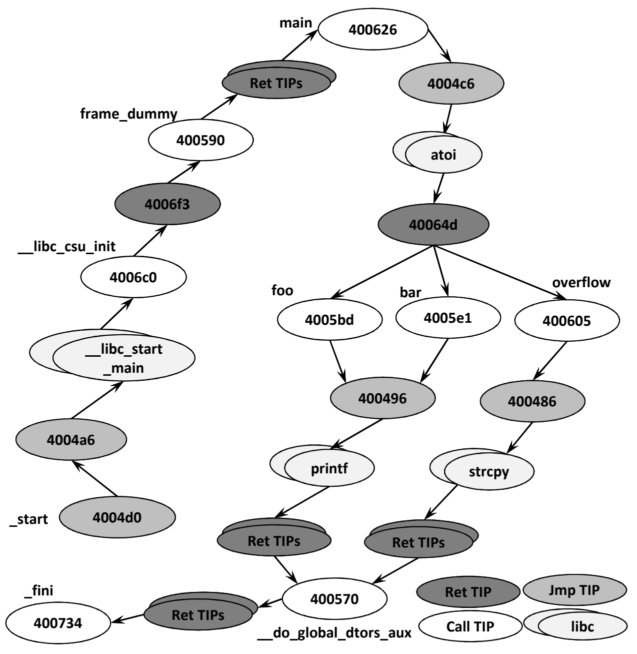

Figure 2: A Partial TIP Sequence Graph for Our Running Example.

PT packet and rebuild a dynamic CFG with original binary code usually takes a large amount of time.

Fortunately, we notice that we can actually use a lightweight, TIP sequence graph to detect the anomalies. In particular, all indirect control flow transfers (call, ret, jmp) will trigger a corresponding TIP packet. Therefore, we can build a TIP sequence graph, and compare this graph with the legitimate TIP graph. As illustrated in Fig. 2, the node of this graph is an indirect control flow transfer point, and the edge captures the transition between the two points. For instance, considering the first two sequences shown in Fig. 1(d), namely `TIP 0x4004d0` and `TIP 0x4a6`, the TIP sequence graph of this two nodes shown in Fig. 2 captures the transition from `_start` to the PLT entry of `__libc_start_main`.

While we cannot build the legitimate TIP graph statically since we cannot resolve the corresponding destination addresses without any additional sophisticated point-to analysis, we can build it from the TIP trace since they exactly capture the indirect control flow transitions. Then to really differentiate between legal and illegal control flow transfers (at least for the backward return edges), we can use a deep inspection technique discussed below. We will detail how we extract the CFI policy in Section 4.1.

- **Using syscall interposition for the enforcement**. Once we have defined the CFI policy, we must check it at runtime to ensure the software adhere to the legitimate control flow path. The first CFI approach by Abadi *et al.* uses the binary rewriting (with debugging symbols) to enforce the policy. However, without debugging symbols, it is very hard to correctly rewrite a binary. Therefore, we would like to seek alternative approaches.

  Inspired by ROPecker and kBouncer, where they enforce the security policy at selected syscall execution point, we can also adopt such an approach. In particular, while the PT packet generation is asynchronized, we can ask PT to stop at selected system call execution point, and the execution can continue only there is no violation of our CFI policy. It is true that an attacker might have already executed a number of gadgets before being detected, but they must invoke system calls for any malicious actions. Therefore, it is a proved viable approach by using system call interposition as shown in many of the prior efforts. Then, the rest challenge becomes how we synchronize the execution at the syscall execution point with our CFI policy checking. The detailed design of how we perform our enforcement is presented in Section 4.2.

- **Using deep inspection when needed**. If there is no malicious attack, the TIP graph extracted from the PT packet trace faithfully represents the legitimate control flow path. Only when an unknown TIP sequence occurs, we invoke the slow and expensive PT packet parsing to parse the runtime control flow transfer, and compare with our CFI policy (discussed in Section 3.4) for the detection. Such a process also works similarly to network packet inspection. Most of the time, we directly just parse the headers of most packet, and only when unknown packet arrives, we parse its content. Therefore, we call this slow PT parsing and CFI verification process *deep inspection*.

  The purpose of designing this component is to speed up the performance of our CFI. As discussed earlier, one extreme case is to parse every PT packet to compute the dynamic CFG, and then inspect the binary code to check whether it conforms the legal control flow transfers. But this will render our system impractical because of its huge overhead. Our deep inspection solves this problem, and it only gets invoked when the CFI policy is incomplete. We will present in greater details of how we perform deep inspection in Section 4.1.

