StreamCFI: Streamed Per-Input Control-Flow Integrity

Konstantinos Kleftogiorgos
Stevens Institute of Technology
Hoboken, NJ, USA
kkleftog@gmail.com

Cristiano Giuffrida
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
c.giuffrida@vu.nl

Georgios Portokalidis
IMDEA Software Institute
Madrid, Spain
georgios.portokalidis@imdea.org

***Abstract*—Control-Flow Integrity (CFI) mitigates code-reuse attacks by restricting indirect control-flow transfers to a predetermined set of valid targets. While prior work has demonstrated that finer-grained, dynamic policies can provide stronger security guarantees, enforcing such policies typically incurs substantial performance overhead. Recent approaches have explored offloading security checks to parallel monitors as a means to reduce this cost. In particular, Software-Driven Logging (SDL), enabled by processor extensions such as Intel Processor Trace (PT), has shown promise for secure and efficient offloading of CFI enforcement to a separate monitor process. However, existing SDL-based solutions are limited to conventional, statically-derived CFI policies (e.g., LLVM-CFI) and only marginally improve their performance.**

We present StreamCFI, an SDL-based framework that efficiently enforces dynamic CFI through a hybrid approach combining type-based and per-input policies. StreamCFI is built for x86-64 and leverages Intel PT along with the PTWRITE instruction to capture run-time metadata, introducing several novel optimizations to minimize logging overhead. Our evaluation on SPEC CPU2017 benchmarks shows that StreamCFI reduces geometric-mean overhead from 26.67% to 8.63% compared to the state-of-the-art per-input CFI solution, while reducing code-size overhead by 5×. Notably, StreamCFI makes dynamic CFI practical for deployment on commodity hardware, remaining reasonably close to LLVM-CFI's 3.45% geometric-mean overhead while providing stronger security by reducing the set of allowed targets at run time.

1. Introduction

Memory-unsafe languages, such as C and C++, are ubiquitous in performance-critical systems despite their inherent susceptibility to memory errors [1], [2]. These errors, including buffer overflows and use-after-free vulnerabilities, enable attackers to corrupt code pointers, such as function pointers and virtual table pointers, and thereby manipulate program control flow. Although defenses like Data Execution Prevention (DEP) and W^X policies [3], [4] have curtailed direct code injection, adversaries have transitioned to code-reuse attacks such as return-oriented programming (ROP) [5], [6], jump-oriented programming (JOP) [7], [8], and counterfeit object-oriented programming (COOP) [9]–[11]. These sophisticated attacks not only bypass traditional defenses like stack canaries [12] and address space layout randomization (ASLR) [13], but also exploit subtle info leaks [14] and even speculative execution vulnerabilities [15]–[17].

To counter such threats, Control-Flow Integrity (CFI) solutions seek to ensure that a program's execution adheres to a predetermined (static) control-flow graph (CFG) [18], [19]. Early CFI implementations used very coarse-grained strategies, placing all address-taken functions into a single equivalence class. While efficient, this approach left significant room for exploitation [20], [21]. Subsequent techniques, such as type-based CFI (e.g., LLVM-CFI) [22], [23], improved precision by grouping functions by signature. Nonetheless, type-based CFI solutions still suffer from static analysis overapproximations, allowing attackers to target any function sharing the same type [24].

To provide stronger security guarantees, dynamic CFI techniques incorporate additional runtime context (e.g., call-site history, pointer origin, etc.) to narrow the set of legitimate targets [25]–[28]. A notable example is πCFI [29], which introduces a *per-input* CFI policy labeling indirect call targets as initially unreachable and thus *inactive* at program startup. These targets are dynamically *activated* during execution when a function address is assigned to a function pointer. While this approach significantly reduces the attack surface (i.e., to the activated targets), it also relies on complex sandboxing and extensive inline instrumentation, with nontrivial performance overhead.

To mitigate the performance overhead of inline CFI enforcement, researchers have proposed *decoupled* mechanisms that offload security checks from the application to an external monitor or specialized hardware. Hardware-assisted solutions [30]–[32] enable efficient and secure enforcement, but their reliance on custom hardware support limits practicality and widespread deployment. In response, several efforts [33]–[36] have leveraged existing hardware tracing features, such as Intel Processor Trace (Processor Trace) [37], to implement CFI with lower overhead on commodity systems. Among these, some [33], [38] rely on execution tracing to offload static CFI checks, but cannot support additional run-time context required by more precise, dynamic CFI policies. In con-

trast, SideCar [36] adopts *Software-Driven Logging* (SDL), enabling applications to selectively emit compact, security-relevant metadata at run time. This approach has proven effective in reducing the overhead of high-cost CFI checks, but prior work has focused exclusively on static policies. Whether SDL can be extended to support dynamic CFI enforcement, such as the per-input policy proposed in πCFI [29], remains an open question. This paper addresses that gap.

In this paper, we present StreamCFI, a framework that combines per-input CFI with classic type-based CFI and offloads enforcement to an external monitor using selective SDL-based instrumentation. StreamCFI targets a best-of-both-worlds design point, marrying the strong security of dynamic CFI with the low overhead and improved isolation of decoupled enforcement on commodity hardware.

