Practical Context-Sensitive CFI

Victor van der Veen[†‡] Dennis Andriesse[†*] Enes Göktaş[*] Ben Gras[*]
Lionel Sambuc[*] Asia Slowinska[§] Herbert Bos[‡] Cristiano Giuffrida[‡]
[†]Equal contribution joint first authors

[‡ * §]Department of Computer Science
VU University Amsterdam, The Netherlands

[§]Lastline, Inc.
Santa Barbara, CA

[‡]{vvdveen,herbertb,giuffrida}@cs.vu.nl
[*]{d.a.andriesse,e.goktas,b.j.gras,l.a.sambuc}@vu.nl

[§]asia@lastline.com

ABSTRACT

Current Control-Flow Integrity (CFI) implementations track control edges individually, insensitive to the context of preceding edges. Recent work demonstrates that this leaves sufficient leeway for powerful ROP attacks. Context-sensitive CFI, which can provide enhanced security, is widely considered impractical for real-world adoption. Our work shows that Context-sensitive CFI (CCFI) for both the backward and forward edge can be implemented efficiently on commodity hardware. We present *PathArmor*, a binary-level CCFI implementation which tracks paths to sensitive program states, and defines the set of valid control edges *within the state context* to yield higher precision than existing CFI implementations. Even with simple context-sensitive policies, *PathArmor* yields significantly stronger CFI invariants than context-insensitive CFI, with similar performance.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Control-Flow Integrity, Context-sensitive CFI

1. INTRODUCTION

Control-Flow Integrity (CFI) [8] has developed into one of the most promising techniques to stop code reuse attacks against C and C++ programs. Typically, such attacks circumvent common defenses such as DEP/W$\oplus$X by diverting a program's control flow to a set of Return-Oriented Programming (ROP) gadgets [17, 42]. Likewise, they defeat widely deployed ASLR by either targeting gadgets at fixed (non-randomized) addresses [13], or by dynamically disclosing the addresses of randomized gadgets [45]. CFI promises to prevent all such attacks by ensuring that all control transfers conform to the program's original Control Flow Graph (CFG). In theory, CFI is very powerful and, in its purest and ideal form, provably secure against most integrity violations of the control flow [7].

Ten years after the original CFI proposal [6], however, researchers are still working to find practical CFI implementations [19, 26, 31, 36, 47, 53, 55], able to approximate the security of the purest form of CFI with acceptable performance. Common CFI solutions, including state-of-the-art binary-level implementations such as bin-CFI [55] and CCFIR [53], attempt to substantially relax constraints on the set of legal targets for both the backward (e.g., `ret` instructions) and forward (e.g., indirect `call` instructions) control edges. While doing so reduces the performance overhead to a few percent only, it also provides more degrees of freedom for the attackers. Other even more lightweight CFI solutions, such as ROPecker [19] and kBouncer [36], build on heuristics and hardware support to detect anomalous control flows—which resemble ROP gadget chains—and stop many current exploitation attempts at low performance overheads. Unfortunately, a string of recent publications comprehensively shows that it is possible to circumvent all these lightweight CFI solutions with relatively little effort [16, 24, 27, 28, 41].

A key problem with traditional CFI solutions—even recent source-level fine-grained ones [47]—is that they enforce only context-insensitive CFI policies, which examine control edges in isolation and attempt to statically derive the resulting superset of all the possible targets according to the CFG. The lack of context inevitably results in weak CFI invariants, allowing attackers to freely chain edges together and form paths that are even trivially infeasible in the original CFG (e.g., returning to a function never on the active call stack [27]).

Context-sensitive CFI techniques are a promising way to address this problem, since they rely on context-sensitive static analysis to associate CFI invariants to control-flow *paths*—i.e., multiple consecutive edges—in the CFG and enforce such invariants on execution paths at runtime. The stronger security guarantees provided by context-sensitive CFI techniques have been acknowledged as early as in the

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813673.

original CFI proposal, but their real-world adoption has been rapidly dismissed as impractical [6].

In this paper, we demonstrate that Context-sensitive CFI (CCFI) can indeed be implemented in an efficient, reliable, and practical way for real-world applications. We present *PathArmor*[1], the first binary-level CCFI solution which enforces context-sensitive CFI policies on both the backward and forward edges. *PathArmor* relies on commodity hardware support to efficiently and reliably monitor execution paths to sensitive functions which can be used to mount control-flow diversion attacks [36], and uses a carefully optimized binary instrumentation design to enforce CCFI invariants on the monitored paths. *PathArmor*'s path invariants are derived by a scalable context-sensitive static analysis performed over the CFG on-demand, which uses caching of path verification steps to achieve high efficiency. Verification itself is also very efficient, since all the CFI checks are batched at sensitive program points.

To show the practicality of our design, we have prototyped two context-sensitive and binary-level CFI policies (for the backward and forward edges, respectively) on top of *PathArmor*. Moreover, our framework can also serve as a general foundation for even stronger CCFI implementations, for instance using context-sensitive data-flow analysis at the source level. Even in the current setup, *PathArmor* provides a comprehensive CCFI protection system with much stronger security guarantees than traditional CFI solutions, while matching or even improving their performance. Moreover, due to its optimized design, *PathArmor* can also serve as an efficient basis for fine-grained context-insensitive CFI ($\overline{\text{C}}$CFI) policies.

Contributions.

Our contribution is threefold:

- We identify the key challenges towards practical CCFI implementations and investigate opportunities to address these challenges in real-world applications and commodity platforms.

- We present *PathArmor*, a framework to efficiently support arbitrary context-sensitive and context-insensitive CFI policies on commodity platforms. To fulfill its goals, *PathArmor* relies on hardware support, binary instrumentation, and on-demand static analysis to batch even sophisticated CFI checks at the relevant sensitive points in a binary. We complement our solution with fine-grained $\overline{\text{C}}$CFI policies and simple but comprehensive (backward and forward edge) CCFI policies, making *PathArmor* the first practical end-to-end CCFI implementation.

- We evaluate *PathArmor* using popular server applications and the SPEC CPU2006 benchmarks. Our results show that *PathArmor* can significantly restrict the number of legal control flows compared to traditional CFI solutions (-70% across all our applications, geometric mean), while yielding bounded memory usage (+18-74 MB on our applications) and low run-time performance overhead (3% on SPEC and 8.4% on our applications, geometric mean).

[1] *PathArmor* is open source, available via `https://github.com/dennisaa/patharmor`

2. CONTEXT-SENSITIVE CFI

The general goal of every CFI solution is to allow all the control flows which occur in the interprocedural control-flow graph (CFG) defined by the programmer, and reject the largest possible fraction of the other flows as illegal [8]. This section formalizes the definition of a legal flow adopted in existing practical CFI solutions, contrasts it with the stricter definition adopted in Context-sensitive CFI (CCFI), and details the key challenges towards practical CCFI.

2.1 Legal flows

We model a CFG as a digraph $G = (V, E)$ where V is the set of basic blocks, and E the set of control edges in the CFG defined by the program.

Traditional CFI [8] enforces that each individual (indirect) control transfer taken by the program during the execution must match an edge in the CFG:

Context-insensitive CFI ($\overline{\text{C}}$CFI). *For each control transfer $e_i = (v_x, v_y)$ between basic blocks v_x and v_y, $\overline{\text{C}}$CFI enforces that $e_i \in E$.*

In other words, $\overline{\text{C}}$CFI checks conformance to the current position in the CFG and does not distinguish between different paths in the CFG that lead to a given control transfer. For instance, consider the following two paths that both lead to *E()*:

```
A(){ indirect call to B(); } C(){ indirect call to D(); }
B(){ indirect call to E(); } D(){ indirect call to E(); }
```

Disregarding the context would allow function E to return to either B or D. However, we should only allow a return (backward edge) to B, when coming from A (and B). Likewise, we should only allow a return to D if the program got there via C.

We can easily construct a similar example for the CFG's forward edges, for instance by considering callbacks. Suppose B and D both call E with callback argument cb_B and cb_D, respectively. When E invokes the callback, $\overline{\text{C}}$CFI would allow either of the (cb_B and cb_D) targets, while taking the context into consideration would allow us to (rightly) conclude that cb_B is only legal if we reached E via B.

