Efficient Protection of Path-Sensitive Control Security

Ren Ding and Chenxiong Qian, *Georgia Tech;* Chengyu Song, *UC Riverside;*
Bill Harris, Taesoo Kim, and Wenke Lee, *Georgia Tech*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Efficient Protection of Path-Sensitive Control Security

Ren Ding* Georgia Tech
Chenxiong Qian* Georgia Tech
Chengyu Song UC Riverside
William Harris Georgia Tech
Taesoo Kim Georgia Tech
Wenke Lee Georgia Tech
* *Equal contribution joint first authors*

Abstract

Control-Flow Integrity (CFI), as a means to prevent control-flow hijacking attacks, enforces that each instruction transfers control to an address in a set of valid targets. The security guarantee of CFI thus depends on the definition of valid targets, which conventionally are defined as the result of a static analysis. Unfortunately, previous research has demonstrated that such a definition, and thus any implementation that enforces it, still allows practical control-flow attacks.

In this work, we present a path-sensitive variation of CFI that utilizes runtime path-sensitive point-to analysis to compute the legitimate control transfer targets. We have designed and implemented a runtime environment, PITTYPAT, that enforces path-sensitive CFI efficiently by combining commodity, low-overhead hardware monitoring and a novel runtime points-to analysis. Our formal analysis and empirical evaluation demonstrate that, compared to CFI based on static analysis, PITTYPAT ensures that applications satisfy stronger security guarantees, with acceptable overhead for security-critical contexts.

1 Introduction

Attacks that compromise the control-flow of a program, such as return-oriented programming [33], have critical consequences for the security of a computer system. Control-Flow Integrity (CFI) [1] has been proposed as a restriction on the control-flow transfers that a program should be allowed to take at runtime, with the goals of both ruling out control-flow hijacking attacks and being enforced efficiently.

A CFI implementation can be modeled as program rewriter that (1) before a target program P is executed, determines feasible targets for each indirect control transfer location in P, typically done by performing an analysis that computes a sound over-approximation of the set of all memory cells that may be stored in each code pointer (i.e., a static *points-to analysis* [2, 34]). The rewriter then (2) rewrites P to check at runtime before performing each indirect control transfer that the target is allowed by the static analysis performed in step (1).

A significant body of work [1, 21, 41] has introduced approaches to implement step (2) for a variety of execution platforms and perform it more efficiently. Unfortunately, the end-to-end security guarantees of such approaches are founded on the assumption that if an attacker can only cause a program to execute control branches determined to be feasible by step (1), then critical application security will be preserved. However, recent work has introduced new attacks that demonstrate that such an assumption does not hold in practice [5, 12, 32]. The limitations of existing CFI solutions in blocking such attacks are inherent to *any* defense that uses static points-to information computed per control location in a program. Currently, if a developer wants to ensure that a program only chooses valid control targets, they must resort to ensure that the program satisfies *data integrity*, a significantly stronger property whose enforcement typically incurs prohibitively large overhead and/or has deployment issues, such as requiring the protected program being recompiled together with all dependent libraries and cannot be applied to programs that perform particular combinations of memory operations [17, 22–24].

In this work, we propose a novel, *path-sensitive* variation of CFI that is stronger than conventional CFI (i.e., CFI that relies on static points-to analysis). A program satisfies path-sensitive CFI if each control transfer taken by the program is consistent with the program's *entire* executed control path. Path-sensitive CFI is a stronger security property than conventional CFI, both in principle and in practice. However, because it does not place any requirements on the correctness of data operations, which happen much more frequently, it can be enforced much more efficiently than data integrity. To demonstrate this, we present a runtime environment, named PITTYPAT, that enforces path-sensitive efficiently using a combination of commodity, low-overhead hardware-based monitoring and a new runtime points-to analysis.

PITTYPAT addressed two key challenges in building an efficient path-sensitive CFI solution. The first challenge is how to efficiently collect the path information about a program's execution so as to perform the analysis and determine if the program has taken only valid control targets. Collecting such information is not straightforward for dynamic analysis. An approach that maintains information inside the same process address space of the monitored program (e.g., [17]) must carefully protect the information; otherwise it would be vulnerable to attacks [11]. On the other hand, an approach that maintains information in a separate process address space must efficiently replicate genuine and sufficient data from the monitored program.

The second key challenge is how to use collected information to precisely and efficiently compute the points-to relationship. Niu et al. [26] have proposed leveraging execution history to dynamically activate control transfer targets. However, since the activation is still performed over the statically computed control-flow graph, its accuracy can degrade to the same as pure static-analysis-based approach. We compare PITTYPAT to such approaches in detail in §6.

PITTYPAT applies two key techniques in addressing these two challenges. First, PITTYPAT uses an event-driven kernel module that collects all chosen control-transfer targets from the *Processor Tracing (PT)* feature available on recent Intel processors [31]. PT is a hardware feature that efficiently records conditional and indirect branches taken by a program. While PT was originally introduced to enable detailed debugging through complete tracing, our work demonstrates that it can also be applied as an effective tool for performing precise, efficient program analysis for security.

The second technique is an abstract-interpretation-based incremental points-to analysis. Our analysis embodies two key innovations. First, raw PT trace is highly compressed (see §3 for details). As a result, reconstructing the control-flow (i.e., source address to destination address) itself is time consuming and previous work has utilized multiple threads to reduce the decoding latency [13]. Our insight to solve this problem is to sync up our analysis with the execution, so that our analysis only needs to know what basic blocks being executed, not the control transfer history. Therefore, we can directly map the PT trace to basic blocks using the control-flow graph (CFG). The second optimization is based on the observation that static points-to analyses collect and solve a system of constraints over *all pairs* of pointer variables in the program [2, 15]. While this approach has good throughput, it introduces unacceptable latency for online analysis. At the same time, to enforce CFI, we only need to know the points-to information of code pointers. Based on this observation, our analysis eagerly evaluates control relevant points-to constraints as they are generated.

We implemented PITTYPAT as an instrumenting compiler for the LLVM compiler [20] and a tool for Linux; the instrumenting compiler is an artifact of the current version of our prototype: PITTYPAT does not fundamentally rely on the ability to compile and instrument a target program. To evaluate PITTYPAT, we used it to enforce path-sensitive CFI for a set of security benchmarks developed in independent work. The results demonstrate that PITTYPAT can detect recent attacks on the control flow of benign benchmarks [5], as well as subversion of control flow in programs explicitly crafted to contain control vulnerabilities that are difficult to detect [12, 32]. In common cases where CFI allows a program to choose from tens of control transfer targets, PITTYPAT typically determines that only a *single* target is valid, based on the program's executed control path. On even compute-intensive benchmarks, PITTYPAT incurs reasonable performance overhead: a geometric mean of 12.73% over all SPEC CPU2006 benchmarks, whereas techniques that enforce data integrity incur 122.60%.

The rest of this paper is organized as follows. In §2, we illustrate PITTYPAT by example. In §3, we review previous work on which PITTYPAT is based. In §4, we present the security guarantees that PITTYPAT establishes, and describe the design of PITTYPAT. In §5, we describe the implementation of PITTYPAT in detail. In §6, we present an empirical evaluation of PITTYPAT. In §7, we compare PITTYPAT to related work. In §8, we conclude our work.

2 Overview

In this section, we present PITTYPAT by introducing a running example. In §2.1, we present a program `dispatch` that contains a control-flow vulnerability. In §2.2, we use `dispatch` to illustrate that any defense that enforces conventional CFI allows effective attacks on control-flow. In §2.3, we illustrate that *path-sensitive CFI* enforced by PITTYPAT does not allow the attack introduced in §2.2. In §2.4, we illustrate how PITTYPAT enforces path-sensitive CFI.

2.1 Subverting control flow

Figure 1 contains a C program, named `dispatch`, that we will use to illustrate PITTYPAT. `dispatch` declares a pointer `handler` (line `L7`) to a function that takes an argument of a struct `request` (defined at line `L1`–`L4`), which has two fields: `auth_user` represents a user's identity, and `args` stores the arguments. `dispatch` contains a loop (line `L10`–`L23`) that continuously accepts requests from users,

```
1  struct request {
2   int auth_user;
3   char args[100];
4  };
5
6  void dispatch() {
7   void (*handler)(struct request *) = 0;
8   struct request req;
9
10  while(1) {
11    // parse the next request
12    parse_request(&req);
13    if (req.auth_user == ADMIN) {
14      handler = priv;
15    } else {
16      handler = unpriv;
17      // NOTE. buffer overflow, which can overwrite
18      //  the handler variable
19      strip_args(req.args);
20    }
21    // invoke the hanlder
22    handler(&req);
23  }
24 }
```

Figure 1: A motivating example that illustrates the advantages of control-path validity.

and calls `parse_request` (line 12) to parse the next request. If the request is an administrator (line L13), the function pointer `handler` will be assigned with `priv`. Otherwise, `handler` is assigned to `unpriv` (line L16), and `dispatch` will call `strip_args` (line L19) to strip the request's arguments. At last, `dispatch` calls `handler` to perform relevant behaviors.

However, the procedure `strip_args` contains a buffer-overflow vulnerability, which allows an attacker with control over input to `strip_args` to potentially subvert the control flow of a run of `dispatch` by using well-known techniques [28]. In particular, the attacker can provide inputs that overwrite memory outside of the fixed-size buffer pointed to by `req.args` in order to overwrite the address stored in `handler` to be the address of a function of their choosing, such as `execve`.

2.2 Limitations of existing CFI

Protecting `dispatch` so that it satisfies conventional control-flow integrity (CFI) [1] does not provide strong end-to-end security guarantees. An implementation of CFI attempts to protect a given program P in two steps. In the first step, the CFI implementation computes possible targets of each indirect control transfer in P by running a flow-sensitive points-to analysis[1] [2, 15, 34]. Such an approach, when protecting `dispatch`, would determine that when the execution reaches each of the following control locations L, the variable `handler` may store the following addresses $p(\texttt{L})$:

$$p(\texttt{L7}) = \{0\} \qquad p(\texttt{L14}) = \{\texttt{priv}\}$$
$$p(\texttt{L16}) = \{\texttt{unpriv}\} \qquad p(\texttt{L22}) = \{\texttt{priv}, \texttt{unpriv}\}$$

While flow-sensitive points-to analysis may implement various algorithms, the key property of each such analysis is that it computes points-to information per control location. If there is any run of the program that may reach control location L with a pointer variable `p` storing a particular address `a`, then the result of the points-to analysis must reflect that `p` may point to `a` at L. In the case of `dispatch`, any flow-sensitive points-to analysis can only determine that at line L22, `handler` may point to either `priv` or `unpriv`.

After computing points-to sets p for program P, the second step of a CFI implementation rewrites P so that at each indirect control-transfer instruction in each run, the rewritten P can only transfer control to a control location that is a points-to target in the target register according to p. Various implementations have been proposed for encoding points-to sets and validating control transfers efficiently [1, 9, 41].