3.4 Our CFI Policy

Nearly all control flow hijacking attack targets indirect control flow transfers that occur at instruction `call`, `jmp`, `ret`, because direct control flow transfers are hardened in the read-only binary code, which typically cannot be altered by attackers. The key idea of CFI is to build a CFG, and then at runtime verify whether an indirect control flow transfer follows an edge in the CFG. Based on the edge direction in the CFG, we can classify them into:

- **Backward Edge** where a control flow transfers back to a node in the CFG. Such an edge exists because of the `ret` instruction, which transfers control flow back to the next instruction right after a call-site. A large amount of modern exploits (e.g., ROP) target manipulating the backward edge by controlling the return addresses. The primary goal of PT-CFI is to design a perfect policy that captures various backward edge violation attacks.

  In PT-CFI, indeed we can have a precise policy to detect the illegal backward edge thanks to the design of PT as well as our deep inspection capability. In particular, as acknowledged by many prior works (e.g., [10, 46]), using a shadow stack can really stop various ROP attacks because fundamentally attackers have to redirect the return address to some other locations, which will inevitably make the executed return address mismatching with the legal one. Since PT provides a complete trace of all indirect control flow transfers, we are able to build a perfect shadow stack based on the TIP traces and examine with the original binary code to detect ROPs. That is why we call PT-CFI backward-edge CFI since it has a complete protection for all backward edges.

- **Forward Edge** where a control flow transfers to a new target. There are two types of forward edges: one is caused by indirect `call` and the other is caused by indirect `jmp`. One of the biggest challenges in any CFI is how to get the legal forward edges. This is because when an indirect call or jump occurs, e.g., `call eax`, statically it is hard to know what the value of `eax` should be, since it may requires sophisticated point-to analysis but there is no sound and complete solution to this problem yet at binary code level. Therefore, CFI solutions often have to make approximations for forward edges. Unlike in backward-edge cases where we have a perfect CFI policy, we do not have a sound and complete solution to forward-edge yet. Therefore, we leave how to use PT for forward-edge CFI for future work.

3.5 Overview

An overview of our system is presented in Fig. 3. The goal of PT-CFI is to detect control flow hijacking by enforcing a lightweight CFI model. Unlike the traditional CFI where inlined reference monitoring is used, PT-CFI uses a separate dedicated monitoring process to detect any control flow violation of the monitored process.

There are four components of PT-CFI: When PT packets are generated for the monitored process, the first component *Packet Parsing* will parse each packet and generate the TIP sequences, which will be fed to our second component, *TIP Graph Matching*.

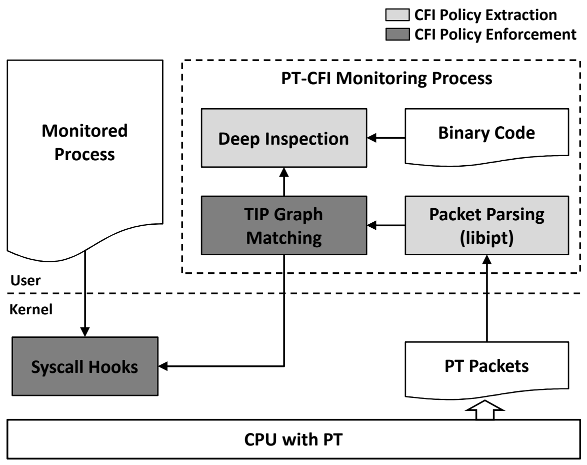

Figure 3: Architectural Overview of PT-CFI.

If a stream of TIP sequences matches with the TIP graph, execution continues. Otherwise, it invokes our third component *Deep Inspection* to decode the packets and construct the shadow stack. If the decoded return addresses all are matched in the shadow stack, the new TIP sequence will be considered legal, and added to our TIP graph; Otherwise, it will inform the the last component *Syscall Hooking* to terminate the execution of the monitored process since there is a control flow violation.

Scope, Assumptions, and Threat Model. We focus on protecting x86 ELF binaries in Linux platform, and we assume they are not obfuscated since we need to disassemble the binary code to decide the TIP type. We do not assume perfect disassembling since our disassembler can leverage the runtime information such as the exercised code address to disassemble the code.