To mimic context-sensitive behavior to some degree (backward edges only) a number of existing CFI solutions complement their operations with a shadow stack [11, 18, 20, 21, 25, 38, 39, 43, 51]. However, shadow stacks are typically expensive at the binary level [18, 23, 43]. Moreover, unlike CFI techniques, their security relies on the integrity of in-process run-time information, which state-of-the-art implementations typically protect using system-enforced ASLR—with its known security limitations and probabilistic guarantees against arbitrary memory write vulnerabilities. All the other existing CFI solutions, in turn, implement fully context-insensitive ($\overline{\text{C}}$CFI) policies as described above.

In addition, state-of-the-art binary-level CFI solutions, such as CCFIR [53] or binCFI [55], further relax their $\overline{\text{C}}$CFI policies for performance reasons. These context-insensitive implementations group control transfer sources and destinations based on a general definition of type, and enforce that the source and destination type match:

Practical $\overline{\text{C}}$CFI. *For each control transfer $e_i = (v_x, v_y)$ between basic blocks v_x and v_y, practical $\overline{\text{C}}$CFI ensures $x \in sources(type(e_i)) \wedge y \in sinks(type(e_i))$, where $sources(\tau)$*

and $sinks(\tau)$ *denote the sets of program locations having outbound or inbound edges of type* τ*, respectively.*

Practical $\overline{\text{C}}$CFI precludes malicious control transfers like jumps into the middle of a function, or returns to non-call sites. Attackers, however, can still successfully mount powerful attacks using gadgets which adhere to the imposed type restrictions [15, 16, 24, 27, 41].

CCFI provides stronger CFI invariants than both practical and ideal $\overline{\text{C}}$CFI. Rather than considering control transfers individually, CCFI examines each transfer in the context of recently executed transfers:

CCFI. *Given a path* $p = (e_1, e_2, \ldots, e_n)$ *of control transfers leading to a given program point* P*, CCFI verifies the validity of* P *by checking that* $\forall i \in \{1, 2, \ldots, n\}$*, edge* e_i *is consecutively valid in the context of all the preceding CFG edges* $e_1, \ldots, e_{i-1}$.

Since CFI checks are enforced *per path* (rather than *per edge*), CCFI can enable arbitrarily powerful context-sensitive policies on both the backward and forward edges.

2.2 Challenges

In this section, we discuss the three fundamental challenges towards practical CCFI, and in the remainder of the paper, we present *PathArmor*—the first practical binary-level solution to these problems—proving CCFI effective in practice.

C1. Efficient path monitoring A major challenge in implementing a practical CCFI solution is identifying an efficient mechanism to constantly monitor paths of executed control flow transfers at runtime. Other than imposing minimal performance overhead, the path monitoring mechanism should also be reliable, that is neither the program nor the attacker should be able to tamper with the recorded data. All these requirements were considered the key obstacle to the real-world adoption of context-sensitive CFI in the original CFI proposal [6].

To address this challenge, *PathArmor* relies on branch recording features available in modern x86_64 processors to implement an efficient and reliable path monitoring mechanism at runtime.

C2. Efficient path analysis To verify the validity of a path towards a given program point P, CCFI needs to statically analyze the CFG and identify the legal paths towards P in a context-sensitive fashion, validating all the edges in the path. The naive solution—statically enumerating *all* the legal paths towards *all* the relevant program points—cannot scale efficiently to large and complex CFGs, with the number of paths growing exponentially with $|V|$ and $|E|$. This *path explosion* problem is well known in several application domains (symbolic execution, among others [30]). Even focusing our static analysis on a particular program point and sequence of indirect control transfers derived by run-time information only partially eliminates this problem. Path explosion can still occur between any two indirect control edges, especially in presence of loops and long sequences of direct jumps and calls.

To address this challenge, *PathArmor* relies on an on-demand, constraint-driven context-sensitive static analysis over a normalized CFG representation. The constraints, derived by run-time information recorded by our path monitor, allow our context-sensitive path analysis to efficiently scale to arbitrarily large and complex CFGs.

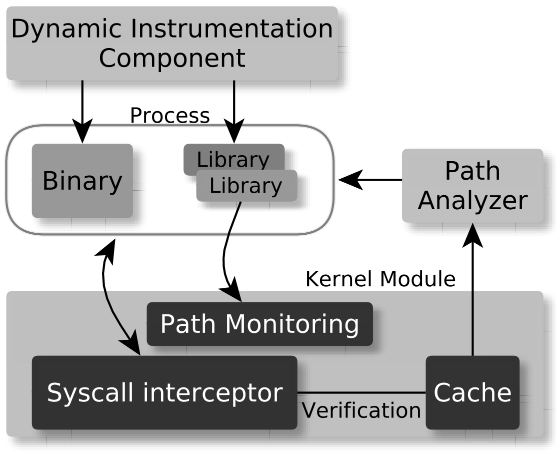

Figure 1: Overview of *PathArmor*.

C3. Efficient path verification To detect control-flow diversion attacks, CCFI needs to carefully select program points to verify the current execution path for validity. To provide strong security guarantees, path verification needs to be performed in all the possible execution states that are potentially harmful. The naive solution—performing path verification after every executed control transfer—is clearly inefficient and scales poorly with the path length.

To address this challenge, *PathArmor* relies on a kernel module to efficiently verify only the paths to well-defined sensitive functions in the program. While the verification still needs to run for each path to these functions encountered during the execution, *PathArmor* aggressively caches verification results to minimize the resulting impact on runtime performance. Since the number of paths to sensitive functions is limited in practice (as shown in Section 5.3 for popular server programs), caching is effective in amortizing path verification costs throughout the execution.

3. PATHARMOR

Figure 1 presents the high-level overview of *PathArmor* and details its three main components: (i) a kernel module, (ii) an on-demand static analyzer, and (iii) an instrumentation component.

PathArmor relies on a *kernel module* which provides a Branch Record core to support per-thread control transfer monitoring in multi-process and multi-threaded programs. For this purpose, our module uses the 16 *Last Branch Record* (*LBR*) registers available in modern Intel processors and only accessible from ring 0. This strategy allows our module to monitor paths of (16) recently exercised control transfers in an efficient and reliable way (addressing **C1**).

In addition to path monitoring, our kernel module triggers path verification steps upon security-sensitive system calls issued by the program—but also other special sensitive operations, as detailed later. To further improve the performance of path verification, our module also maintains a path cache, which stores hashes of previously verified paths and eliminates the need to enforce more expensive CCFI checks at each verification (addressing **C3**). We discuss our kernel module in more detail in Section 3.1.

Once the kernel module is loaded, protected program binaries run with *PathArmor*'s *dynamic instrumentation component.* This component first starts our *path analyzer*, an external trusted component which runs in the background and waits for path verification requests from the kernel module via a dedicated upcall interface. To satisfy path verification requests, our analyzer receives all the necessary LBR-based path information—and constraints on indirect but also interprocedural direct control transfers—from our kernel module and performs static analysis on-demand to enforce our CCFI policies. For this purpose, the analyzer needs to reconstruct the CFG of the target binary and preprocess it with a preliminary *CFG reduction* step that prunes all the irrelevant intraprocedural edges from the control-flow graph. This step and our constraint-based strategy eliminate all the intraprocedural and interprocedural (respectively) path explosion threats, ensuring a scalable on-demand path analysis (addressing **C2**). After determining whether a path is valid, our analyzer reports its findings back to the kernel module, which, in response, stops the program (if verification fails) or populates the path cache (otherwise). We elaborate more on our path analyzer in Section 3.2.

After initializing *PathArmor*'s path analyzer, our *dynamic instrumentation component* sets up an in-program communication channel with the kernel module to enable (and later manage) path monitoring for the target binary. Finally, our instrumentation component instruments the binary according to a predetermined sensitive path termination policy. *PathArmor* can, in principle, be configured to verify either entire paths to sensitive system calls or limit such paths to the library call interface. The current *PathArmor* implementation uses the latter mode of operation by default, given that, on commodity hardware, the LBR can only record the 16 most recently executed control transfers and allowing branch tracing inside the libraries can potentially "pollute" paths and thus "erase" program context—an observation also made in prior work [36]. The trade off—which can, however, be reconsidered with future hardware extensions—is that *PathArmor*'s default configuration can defend against control-flow diversion attacks only in the program, excluding those originating from vulnerabilities in the libraries from the threat model. For completeness, we evaluate the feasibility of future in-library path tracking in Section 5.5. We discuss our instrumentation component in more detail in Section 3.3.

3.1 Kernel Module

As illustrated in Figure 1, the kernel module consists of two main components: (i) a system call interceptor that sends validation requests (via a cache) to the on-demand static analyzer, and (ii) a Branch Record core (LBR API) that monitors and records branches occurring within the target's main address space.