However, all such schemes are fundamentally limited by the fact that they can only validate if a transfer target is allowed by checking its membership in a flow-sensitive points-to set, computed per control location. `dispatch` and the points-to sets p illustrate a case in which any such scheme *must* allow an attacker to subvert control flow. In particular, an attacker can send a request with the identity of anonymous user. When `dispatch` accepts such a request, it will store `unpriv` in `handler`, and then strip the arguments. The attacker can provide arguments crafted to overwrite `handler` to store `priv`, and allow execution to continue. When `dispatch` calls the function stored in `handler` (line L22), it will attempt to transfer control to `priv`, a member of the points-to set for L22. Thus, `dispatch` rewritten to enforce CFI must allow the call. Let the sequence of key control locations visited in the above attack be denoted $p_0 = [\texttt{L7}, \texttt{L16}, \texttt{L22}]$.

Although PathArmor [37] enforces context-sensitive CFI by inspecting the history of branches taken at runtime before allowing the monitored execution to perform a security-sensitive operation, it decides to allow execution to continue if the path contains a sequence of control transfers that are feasible according to a static, flow-sensitive points-to analysis computed before the program is run. As a result, PathArmor is susceptible to a similar attack.

Per-input CFI (denoted π-CFI) [26] avoids some of the vulnerabilities in CFI inherent to its use of flow-sensitive points-to sets, such as the vulnerability described above for `dispatch`. π-CFI updates the set of valid targets of control transfers of each instruction dynamically, based on operations performed during the current program execution. For example, π-CFI only allows a program to

[1]Some implementations of CFI [25, 41, 42] use a type-based alias analysis to compute valid targets, but such approaches are even less precise.

perform an indirect call to a function whose address was taken during an earlier program operation. In particular, if `dispatch` were rewritten to enforce π-CFI, then it would block the attack described above: in the execution of π-CFI described, the only instruction that takes the address of `handler` (line `L14`) is never executed, but the indirect call at `L22` uses `priv` as the target of an indirect call.

However, in order for π-CFI to enforce per-input CFI efficiently, it updates valid points-to targets dynamically using simple, approximate heuristics, rather than a precise program analysis that accurately models the semantics of instructions executed. For example, if a function f appears in the static points-to set of a given control location L and has its address taken at any point in an execution, then f remains in the points-to set of L for the rest of the execution, even if f is no longer a valid target as the result of program operations executed later. In the case of `dispatch`, once `dispatch` takes the address of `priv`, `priv` remains in the points-to set of control location `L22` for the remainder of the execution.

An attacker can thus subvert the control flow of `dispatch` rewritten to enforce π-CFI by performing the following steps. **(1)** An administrator sends a request, which causes `dispatch` to store `priv` in `handler`, call it, and complete an iteration of its loop. **(2)** The attacker sends an anonymous request, which causes `dispatch` to set `unpriv` in `handler`. **(3)** The attacker provides arguments that, when handled by `strip_args`, overwrite the address in `handler` to be `priv`, which causes `dispatch` to call `priv` with arguments provided by the attacker.

Because `priv` will be enabled as a control target as a result of the operations performed in step (1), `priv` will be a valid transfer target at line `L22` in step (3). Thus, the attacker will successfully subvert control flow. Let the key control locations in the control path along which the above attack is performed be denoted $p_1 = [\texttt{L7}, \texttt{L14}, \texttt{L22}, \texttt{L16}, \texttt{L22}]$.

2.3 Path-sensitive CFI

In this paper, we introduce a path-sensitive version of CFI that addresses the limitations of conventional CFI illustrated in §2.2. A program satisfies path-sensitive CFI if at each indirect control transfer, the program only transfers control to an instruction address that is in the points-to set of the target register according to a points-to analysis of the whole executed control *path*.

`dispatch` rewritten to satisfy path-sensitive CFI would successfully detect the attacks given in §2.2 on existing CFI. One collection of valid points-to sets for `handler` for each control location in subpath p_0 (§2.2) are the following:

$$(\texttt{L7}, \{0\}), (\texttt{16}, \{\texttt{unpriv}\}), (\texttt{L22}, \{\texttt{unpriv}\})$$

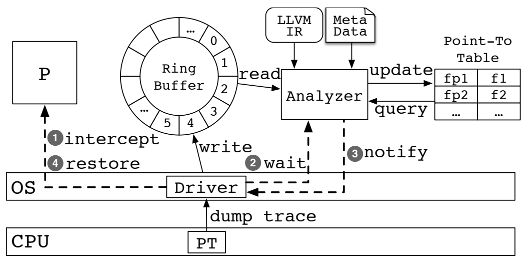


Figure 2: The architecture of PITTYPAT. *P* denotes a target program. The *analyzer* and *driver* modules of PITTYPAT are described in §2.4.

When execution reaches `L22`, `priv` is not in the points-to set of `handler`, and the program halts.

Furthermore, `dispatch` rewritten to satisfy path-sensitive CFI would block the attack given in §2.2 on π-CFI. One collection of valid points-to sets for `handler` for each control location in subpath p_1 are the following:

$$(\texttt{L7}, \{0\}) \quad (\texttt{L14}, \{\texttt{priv}\}) \quad (\texttt{L22}, \{\texttt{priv}\})$$
$$(\texttt{L16}, \{\texttt{unpriv}\}) \quad (\texttt{L22}, \{\texttt{unpriv}\})$$

When execution reaches `L22` in the second iteration of the loop in `dispatch`, `priv` is not in the points-to set of `handler`, and the program determines that the control-flow has been subverted.

2.4 Enforcing path-sensitive CFI efficiently

The points-to sets for control paths considered in §2.3 illustrate that if a program can be rewritten to satisfy path-sensitive CFI, it can potentially satisfy a strong security guarantee. However, ensuring that a program satisfies path-sensitive CFI is non-trivial, because the program must be extended to dynamically compute the results of sophisticated semantic constraints [2] over the exact control path that it has executed.

A key contribution of our work is the design of a runtime environment, PITTYPAT, that enforces path-sensitive CFI efficiently. PITTYPAT's architecture is depicted in Figure 2. For program *P*, the state and code of PITTYPAT consist of the following modules, which execute concurrently: **(1)** a user-space process in which *P* executes, **(2)** a user-space *analysis module* that maintains points-to information for the control-path executed by *P*, and **(3)** a kernel-space *driver* that sends control branches taken by *P* to the analyzer and validates system calls invoked by *P* using the analyzer's results.

Before a program *P* is monitored, the analysis module is given (1) an intermediate representation of *P* and (2) meta data including a map from each instruction address in the binary representation of *P* to the instruction in the intermediate representation of *P*. We believe that it would also be feasible to implement PITTYPAT to protect a program given only as a binary, given that the analyzer module only must perform points-to analysis on the sequence of executed instructions, as opposed to inferring the program's complete control-flow graph.

As P executes a sequence of binary instructions, the driver module copies the targets of control branches taken by P from PT's storage to a ring buffer shared with the analyzer. PT's storage is privileged: it can only be written by hardware and flushed by privileged code, and cannot be tampered with by P or any other malicious user-space process. The analyzer module reads taken branches from the ring buffer, uses them to reconstruct the sequence of IR instructions executed by P since the last branch received, and updates the points-to information in a table that it maintains for P's current state by running a points-to analysis on the reconstructed sequence.

When P invokes a system call, the driver first intercepts P (❶), while waiting for the analyzer module to determine in parallel if P has taken a valid sequence of control targets over the *entire* execution up to the current invocation (❷ and ❸). The analyzer validates the invocation only if P has taken a valid sequence, and the driver allows execution of P to continue only if the invocation is validated (❹).

There are two key challenges we must address to make PittyPat efficient. First, trace information generated by PT is highly compressed; e.g., for each conditional branch that a program executes, PT provides only a single bit denoting the value of the condition tested in the branch. Therefore additional post-processing is necessary to recover transfer targets from such information. The approach used by the `perf` tool of Linux is to parse the next branch instruction, extract the offset information, then calculate the target by adding the offset (if the branch is taken) or the length of instruction (if branch is not taken). However, because parsing x86 instructions is non-trivial, such an approach is too slow to reconstruct a path online.

Our insight to solve this problem is that, to reconstruct the executed path, an analysis only needs to know the basic blocks executed. We have applied this insight by designing the analysis to maintain the current basic block executed by the program. The analysis can maintain such information using the compressed information that PT provides. E.g., if PT provides only a bit denoting the value of a condition tested in a branch, then the analysis inspects the conditional branch at the end of the maintained block, and from the branch, updates its information about the current block executed.

The second key challenge in designing PittyPat is to design a points-to analysis that can compute accurate points-to information while imposing sufficiently low overhead. Precise points-to analyses solve a system of constraints over all pairs of pointer variables in the program [2, 15]; solving such constraints uses a significant amount of time that is often acceptable in the context of an offline static analysis, but would impose unacceptable overhead if used by PittyPat's online analysis process. Other analyses bound analysis time to be nearly linear with increasing number of pointer variables, but generate results that are often too imprecise to provide strong security guarantees if used to enforce CFI [34].

Packet	Description
TIP.PGE	IP at which the tracing begin
TIP.PGD	Marks the ending of tracing
TNT	Taken/non-taken decisions of conditional branches
TIP	Target addresses of indirect branches
FUP	The source addresses of asynchronous events

Table 1: Control-relevant trace packets from Intel PT.

To address the limitations of conventional points-to analysis, we have designed an online points-to analysis that achieves the precision of precise analysis at high performance. The analysis eagerly evaluates control relevant points-to constraints as they are generated, while updating the points-to relations table used for future control transfer validation. The analysis enables PittyPat, when analyzing runs of `dispatch` that execute paths p_0 and p_1, to compute the accurate points-to information given in §2.3. On practical benchmarks, it allows significantly smaller sets of control targets to be taken at each control branch, and detects attacks on control flow not detected by state-of-the-art defenses. Combined with our efficient path-reconstruction process, it also enables PittyPat to execute with an average of 12.73% overhead (geometric mean) on even compute-intensive benchmarks, such as SPEC CPU2006 (see §6).

3 Background

3.1 Intel Processor Trace

Intel PT is a commodity, low-overhead hardware designed for debugging by collecting *complete* execution traces of monitored programs. PT captures information about program execution on each hardware thread using dedicated hardware facilities so that after execution completes, the captured trace data can be reconstructed to represent the exact program flow.

The captured control flow information from PT is presented in encoded data packets. The control relevant packet types are shown in Table 1. PT records the beginning and the end of tracing through `TIP.PGE` and `TIP.PGD` packets, respectively. Because the recorded control flow needs to be highly compressed in order to achieve the efficiency, PT employs several techniques to achieve this goal. In particular, PT only records the taken/non-taken decision of each conditional branches through `TNT`, along with the target of each indirect branches through `TIP`. A direct branch does not trigger a PT packet because the control target of a direct branch is fixed.