We design PT-CFI to detect various return-based control flow hijacks, and we assume the OS kernel and the CPU hardware are not compromised during the attacks. In particular, we primarily focus on ROP attacks against user-space applications from remote adversaries who, by manipulating the input to the vulnerable applications, aim to hijack the control flow of the target applications. Again, we do not attempt to address attacks that use JOP or COP since these attacks violate the forward-edge CFI policy whereas PT-CFI is designed as a backward-edge CFI solution.

4. DETAILED DESIGN

In this section, we present the detailed design of each component of PT-CFI. Based on how a typical CFI system works, we first present how we extract the CFI policies by our *Packet Parsing* and *Deep Inspection* in Section 4.1, and then describe how we enforce the CFI policy by our *TIP Graph Matching* and *Syscall Hooking* in Section 4.2.

4.1 CFI Policy Extraction

4.1.1 Packet Parsing

The goal of packet parsing is to parse various types of PT packet (e.g., TNT, TIP, PUF), to facilitate the construction of legal TIP graph (if it has not been created yet or incomplete) and meanwhile send the parsed TIP packet generated by each indirect call, jump, or return, to our *TIP Graph Matching* component.

TIP Graph (TIP-G) Construction. The detection of control flow violation in our PT-CFI is based on the TIP-G, which is defined

Algorithm 1: TIP-G Construction

```
Input: TIP Packets: $P$ ($p_i \in P$)
Result: The desired TIP Graph $G$
1  begin
2      $G$.node ← $p_0$;
3      $G$.edge ← $\emptyset$;
4      $i \leftarrow 1$;
5      for each $p_i \in P$ do
6          $t \leftarrow$ GetTIPType($p_i$);
7          if $p_i \notin G.node$ then
8              $G$.node ← $G$.node $\cup\, p_i$;
9          end
10         if $<p_{i-1}, p_i, t> \notin G.edge$ then
11             $G$.edge ← $G$.edge $\cup <p_{i-1}, p_i, t>$;
12         end
13     end
14 end
```

$<N, E>$, where N denotes the set of nodes, each of which is the indexed by each unique TIP packet, and E denotes a set of directed edges. There is an edge from A to B if and only if right after the execution of an indirect control flow transfer A, it will execute the indirect control flow transfer B. That is, the edge captures the sequential execution of two indirect control flow transfers.

N is further divided into three different types based on the three different types of indirect control flow transfers. As such, we will have N_{ret} if the TIP node is corresponding to a `ret` instruction, N_{call} if it is an indirect `call`, and N_{jmp} if it is an indirect `jmp`. There are several ways to build our TIP-G. One intuitive approach is to statically disassemble the binary code to first build a CFG, and then only keep those indirect control flow transfer nodes in the CFG, since statically we cannot resolve the target address but we can leverage the runtime values to connect the missing edges and nodes. While we can use this approach, we realize in fact we do not have to disassemble the code and instead we can directly use the traced TIP packet on the fly to build our TIP-G.

More specifically, the construction of TIP-G is quite simple, as illustrated in Algorithm 1. Initially, the node of TIP-G will be just the first TIP packet (p_0), and the edge will be empty (line 2 and line 3). Whenever there is a new TIP packet p_i generated, we parse the type of p_i by a helper function GetTIPType, and the result could be N_{call}, N_{jmp}, and N_{ret} (line 6). Next, we insert p_i to the node of TIP-G if it has not been added yet (note that there will be only one instance of p_i in TIP-G). Meanwhile, we will also insert an edge $<p_{i-1}, p_i, t>$ with label t (which is acquired at line 6) from p_{i-1} to p_i if this edge has not been added before. The label of the edge indicates the three different indirect control flow transfers, which is important for PT-CFI to enforce the backward edges (essentially only the return control flow transfers). We keep iterating this process until all TIP packets have been processed (from line 5 to line 13). The resulting graph will be the desired TIP-G.

To build a complete TIP-G, we have two complementary approaches. One is to use *training*, and the other is to use the *deep inspection* discussed below. *Training* can be viewed as cached data, and when a cache misses we invoke the *deep inspection* for remediation. The reason why training works is because if we are running the protected software with all benign input, all p_i should be legitimate and we do not have to perform any deep inspection. Only when p_i is unknown (a new TIP node) or $<p_{i-1}, p_i, t>$ is unknown (a new edge), namely our CFI policy is incomplete, we invoke *deep inspection* to decide whether p_i or transition from p_{i-1} to p_i is legal.