3.1.1 System Call Interception

As mentioned in Section 2.2, *PathArmor* limits verification to a small number of security sensitive path endpoints in order to maintain minimal runtime overhead. In particular, these endpoints consist of a set of dangerous system calls an attacker requires to deploy a meaningful exploit, like `exec` and `mprotect` (and other dedicated sensitive operations, see Section 3.3.3). We refer to them as sensitive calls. Like other work in this area [19], we propose to detect only these dangerous endpoints, rather than every possible library and system call.

To intercept system calls at runtime, the kernel module installs an alternative syscall handler. When our target requests execution of a dangerous system call, we pause execution, collect LBR data, and forward it to the on-demand static analyzer in user space. If the analyzer returns `true` (meaning that the path was found in the CFG and thus is valid), the kernel module stores a hash of the path in a cache data structure before permitting the system call. We use cryptographically secure second-preimage resistant[2] hash algorithms (MD4 in our evaluation) to prevent path crafting attacks, where attackers craft an invalid path with a hash that collides with that of a valid path.

If the exact same path is executed a second time, *PathArmor* looks for its hash in the cache, and only sends a request to the on-demand static analyzer if no match was found in the cache. This limits the amount of overhead caused by traversing the CFG.

In the event that on-demand static analysis returns a negative result (no valid path was found in the CFG), the module stops the program and reports that an attack was detected. With the LBR data still in place, this can also help pinpoint the exact location of the attack.

3.1.2 Branch Recording

In addition to path verification, the kernel module provides a Branch Recording core that implements support for tracking branches on a per process-thread basis. In addition, it exposes an interface to the instrumented libraries that is used to disable branch recording during library execution. It can do this either using the LBR (the current default) or Intel's Branch Trace Storage (BTS) feature. Although prior work has shown that BTS imposes a significant performance slowdown (typically in the order of 20-40x [46]), its 'unlimited' nature provides a useful means to measure how many LBR registers are required to approach optimal security (Section 5).

Ideally, we would configure the Branch Recording core to collect only indirect branches (indirect jumps, indirect calls and returns), as only these branches can be modified by an attacker. However, armed only with information about indirect branches exercised by the program, we cannot eliminate the path explosion problem. To solve this issue, we instruct the Branch Recording core to keep track of direct call instructions as well, which can be used by the on-demand static analysis to eliminate path explosion, rendering *PathArmor* efficient in practice. We elaborate more on this in Section 3.2.

To disable branch recording during library execution, we expose two `ioctl()` requests to libraries: `LIB_ENTER` and `LIB_EXIT`. The dynamic instrumentation component detailed in Section 3.3 inserts these requests for each used library function by instrumenting their entry and exit points. We discuss related implementation challenges such as how to enable branch recording again for callbacks, in depth in Section 4. Note that attackers cannot abuse `ioctl` requests to disable *PathArmor*, as discussed in Section 6.3.

[2]For a second-preimage resistant hash algorithm h and input x, it is computationally hard to find a second input $x' \neq x$ such that $h(x) = h(x')$.

3.2 Path Analyzer

The role of the path analyzer is to verify (on the static CFG) at runtime if a particular path observed at an endpoint is valid. It consults the CFG of the binary and searches it for the path. We now discuss the three main steps of this analysis: CFG generation, a reduction of the CFG to eliminate the path explosion problem, and path validation.

3.2.1 CFG Generation

To validate a path, *PathArmor* requires an accurate CFG of the protected binary. To obtain a CFG, we use existing binary analysis frameworks to disassemble and analyze binaries, as detailed in Section 4. Previous work has shown that the results are accurate enough in practice [14, 54]. To err on the safe side, *PathArmor* tolerates potential errors by overestimating the CFG when necessary. In the worst case, this may cause *PathArmor* to accept invalid paths, but it will never reject legitimate ones.

Furthermore, *PathArmor* implements indirect edge resolution policies to augment a CFG walk with indirect edges in a context-sensitive manner. If these policies fail, we resort to a fine-grained context-insensitive policy instead [53, 55].

For **backward edges** (i.e., returns), our policies implement a fully context-sensitive resolution strategy, to which we refer as *call/return matching*. This strategy emulates a runtime call stack by tracking call and return edges as these are encountered.

For **forward edges** (i.e., indirect calls), our current prototype supports a simple context-sensitive strategy which resolves code pointers propagated across caller-callee pairs with no contrived data flow. This policy lets us unambiguously resolve indirect call sites, at which call targets are loaded as constants and passed as a callback argument. However, our path abstraction, in principle, enables much more complex context-sensitive extensions. We evaluate the additional security provided by forward-edge context-sensitivity in Section 5. In cases where our current policy is unable to trace a code pointer (e.g., in case of a long-lived code pointer stored on the heap), *PathArmor* resorts to a $\overline{\text{C}}$CFI policy which matches all indirect call sites with all the functions having their address taken. Indirect jumps, in turn, are conservatively resolved by the underlying binary analysis framework.

We also implement a strategy to augment the precision of indirect jumps found in PLT entries. The CFG is updated with data received from the instrumentation component, enabling unambiguous target resolution. We discuss this resolution in more detail in Section 3.3.1.

3.2.2 Addressing Path Explosion

As discussed in Section 2, static analysis of large CFGs may lead to a path explosion problem, where the number of paths to explore increases exponentially with the exploration depth. *PathArmor* takes two measures to address the problem and perform efficient path verification.

First, as a preprocessing round, *PathArmor* performs a *CFG reduction* step that significantly prunes the CFG, and preserves reachability relations with respect to indirect edges and interprocedural direct edges. This step finds all possible paths of direct edges between entry and exit points of each function, and then collapses these paths down to a single edge between each entry point and the exit points reachable from it. This makes the subsequent search much faster, as needless (re-)explorations of direct edges can be avoided (e.g., loops).

Second, call/return matching (discussed in Section 3.2.1) allows us to recognize and discard impossible paths, such as paths that call a function from one call site, and subsequently return to another call site. Without call/return matching, the path search would have no way of identifying such mismatches.

3.2.3 Path Verification

The path analyzer is given a path that must be verified. The path is an LBR state containing direct and indirect calls, indirect jumps, and returns. To verify whether it is valid, the analyzer performs a Depth-First Search (DFS) on the CFG to find a path that contains the provided edges in the same order as they were recorded by the LBR. A recorded path is thus considered valid iff: (i) all edges in the LBR state exist within the CFG, and (ii) these edges can be linked together via a valid path of direct edges within the CFG. To ensure that the search terminates quickly if a path does *not* exist (e.g., the LBR state is malicious), the DFS does not follow indirect edges or direct call edges. Following such edges would not make sense, because by definition, such edges would be in the LBR state if they occurred on the path under analysis.

Note that in the presence of (i) direct call recording and (ii) the CFG reduction, the DFS cannot get stuck on cycles within the CFG. Indeed, it first consults the LBR for the oldest recorded branch, from a basic block `A` to a basic block `B`, and then loops over all possible outgoing edges of `B` to see which one to follow. Due to the CFG reduction, direct jump edges are collapsed, so the outgoing edges of `B` are all either indirect edges or direct call edges. For each edge the DFS examines, it checks whether this edge is the next recorded branch. If this does not hold, it tries the next edge, until it finds one that matches the following LBR state. From here, it restarts analysis, starting from this new edge. This process continues until the last edge (the most recently recorded branch) is found.

3.3 Dynamic Instrumentation

The instrumentation component consists of both a library instrumentation (in the form of a special loader), and dynamic binary instrumentation module. Its main objectives are (i) collecting address offsets (for both libraries and the target program) and passing these to the static analysis component, (ii) instrumenting libraries such that they disable LBR tracking before their execution starts and re-enable it again once finished, and (iii) starting the actual target process. In addition, the instrumentation component opens a communication channel with the kernel module that can be used by the inserted instrumentation snippets to communicate with the Branch Recording core. We now discuss the two main instrumentation modules in more detail.

3.3.1 Loader

The loader is responsible for setting up the *PathArmor* environment before starting the protected binary. It is implemented as a pre-loaded shared library using `LD_PRELOAD` and instruments the target binary's `main()` function. This hook is then used to open an `ioctl()` interface with the LBR API of the kernel module, which in turn is used by the inserted code snippets to notify the kernel module of specific events (e.g., `LIB_ENTER`).

In addition, the loader collects the program's PLT and GOT entries as well as the base addresses of the different libraries that are in place. This information is then passed via the kernel module to the on-demand static analyzer where it is used to distinguish calls to library functions from branches within the program's main address space. For this to work, the target program is started with `LD_BIND_NOW=1`, which causes the dynamic linker to resolve all symbols at the program startup instead of deferring function call resolution to the point when they are first referenced.