Besides the limited packet types necessary for recovering *complete* execution traces, PT also adopts compact packet format to reduce the data throughput aggressively. For instance, `TNT` packets use one bit to indicate the direction of each conditional branches. `TIP` packets, on the other hand, contain compressed target address if the upper address bytes match the previous address logged. Thus on average, PT tracing incurs less than 5% overhead [13].

When configured appropriately, PT monitors a single program as well as its descendants based on CR3 filtering, and outputs all collected packets to physical memory allocated by its kernel driver. In the current implementation of PITTYPAT, a ring buffer is allocated so that it can be reused throughout execution. The details of its implementation are described in §5.1.

3.2 Conventional CFI

A control analysis, given program P, computes a sound over-approximation of the instruction pointers that may be stored in each pointer when P executes each instruction. An abstract domain D [8] consists of a set of abstract states, a concretization relation from abstract states to the program states that they represent, and for each program instruction $\texttt{i}$, an abstract transformer $\tau_D[\texttt{i}] : D \rightarrow D$ that describes how each abstract state is updated by a program. Each abstract domain defines a transition relation ρ_D of steps valid according to D. In particular, for each instruction $\texttt{i}$, domain element D, and all states σ and σ', if σ represented by D and σ' is represented by $\tau_D[\texttt{i}](D)$, then $(\sigma, \texttt{i}, \sigma') \in \rho_D$. A control-analysis domain D is an abstract domain extended with a relation from each abstract domain element and instruction pointer to code pointers in states represented by D.

A valid flow-sensitive description in D of a program P is a map from each program point in P to an element in D that is consistent with the semantics of program instructions. There is always a most-precise valid flow-sensitive description in D, denoted $\mu[D]$.

Definition 1 *For control domain D, program P satisfies* (conventional) CFI *modulo D if, in each run of P, at each indirect branch point* $\texttt{L}$, *P transfers control to a control target in* $\mu[D](\texttt{L})$.

We provide a complete formal definition of conventional CFI in §C.1.

An analysis that computes such a description is a *control analysis*. Control analyses conventionally are implemented as *points-to* analyses, such as Andersen's analysis [2] or Steensgard's analysis [34].

4 Design

A program P satisfies path-sensitive CFI under control domain D if each step of P is valid according to D (as described in §3.2).

Definition 2 *For control domain D, program P satisfies* path-sensitive CFI *modulo D if, in each run of P consisting of states* $\sigma_0, \ldots, \sigma_n$, *for each* $0 \leq j < n$ *where* σ_j *steps to* σ_{j+1} *on instruction* $\texttt{i}$, $(\sigma_j, \texttt{i}, \sigma_{j+1}) \in \rho_D$.

A formal definition of path-sensitive CFI, along with results establishing that path-sensitive CFI is strictly stronger than conventional CFI, are given in §C.2.

PITTYPAT enforces path-sensitive CFI by maintaining a shadow execution/analysis that only examines control relevant data, while running concurrently with the monitored process. Using the complete traces reconstructed from Intel PT, only control-relevant data are computed and maintained as points-to relations throughout the execution, using an online points-to analysis. Analyzing only control-relevant data satisfies the need to validate control-transfer targets but significantly optimizes the analysis, because only parts of the program will be examined in the shadow execution/analysis. Such an analysis, along with the low overhead incurred by commodity hardware, allow PITTYPAT to achieve path-sensitive CFI with practical runtime overhead.

The architecture of PITTYPAT is depicted in §2.4, Figure 2. PITTYPAT consists of two modules. The first module executes a given program P in a designated monitor process and collects the targets of control transfers taken by P. We describe the operation of this module in §4.1 and give the details of its implementation in §5.1. The second module receives control-branch targets taken by P from the first module, reconstructs the control path executed by P from the received targets, and performs a points-to analysis along the reconstructed control path of P. We describe the operation of the analysis module in §4.2 and describe details of its implementation in §5.2.

4.1 Sharing taken branches efficiently

PITTYPAT uses the PT extension for Intel processors [31] to collect the control branches taken by P. A naive implementation of PITTYPAT would receive from the monitoring module the complete target address of each branch taken by P in encoded packets and decode the traces offline for analysis. PITTYPAT, given only Boolean flags from PT, decodes complete branch targets on the fly.

To do so, PITTYPAT maintains a copy of the current control location of P. For example, in Figure 1, when `dispatch` steps through the path [`L10`,`L16`,`L22`], the relevant PT trace contains only two `TNT` packets and one `TIP` packet. A `TNT` packet is a two-bit stream: 10. The first bit, 1, represents the conditional branch at `L10` is taken (i.e., the execution enters into the loop). The second bit, 0, indicates the conditional branch at `L13` is not taken, and the executed location is now in the else branch. The `TIP` packet contains the address of function `unpriv`, which shows an indirect jump to `unpriv`.

PittyPat uses the Linux `perf` infrastructure to extract the execution trace of *P*. In particular, PittyPat uses the `perf` kernel driver to **(1)** allocate a ring buffer shared by the hardware and itself and **(2)** mark the process in which the target program executes (and any descendant process and thread) as traced so as to enable tracing when context switching into a descendant and disable tracing when context switching out of a descendant. The driver then transfers the recorded PT packets, together with thread ID and process ID, to the analyzer module through the shared buffer. This sharing mechanism has proved to be efficient on all performance benchmarks on which we evaluated PittyPat, typically incurring less than 5% overhead.

PittyPat intercepts the execution of a program at security-sensitive system calls in the kernel and does not allow the program to proceed until the analyzer validates all control branches taken by the program. The list of intercepted system calls can be easily configured; the current implementation checks `write`, `mmap`, `mprotect`, `mremap`, `sendmsg`, `sendto`, `execve`, `remap_file_pages`, `sendmmsg`, and `execveat`. The above system calls are intercepted because they can either disable DEP/W$\oplus$X, directly execute an unintended command, write to files on the local host, or send traffic over a network.

4.2 Online points-to analysis

The analyzer module executes in a process distinct from the process in which the monitored process executes. Before monitoring a run of the program, the analyzer is given the monitored program's LLVM IR and meta information about mapping between IR and binary code. At runtime, the analyzer receives the next control-transfer target taken by the protected program from the monitor module, and either chooses to raise an alarm signaling that the control transfer taken would violate path-sensitive CFI, or updates its state and allows the original program to take its next step of execution.

The updated states contain two components: (1) the callstack of instructions being executed (i.e., the `pc`'s) and (2) points-to relations over models of memory cells that are control relevant only. The online points-to analysis addresses the limitations of conventional points-to analyses. In particular, it reasons precisely about the calling context of the monitored program by maintaining a stack of register frames. It avoids maintaining constraints over pairs of pointer variables by eagerly evaluating the sets of cells and instruction addresses that may be stored in each register and cell. It updates this information efficiently in response to program actions by performing updates on a single register frame and removing register frames when variables leave scope on return from a function call.

In general, a program may store function pointers in arbitrarily, dynamically allocated data structures before eventually loading the pointer and using it as the target of an indirect control transfer. If the analyzer were to maintain precise information about the points-to relation of all heap cells, then it would maintain a large amount of information never used and incur a significant cost to performance. We have significantly optimized PittyPat by performing aggressive analyses of a given program *P* offline, before monitoring the execution of *P* on a given input. PittyPat runs an analyzer developed in previous work on *code-pointer integrity* (CPI) [17] to collect a sound over-approximation of the instructions in a program that may affect a code pointer used as the target of a control transfer. At runtime, the analyzer only analyzes instructions that are control relevant as determined by its offline phase.

A program may contain many functions that perform no operations on data structures that indirectly contain code pointers, and do not call any functions that perform such operations. We optimized PittyPat by applying an offline analysis based on a sound approximation of the program's call graph to identify all such functions. At runtime, PittyPat only analyzes functions that may indirectly perform relevant operations.

To illustrate the analyzer's workflow, consider the execution path [`L10`, `L12`, `L16`, `19`, `L22`] in Figure 1 as an example. Initially, the analyzer knows that the current instruction being executed is `L10`, and the points-to table is empty. The analyzer then receives a taken `TNT` packet, and so it updates the `pc` to `L12`, which calls a non-sensitive function `parse_request`. However instead of tracing instructions in `parse_request`, the analyzer waits until receiving a `TIP` packet signaling the return from `parse_request` before continue its analysis. Next, it updates the `pc` to `L16` after receiving a non-taken `TNT` packet, which indicates that the else branch is taken. Here, the analyzer updates the points-to table to allow `handler` to point to `unpriv` when it handles `L16`. Because the program also calls a non-sensitive function at `L19`, the analyzer waits again and updates the `pc` to `L22` only after receiving another `TIP` packet. Finally, at `L22`, the analyzer waits for a `TIP` packet at the indirect call, and checks whether the target address collected by the monitor module is consistent with the value pointed by `handler` in the points-to table. In this case, if the address in the received `TIP` packet is not `unpriv`, the analyzer throws an alarm.

We have described the analyzer as validating taken control branches and eagerly throwing alarms when it detects an incorrect branch in order to simplify its description.

The actual implementation of the analyzer only provides such an alarm in response to a request from PittyPat's kernel module when a monitored process attempts to invoke a system call, as described in §5.1.

5 Implementation

5.1 Monitor module

PittyPat controls the Intel PT extension and collects an execution trace from a monitored program by adapting the Linux v4.4 `perf` infrastructure. Because `perf` was originally designed to aid debugging, the original version provided with Linux 4.4 only supports decoding and processing traces offline. In the original implementation, the `perf` kernel module continuously outputs packets of PT trace information to the file system in user space as a log file to be consumed later by a userspace program. Such a mechanism obviously cannot be used directly within PittyPat, which must share branch information at a speed that allows it to be run as an online monitor.

We modified the kernel module of `perf`, which begins and ends collection of control targets taken after setting a target process to trace, allocates a ring buffer in which it shares control branches taken with the analyzer, and monitors the amount of space remaining in the shared buffer. The module also notifies the analyzer when taken branches are available in its buffer, along with how many chosen control targets are available. The notification mechanism reuses the pseudo-file interface of the `perf` kernel module. The analyzer creates one thread to wait (i.e., poll) on this file handler for new trace data. Once woken up by the kernel, it fetches branches from the shared ring buffer with minimal latency.

System calls are intercepted by a modified version of the system-call mechanism provided by the Linux kernel. When the monitored process is created, it—along with each of its sub-processes and threads created later—is flagged with a true value in a `PT_CPV` field of its `task_struct` in kernel space. When the kernel receives a request for a system call, the kernel checks if the requesting process is flagged. If so, the kernel inspects the value in register `rax` to determine if it belongs to the configured list of marked system calls as described in §4.1. The interception mechanism is implemented as a semaphore, which blocks the system call from executing further code in kernel space until the analyzer validates all branches taken by the monitored process and signals the kernel.