It is important to note that there is no policy coverage issues in

PT-CFI even though we use a training approach. This is because our *deep inspection* component can always return a policy to determine whether an indirect control flow transfer is legal or not. We can run PT-CFI without any training by invoking *deep inspection* every time when we observe a p_i or a transition from p_{i-1} to p_i to decide the security policy. However, such an approach will be extremely slow. Therefore, training is just to improve the performance. Meanwhile, training can be performed offline and TIP-G can be reused across different machines for the same software.

4.1.2 Deep Inspection

When our CFI policy is incomplete, our *Deep Inspection* component will be invoked to disassemble the corresponding binary code based on the runtime information and determine the type of the TIP packet and also whether it is legal or not. Specifically, we must parse the PT packet to determine the type of the TIP packet that causes the deep inspection; namely, whether it is N_{call} or N_{jmp}, or N_{ret}. Since PT trace is a sequence of various PT packets and there is no information of the type of p_i, unless we correlate the virtual address with the binary code. A rigorous way of deciding the type of p_i needs to disassemble and walk through the code based on the closest known virtual address in the PT packets. Note that hardware will generate an alignment PT packet that contains the virtual address of the executed program code. Based on this known virtual address, an offline analysis is able to reconstruct the program behavior and precisely know the type of the TIP packet.

Since a program often contains loops, an observed TIP sequences may be observed again. To avoid parsing the same set of sequences again, we use a caching mechanism to avoid the re-disassembling and re-walking of the binary code in order to identify the corresponding TIP type. Then, at runtime, only unknown TIP will trigger the deep inspection. According to the specific unknown TIP packet, we will take different actions. If it is an indirect call or an indirect jmp, we will add them to our TIP-G because we do not have a perfect policy to precisely determine their legal target. If it is a return, we will build a shadow stack based on the PT traces. If the return address has matched in the shadow stack, this N_{ret} node and the corresponding edge will be added into our TIP-G. Otherwise, it is an attack, and our CFI enforcement will stop the execution of the monitored process. Note that for tail-recursive calls, we use the approaches in [35] to make sure our shadow stack is balanced.

4.2 CFI Policy Enforcement

Once a TIP-G is constructed (by the offline training), we can then use it to detect the control flow violations. The detection is done by our *TIP Graph Matching* component (which may also call our *Deep Inspection* discussed above). If it detects a real violation, it will inform our last component *Syscall Hooking* to terminate the execution of the monitored process.

4.2.1 TIP Graph Matching

When given a TIP packet p_j, the TIP-G matching becomes quite straightforward. Assume the CFI policy is complete, then at a give node n_i in TIP-G, there is only a set of allowed transition node, assume it is n_j, if p_j belongs to n_j, then there is no CFI violation. Otherwise, p_j is not known to TIP-G, and in this case, we will invoke our *Deep Inspection* component to decide whether p_j is a legal transfer. If not, an attack is detected. If it is not a CFI violation, p_j will be added to TIP-G.

More specifically, to detect whether p_j violates CFI during the deep inspection, we will use its type (recall all the edge has a type in our TIP-G). If it is N_{ret} for our aimed back-edge CFI enforcement, our deep inspection will check with the shadow stack built based on the PT traces. If the returning location is not the legal return address, it is an attack. Otherwise, this missing legal CFI transition will be added to TIP-G. If it is N_{call} or N_{jmp}, we do not have a precise policy and we will allow the execution by adding the missing node and edge in our TIP-G. Therefore, PT-CFI will not detect any JOP or COP attacks as discussed in Section 3.

4.2.2 Syscall Hooking

Once we have detected there is a control flow violation, we must terminate the execution of the running process. To make PT-CFI get an control of the monitored process execution, we take a system call interposition approach, which has been widely used by many other systems such as kBouncer and ROPecker. Basically, we selectively hook a number of security sensitive system calls including `execve`, `write`, `mprotect`, `munmap`, `clone`, `fork`, `open`, `close` and `exit_group`. We introduce a lock at the entry point of these system calls. It will be only unlocked by our monitoring process to continue its execution, when there is no violation of CFI given the current parsed TIP packets. Otherwise, the monitored process will be terminated at the execution of these system calls. Since syscall hooking is a standard approach, we omit its technical details here.

5. IMPLEMENTATION

We have implemented PT-CFI. We implemented the kernel component *Syscall Hooking* by using a kernel extension, and implemented the rest component, especially *Packet Parsing* by borrowing a large amount of code from a user level program `perf`, the first and the industry strength tool for Intel PT.