3.3.2 *Rewriter*

In its default configuration, *PathArmor* terminates all sensitive paths at library boundaries. For this purpose, our dynamic instrumentation module uses Dyninst to rewrite all library functions that are used by the program (i.e., those that can be found in the process' PLT table, as well as those dynamically loaded using `dlsym()`). The inserted code snippets ensure that library functions first send an LBR disable request to the LBR API in the kernel module before executing, and finish with an LBR enable request before returning to the program.

Disabling the LBR of course comes at a price: a library function may at some point invoke a callback handler which may or may not reside in the target's address space. If we do not re-enable the LBR again on callbacks, a bug in the callback handler could still be exploited by an attacker as we lose vital information on executed paths. To overcome this problem, we apply another round of dynamic instrumentation, only this time to make sure that whenever such a callback is invoked, LBR tracking is enabled again. We discuss this process in more detail in Section 3.3.3.

The dynamic instrumentation module of the initialization component performs necessary rewriting tasks at load time (when dynamically linked libraries are available) and at runtime (every time a new shared library is dynamically loaded into memory). Note that we only need to instrument shared libraries. No instrumentation is required in the protected applications, leaving the original target's code space intact.

3.3.3 *Callbacks*

As mentioned above, a second dynamic instrumentation round is required in order to enable branch recording again when a library function invokes a callback that lies in the program's binary (e.g., `qsort()`). Instrumenting callback sites is done by looping over all shared library functions and searching for indirect call instructions. For each indirect call instruction, a short snippet of code is inserted that (i) tests if the target of the indirect call lies in the target program's address space, and (ii) if this is the case, wraps the call instruction in two `ioctl()` system calls that notify the kernel module that a callback function is entered or exited: (`CALLBACK_ENTER` and `CALLBACK_EXIT`, respectively).

Whenever the kernel module receives the `CALLBACK_ENTER` request, it pushes the current LBR state (i.e., the content of the LBR registers as seen before the library function that performs the callback) to an internal stack of LBR contexts. When the callback exits (`CALLBACK_EXIT`), the kernel module pops the top of this LBR stack back into the actual registers. To support code that forks within a callback, the kernel module copies the stack of LBR contexts to the newly created process, so that parent and child both safely return to their callback handlers without inconsistent branch records.

Observe that signals are essentially a specialized form of callbacks and can be processed in a similar manner. The only difference is that instead of instrumenting code, we install a hook on the kernel's signal delivery function. This hook is executed before control is returned to the signal handler, allowing us to save the current LBR context so that it can be restored upon the `sigreturn` system call.

This approach of switching LBR contexts at the moment callback handlers are invoked raises a specific security threat where an attacker could install a different handler than normally enforced by the CFG. Consider the example where an attacker exploits a memory corruption bug to install a callback handler that fits his needs. Without applying any additional security measurements, this operation may go unnoticed (control-flow diversion happens indirectly in the kernel or in the libraries). To overcome this situation, *PathArmor* (i) considers signal handler registration and LBR management operations (i.e., push context, pop context) as sensitive operations and (ii) always copies the last branch entry during LBR context switching as the first branch entry for the new context, allowing on-demand static analysis to apply our indirect edge resolution policies on the library-originated indirect call edge before allowing the callback. A symmetric approach is used to avoid false positives for library-originated function pointers (e.g., returned by `dlsym()`) which are used for indirect call invocations by the program. Our static analyzer resolves the "special" library target in a dedicated way without resorting to more sophisticated modular CFI policies [34].

3.3.4 *Special Constructs*

Similarly to our callback support, *PathArmor* supports the `longjmp()` construct by implementing a special handler for this in the kernel module: for each `setjmp()`, the kernel stores the existing LBR contents along with the provided `env` argument. When a `longjmp()` is executed, our module verifies the LBR contents, flushes them and restores the LBR with the appropriate state as stored earlier (matching `env`). Similarly to callbacks, we rely on our dynamic instrumentation component to insert dedicated `SETJMP` and `LONGJMP` `ioctl()` requests for each construct.

4. IMPLEMENTATION

We implemented *PathArmor* on Linux v3.13 for x86_64 with support for multi-process and multi-threaded applications. Our kernel module is implemented as a standard loadable module for the Linux kernel in 1,752 LOC. The on-demand static analysis component is implemented as a plugin for the Dyninst binary analysis and rewriting framework [10] in 6,741 LOC overall. The library instrumentation is implemented as another Dyninst plugin in 1,625 LOC.

To intercept sensitive system calls, we install an alternative syscall handler by overwriting the `MSR_LSTAR` register. *PathArmor* will forward most system calls directly to their vanilla implementation, imposing little to zero extra overhead. However, we consider a total of seven system call families as dangerous, and start verification whenever these are encountered: `mprotect` and the `mmap` family (which can be used to disable DEP/W$\oplus$X), and the `exec` family (which can be used to start a malicious command) are obvious choices and have been considered in prior work in the area [19]. To address the challenges related to signal handling as detailed in Section 3.3.3, *PathArmor* also intercepts the **`sigaction`** and **`sigreturn`** system calls. *PathArmor* can also be configured to protect I/O system calls, to prevent attacks like data leaks and script injection in (for instance) web servers.

Since Linux does not currently support per-task LBR context management, we implemented it to avoid pollution from other processes. We used the standard preemption notifier functionality (**`preempt_notifier_register`**) provided by the Linux kernel to install hooks on context-switches. During a context-switch-out (**`sched_out`**), *PathArmor* stores the LBR state of the current process into an LBR process table, to restore it later when the thread is scheduled in again (**`sched_in`**). This approach allows *PathArmor* to support binaries that make use of multi-threading.

Our current *PathArmor* prototype is based on the Dyninst binary rewriting framework, and as a consequence does not support C++ exceptions. This limitation is not fundamental to *PathArmor*, and can be addressed in future work with additional engineering effort.

5. EVALUATION

We evaluated *PathArmor* on a workstation equipped with an Intel i5-2400 CPU 3.10 GHz and 8 GB of RAM. We ran all our tests on an Ubuntu 14.04 installation running Linux kernel 3.13 (x86_64). To measure the performance impact of *PathArmor* for the worst case, we default *PathArmor* to run in non-library operation mode, but we evaluate the effects of enabling in-library tracking in Section 5.5.

We focus our evaluation on popular Linux server applications, given that (i) they are widely known and adopted in the research community for evaluation purposes, (ii) they are popular exploitation targets for both local and remote attacks, and (iii) they naturally contain a relevant number of security-sensitive functions and can greatly benefit from the protection guarantees provided by *PathArmor*. Specifically, we evaluated our prototype with three popular FTP servers (namely, vsftpd v1.1.0, proftpd v1.3.3, and pureftpd v1.0.36), two popular web servers (nginx v0.8.54 and lighttpd v1.4.28), a popular SSH server (the OpenSSH Daemon v3.5), and a popular email server (exim v4.69). We also evaluated *PathArmor*'s performance on SPEC CPU2006.

To benchmark our web servers, we used the Apache benchmark [1] configured to issue 25,000 requests with 10 concurrent connections and 10 requests per connection. To benchmark our FTP servers, we relied on the pyftpbench benchmark [4] configured to open 100 connections and request 100 1 KB-sized files per connection. To benchmark OpenSSH and exim, finally, we used the OpenSSH test suite [3] and a homegrown script which repeatedly launches the sendemail program [5], respectively. We configured all our applications and benchmarks with their default settings. We ran all our experiments 11 times (checking that the CPUs were fully loaded throughout our tests) and report the median with marginal variations observed across runs.

Our evaluation answers 4 key questions: (i) *Security*: Is *PathArmor* effective in improving the security of existing CFI techniques against control-flow diversion attacks? (ii) *Memory usage*: How much memory does *PathArmor* require? (iii) *Analysis time*: Does *PathArmor*'s static analysis complete in reasonable time? (iv) *Run-time performance*: Does *PathArmor* yield low run-time overhead while protecting a relevant set of sensitive functions?

5.1 Security

To evaluate the security guarantees offered by *PathArmor* and, in particular, the improvements offered by CCFI over existing $\overline{\text{C}}$CFI techniques, we measured the strength of the CFI invariants extracted by our static analysis and enforced by *PathArmor*'s run-time verification. For this purpose, we instructed our static analyzer to generate CFI statistics during the execution of our benchmarks and compare the results against fully context-insensitive CFI policies. Note that these statistics (and metrics) are intended only to provide a clear picture of the strength of *PathArmor*'s invariants compared to other CFI solutions. As such, the following discussion focuses on a relative comparison across CFI implementations, rather than on absolute numbers.