The driver module and modifications to the kernel consist of approximately 400 lines of C code.

5.2 Analyzer module

PittyPat's analyzer module is implemented as two core components. The first component consists of a LLVM compiler pass, implemented in 500 lines, that inserts an instruction at the beginning of each basic block before the IR is translated to binary instructions. Such instructions are used to generate a map from binary basic blocks to LLVM IR basic blocks. Thus when PittyPat receives a `TNT` packet for certain conditional branch, it knows the corresponding IR basic block that is the target of the control transfer. The inserted instructions are removed when generating binary instructions; therefore no extra overhead is introduced to the running program.

The second component, implemented in 5,800 lines C++ code, performs a path-sensitive points-to analysis over the control path taken by the monitored process, and raises an error if the monitored process ever attempts to transfer control to a branch not allowed by path-sensitive CFI. Although the analysis inspects only low-level code, it directly addresses several challenges in analyzing code compiled from high-level languages. First, to analyze exception-handling by a C++ program, which unwinds stack frames without explicit calls to return instructions, the analyzer simply consumes the received `TNT` packets generated when the program compares the exception type and updates the `pc` to the relevant exception handler.

To analyze a dynamic dispatch performed by a C++ program, the analyzer uses its points-to analysis to determine the set of possible objects that contain the vtable at each dynamic-dispatch callsite. The analyzer validates the dispatch if the requested control target stored in a given `TIP` packet is one of the members of the object from which the call target is loaded. At each call to `setjmp`, the analyzer stores all possible `setjmp` buffer cells that may be used as arguments to `setjmp`, along with the instruction pointer at which `setjmp` is called, in the top stack frame. At each call to `longjmp`, the analyzer inspects the target `T` of the indirect call and unwinds its stack until it finds a frame in which `setjmp` was called at `T`, with the argument buffer of `longjmp` may have been the buffer passed as an argument to `setjmp`.

6 Evaluation

We performed an empirical evaluation to answer the following experimental questions. **(1)** Are benign applications transformed to satisfy path-sensitive CFI less susceptible to an attack that subverts their control security? **(2)** Do applications that are explicitly written to perform malicious actions that satisfy weaker versions of CFI fail to satisfy path-sensitive CFI? **(3)** Can PittyPat enforce path-sensitive CFI efficiently?

To answer these questions, we used PittyPat to enforce path-sensitive CFI on a set of benchmark programs and workloads, including both standard benign applications and applications written explicitly to conceal malicious behavior from conventional CFI frameworks. In summary, our results indicate that path-sensitive CFI provides a stronger security guarantee than state-of-the-art CFI mechanisms, and that PITTYPAT can enforce path-sensitive CFI while incurring overhead that is acceptable in security-critical contexts.

6.1 Methodology

We collected a set of benchmarks, each described in detail in §6.2. We compiled each benchmark with LLVM 3.6.0, and ran them on a set of standard workloads. During each run of the benchmark, we measured the time used by the program to process the workload. If a program contained a known vulnerability that subverted conventional CFI, then we ran the program on inputs that triggered such a vulnerability as well, and observed if PITTYPAT determined that control-flow was subverted along the execution. Over a separate run, at each control branch taken by the program, we measured the size of the points-to set of the register that stored the target of the control transfer.

We then built each benchmark to run under a state-of-the-art CFI framework implemented in previous work, π-CFI [26]. While π-CFI validates control targets per control location, it instruments a subject program so that control edges of the program are disabled by default, and are only enabled as the program executes particular triggering actions (e.g., a function can only be called after its address is taken). It thus allows sets of transfer targets that are no larger than those allowed by conventional implementations of CFI, and are often significantly smaller [26]. For each benchmark program and workload, we observed whether π-CFI determined that the control-flow integrity of the program was subverted while executing the workload and measured the runtime of the program while executed under π-CFI. We compared PITTYPAT to π-CFI because it is the framework most similar to PITTYPAT in concept: it validates control-transfer targets based not only on the results of a static points-to analysis, but collecting information about the program's dynamic trace.

6.2 Benchmarks

To evaluate the ability of PITTYPAT to protect long-running, benign applications, and to evaluate the overhead that it incurs at runtime, we evaluated it on the SPEC CPU2006 benchmark suite, which consists of 16[2] C/C++ benchmarks. We ran each benchmark three times over its provided reference workload. For each run, we measured the runtime overhead imposed by PITTYPAT and the number of control targets allowed at each indirect control transfer, including both indirect calls and returns. We also evaluated PITTYPAT on the NGINX server—a common performance macro benchmark, configured to run with multiple processes.

[2]We don't include `447.dealII`, `471.omnetpp`, and `483.xalancbmk` because their LLVM IR cannot be completely mapped to the binary code.

To evaluate PITTYPAT's ability to enforce end-to-end control security, we evaluated it on a set of programs explicitly crafted to contain control vulnerabilities, both as analysis benchmarks and in order to mount attacks on critical applications. In particular, we evaluated PITTYPAT on programs in the RIPE benchmark suite [39], each of which contains various vulnerabilities that can be exploited to subvert correct control flow (e.g. *Return-Oriented Programming* (ROP) or *Jump-oriented Programming* (JOP)). We compiled 264 of its benchmarks in our x64 Linux test environment and evaluated PITTYPAT on each. We also evaluated PITTYPAT on a program that implements a proof-of-concept COOP attack [32], a novel class of attacks on the control-flow of programs written in object-oriented languages that has been used to successfully mount attacks on the Internet Explorer and Firefox browsers. We determined if PITTYPAT could block the attack that the program attempted to perform.

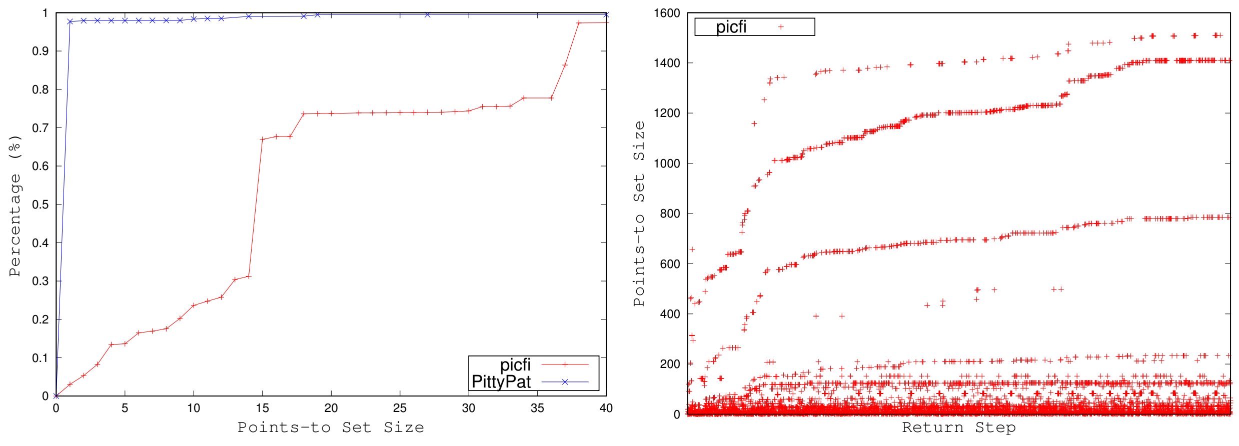
6.3 Results

6.3.1 Protecting benign applications

Figure 3 contains plots of the control-transfer targets allowed by π-CFI and PITTYPAT over runs of example benchmarks selected from §6.2. In the plots, each point on the x-axis corresponds to an indirect control transfer in the run. The corresponding value on the y-axis contains the number of control targets allowed for the transfer.

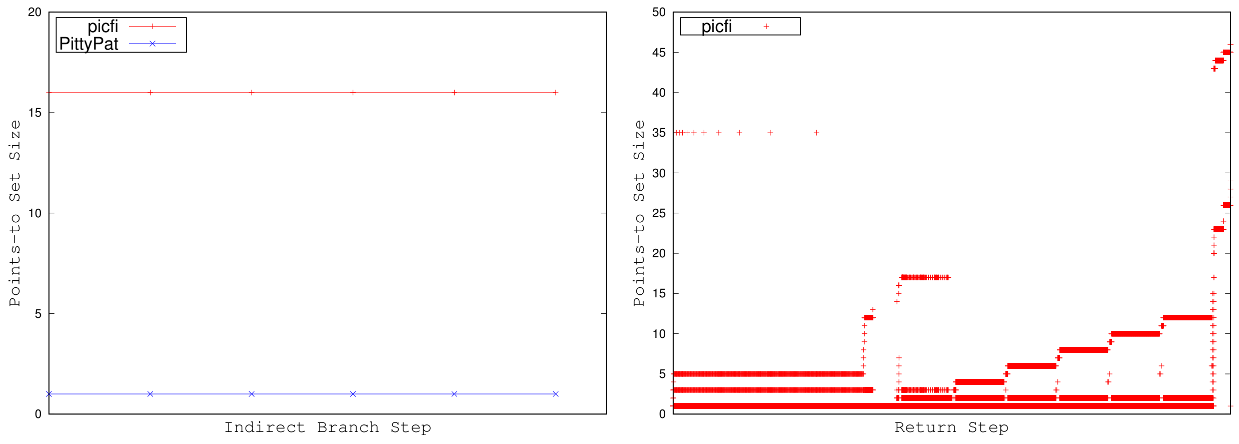
Previous work on CFI typically reports the average indirect-target reduction (AIR) of a CFI implementation; we computed the AIR of PITTYPAT. However, the resulting data does not clearly illustrate the difference between PITTYPAT and alternative approaches, because all achieve a reduction in branch targets greater than 99% out of all branch targets in the program. This is consistent with issues with AIR as a metric established in previous work [4]. Figure 3, instead, provides the absolute magnitudes of points-to sets at each indirect control transfer over an execution.

Figure 3a contains a Cumulative Distribution Graph (CDF) of all points-to sets at *forward* (i.e., jumps and calls) indirect control transfers of size no greater than 40 when running `403.gcc` under π-CFI and PITTYPAT. We used a CDF over a portion of the points-to sets in order to display the difference between the two approaches in the presence of a small number of large points-to sets,



(a) Partial CDF of allowed targets on forward edges taken by `403.gcc`.

(b) π-CFI points-to set of backward edges taken by `403.gcc`.



(c) π-CFI and PittyPat points-to sets for forward edges taken by `444.namd`.

(d) π-CFI points-to sets for backward edges taken by `444.namd`.