More specifically, we use *Syscall Hooking* to create a sandboxed execution environment to the monitored program by hooking only a set of sensitive system calls in the system call table. Upon entering these system calls, PT-CFI will consult with the user level component *TIP-G Matching* to ensure the backward-edge CFI of the monitored program. The monitored program will be paused on the sensitive syscall until there is no attack detected by our *TIP-G Matching*.

To control PT execution and parse PT packets at runtime, we extended `perf`, which is available since Linux kernel 4.3. Since PT is a system wide hardware level feature, we need to configure it to trace only user level program by setting the corresponding MSRs. To this end, we use the available `sys_perf_event_open` to initialize the PT hardware, and this system call allows us to specify process ID and CPU Core number as well as other filtering for Intel PT hardware.

Since PT packet parsing is a slow process, we use a thread pool to handle PT packets in parallel in order to improve the performance of the whole system. In addition, to better dispatcher threads, we bind the monitored program and our monitoring program to different cores and bind our working threads to the rest cores. For example, for an 8 core computers, we bind the monitored program to core 0 and the monitoring program to core 1 and working threads to core 2 to 7. We use function `sched_setaffinity` to bind processes to a specific core and function `pthread_setaffinity_np` to bind threads to cores.

During the *Deep Inspection*, PT-CFI needs to disassemble the binary code to determine the type of TIP packet, and check the shadow stack. To do that, PT-CFI first scans the decoder synchronization packet, usually a PSB at the beginning of the PT buffers, then it can find a TIP.PGE packet indicating the full starting instruction pointer (IP). With the full IP and PT packets, PT-CFI can follow the execution path to disassemble the binary code and tell the type of the TIP packet. For the shadow stack, it is incremen-

Table 2: **Experimental Result with SPEC2006 CPUINT benchmark**

Program Name	$\|N\|$	$\|E\|$	#Syscall	PT Packet Size (MB)	Training Time (ms)
400.perlbench	3486	7294	160	1.27	3408.6
401.bzip2	483	677	77	210.94	1021.4
403.gcc	20233	71545	238	498.06	1115.8
429.mcf	456	642	357	158.5	1373.2
433.milc	1290	1920	6511	584.92	4519
445.gobmk	1361	2740	365	10.22	62.6
456.hmmer	964	1662	88	20.64	1356.4
458.sjeng	960	1911	608	498.05	3009.2
462.libquantum	850	1323	96	9.48	33
464.h264ref	1696	2940	541	443.97	15464.2
470.lbm	640	850	711	54.87	1235.4
482.sphinx3	2823	4231	3168	468.07	1049.2
AVG	2936.84	8144.59	1076.67	246.59	2804

tally built based on the traces from the beginning of the execution to the suspicious point. If the TIP packet is of return type, and if the shadow stack is not matched, then an attack is detected.

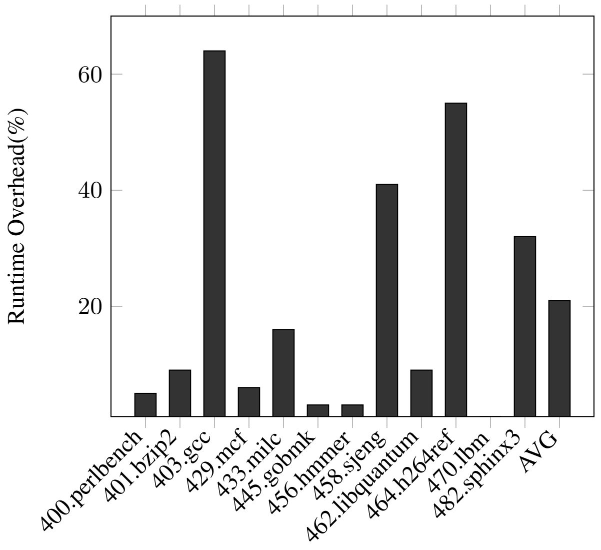

Figure 4: SPEC CPU2006 Benchmark Overhead

6. EVALUATION

In this section, we evaluate the effectiveness of PT-CFI for detecting the ROP attacks, and its efficiency in terms of runtime performance. Our testing platform is a desktop computer with an Intel i7-6700K Skylake 4.00 GHz CPU and 8G memory, running Ubuntu 14.04.1 LTS with Linux kernel 4.3.0.