Table 1 presents the resulting statistics aggregated across all our applications. The first, second, and third group of columns provide an overview of all the applications analyzed, their sensitive functions, and their interprocedural CFG (or simply CFG) information generated by our analyzer with fully context-insensitive indirect edge resolution policies. As the table shows, the number of sensitive functions as well as the number of nodes and edges in the CFG ($|V|$ and $|E|$, respectively) varies greatly across applications, reflecting their different internal structure.

The fourth group of columns, in turn, reports the fraction of indirect backward edges (*IB*), indirect forward edges (*IF*), and direct forward edges (*DF*) in the LBR averaged across all the sensitive function calls during the execution of our benchmarks. As expected, the overall distribution is relatively stable across applications, with backward edges largely dominating (indirect) forward edges (37% vs. 25% geometric mean). Encouragingly, direct forward edges—which, however necessary to scalably enforce our CCFI policies, also naturally decrease the number of LBR entries subject to CFI enforcement—have a significant but non-dominant impact in practice (37% geometric mean).

Finally, the last three column groups present averaged gadget statistics for the coarse-grained $\overline{\text{C}}$CFI, fine-grained $\overline{\text{C}}$CFI and CCFI policies (respectively). In detail, the $|G|$ column reports the average number of targets (and thus gadgets) allowed by the given CFI policy for each indirect edge observed in the LBR. The $\min[G_{\text{Len}}]$ column, in turn, provides more qualitative information on the resulting CFI-allowed gadgets, by averaging the minimum allowed gadget length for each edge observed in the LBR. As shown in the table, CCFI yields a significantly lower average number of gadgets compared to coarse-grained and fine-grained $\overline{\text{C}}$CFI (respectively, -99.7% and -61.6% geometric mean). Figure 2 also details the CDF of the number of allowed targets for the two applications with most sensitive calls (exim and proftpd). We observed similar trends for the other applications. The CDF confirms that CCFI allows very few targets for the vast majority of control flow transfers—for instance, on exim, 98% have less than 13 targets compared to around 86% for fine-grained $\overline{\text{C}}$CFI and 72% for coarse-grained $\overline{\text{C}}$CFI (the common policy for binary-level CFI solutions [53, 55]). This demonstrates the effectiveness of our context-sensitive CFI policies, which can drastically restrict the number of legal targets for most LBR entries.

Our improvements are naturally also reflected in the overall complexity of the gadgets left to the attacker, with the average minimum allowed gadget length ($\min[G_{\text{Len}}]$) substantially increasing compared to the coarse-grained and fine-

Server	**Functions**	**CFG**		**LBR (Avg)**			**$\overline{\text{C}}$CFI$_{\text{cg}}$ (Avg)**		**$\overline{\text{C}}$CFI$_{\text{fg}}$ (Avg)**		**CCFI (Avg)**	
		$\lvert V\rvert$	$\lvert E\rvert$	$\frac{\lvert E_{\text{IB}}\rvert}{\lvert E\rvert}$	$\frac{\lvert E_{\text{IF}}\rvert}{\lvert E\rvert}$	$\frac{\lvert E_{\text{DF}}\rvert}{\lvert E\rvert}$	$\lvert G\rvert$	$\min[G_{\text{Len}}]$	$\lvert G\rvert$	$\min[G_{\text{Len}}]$	$\lvert G\rvert$	$\min[G_{\text{Len}}]$
vsftpd	sa,mm,mp	4,052	9,269	0.33	0.23	0.44	543.26	3.5	3.17	8.0	1.27	13.1
proftpd	sa,sg,ki,mm	29,682	210,489	0.38	0.27	0.35	3249.55	2.2	19.96	4.0	6.11	7.5
pure-ftpd	sa,	5,702	19,910	0.32	0.33	0.35	403.57	2.2	5.37	4.5	1.94	5.1
lighttpd	sa,sg,ki,m4,el	7,380	38,006	0.38	0.22	0.40	561.00	2.0	2.77	4.8	1.00	5.5
nginx	sa,ra,ki,m4,ee	26,029	432,829	0.45	0.20	0.35	1482.08	2.8	23.40	9.3	14.90	9.9
openssh	sa,sg,mm,el,ev,ee	14,749	63,644	0.38	0.26	0.36	1725.20	2.1	16.02	3.9	4.37	7.2
exim	sa,sg,ki,ev,ee	37,906	167,867	0.34	0.28	0.38	2588.53	2.2	25.10	4.4	11.05	11.1

Table 1: CFI statistics gathered during the execution of our benchmarks. Function endpoints: sa=`sigaction`, sg=`signal`, ra=`raise`, ki=`kill`, mm=`mmap`, m4=`mmap64`, mp=`mprotect`, el=`execl`, ev=`execv`, ee=`execve`. The CFG group reports the number of nodes and edges in the CFG. The LBR group reports the average number of indirect backward edges, indirect forward edges, and direct forward edges in the LBR. The last three groups compare coarse-grained, fine-grained, and context-sensitive CFI (average number of legal targets and minimum gadget length).

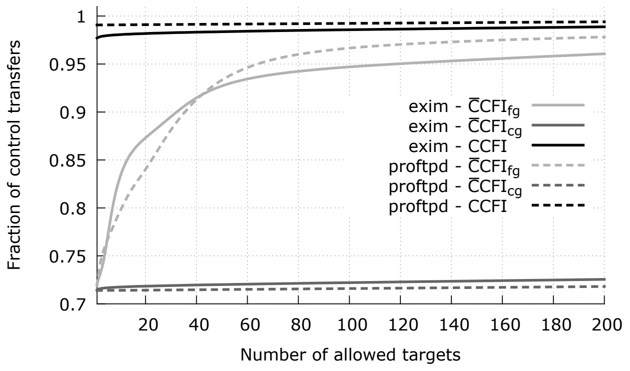

Figure 2: CDF of allowed gadgets for $\overline{\text{C}}$CFI and CCFI (two applications with most sensitive calls).

	#icalls	$\frac{\text{targets}_{cs}}{\text{targets}_{ci}}$
vsftpd	6	0.38
proftpd	120	0.99
pure-ftpd	11	1.00
lighttpd	66	0.84
nginx	271	0.82
openssh	131	0.82
exim	99	0.89
geomean	56	0.78

Table 2: Fraction of legal indirect targets for (ideal binary-level) context-sensitive vs. context-insensitive forward-edge CFI.

grained versions of $\overline{\text{C}}$CFI (respectively, +245% and +53% geometric mean). In general, shorter gadgets are easier to fit together and are more preferred than longer gadgets for building a ROP chain. By reducing the possible indirect edge targets, the attacker's gadget arsenal gets diminished and the bar for exploitation increased. As an example, Table 1 shows that the reduction in the average number of indirect edge targets from 17 to 2.3 for exim resulted in an increase of the average number of instructions in the shortest allowed gadgets from 4.4 to 11. With CCFI, a deeper gadget analysis also revealed a significant increase in the average number of register accesses in the shortest allowed gadgets compared to the coarse-grained and fine-grained versions of $\overline{\text{C}}$CFI. The geometric means of these accesses for the coarse-grained $\overline{\text{C}}$CFI, the fine-grained $\overline{\text{C}}$CFI and CCFI are respectively 1.3, 4.5 and 7.7. This further confirms the increased gadget complexity when using CCFI policies.

To evaluate the effectiveness of the particular CCFI techniques implemented in *PathArmor*, we also examined the impact of context sensitivity on both edges in more detail. For this purpose, we first compared our (static) backward-edge CCFI policy with that enforced by a (dynamic) shadow stack, the only known (run-time) solution which mimics context-sensitive control-flow policies—albeit only on the backward edge and using tamper-prone and more heavyweight instrumentation at the binary level. For a fair comparison, we focused our measurements on the fraction of backward edges observed in the LBR which are allowed only one target (in a fully context-sensitive fashion) by our CCFI techniques and also relied on Intel's BTS feature to simulate an LBR of arbitrary size—overcoming the restrictions imposed by commodity hardware.