Figure 3: Control-transfer targets allowed by π-CFI and PittyPat over `403.gcc` and `444.namd`.

explained below. Figure 3a shows that PittyPat can consistently maintain significantly smaller points-to sets for forward edges than that of π-CFI, leading to a stronger security guarantee. Figure 3a indicates that when protecting practical programs, an approach such as π-CFI that validates per location allows a significant number of transfer targets at each indirect callsite, even using dynamic information. In comparison, PittyPat uses the entire history of branches taken to determine that at the vast majority of callsites, only a *single* address is a valid target. The difference in the number of allowed targets can be explained by the different heuristics adopted in π-CFI, which monotonically accumulates allowed points-to targets without any disabling schemes once targets are taken, and the precise, context-sensitive points-to analysis implemented in PittyPat. Similar difference between π-CFI and PittyPat can also be found in all other C benchmarks from SPEC CPU2006.

For the remaining 4% of transfers not included in Figure 3a, both π-CFI and PittyPat allowed up to 218 transfer targets; for each callsite, PittyPat allowed no more targets than π-CFI. The targets at such callsites are loaded from vectors and arrays of function pointers, which PittyPat's current points-to analysis does not reason about precisely. It is possible that future work on a points-to analysis specifically designed for reasoning precisely about such data structures over a single path of execution—a context not introduced by any previous work on program analysis for security—could produce significantly smaller points-to sets.

A similar difference between π-CFI and PittyPat is demonstrated by the number of transfer targets allowed for other benchmarks. In particular, Figure 3c contains similar data for the `444.namd` benchmark. `444.namd`, a C++ program, contains many calls to functions loaded from vtables, a source of imprecision for implementations of CFI that can be exploited by attackers [32]. PittyPat allows a *single* transfer target for *all* forward edges as a result of its online points-to analysis. The difference between π-CFI and PittyPat are also found for other C++ benchmarks, such as `450.soplex`, `453.povray` and `473.astar`.

π-CFI and PittyPat consistently allow dramatically different numbers of transfer targets for return instructions. While monitoring `403.gcc`, π-CFI allows, for some return instructions, over $1,400$ return targets (Figure 3b). While monitoring `444.namd`, π-CFI allows, for some return instructions, more than 46 transfer targets (Figure 3d). Because PittyPat maintains a stack of points-to information during its analysis, it will *always* allow only a single transfer target for each return instruction, over all programs and workloads. PittyPat thus significantly improves defense against ROP attacks, which are still one of the most popular attacks software.

6.3.2 Mitigating malicious applications

To determine if PittyPat can detect common attacks on control, we used it to monitor selected RIPE benchmarks [39]. For each of the 264 benchmarks that ran in our experimental setup, PittyPat was able to successfully detect attacks on the benchmark's control security.

We constructed a proof-of-concept program vulnerable to a COOP [32] attack that corrupts virtual-function pointers to perform a sequence of method calls not possible by a well-defined run of the program. In Figure 4, the program defines two derived classes of `SchoolMember` (line `L1`–`L4`), `Student` (line `L5`–`L10`) and `Teacher` (line `L11`–`L16`). Both `Student` and `Teacher` define their own implementation of the virtual function `registration()` (lines `L7`–`9` and `L13`–`15`, respectively). `set_buf()` (line `L17`–`L21`) allocates a buffer `buf` on the stack of size 4 (line `L18`), but does not bound the amount of data that it reads into `buf` (line `L20`). The `main` function (line `L22`–`L37`) constructs instances of `Student` and `Teacher` (lines `L23` and `L24`, respectively), and stores them in `SchoolMember` pointers (lines `L26` and `27` respectively). `main` then calls the `registration()` method of each instance (lines `L29`–`L31`), reads input from a user by calling `set_buf()` (line `L33`), and calls `Student::registration()` a second time (line `L35`). A malicious user can subvert control flow of the program by exploiting the buffer overflow vulnerability in `set_buf` to overwrite the vptr of `Student` to that of `Teacher` and run `Teacher::registration()` at line `L35`.

Previous work introducing COOP attacks [32] established such an attack cannot be detected by CFI. π-CFI was not able to detect an attack on the above program because it allows a dynamic method as a call target once its address is taken. However, PittyPat detected the attack because its analyzer module accurately models the effect of each load of a function pointer used to implement the dynamic calls over the program's well-defined runs.

```
1  class SchoolMember {
2    public:
3      virtual void registration(void){}
4  };
5  class Student : public SchoolMember{
6    public:
7      void registration(void){
8        cout << "I am a Student\n";
9      }
10 };
11 class Teacher : public SchoolMember{
12   public:
13     void registration(void){
14       cout << "This is sensitive!\n";
15     }
16 };
17 void set_buf(void){
18   char buf[4];
19   //change vptr to that of Teacher's sensitive func
20   gets(buf);
21 }
22 int main(int argc, char *argv[]){
23   Student st;
24   Teacher te;
25   SchoolMember *member_1, *member_2;
26   member_1 = &te;
27   member_2 = &st;
28   //Teacher calling its virtual functions
29   member_1->registration();
30   //Student calling its virtual functions
31   member_2->registration();
32   //buffer overflow to overwrite the vptr
33   set_buf();
34   //Student calling its virtual functions again
35   member_2->registration();
36   return 0;
37 }
```

Figure 4: A program vulnerable to a COOP attack.

6.3.3 Enforcing path-sensitive CFI efficiently

Table 2 contains measurements of our experiments that evaluate performance of PittyPat when monitoring benchmarks from SPEC CPU2006 and NGINX server, along with the performance results replicated from the paper that presented π-CFI [26]. A key feature observable from Table 2 is that PittyPat induces overhead that is consistently larger than, but often comparable to, the overhead induced by π-CFI. The results show that PittyPat incurs a geometric mean of 12.73% overhead across the 16 SPEC CPU2006 benchmarks, along with a 11.9% increased response time for NGINX server over one million requests with concurrency level of 50. Overhead of sharing branch targets taken is consistently less than 5%. The remaining overhead, incurred by the analysis module, is proportional to the number of memory operations (e.g., loads, stores, and copies) performed on memory cells that transitively point to a target of an indirect call, as well as the number of child processes/threads spawned during execution of multi-process/-threading benchmarks.

Another key observation from Table 2 is that PittyPat induces much smaller overhead than CETS [23] and SoftBound [22], which can only be applied to a small selection of the SPEC CPU2006 benchmarks. CETS provides

Program Features		Payload Features		π-CFI Features		PITTYPAT Features		CETS+SB Features	
Name	**KLoC**	**Exp**	**Tm (sec)**	**Alarm**	**Overhd (%)**	**Alarm**	**Overhd (%)**	**Alarm**	**Overhd (%)**
`400.perlbench`	128	No	332	No	8.7%	No	47.3%	Yes	–
`401.bzip2`	6	No	317	No	1.3%	No	17.7%	No	91.4%
`403.gcc`	383	No	179	No	6.2%	No	34.1%	Yes	–
`429.mcf`	2	No	211	No	4.3%	No	32.2%	Yes	–
`433.milc`	10	No	514	No	1.9%	No	1.8%	Yes	–
`444.namd`	4	No	556	No	-0.3%	No	28.8%	Yes	–
`445.gobmk`	158	No	328	No	11.4%	No	4.0%	Yes	–
`450.soplex`	28	No	167	No	-1.1%	No	27.5%	Yes	–
`453.povray`	79	No	100	No	11.9%	No	16.0%	Yes	–
`456.hmmer`	21	No	258	No	0.2%	No	20.2%	Yes	–
`458.sjeng`	11	No	359	No	8.5%	No	6.7%	No	80.1%
`462.libquantum`	3	No	234	No	-1.5%	No	14.1%	Yes	–
`464.h264ref`	36	No	339	No	8.0%	No	11.8%	No	251.7%
`470.lbm`	1	No	429	No	1.4%	No	0.7%	Yes	–
`473.astar`	4	No	289	No	2.2%	No	22.5%	Yes	–
`482.sphinx3`	13	No	338	No	1.7%	No	16.0%	Yes	–
Geo. Mean	15	–	285	–	3.30%	–	12.73%	–	122.60%
`nginx-1.10.2`	122	No	25.41	No	2.7%	No	11.9%	Yes	–

Table 2: "Name" contains the name of the benchmark. "KLoC" contains the number of lines of code in the benchmark. Under "Payload Features," "Exp" shows if the benchmark contains an exploit and "Tm (sec)" contains the amount of time used by the program, when given the payload. Under "π-CFI Featues", "PITTYPAT Features," and "CETS+SB Features," "Alarm" contains a flag denoting if a given framework determined that the payload was an attack and aborted; "Overhd (%)" contains the time taken by the framework, expressed as the ratio over the baseline time.

temporal memory safety and SoftBound provides spatial memory safety; both enforce full data integrity for C benchmarks, which entails control security. However, both approaches induce significant overhead, and cannot be applied to programs that perform particular combinations of memory-unsafe operation [17]. Our results thus indicate a continuous tradeoff between security and performance among exisiting CFI solution, PITTYPAT, and data protection. PITTYPAT offers control security that is close to ideal, i.e. what would result from data integrity, but with a small percentage of the overhead of data-integrity protection.

7 Related Work

The original work on CFI [1] defined control-flow integrity in terms of the results of a static, flow-sensitive points-to analysis. A significant body of work has adapted the original definition for complex language features and developed sophisticated implementations that enforce it. While CFI is conventionally enforced by validating the target of a control transfer before the transfer, control-flow locking [3] validates the target after the transfer to enable more efficient use of system caches. Compact Control Flow Integrity and Randomization (CCFIR) [41] optimizes the performance of validating a transfer target by randomizing the layout of allowable transfer targets at each jump. Opaque CFI (O-CFI) [21] ensures that an attacker who can inspect the rewritten code cannot learn additional information about the targets of control jumps that are admitted as valid by the rewritten code.

All of the above approaches enforce security defined by the results of a flow-sensitive points-to analysis; previous work has produced attacks [5, 12, 32] that are allowed by any approach that relies on such information. PITTYPAT is distinct from all of the above approaches because it computes and uses the results of a points-to analysis computed for the exact control path executed. As a result, it successfully detects known attacks, such as COOP [32] (see §6.3.2).

Previous work has explored the tradeoffs of implementing CFI at distinct points in a program's lifecycle. CF restrictor [30] performs CFI analysis and instrumentation completely at the source level in an instrumenting compiler, and further work developed CFI integrated into production compilers [36]. BinCFI [42] implements CFI without access to the program source, but only access to a stripped binary. Modular CFI [25] implements CFI for programs constructed from separate compilation units. Unlike each of the above approaches, PITTYPAT consists of a background process that performs an online analysis of the program path executed.

Recent work on *control-flow bending* has established limitations on the security of any framework that enforces only conventional CFI [5], and proposes that future work explore CFI frameworks that validate branch targets using an auxiliary structure, such as a shadow stack. The conclusions of work on control-flow bending are strongly consistent with the motivation of PITTYPAT: the key contribution of PITTYPAT is that it enforces path-sensitive CFI, provably stronger than conventional CFI, and does so not only by maintaining a shadow stack of points-to information, but by validating the targets of indirect branches using *path*-sensitive points-to analysis. Per-input CFI (π-CFI) [26] only enables control transfers to targets that are enabled depending on previous operations taken by a program in a given run; §6 contains a detailed comparison of π-CFI to PITTYPAT.