6.1 Security Evaluation

By design, PT-CFI is able to detect all ROP attacks, thanks to our deep inspection and the shadow stack constructed from PT trace. To measure how PT-CFI really works in detecting the control flow violations, we use both a contrived attack against our running example in Fig. 1 and a real attack against Nginx HTTP daemon (*i.e.*, Nginx-1.4.0) to evaluate the capability of PT-CFI in detecting the control flow violations. Such a security evaluation methodology has been used in many other ROP defenses such as ROPecker.

More specifically, to construct our attack payload, we leveraged a widely used ROP gadget searching and linking tool ROPgadget to analyze both the executable and linked libraries of these two programs. To make our gadget construction easier, we disabled the ASLR protection. We successfully constructed two ROP payloads and both spawned a shell without the protection from PT-CFI.

Then we applied PT-CFI to protect them. We first trained each of them: using the three benign inputs to our running example, and using 1,000 requests with varied length generated from `Httperf` [1] for the Nginx daemon, respectively. Then we injected the constructed attack payload to these two victim programs. As expected, both of them triggered our deep inspection, which took a 0.42 seconds for our running example, and 0.63 seconds for Nginx to report that a ROP attack is detected. The reason of why Nginx took more time is because more packets need to be used to disassemble the code and walk through the binary to build the shadow stack, which is a slow process. Also note that no false positive or false negatives occurred in these two security tests.

6.2 Performance Evaluation

To evaluate the performance overhead of our system, we tested with a set of SPEC2006 benchmark programs and the Nginx HTTP daemon.

SPEC2006 CPUINT. We used the 12 CPUINT benchmark programs from SPEC2006 in our evaluation. We compiled them with the default configuration by using gcc-4.8.4, then we executed and trained each of them with the default configured input to get their corresponding TIP-G. The detailed result for our training phase is presented in Table 2.

Specifically, it took a variety of amount time to train each of the benchmark, as reported in the last column in Table 2. To train the program, we run each SPEC CPUINT program with their default configured input. Some of them (*e.g.* 464.h264ref) took 15.46 seconds, and some of them (*e.g.* 462.libquantum) only costed 0.03 seconds. On average, it took 2.80 seconds to train each benchmark. During the training phase, we also observed 1076.67 syscalls on average (the 3rd column), and 2936.84 of nodes (2nd column) and 8144.59 edges (3rd column) in their TIP-G. The PT packet size is reported in the 4th column, and on average we collected 246.58 MB traced PT packet during each run of these programs.

Next, we applied the obtained the TIP-G for the CFI enforcement. We run these benchmark again with their default input. The purpose of this experiment is to measure how slow our TIP-G matching is when used in real software. The performance overhead for this experiment is shown in Fig. 4. While some programs such as `gcc` have high overhead (up to 65%), most of them has less than 10%. On average, it is 21% for these CPUINT benchmarks. The reason of why `gcc` has high overhead is that it has many more packets to process, and more nodes and edges in the TIP-G for the matching than that of others.

Network Daemon. While SPEC benchmark can provide an estimation of how slow our PT-CFI is for real software, in practice we believe PT-CFI will be mostly used to protect the network daemons. To understand the performance impact of our approach for network daemons, we again measured the latency and throughput of the HTTP daemon Nginx we tested in our security evaluation.

Note that we have to configure Nginx in the mode of running one worker process (which means only one Nginx thread is executing), since tracing multiple-threads simultaneously is not supported in Intel Process Trace (fundamentally PT hardware only uses CR3 to differentiate the packet and the packet generated by each thread will be merged as one process' PT packet). To train Nginx, again

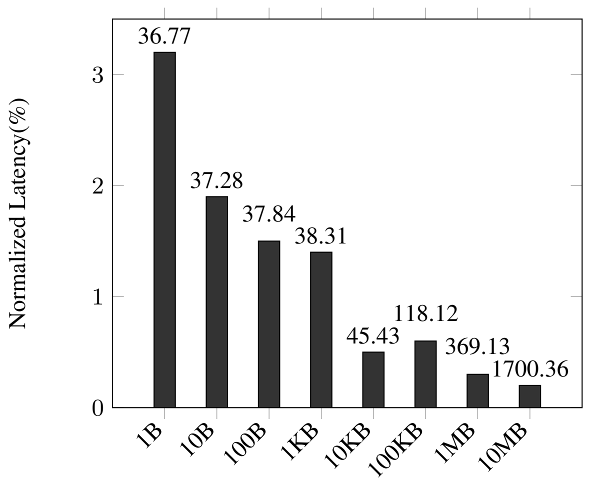

Figure 5: Nginx File Download Latency

we used the same configuration as in our security evaluation by generating 1,000 client request messages with `Httperf`. During this training phase, we observed 391 nodes, and 1213 edges in our TIP-G. To monitor the Nginx, we first get the PID of the worker process, then we attach PT-CFI to this process.