Figure 3 presents our results for increasing LBR sizes and the two applications with most sensitive calls (exim and proftpd). We observed similar trends for the other applications. On commodity hardware (16 LBR entries), *PathArmor* can enforce a single target for nearly 75% of the backward edges observed in the LBR. In the remaining cases, the limited LBR size causes *PathArmor* to lose program context and resort to $\overline{\text{C}}$CFI policies. While the current LBR size limit prevents *PathArmor* from fully reaching the ideal shadow stack performance (100%), these results are still encouraging given the small default LBR size. In addition, Figure 3 shows that future hardware extensions can help fill the gap, e.g., enforcing a single target in 90% of cases with 70 LBR entries.

To evaluate the effectiveness of our forward-edge CCFI policy, we examined the reduction in the number of allowed indirect call targets caused by context sensitivity. Due to the very limited number of indirect call entries in the LBR for our test programs (which rarely use indirect calls close to sensitive function points), however, we did not observe any significant reduction in our experiments. To generalize our results and eliminate any application-specific bias, we applied our policy to all the code paths. This still resulted in a relatively small reduction overall (less than 5% in most cases). This is, however, expected, given that our current binary-level forward-edge CCFI policy is very simple—only propagating function pointers passed in call arguments in a

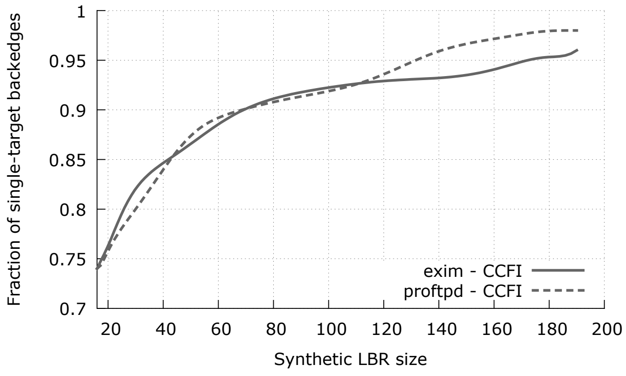

Figure 3: Fraction of single-target backedges for CCFI (two applications with most sensitive calls) when simulating an increasingly large LBR.

straightforward way—and only intended to demonstrate the practicality of implementing arbitrary forward-edge CCFI policies in *PathArmor*. To examine the potential for more sophisticated forward-edge CCFI policies, we approximated an *ideal* binary-level context-sensitive forward-edge analysis using higher-level language semantics—i.e, implemented on top of LLVM 2.9 Data Structure Analysis (DSA) [32].

Table 2 shows the effect of the resulting forward-edge CCFI policy on our set of server programs. The policy causes a significant geometric mean reduction of 22% for the average number of indirect call targets. The reduction varies depending on the context-sensitive function pointer resolution accuracy. For vsftpd, we obtain a reduction of 62%, while numbers decrease for applications with more complex pointer resolutions. We believe these results are encouraging, simulating research on more sophisticated forward-edge CCFI policies—which *PathArmor* can serve as a basis for. Moreover, DSA's flow-insensitive and unification-based design aggressively merges data-flow information, improving speed but also resulting in overly conservative results [32]. In addition, due to implementation limitations, DSA is known to produce even more conservative, and thus pessimistic, results on modern LLVM releases [2]. Thus, an updated version of DSA (or a more precise, but also less scalable analysis) would already likely yield substantially improved forward-edge results.

Overall, our analysis shows that CCFI is effective in generating robust CFI invariants to defend against even sophisticated control-flow diversion attacks. While attacks are still theoretically possible—and they might be even for an ideal CCFI solution—the adoption of context sensitivity sensibly limits the quantity and quality of gadgets available to the attacker. This is in stark contrast, for example, with unrestrictedly allowing simple *call-site gadgets*, which have been used to mount attacks against prior CCFI techniques [27].

5.2 Memory Usage

PathArmor instrumentation increases memory usage at runtime. To evaluate this impact, we measured the physical memory used by instrumented applications compared to the baseline. Deploying our kernel module alone has a constant and marginal memory usage impact (+1 MB). Our static analyzer, in turn, yields a memory usage impact proportional to the size of the CFGs under active analysis, resulting in an increase of +18-74 MB across all our applications.

More important is to assess the memory usage impact of our path caching strategy, given that caching static analysis results is important to minimize the performance impact on instrumented applications. Encouragingly, our measurements indicate a very small memory usage impact induced by our in-kernel path cache, resulting in a worst-case increase of only 2 KB across all our applications during the execution of our benchmarks. This suggests that our path caching strategy is practical even for applications which periodically issue several different sensitive function calls, and even provides evidence that deploying a system-wide path cache that persists across application restarts (thus eliminating cache warmup-phase penalties for applications with strong real-time guarantees) may be a realistic option.

	Time (ms)		**Cache Stats**	
Server	*Total*	*Avg*	*# Misses*	*# Hits*
vsftpd	24	3	9	2,283
proftpd	140	4	39	2,495
pure-ftpd	56	2	27	1,915
lighttpd	28	2	13	2
nginx	24	5	5	10
openssh	52	2	22	49
exim	100	3	40	1,871
geomean	49	3	18	213

Table 3: Path analysis time and cache statistics gathered during the execution of our benchmarks.

5.3 Analysis Time

PathArmor's on-demand path analysis translates to increased application run-time. To evaluate the resulting impact, we measured the time spent in our analyzer—using our CCFI policies—during the execution of our benchmarks. Table 3 presents our results. The second group of columns details the total and average analysis time measured across all the paths analyzed. As shown in the table, the average time spent in our analyzer to inspect each path—with little time variations across paths—is relatively low (3 ms, geometric mean). This demonstrates that our optimizations—pre-normalizing the CFG and recording direct forward edges in the LBR—are effective in implementing a scalable context-sensitive path analysis even for programs with a large and complex CFG. In addition, the total time spent in our analyzer is marginal compared to the total benchmark run time (49 ms, geometric mean vs. several seconds). This shows the effectiveness of our path cache which, as reported in Table 3, was consulted thousands of times with only dozens of misses for most applications. We elaborate on the end-to-end impact of our on-demand path analysis strategy on run-time performance in the next section.

5.4 Run-time Performance

To evaluate the impact of *PathArmor*'s instrumentation and path verification strategy on run-time performance, we measured the time to complete the execution of our benchmarks and compared against the baseline. Table 4 presents our results. The second group of columns details the normalized run-time across a number of *PathArmor* configurations. The *LBR only* configuration refers to *PathArmor* solely de-

	Normalized Run Time				**Event Stats**		
Server	*LBR only*	*+LInstr*	*+CBInstr*	*+PathVer*	*#LCalls*	*#SCalls*	*#Signals*
vsftpd	1.000	1.000	1.000	1.000	35,883	42,446	208
proftpd	1.000	1.000	1.000	1.000	171,440	48,562	6
pure-ftpd	1.003	1.053	1.031	1.074	115,897	57,843	64
lighttpd	1.097	1.236	1.226	1.275	1,209,081	200,564	0
nginx	1.053	1.178	1.168	1.174	1,500,021	200,002	0
openssh	1.003	1.003	1.031	1.020	24,313	720	8
exim	1.025	1.019	1.036	1.079	67,849	4,149	50
geomean	1.025	1.066	1.067	1.085	154,831	28,229	12

Table 4: Run-time normalized against the baseline and stats gathered during the execution of our benchmarks.

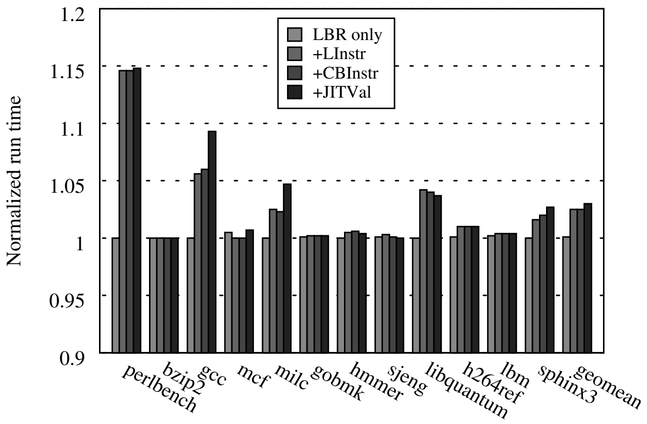

Figure 4: Run-time normalized against the baseline for the SPEC CPU2006 benchmarks.