Several implementations of CFI use hardware features that efficiently record control targets chosen by a program. CFIMon [40] collects the transfer targets chosen by the program from the processor's branch tracing store, and validates the chosen target against the results of a flow-sensitive points-to analysis. Previous work has also proposed customized architectures with extended instruction sets that directly implement primitive operations required in order to enforce CFI [9]. Such approaches are thus distinct from our approach for the same reason as all approaches that use the results of a flow-sensitive analysis. kBouncer [29] interposes when a program attempts to execute a system call and inspects the Last Branch Record (LBR) provided on Intel processors to detect patterns of transfer targets that indicate an ROP attack. ROPecker [7] similarly interposes at key security events and inspects the LBR, but combines information from inspecting the history of chosen branches with a forward analysis. PathArmor [37] interposes key system calls, collects the last transfer targets collected in the LBR, and determines if there is a feasible path through the program's control-flow graph that reaches each transfer target. Further work [6] introduced counterattacks against such defenses that exploit the fact that each of the defenses only inspects the LBR to analyze a bounded number of transfer targets chosen immediately before a system call.

The above approaches are similar to PITTYPAT in that they inspect the results of hardware features that collect some subset of the control targets taken by a program at runtime. However, they are all distinct from PITTYPAT because PITTYPAT uses hardware features to maintain accurate points-to information by inspecting *all* branch targets chosen by a program over its execution. Recent work has proposed approaches that leverage Intel PT. Most such approaches use PT to debug programs [16, 35], whereas PITTYPAT uses PT to protect their control security. Some approaches [13, 14, 19] use PT to enforce that an application satisfies CFI as defined by a static flow-sensitive analysis; PITTYPAT uses PT to ensure that a program satisfies a stronger, path-sensitive variation of CFI.

Points-to analysis is a classic problem in static program analysis, with different approaches that achieve distinct tradeoffs in either higher precision [2] or scalability [34]. Points-to analyses are characterized on multiple dimensions, including flow-sensitivity [2, 34] and context-sensitivity [10, 18, 27, 38, 43]. However, a key property of all such analyses is that they are performed statically, and thus compute information either per program point or per group of stack configurations [15]. PITTYPAT uses a points-to analysis to compute points-to information based on the exact program path executed. As a result, PITTYPAT does not merge points-to information over multiple paths that reach a given control location or stack configuration, which heavily influenced the design of the novel points-to analysis that it uses. Recent work [17] has introduced *Code-Pointer Integrity (CPI)*, which protects the integrity of all addresses that indirectly affect the value of a function pointer used as the target of an indirect branch. A key finding of the original work on CPI is that CPI is relatively expensive to enforce for programs that contain a large number of code pointers, such as binaries compiled from programs in object-oriented languages. As a result, CPI was proposed along with *code-pointer separation* (CPS), in which the values of code pointers are protected, but pointers to cells containing code pointers are left unprotected. Subsequent work on *counterfeit object-oriented programming* [32] demonstrated that CPS is insufficiently strong to block code-reuse attacks on object-oriented programs.

PITTYPAT, along with all approaches for enforcing various versions of CFI, differs fundamentally from CPI in that it does not attempt to protect any segment of a program's data at runtime. Instead, PITTYPAT validates candidate targets of indirect control transfers based only on the history of control branches taken. CPI and PITTYPAT have complementary strengths and should be applied in complementary security settings. In particular, CPI often incurs slightly lower overhead, but can only be applied in scenarios in which the source code of the entire program to be protected is available to be analyzed and instrumented. Such conditions are not satisfied in cases in which a program relies on large, untrusted third-party or shared libraries. PITTYPAT can potentially incur larger performance overhead than CPI. However, because it performs an points-to analysis that can be easily run on sequences of low-level instructions, it can be applied to protect program modules that are only available as binaries. It also need not instrument any code of a protected application. Our current implementation of PITTYPAT uses an analysis proposed in the work on CPI only to optimize the points-to analysis performed at runtime to validate branch targets.

8 Conclusion

We introduced a path-sensitive variation of CFI and an efficient runtime enforcement system, PITTYPAT. Our formal analysis and empirical evaluation demonstrate that, PITTYPAT provides strictly stronger security guarantees than conventional CFI, while incurring an acceptable amount of runtime overhead.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity. In *CCS* (2005).

[2] ANDERSEN, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, U. Cophenhagen, 1994.

[3] BLETSCH, T., JIANG, X., AND FREEH, V. Mitigating code-reuse attacks with control-flow locking. In *ACSAC* (2011).

[4] BUROW, N., CARR, S. A., BRUNTHALER, S., PAYER, M., NASH, J., LARSEN, P., AND FRANZ, M. Control-flow integrity: Precision, security, and performance. *arXiv preprint arXiv:1602.04056* (2016).

[5] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security* (2015).

[6] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security* (2014).

[7] CHENG, Y., ZHOU, Z., YU, M., DING, X., AND DENG, R. H. Ropecker: A generic and practical approach for defending against ROP attacks. In *NDSS* (2014).

[8] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977).

[9] DAVI, L., KOEBERL, P., AND SADEGHI, A.-R. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *DAC* (2014).

[10] EMAMI, M., GHIYA, R., AND HENDREN, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI* (1994).

[11] EVANS, I., FINGERET, S., GONZÁLEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point (er): On the effectiveness of code pointer integrity. In *SP* (2015).

[12] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS* (2015).

[13] GE, X., CUI, W., AND JAEGER, T. Griffin: Guarding control flows using intel processor trace. In *ASPLOS* (2017).

[14] GU, Y., ZHAO, Q., ZHANG, Y., AND LIN, Z. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *CODASPY* (2017).

[15] HARDEKOPF, B., AND LIN, C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI* (2007).

[16] KASIKCI, B., SCHUBERT, B., PEREIRA, C., POKAM, G., AND CANDEA, G. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *SOSP* (2015).

[17] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *OSDI* (2014).

[18] LATTNER, C., LENHARTH, A., AND ADVE, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI* (2007).

[19] LIU, Y., SHI, P., WANG, X., CHEN, H., ZANG, B., AND GUAN, H. Transparent and efficient cfi enforcement with intel processor trace. In *HPCA* (2017).

[20] The LLVM compiler infrastructure project. `http://llvm.org/`, 2016. Accessed: 2016 May 12.

[21] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque control-flow integrity. In *NDSS* (2015).

[22] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Softbound: highly compatible and complete spatial memory safety for c. In *PLDI* (2009).

[23] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Cets: compiler enforced temporal safety for c. In *ISMM* (2010).

[24] NECULA, G. C., MCPEAK, S., AND WEIMER, W. Ccured: Type-safe retrofitting of legacy code. In *PLDI* (2002).

[25] NIU, B., AND TAN, G. Modular control-flow integrity. In *PLDI* (2014).

[26] NIU, B., AND TAN, G. Per-input control-flow integrity. In *CCS* (2015).

[27] NYSTROM, E. M., KIM, H.-S., AND WEN-MEI, W. H. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *International Static Analysis Symposium* (2004).

[28] ONE, A. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996).

[29] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security* (2013).

[30] PEWNY, J., AND HOLZ, T. Control-flow restrictor: Compiler-based CFI for iOS. In *ACSAC* (2013).

[31] REINDERS, J. Processor tracing - Blogs@Intel. `https://blogs.intel.com/blog/processor-tracing/`, 2013. Accessed: 2016 May 12.

[32] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *SP* (2015).

[33] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS* (2007).

[34] STEENSGAARD, B. Points-to analysis in almost linear time. In *POPL* (1996).

[35] THALHEIM, J., BHATOTIA, P., AND FETZER, C. Inspector: Data provenance using intel processor trace (pt). In *ICDCS* (2016).

[36] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Sec.* (2014).

[37] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical context-sensitive CFI. In *CCS* (2015).

[38] WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI* (2004).

[39] WILANDER, J., NIKIFORAKIS, N., YOUNAN, Y., KAMKAR, M., AND JOOSEN, W. RIPE: Runtime intrusion prevention evaluator. In *ACSAC* (2011).

[40] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. Cfimon: Detecting violation of control flow integrity using performance counters. In *DSN* (2012).

[41] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *SP* (2013).

[42] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Usenix Sec.* (2013).

[43] ZHU, J., AND CALMAN, S. Symbolic pointer analysis revisited. In *PLDI* (2004).

$$\begin{aligned}
\mathsf{instrs} := {} & \texttt{ops REGS, REGS, REGS} \mid \texttt{alloc REGS} & (1)\\
& \mid \texttt{ld REGS,REGS} \mid \texttt{store REGS,REGS} & (2)\\
& \mid \texttt{br REGS,REGS} \mid \texttt{call REGS} \mid \texttt{return} & (3)
\end{aligned}$$

Figure 5: A space of instructions, Instrs, in a target language. Instrs is defined over registers `Regs` and data operations `Ops`.

Appendix

A Language definition

In this section, we define the syntax (§A.1) and semantics (§A.2) of programs in PITTYPAT's target language.

A.1 Syntax

Figure 5 contains the syntax of a space of program instructions, Instrs. An instruction may compute the value of an operation in `ops` over values stored in registers and store the result in a register, may allocate a fresh memory cell (Eqn. 1), may load a value stored in the address in one operand register into a target register, may store a value in an operand register at the address stored in a target register (Eqn. 2), may test if the value in a register is non-zero and if so transfer control to an instruction at the address stored in an operand register, may perform an indirect call to a target address stored in an operand, or may return from a call (Eqn. 3). Although all operations are assumed to be binary, when convenient we will depict operations as using fewer registers (e.g., a copy instruction `copy r0,r1` in §4.2).

A program is a map from instruction addresses to instructions. That is, for space of instruction addresses IAddrs containing a designated *initial* address $\iota \in \mathsf{IAddrs}$, the language of programs is $\mathsf{Lang} = \mathsf{IAddrs} \to \mathsf{Instrs}$.

Instrs does not contain instructions similar to those in an architecture with a complex instruction-set, which may, e.g., perform operations directly on memory. The design of PITTYPAT directly generalizes to analyze programs that use such an instruction set. In particular, the actual implementation of PITTYPAT monitors programs compiled for `x86`.

A.2 Semantics

Each program $P \in \mathsf{Lang}$ defines a language of sequences of program states, called runs, that are generated by executing a sequence of instructions in P from an initial state. In particular, each program P defines two languages of runs. The first is the language of *well-defined* runs, in which each step from the current state is defined by the semantics of the next instruction in P. The second is the language of *feasible* runs contain some state q from which P executes an instruction that is not defined at q (e.g., dereferencing an invalid address). When the successive state of q is not defined and the program takes a step of execution, the program may potentially perform an operation that subverts security.