- **Latency**. To evaluate the latency of Nginx with PT-CFI, we use Apache HTTP server benchmarking tool (`ab`) to send 10,000 requests to Nginx to download different sizes of file, from one byte to 10M bytes. The normalized latency compared without PT-CFI protection is reported in Figure 5. We can see that for small size downloaded packets, the latency is slight larger than those bigger size ones, though they all appear to be quite negligible (less than 5%). Note that we also reported the absolute download time on top of each bar with unit milliseconds in this figure. For instance, when the request file is 1 byte, it took 36.77 milliseconds to download this file.

- **Throughput**. We also evaluate the file download throughput of Nginx. We use the tool `Httperf` to generate different numbers of concurrent requests to access the same 300K-bytes file, then report the throughput of Nginx without and with PT-CFI in Figure 6. We can see that with different numbers of concurrent requests per seconds (x-axis), there is negligible impact in the replies per seconds (y-axis) (almost the same height of bar in both cases). When the concurrent number of requests exceeds 380 (which appears to be the maximum number of requests Nginx can handle simultaneous), the throughput goes down as without PT-CFI protection.

Overall, we can observe for network daemons such as Nginx, our system does not have noticable performance impact against normal users (less than 5% latency and negligiable throughput impact). We believe this represents a practical CFI approach we have aimed to achieve.

7. DISCUSSION AND FUTURE WORK

While PT-CFI has made a first step of using Intel PT to build a practical CFI model, it is still not perfect and has a number of limitations. In this section, we examine these limitations and outline our future work.

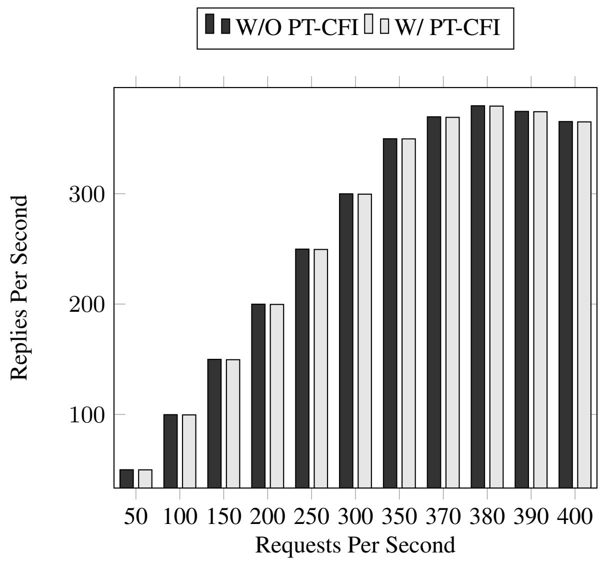

Figure 6: Nginx Throughput Impact

First, PT-CFI has a clear policy for all `ret` based exploits. Unlike existing ROP defenses such as ROPecker that uses heuristics, PT-CFI can precisely detect all ROP instances thanks for the deep inspection. However, we currently do not have policies for all those forward-edges and we allow the monitored process continue the execution when encountering these `call` and `jmp` TIPs. As such, JOP or COP attacks are still possible.

Meanwhile, we have to note that there is no perfect solution to resolve the forward-edges at binary code level because the challenges from point-to analysis, though there are solutions at source code level such as the forward-edge CFI [44]. There could exist some approaches that use value-set analysis to approximate the possible indirect target, or use some loose security policy such as allowing the indirect call target always starts from the entry point of a function as in BinCFI. We plan to investigate how to address these forward edge issues in one of our future works.

Finally, there are also a number of avenues to optimize the performance of PT-CFI, especially its deep inspection component. For instance, to differentiate the type of each TIP packet, we have to perform disassembling of the protected binary code whenever we encounter an unknown TIP. This disassembling process can be optimized by considering the history of the disassembling process, namely, if we have already disassembled some code, we do not have to disassemble it again. While we have already explored using the cache to optimize the re-disassembling, we have not systematically investigated the cache size factor yet. We plan to explore how to optimize PT-CFI further in our another future work.