	#libcalls	**#polluted**	**%polluted**
gcc	3,373,862	13,086,146	24.24
bzip2	449	1,284	17.87
perlbench	60,495,412	253,246,721	26.16
mcf	470,597	5,705,524	75.78
milc	28,807,387	65,657,612	14.24
gobmk	299,877	1,004,581	20.94
hmmer	4,098,071	18,395,790	28.06
sjeng	11,602	176,683	95.18
libquantum	52,609,059	105,222,996	12.50
h264ref	2,449,569	12,515,117	31.93
lbm	2,626,460	5,263,308	12.52
sphinx3	48,625,654	187,711,907	24.13
geomean	1,604,689	6,595,149	25.68

Table 5: LBR pollution caused by library calls for SPEC CPU2006. #libcalls=overall library calls, #polluted=overall polluted LBR entries, %polluted=LBR entries polluted (avg).

ploying its kernel module and saving/restoring the current LBR state at application thread context switching time. As shown in the table, this configuration introduces marginal performance impact (2.5%, geometric mean). The overhead is somewhat more pronounced in the *+LInstr* and *+CBInstr* configurations (6.6% and 6.7%, geometric mean), which additively account for our library entry point and callback instrumentation (respectively), but omit the path verification step in our kernel module. The *+PathVer* configuration, finally, refers to the default *PathArmor* setup, enabling full instrumentation and path verification using our on-demand static analyzer. As shown in the table, our cache-aware path analysis has relatively little impact on run-time performance (+1.7%, geometric mean), resulting in the final average run-time overhead of 8.5% (geometric mean).

To shed some light on the key factors contributing to the performance overhead, we also instructed *PathArmor* to report statistics on the run-time events of interest, as shown in the third group of columns in Table 4. Our results confirm that library calls (*#LCalls*) are the most prevalent contributing factors in the mean case, also inducing the worst-case performance impact on lighttpd (27.3%). More aggressively instrumented operations like callback invocations (marginal, not reported in table), sensitive function calls (*# SCalls*) and signals (*#Signals*) have a less prominent impact and can thus be better amortized over the execution.

To obtain standard and comparable performance results across *PathArmor*'s configurations, we also measured the time to complete all the C programs in the SPEC CPU2006 benchmarks and compared against the baseline. Figure 4 presents our findings. Our results confirm the general behavior observed for our server applications, but the performance overhead is generally much lower (3% in *PathArmor*'s default configuration, geometric mean). This result stems from the lower number of library and system calls issued by SPEC programs, as expected for standard CPU-intensive (as opposed to syscall-intensive) benchmarks.

Overall, *PathArmor* imposes a relatively low run-time performance impact on all the test programs considered. This confirms that *PathArmor*'s lightweight instrumentation and cache-aware path analysis are successful in producing a run-time overhead comparable to the most efficient (source-level and forward-edge only) $\overline{\text{C}}$CFI techniques [47], while enforcing much more advanced context-sensitive CFI policies on both the forward and backward edge and operating entirely at the binary level.

5.5 LBR Pollution

As discussed in Section 3, *PathArmor*'s design supports two modes of operation: (i) stop tracking branches at the library level, or (ii) continue tracking within libraries. The current implementation of *PathArmor* uses the first mode by default, effectively increasing the control flow context of the protected binary during path verification. To also protect against control flow diversion triggered within library code, *PathArmor* can be configured with the second mode of operation. When running in this mode, branch tracking is never disabled at the cost of (partially) "polluting" the LBR from (self-contained) library code.

To evaluate the LBR pollution cost of running in full-library mode, we configured *PathArmor* to compare LBR contents right before and right after each library call and reran the SPEC CPU2006 benchmark. Table 5 shows the results. The average pollution rate of 25.68% overall (geometric mean) is likely acceptable in environments where untrusted, potentially vulnerable libraries are in place.

Tracking inside libraries leads to better performance, as this removes the jump to kernel during program-library transitions. Thus, as mentioned earlier, the results provided in this section show worst-case performance. As discussed above, the tradeoff of in-library tracking is increased LBR pollution, which, however, can also be mitigated with complementary techniques, such as inlining library code or using hardware that provides a larger branch record.

6. DISCUSSION

This paper outlined and evaluated the design decisions made in *PathArmor*. We now discuss evasion techniques an attacker may employ to bypass *PathArmor*, analyzing their impact and the limitations of our current solution.

6.1 History-flushing Attacks

An attacker may attempt to mount a *history-flushing attack* to clear any traces of a ROP chain from the LBR. History-flushing attacks previously described in the literature first execute 16 innocuous *NOP-like gadgets* followed by a long *termination gadget* that restores argument registers and ultimately performs a security-sensitive system call [16]. The long termination gadget bypasses heuristics used in prior LBR-based solutions such as kBouncer [36] and ROPecker [19], which rely on weak security invariants based on gadget size (which they assume to be small) and frequency.

PathArmor is not vulnerable to this simple attack, as history flushing in *PathArmor* is equivalent to the attacker crafting a valid CCFI-permitted path of 16 NOP-like gadgets (using direct calls or indirect branches). This is much more difficult than chaining arbitrary and CFG-agnostic gadgets. In other words, the notion of a path in *PathArmor* is stronger than that of regular (context-insensitive) CFI and much stronger than that of kBouncer and ROPecker. Hence, while history-flushing attacks generally remain of concern, *PathArmor*'s stronger invariants significantly raise the bar for the attacker. For example, we have shown in Section 5.1 that it is generally much harder to maintain register states over that many branches.

A related attack is to force context switches to clear the LBR and indirectly mount a history-flushing attack. This attack is also ineffective against *PathArmor*, given that, as outlined in Section 4, *PathArmor* stores and restores LBR states during context switches on a per-thread basis.

6.2 Non-control Data Attacks

An attacker may attempt to mount a non-control data attack to indirectly influence the execution of existing security-sensitive functions in the program without directly diverting control flow. For example, an attacker can exploit an arbitrary memory write vulnerability to overwrite sensitive function arguments that are maintained in a data region. Similarly to all the existing (and even ideal) CFI solutions, *PathArmor* cannot protect against these and other data-only attacks. Unlike existing whole-program CFI solutions, however, *PathArmor*'s history-based strategy would also allow an attacker to craft a ROP-based memory write primitive before jumping to the beginning of a valid execution path leading to a security-sensitive function. Nevertheless, since ROP is not necessary to perform an attacker-controlled memory write and arbitrary memory write vulnerabilities are actually very common, we do not believe this is a limiting factor within our threat model. We also note that binary-level defenses against non-control data attacks are explored in orthogonal work [44].

6.3 Endpoint-pruning Attacks

An attacker may attempt to evade detection by avoiding calls to sensitive endpoints recognized by *PathArmor*. This is because, similarly to prior endpoint-driven solutions [19, 36], *PathArmor* enforces security invariants only at predetermined sensitive function calls. Assuming *PathArmor*'s default configuration, such *endpoint-pruning* attacks require the attacker to find alternative means to affect the system environment without relying on system calls such as `exec`, and `mprotect`. While this is generally of concern depending on the goals of the attacker, *PathArmor* allows users to configure the list of sensitive endpoints according to their needs. For programs in which our default configuration is not sufficient to provide the required guarantees, users can custom tune the list of endpoints and balance security and run-time performance.

Nevertheless, we believe that *PathArmor*'s default configuration alone drastically reduces the freedom of an attacker. Although ROP may still be used to perform arbitrary Turing-complete computations, without the ability to execute core security-sensitive system calls, the impact on the system remains limited.

6.4 Instrumentation-tampering Attacks

An attacker may attempt to abuse the instrumentation employed by *PathArmor*'s default mode of operation (which disables branch tracking in library code) to alter the branch record. Nevertheless, this attack would still fail to circumvent *PathArmor*'s detection strategy. Consider the scenario wherein an attacker sets up a ROP chain that invokes the `ioctl` system call with a dedicated *PathArmor*-specific argument to tamper with the branch-tracking instrumentation. Depending on the request type, this attack will result in two possible outcomes. In the case of a `CALLBACK_EXIT` request, *PathArmor*'s kernel module will immediately verify the current LBR state (see Section 3.3.3) and detect CCFI invariants violations caused by the originating ROP-based control flow. In the case of a `LIB_ENTER` request, in turn, *PathArmor*'s kernel module will immediately return control to userland after disabling branch tracking, allowing the attack to resume in LBR-free execution. As soon as the attacker invokes a security-sensitive function, however, *PathArmor*'s kernel module will perform verification as normal. At that point, the LBR state will still reflect the branch record generated by the attacker's original ROP chain (leading to the previously issued `ioctl` system call), resulting, again, in *PathArmor* detecting the attack. Note that an attacker can also attempt to later re-enable branch tracking via a `LIB_EXIT` operation, but a *PathArmor*-legal path of 16 indirect branches is then required to clear any traces of the original ROP attack—essentially equivalent to the history-flushing attacks discussed earlier.