A state is a stack of assignments from registers to values and a memory, which maps each memory cell to a value. Let Words be a space of data words and let Cells be a space of memory cells. A value is an instruction address (§A.1), a data word, or a memory cell; i.e., $\mathsf{Values} = \mathsf{IAddrs} \cup \mathsf{Words} \cup \mathsf{Cells}$. Let the space of *registers* be denoted Regs. A *register frame* is the address of the current instruction and a map from each register to a value; i.e., the space of register frames, for $\mathsf{RegMaps} = \mathsf{Regs} \to \mathsf{Values}$, is denoted

$$\mathsf{Frames} = \mathsf{IAddrs} \times \mathsf{RegMaps}$$

For each register frame $f \in \mathsf{Frames}$, the instruction address of f is denoted as $\mathsf{ip}[f]$.

A *cell memory* is a map from each memory cell to a value; i.e., the space of cell memories is $\mathsf{Mems} = \mathsf{Cells} \to \mathsf{Values}$. A *state* is a pair of a non-empty stack of register frames and a cell memory; i.e., the space of states is denoted

$$\mathsf{States} = \mathsf{Frames}^{+} \times \mathsf{Mems}$$

For each state q, the instruction address of the top frame of q is denoted $\mathsf{ip}[q]$. For each sequence of states $r \in \mathsf{States}^*$, the sequence of corresponding instruction pointers of each state in r is denoted $\mathsf{IPs}(r) \in \mathsf{IAddrs}^*$. The states consisting of a single stack frame whose instruction pointer is ι are the *initial* states, denoted $\mathsf{States}_0 \subseteq \mathsf{States}$.

A transition relation relates each pre-state and instruction to their resulting post-states. I.e., the space of transition relations is $\mathsf{TransRels} = (\mathsf{States} \times \mathsf{Instrs}) \times \mathsf{States}$. The semantics of Lang is defined by the *well-defined* transition relation of Lang, denoted $\rho[\mathsf{WellDef}] \in \mathsf{TransRels}$. Each step of execution that is safe is a step in $\rho[\mathsf{WellDef}]$. The definition of $\rho[\mathsf{WellDef}]$ is standard, and we omit a complete definition.

For each transition relation $\rho \in \mathsf{TransRels}$, the *runs* of ρ in P are the sequences of states r in which each state in r steps to the successive state in r under ρ in P; the language of all such runs is denoted $\mathsf{Runs}[\rho, P]$. The runs of P under $\rho[\mathsf{WellDef}]$ are the *well-defined* runs of P, denoted

$$\mathsf{Runs}[\mathsf{WellDef}, P] = \mathsf{Runs}[\rho[\mathsf{WellDef}], P]$$

The feasible transition relation of Lang is $\rho[\mathsf{WellDef}]$ extended to relate each pre-state and instruction undefined in $\rho[\mathsf{WellDef}]$ to each post-state. The feasible transition relation thus includes safe steps of execution that a program may take, along with unsafe steps taken when the program executes an instruction from a state in which the instruction is not defined (i.e., loading from an address that does not point to allocated memory). The feasible transition relation of Lang is denoted

$$\rho[\mathsf{Feasible}] = \rho[\mathsf{WellDef}] \cup ((\mathsf{States} \times \mathsf{Instrs}) \setminus \mathsf{Dom}(\rho[\mathsf{WellDef}])) \times \mathsf{States}$$

where $\mathsf{Dom}(\rho[\mathsf{WellDef}])$ denotes the domain of $\rho[\mathsf{WellDef}]$.

The runs of P under $\rho[\mathsf{Feasible}]$ are the *feasible* runs of P, denoted $\mathsf{Runs}[\mathsf{Feasible}, P] = \mathsf{Runs}[\rho[\mathsf{Feasible}], P]$.

B Formal definition of points-to analysis

A control analysis takes a program P and computes a sound over-approximation of the instruction pointers that may be stored in each register when P executes a given instruction over a well-defined run. A control-analysis domain is an abstract domain [8] consisting of a set of abstract states, a concretization relation from abstract states to the program states that they represent, and an abstract transformer that describes how each abstract state is updated by a program.

Definition 3 *A* control-analysis domain *is a triple* (A, γ, τ), *with:* ***(1)*** *An* abstract domain A. ***(2)****A* concretization relation $\gamma \subseteq A \times \mathsf{States}$. *There must be* initial *and* empty *elements* $\mathsf{Init}, \mathsf{Empty} \in A$ *such that* ***(a)*** $\{\mathsf{Init}\} \times \mathsf{States}_0 \subseteq \gamma$ *and* ***(b)*** $\{\mathsf{Empty}\} \times \mathsf{States} \cap \gamma = \emptyset$. ***(3)*** *An* abstract transformer $\tau : A \times \mathsf{Instrs} \times \mathsf{IAddrs} \to A$, *where for each abstract state* $a \in A$, *each state* $q \in \mathsf{States}$ *such that* $(a, q) \in \gamma$, *and each instruction* $\mathtt{i} \in \mathsf{Instrs}$ *and state* $q' \in \mathsf{States}$ *such that* $(q, \mathtt{i}, q') \in \rho[\mathsf{WellDef}]$, *it holds that* $(\tau(a, \mathtt{i}, \mathsf{ip}[q']), q') \in \gamma$.

For each control domain D, we refer to the abstract states, concretization relation, and abstract transformer of D as $A[D]$, $\gamma[D]$, and $\tau[D]$, respectively. The space of control-analysis domains is denoted Doms.

The initial and empty elements in $A[D]$ are denoted $\mathsf{Init}[D]$ and $\mathsf{None}[D]$. The binary relation $\sqsubseteq^D \subseteq A[D] \times A[D]$ is defined as follows. For all abstract states $a_0, a_1 \in A[D]$, if for each concrete state $q \in \mathsf{States}$ such that $(a_0, q) \in \gamma[D]$ it holds that $(a_1, q) \in \gamma[D]$, then $a_0 \sqsubseteq^D a_1$.

C Formal definitions of control security

C.1 Conventional CFI

For each control domain D and program P, a valid description of P in D over-approximates the control targets stored bound to registers and memory when control reaches each of instruction address of P. In particular, a valid description δ maps each instruction address to an abstract state of D that such that (1) δ maps ι to $\mathsf{Init}[D]$ and (2) δ is consistent with the abstract transformers of each instruction over D.

Definition 4 *For each control domain* $D \in \mathsf{Doms}$ *and program* $P \in \mathsf{Lang}$, *let* $\delta : \mathsf{IAddrs} \to A[D]$ *be such that* ***(1)*** $\delta(\iota) = \mathsf{Init}[D]$*;* ***(2)*** *for all instruction addresses* $a_0, a_1 \in \mathsf{IAddrs}$ *and instruction* $\mathtt{i} \in \mathsf{Instrs}$, *it holds that* $\tau[D](\delta(a_0), \mathtt{i}, a_1) \sqsubseteq^D \delta(a_1)$. *Then* δ *is a* valid description *of P in D.*

For each control domain $D \in \mathsf{Doms}$ and program $P \in \mathsf{Lang}$, the space of valid descriptions of P in D is denoted $\mathsf{ValidDescs}[D, P]$.

For each control domain $D \in \mathsf{Doms}$ and program $P \in \mathsf{Lang}$ the *most precise* description of P in D, denoted $\mu[D, P] \in \mathsf{ValidDescs}[D, P]$, is the valid description of P in D such that for all valid descriptions $\delta' \in \mathsf{ValidDescs}[D, P]$ and each instruction address $a \in \mathsf{IAddrs}$, $\mu[D, P](a) \sqsubseteq^D \delta'(a)$. Under well-understood conditions [8], D has a most-precise description for each program P that can be computed efficiently [2, 34].

Example 1 *For program* `dispatch` *(§2.1) and any control domain D that maps each instruction pointer to a set of instruction addresses, the most precise description of* `dispatch` *restricted to function pointers is given in §2.2.*

Each program P and domain D define a transition relation in which at each step from each instruction address a, the program only transfers control to an instruction address that is feasible in the most precise description of P under D at a.

Definition 5 *For each program $P \in \mathsf{Lang}$ and control domain $D \in \mathsf{Doms}$, let $\rho \in \mathsf{TransRels}$ be such that for all instruction addresses $a, a' \in \mathsf{Addrs}$, each instruction $\mathtt{i} \in \mathsf{Instrs}$ with $\tau[D](\mu[D,P](a), \mathtt{i}, a') \neq \mathsf{None}[D]$ and all states $q, q' \in \mathsf{States}$ with $(\mu[D,P](a), q)$ and $(\mu[D,P](a'), q')$, it holds that $((q, \mathtt{i}), q') \in \rho$. Then ρ is the* flow-sensitive transition relation *of D and P.*

For each domain D and program P, the flow-sensitive transition relation of D and P is denoted $\mathsf{FS}[D,P]$.

For each control domain D and program P, the most precise flow-sensitive description of P in D (Appendix D) defines an instance of generalized control security that is equivalent to CFI [1].

Definition 6 *For all programs $P, P' \in \mathsf{Lang}$ and each control-analysis domain $D \in \mathsf{Doms}$, if P' satisfies generalized control security under $\mathsf{FS}[D,P]$ (Appendix D, Defn. 5) with respect to P, then P' satisfies* CFI modulo D *with respect to P.*

Defn. 6 is equivalent to "ideal" CFI as defined in previous work to establish fundamental limitations on CFI [5].

C.2 Path-sensitive CFI

The problem of enforcing CFI is typically expressed as instrumenting a given program P to form a new program P' that allows each indirect control transfer in each of its executions only if the target of the transfer is valid according to a flow-sensitive description of the control-flow graph of P. To present our definition of path-sensitive CFI, we will introduce a general definition of control security parameterized on a given transition relation ρ. P' satisfies generalized control security under ρ with respect to P if **(1)** P' preserves each well-defined run of P and **(2)** each feasible run of P' has instruction addresses identical to the instruction addresses of some run of P under ρ.

Definition 7 *For each transition relation $\rho \in \mathsf{TransRels}$, let programs $P, P' \in \mathsf{Lang}$ be such that **(1)** $\mathsf{Runs}[\mathsf{WellDef}, P] \subseteq \mathsf{Runs}[\mathsf{WellDef}, P']$; **(2)** for each run $r' \in \mathsf{Runs}[\mathsf{Feasible}, P']$, there is some run $r \in \mathsf{Runs}[\rho, P]$ such that $\mathsf{IPs}(r) = \mathsf{IPs}(r')$. Then P' satisfies* generalized control security *under ρ with respect to P.*

We now define path-sensitive CFI, an instance of generalized control security that is strictly stronger than CFI. Each control domain D defines a transition relation over program states that are described by abstract states of D connected by the abstract transformer of D.