8. CONCLUSION

We have presented PT-CFI, a new backward-edge CFI model based on a recently introduced Intel hardware feature—Processor Trace. Designed primarily for offline software debugging, PT offers the capability of tracing the entire control flow of a running program. In this paper, we have presented the design, implementation, and evaluation of using PT for security with a new practical CFI model for native COTS binary based on the trace from PT. We have addressed a number of technical challenges such as making sure the control flow policy is complete, making PT enforce our CFI policy, and balancing the performance overhea, by exploring the intrinsic tracing property inside PT with a system synchronization primitive and a deep inspection capability. We have implemented PT-CFI and tested with both SPEC2006 and a popular network daemon. Experimental results show that PT-CFI only introduces small overhead for the monitored program with the capability of detecting all ROP attacks.

9. REFERENCES

[1] Httperf, http://www.labs.hpe.com/research/linux/httperf/.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353. ACM, 2005.

[3] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[4] N. Balakrishnan, T. Bytheway, L. Carata, O. R. A. Chick, J. Snee, S. Akoush, R. Sohan, M. Seltzer, and A. Hopper. Recent advances in computer architecture: The opportunities and challenges for provenance. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15)*, Edinburgh, Scotland, July 2015. USENIX Association.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.

[6] 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 '11, pages 353–362. ACM, 2011.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[8] E. Bosman and H. Bos. Framing signals — return to portable exploits. (working title, subject to change.). In *Security & Privacy (Oakland)*, San Jose, CA, USA, May 2014. IEEE.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., Aug. 2015. USENIX Association.

[10] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[11] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 385–399, Berkeley, CA, USA, 2014. USENIX Association.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[13] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS '09, pages 163–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. USENIX Association, 2005.

[15] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attack. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSSâĂŹ14, 2014.

[16] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[17] 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 *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[18] 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 Conference on Security Symposium*, SEC'14, Berkeley, CA, USA, 2014. USENIX Association.

[19] 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 '11, pages 40–51. ACM, 2011.

[20] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[21] 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 *23rd USENIX Security Symposium*, pages 417–432, San Diego, CA, Aug. 2014. USENIX Association.

[22] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, pages 177–192, Washington, D.C., Aug. 2015. USENIX Association.

[23] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy*. IEEE, 2016.

[24] 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 '15, pages 344–360, New York, NY, USA, 2015. ACM.

[25] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 147–163, Broomfield, CO, Oct. 2014. USENIX Association.

[26] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek. Towards a practical solution to detect code reuse attacks on arm mobile devices. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '15. ACM, 2015.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic

instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[28] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 941–951. ACM, 2015.

[29] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, NDSSâĂŹ15, 2015.

[30] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 577–587. ACM, 2014.

[31] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1317–1328. ACM, 2014.

[32] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926. ACM, 2015.

[33] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[34] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA, 2013. USENIX Association.

[35] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *DIMVA*, 2015.

[36] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.

[37] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses*. Springer International Publishing, 2014.

[38] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, pages 25–41, 2011.

[39] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, pages 552–61. ACM Press, Oct. 2007.

[40] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[41] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62. IEEE Computer Society, 2013.

[42] P. Team. Pax address space layout randomization (aslr). 2003.

[43] J. Thalheim, P. Bhatotia, and C. Fetzer. Inspector: A data provenance library for multithreaded programs, 2016. https://arxiv.org/abs/1605.00498.

[44] 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 *23rd USENIX Security Symposium*, pages 941–955, San Diego, CA, Aug. 2014. USENIX Association.

[45] 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 '15, pages 927–940. ACM, 2015.

[46] V. van der Veen, E. Goktas, 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 (Oakland)*, San Jose, CA, USA, May 2016. IEEE.

[47] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 331–340. ACM, 2015.

[48] R. Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[49] 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 '12, Washington, DC, USA, 2012. IEEE Computer Society.

[50] P. Yuan, Q. Zeng, and X. Ding. Hardware-assisted fine-grained code-reuse attack detection. In *Proceedings of 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAIDâĂŹ15. Springer International Publishing, 2015.

[51] 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 *2013 IEEE Symposium on Security and Privacy*, pages 559–573, May 2013.

[52] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pages 337–352. USENIX, 2013.