7. RELATED WORK

CFI was originally proposed by Abadi et al. [8]. The original (strict) CFI proposal incurs high overheads. This has lead to a myriad of proposals for practical CFI implementations which realize better performance by strategically trading off security guarantees. There are two broad branches of CFI implementations: (i) Control-Flow Graph-based (CFG-based) CFI, and (ii) Heuristic-based CFI.

CFG-based CFI focuses on enforcing properties of the CFG. Compiler-based approaches inherently require source to resolve (indirect) control transfers that are considered legitimate [8, 9, 12, 22, 25, 33, 49, 52]. Due to the availability of source information, these approaches are usually able to derive accurate CFGs. Binary-based approaches, while potentially less accurate (i.e., based on an overapproximated CFG), have the advantage of being applicable to legacy programs where the source code is not available [29, 50, 53–55]. Recently, modular CFI approaches have also been proposed. These are a variant of CFG-based approaches, which resolve part of the CFG at runtime, providing greater flexibility for dynamically computed targets [34, 35, 37].

In contrast to CFG-based CFI, heuristic-based CFI does not require a CFG to enforce integrity. Such approaches include kBouncer [36] and ROPecker [19], which seek to detect anomalous control patterns at sensitive program points. Such approaches are easy to deploy, but are also relatively easy to circumvent, due to their heuristics [27].

Prior work explored devastating attacks against both prior CFG-based and heuristic-based CFI, using combinations of individually legal control transfers [16, 24, 27]. *PathArmor* enables stronger defenses against such attacks by efficiently enabling context-sensitive CFI policies over paths to sensitive functions and disallowing many unnecessary forward and backward edges permitted by prior context-insensitive CFI policies (e.g., backward edges to arbitrary call-site gadgets, a common attack target [27]).

In prior fine-grained CFI techniques, context-sensitive policies have been explored only for backward edges and only using shadow stacks [11, 18, 20, 21, 23, 25, 38, 39, 43, 51].

In contrast to the run-time shadow stack approach, *PathArmor* resolves backward edges using a hardware-supported context-sensitive static analysis over the interprocedural CFG and caches the results at sensitive points in the program, yielding improved performance and security against tampering attacks. Static context-sensitive backward edge resolution strategies have been explored before for security, but only to improve the accuracy of IDS models based on syscall sequences [48]. *PathArmor*, in contrast, shows that enforcing context-sensitive CFG-based policies both on the forward and backward edge at a much finer level of granularity (i.e., control-flow transfers for CFI) is a realistic and efficient option thanks to emerging hardware features. This result contrasts claims in prior work, which, while acknowledging their security advantages, generally dismissed context-sensitive CFI policies as impractical for real-world adoption [8].

Other approaches rely on hardware-supported branch tracing to improve CFI performance. Similar to *PathArmor*, kBouncer [36] and ROPecker [19] rely on Intel's LBR to efficiently implement branch tracing, but only to enforce heuristic CFI policies which can be easily circumvented [16]. CFIMon [50] can enforce hardware-supported CFG-based CFI policies, but relies on the significantly slower Intel BTS [36] and yields high detection latencies, potentially missing attacks [19]. Unlike *PathArmor*, none of these approaches attempt to enforce context-sensitive policies over hardware-monitored control transfers.

Concurrently with our work, Schuster et. al. have developed the COOP attack [40], and show that CFI solutions that do not precisely consider object-oriented semantics in C++ programs can generally be bypassed. While our work mainly focuses on C rather than C++ programs, we believe CCFI can strengthen forward-edge invariants (subject to the precision of the underlying data-flow analysis) in modern vtable protection techniques in mainstream compilers [47], raising the bar against COOP-like attacks.

The recent Control-Flow Bending (CFB) [15] evaluates the general effectiveness of even ideal (context-insensitive) CFI solutions and evidences their limitations against sophisticated CFG-aware attacks. Compared to regular CFI, CCFI makes such attacks harder, given that entire paths (rather than individual CFG edges) are checked for validity. CFB attacks have already been shown to be more difficult against CFI solutions that are complemented by a shadow stack [15]. Compared to such solutions, CCFI does not rely on in-process run-time information and can enforce context-sensitive invariants on both forward and backward edges, thereby providing improved defenses against CFB attacks.

8. CONCLUSION

While Context-sensitive CFI (CCFI) can significantly enhance the security of state-of-the-art defenses against control-flow diversion attacks, it has long been perceived as inefficient and impractical for real-world adoption. This paper has shown that the three fundamental challenges towards fast and practical CCFI—efficient path monitoring, analysis, and verification—can indeed be effectively addressed in a realistic way on commodity platforms.

To substantiate our claims, we implemented *PathArmor*, the first binary-level CCFI solution that efficiently enforces context-sensitive CFI policies on both backward and forward edges. *PathArmor* addresses all the CCFI fundamental challenges using low-overhead hardware registers to track control edges, a scalable on-demand and constraint-driven context-sensitive static analysis, and a path cache accessed at sensitive program points. *PathArmor* yields comparable or better performance than prior context-insensitive CFI solutions, while enforcing much stronger context-sensitive invariants and providing a general framework to implement arbitrarily sophisticated CCFI policies.

Acknowledgements

We would like to thank our shepherd, Mathias Payer, and the anonymous reviewers for their valuable comments and input to improve the paper. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571, by the European Research Council through project ERC-2010-StG 259108 "Rosetta", and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO CSI-DHS 628.001.021.

9. REFERENCES

[1] Apache benchmark. `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] LLVM DSA - Reproduce the Result in PLDI 07 Paper. `http://lists.cs.uiuc.edu/pipermail/llvmdev/2015-May/085390.html`.

[3] OpenSSH portable regression tests. `http://www.dtucker.net/openssh/regress`.

[4] pyftpdlib. `https://code.google.com/p/pyftpdlib`.

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

[6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM CCS*, 2005.

[7] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control-flow. In *ICFEM*, 2005.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM TISSEC*, 13(1), 2009.

[9] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE S&P*, 2008.

[10] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *PASTE*, 2011.

[11] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX SEC*, 2005.

[12] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *ACSAC*, 2011.

[13] E. Bosman and H. Bos. Framing signals—A return to portable shellcode. In *IEEE S&P*, 2014.

[14] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *IJHPCA*, 14(4), 2000.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX SEC*, 2015.

[16] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX SEC*, 2014.

[17] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM CCS*, 2010.

[18] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[19] Y. Cheng, Z. Zhou, M. Yu, X. Ding, , and R. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *NDSS*, 2014.

[20] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS*, 2001.

[21] M. L. Corliss, E. C. Lewis, and A. Roth. Using DISE to protect return addresses from attack. In *ASSAV*, 2004.

[22] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE S&P*, 2014.

[23] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ASIACCS*, 2015.

[24] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX SEC*, 2014.

[25] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[26] I. Fratric. Runtime prevention of return-oriented programming attacks, 2012. Technical report.

[27] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[28] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX SEC*, 2014.

[29] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX SEC*, 2002.

[30] S. Krishnamoorthy, M. Hsiao, and L. Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *IEEE ATS*, 2010.

[31] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[32] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

[33] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM CCS*, 2013.

[34] B. Niu and G. Tan. Modular control-flow integrity. In *PLDI*, 2014.

[35] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM CCS*, 2014.

[36] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX SEC*, 2013.

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

[38] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX ATC*, 2003.

[39] B. G. Roth and E. H. Spafford. Implicit buffer overflow protection using memory segregation. In *ARES*, 2011.

[40] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming. In *IEEE S&P*, 2015.

[41] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *RAID*, 2014.

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

[43] S. Sinnadurai, Q. Zhao, and W.-F. Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2004. Technical report.

[44] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[45] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE S&P*, May 2013.

[46] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting hardware advances for software testing and debugging (nier track). In *ICSE*, 2011.

[47] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Úlfar Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC and LLVM. In *USENIX SEC*, 2014.

[48] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE S&P*, 2001.

[49] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S&P*, 2010.

[50] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE DSN*, 2012.

[51] Y. Younan, D. Pozza, F. Piessens, and W. Joosen. Extended protection against stack smashing attacks without performance loss. In *ACSAC*, 2006.

[52] B. Zeng, G. Tan, and Ú. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX SEC*, 2013.

[53] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control-flow integrity and randomization for binary executables. In *IEEE S&P*, 2013.

[54] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *VEE*, 2014.

[55] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX SEC*, 2013.