Definition 8 *For each control domain $D \in \mathsf{Doms}$ (§3.2, Defn. 3), let $\rho[D] \in \mathsf{TransRels}$, be such that for each abstract state $a \in A[D]$, each state $q \in \mathsf{States}$ such that $(a, q) \in \gamma[D]$, and each instruction $\mathtt{i} \in \mathsf{Instrs}$ and state $q' \in \mathsf{States}$ such that $(\tau[D](a, \mathtt{i}, \mathsf{ip}[q']), q') \in \gamma[D]$, it holds that $(q, \mathtt{i}, q') \in \rho[D]$. Then $\rho[D]$ is the* transition relation modulo D.

For all programs P and P' and each control domain D, P' satisfies path-sensitive CFI modulo D with respect to P if each step of each run of P' corresponds to a step of P over states with the same description under D.

Definition 9 *For all programs $P, P' \in \mathsf{Lang}$ and each control domain $D \in \mathsf{Doms}$, if P' satisfies control security under $\rho[D]$ (Defn. 8) with respect to P, then P' satisfies* path-sensitive CFI modulo D *with respect to P.*

Path-sensitive CFI is conceptually similar to, but stronger than, context-sensitive CFI [37], which places a condition on only bounded suffixes of a program's control path before the program attempts to execute a critical security event, such as a system call.

Path-sensitive CFI is as strong as CFI.

Lemma 1 *For each control domain D and all programs $P, P' \in \mathsf{Lang}$ such that P' satisfies path-sensitive CFI modulo D with respect to P, P' satisfies CFI modulo D with respect to P.*

Lemma 1 follows immediately from the fact that any control-transfer target that is along a given control path must be a valid target in a meet-over-all-paths solution.

Path-sensitive CFI is in fact *strictly* stronger than CFI.

Lemma 2 *For some control domain D and programs $P, P' \in \mathsf{Lang}$, P' satisfies CFI with respect to P modulo D but P' does not satisfy path-sensitive CFI with modulo D respect to P.*

Lemma 2 is immediately proven using any domain D that is sufficiently accurate between two control states and a program P that generates state with either control configuration at a particular program point.

D Formal definition of online analysis

The behavior of the analyzer module is determined by a fixed control-analysis domain D (§3.2, Defn. 3). We refer to PITTYPAT instantiated to use control domain D for points-to analysis as PITTYPAT$[D]$.

As the analyzer module executes, it maintains a control-domain abstract state $d \in A[D]$. In each step of execution, the analyzer module receives from the monitor process the next control-transfer target taken by the monitored program P, and either chooses to raise an alarm that transferring control to the target would cause P to break path-sensitive CFI modulo D, or updates its state and allows P to take its next step of execution.

In each step of execution, the analyzer module receives the next control target $a \in \mathsf{IAddrs}$ taken by the monitored program, and either raises an alarm or updates its maintained control description d as a result. If a is not a feasible target from d over the next sequence of non-branch instructions, then the analyzer module throws an alarm signaling that control flow has been subverted, and aborts.

Theorem 1 *For $D \in \mathsf{Doms}$ and $P \in \mathsf{Lang}$, the program P' simulated by running P in PITTYPAT$[D]$ satisfies path-sensitive CFI modulo D with respect to P (Defn. 9).*

We have given the design of an analyzer module that uses an arbitrary control domain generically; i.e., the analyzer can use any control-analysis domain that satisfies the definition given in §3.2, Defn. 3. However, we have found that the performance of the analyzer module can be improved significantly by using a control domain that takes advantage of the particular context of online path-sensitive analysis by maintaining points-to information about exactly the variables that are live in each live stack frame in the program state. We now define in detail the control domain used by our analysis, $\mathsf{OnlinePtsTo} = (A, \gamma, \tau)$.

Each element in the space A is either $\mathsf{None}[A]$, which represents no states, or a tuple consisting of **(1)** a stack in which each entry is a map from each register `r` to a set of memory cells and instruction pointer that `r` may store and **(2)** a map from each cell to the cells and instruction pointers that it may store. I.e., for

$$\begin{aligned} \mathsf{Addrs} &= \mathsf{IAddrs} \cup \mathsf{Cells} \\ \mathsf{RegPtsMaps} &= \mathsf{Regs} \to \mathscr{P}(\mathsf{Addrs}) \\ \mathsf{FramePtsTo} &= \mathsf{IAddrs} \times \mathsf{RegPtsMaps} \\ \mathsf{CellPtsTo} &= \mathsf{Cells} \to \mathscr{P}(\mathsf{Addrs}) \end{aligned}$$

with $\mathscr{P}(\mathsf{Addrs})$ the powerset of addresses, the abstract states are $A = \mathsf{FramePtsTo}^{+} \times \mathsf{CellPtsTo}$. The stack containing a single frame that maps each register to the empty set of addresses, paired with an empty memory map, is the initial element of A.

Example 2 *§2.3 contains examples of elements of A. In order to simplify the presentation, in §2.3, only bindings to the function pointer `handler` are shown, because these bindings are the only ones that need to be inspected to determine the security of a given run of `dispatch`.*

Concretization relation $\gamma \subseteq A \times \mathsf{States}$ relates each stack and memory of points-to information to each concrete state with a similarly structured stack and heap. For each $n \in \mathbb{N}$, let $\mathtt{a}_0, \ldots, \mathtt{a}_n \in \mathsf{IAddrs}$, $R_0, \ldots, R_n \in \mathsf{RegMaps}$, and $R'_0, \ldots, R'_n \in \mathsf{RegPtsMaps}$ be such that for each $i \leq n$ and each register $\mathtt{r} \in \mathsf{Regs}$, if $R_i(\mathtt{r}) \in \mathsf{Addrs}$, then $R_i(\mathtt{r}) \in R'_i(\mathtt{r})$. Let $m \in \mathsf{Mems}$ and $m' \in \mathsf{CellPtsTo}$ be such that for each cell $c \in \mathsf{Cells}$, $m(c) \in m'(c)$. Then:

$$\begin{aligned} &(([(\mathtt{i}_0, R'_0), \ldots, (\mathtt{i}_n, R'_n)], m'), \\ &\quad ([(\mathtt{i}_0, R_0), \ldots, (\mathtt{i}_n, R_n)], m)) \in \gamma \end{aligned}$$

The abstract transformer $\tau : A \times \mathsf{Instrs} \times \mathsf{IAddrs} \to A$ is defined as follows. For each set of memory cells $C \subseteq \mathsf{Cells}$, let $\mathsf{fresh}(C) \in \mathsf{Cells} \setminus C$ be a fresh memory cell not in C. For all register frames $f_0, \ldots, f_n \in \mathsf{FramePtsTo}$, each register map $m \in \mathsf{RegPtsTo}$, each cell points-to map $c \in \mathsf{CellPtsTo}$, all registers $\mathtt{r0}, \mathtt{r1}, \mathtt{r2} \in \mathsf{Regs}$, and all instruction addresses $a, a' \in \mathsf{IAddrs}$, a store instruction `store r0, r1` updates the cell map so that each cell bound to `r1` points to each cell points to each cell bound to `r0`. I.e., for $c_0, \ldots, c_n \in R(\mathtt{r1})$,

$$\begin{aligned} &\tau(((a,R) :: F, m), \mathtt{store\ r0,\ r1}, a') = \\ &\quad ((a',R) :: F, m[c_0 \mapsto R(\mathtt{r0}), \ldots, c_n \mapsto R(\mathtt{r0})]) \end{aligned}$$

A branch instruction requires that the target instruction address is in the points-to set of the target register of the branch. I.e., if $a' \in R(\mathtt{r0})$, then

$$\tau(((a,R) :: F, m), \mathtt{br\ r0}, a') = ((a',R) :: F, m)$$

Otherwise, τ maps the abstract state to $\mathsf{None}[A]$. A call instruction increments the instruction pointer in the top frame and pushes onto the stack a frame with an empty register map. I.e., if $a' \in R(\mathtt{r})$,

$$\begin{aligned} &\tau(((a,R) :: F, m), \mathtt{call\ r0}, a') = \\ &\quad ((a', \emptyset) :: (a+1, R) :: F, m) \end{aligned}$$

Otherwise, τ maps the abstract state to $\mathsf{None}[A]$. A return instruction pops the top register frame from the stack. I.e., $\tau(((a,R) :: F, m), \mathtt{return}, a') = (F, m)$ A data operation updates only the instruction address:

$$\tau(((a,R) :: F, m), \mathtt{op\ r0,r1,r2}, a') = ((a',R) :: F, m)$$

An allocation `alloc r0` updates the register map in the top frame of the stack so that `r0` points to a fresh memory cell. I.e.,

$$\begin{aligned} &\tau(((a,R) :: F, m), \mathtt{alloc\ r0}, a') = \\ &\quad ((a', R[\mathtt{r0} \mapsto \mathsf{fresh}(\mathsf{Rng}(m))]) :: F, m) \end{aligned}$$

where $(a,R) :: F$ denotes (a,R) prepended to F and $\mathsf{Rng}(m)$ denotes the range of m. A copy instruction `copy r0, r1` updates the register map so that each cell that may be stored in `r0` may be stored in `r1`. I.e.,

$$\begin{aligned} &\tau(((a,R) :: F, m), \mathtt{copy\ r0,\ r1}, a') = \\ &\quad ((a', R[\mathtt{r1} \mapsto R(\mathtt{r0})]) :: F, m) \end{aligned}$$

A load instruction `load r0, r1` updates the register map in the top frame so that each cell that may be pointed to by a cell bound to `r0` is bound to `r1`:

$$\begin{aligned} &\tau(((a,R) :: F, m), \mathtt{ld\ r0,\ r1}, a') = \\ &\quad ((a', R[\mathtt{r1} \mapsto \bigcup_{c \in R(\mathtt{r0})} m(c)]) :: F, m) \end{aligned}$$

The abstract transformers for other instructions, such as data operations that perform pointer arithmetic, are defined similarly, and we do not give explicit definitions here in order to simplify the presentation.

Example 3 *Consider descriptions of states of `dispatch` and its instruction `call handler` (§2.1). For abstract state*

$$A_0 = ([(\mathit{L22}, [\mathit{handler} \mapsto \{\mathit{priv}\}])], \emptyset)$$

$\tau(A_0, \mathit{call\ handler}, \mathit{priv})$ consists of a fresh stack frame for priv pushed onto the stack $[(\mathit{L22}, \mathit{handler} \mapsto \mathit{priv})]$. For abstract state

$$A_1 = ([(\mathit{L22}, [\mathit{handler} \mapsto \{\mathit{unpriv}\}]), \emptyset)$$

$\tau(A_1, \mathit{call\ handler}, \mathit{priv})$ is $\mathsf{None}[A]$.

We have given an online points-to analysis for a simple language with only calls and returns. Practical languages typically support additional interprocedural control instructions that, e.g., resolve calls targets through dynamic dispatch or unwind the callstack. Our complete implementation handles each such instruction using an appropriate abstract transformer.

The fact that (D, γ, τ) defines a sound analysis can be proven using standard techniques from abstract interpretation [8].