For this purpose, StreamCFI builds on the SideCar framework and leverages Intel PT's `PTWRITE` instruction to log security-relevant events efficiently. It selectively emits metadata for both function pointer assignments and indirect function calls, enabling fine-grained policy enforcement without inline checks or complex sandboxing, as required by πCFI [29]. By offloading enforcement to a parallel monitor, StreamCFI ensures that a target function is invoked only if its activation has been previously recorded. This enables adaptive enforcement based on dynamic context while keeping run-time logging lightweight. To further reduce overhead, we introduce several novel optimization techniques that minimize the volume and frequency of `PTWRITE` operations. Specifically, we compress each log entry from 64 to 32 bits by using the instruction's short encoding format and exploit an underutilized feature of `PTWRITE` that triggers the emission of Follow-Up Packets (FUPs), allowing additional metadata to be transmitted without extra `PTWRITE` instructions.

Our evaluation shows that StreamCFI reduces geometric-mean overhead from 26.67% to 8.63% compared to πCFI on the SPEC CPU2017 integer benchmarks, and it reduces code-size overhead by 5×. StreamCFI also remains close to the state-of-the-art CFI implementation in LLVM (LLVM-CFI), which incurs 3.45% geometric-mean overhead, while reducing the set of allowed targets at run time. Finally, our evaluation on the RIPE64 [39] benchmark suite confirms that StreamCFI reliably detects attacks using function pointers without introducing false positives.

Contributions. We make the following contributions:

- We design a fully decoupled, dynamic CFI mechanism that combines a per-input target activation policy with type-based checks. Our design relies on SDL to enforce forward-edge CFI, improving isolation and reducing the attack surface compared to pure type-based approaches.
- We develop an LLVM instrumentation pass in the SideCar framework to detect function pointer assignments, extract metadata, and insert instrumentation to efficiently log function activation events at assignments and invocation events at indirect calls via SDL.
- We introduce optimizations to compactly encode CFI metadata in SDL logs, reducing bus load with the 32-bit short encoding format of `PTWRITE` instructions and transferring additional metadata with the FUP packet capabilities of Intel PT.
- We implement StreamCFI on x86-64 (validated with RIPE64), evaluate its performance on SPEC CPU2017, and compare it against πCFI and LLVM-CFI.

2. Background and Motivation

2.1. Control-Flow Integrity

Control-Flow Integrity (CFI), first introduced by Abadi et al. [18], is a security mechanism that ensures a program's run-time control transfers adhere strictly to a statically derived control-flow graph (CFG). This prevents adversaries from redirecting control flow to arbitrary locations (and code gadgets), even if they are able to corrupt function pointers in memory. For example, consider the code snippet in Listing 1, where the global variable `fdevents` (line 6) holds the function pointer `event_set`. Depending on the configuration of the Lighttpd server, this pointer may be assigned either `fdevent_poll_event_set()` or `fdevent_select_event_set()`, both of which are potential call targets at line 32. CFI mechanisms instrument the program to constrain the possible targets of such indirect calls, ensuring that the jump at line 32 can only transfer control to valid, pre-identified functions.

Over the years, numerous CFI techniques have been proposed [40], each enforcing different levels of strictness and incurring various trade-offs. In this paper, we categorize these techniques into two broad classes: (i) approaches based on purely static policies, and (ii) dynamic approaches that impose stricter enforcement based on run-time program behavior.

CFI with Static Policies. Static CFI techniques [20]–[22], [41]–[43] derive a program's CFG, and the corresponding enforcement policy, either through static analysis or at compile time. *Coarse-grained* approaches [20], [21] enforce broad policies that permit any function whose address is taken to serve as a valid target for all indirect control transfers. In the example from Listing 1, this means the indirect call at line 32 would be allowed to target any address-taken (AT) function. In contrast, *fine-grained* techniques [22], [42] aim to reconstruct and enforce a more precise CFG by analyzing control-flow relationships more thoroughly. For instance, the CFI implementation in the Clang compiler toolchain [44] groups functions into equivalence classes (ECs) based on their signatures (*type-based CFI*). Indirect calls are then allowed only to targets within the same EC. Returning to our example, the indirect call in line 32 would be permitted to jump only to `fdevent_poll_event_set()` or `fdevent_select_event_set()`, since both share the same signature and thus belong to the same EC. Clang's CFI offers the additional advantages of sup-

```
1  /* fdevents structure uses function pointer to support
       different event handlers in lighttpd */
2  struct fdevents {
3      int (*event_set)(struct fdevents *ev, fdnode *fdn,
           int events);
4      ...
5  };
6  typedef struct fdevents fdevents;
7
8  /* Possible values for event_set */
9  static int fdevent_poll_event_set(fdevents *ev, fdnode
       *fdn, int events);
10 static int fdevent_select_event_set(fdevents *ev, fdnode
       *fdn, int events);
11
12 /* Value of event handler function determined by
       server.event-handler configuration option */
13 fdevents *fdevent_init(const char *event_handler, ...)
14 {
15     ...
16     int type = fdevent_config(&event_handler, errh);
17     switch(type) {
18         case FDEVENT_HANDLER_POLL:
19             ev->event_set = fdevent_poll_event_set;
20             break;
21         case FDEVENT_HANDLER_SELECT:
22             ev->event_set = fdevent_select_event_set;
23             break;
24     }
25     ...
26 }
27
28 /* Event handler function pointer is used here */
29 static void fdevent_fdnode_event_setter(fdevents *ev, ...) {
30     ...
31
32     if (0 == ev->event_set(ev, fdn,
           events) /* Indirect call */
33         || fdevent_fdnode_event_setter_retry(ev, fdn,
               events))
34     {
35         fdn->events = events;
36     }
37 }
```

Listing 1: A Motivating Example from Lighttpd

porting dynamically shared objects (i.e., libraries) and introducing minimal performance overhead.

CFI with Dynamic Policies. While static CFI policies can be effective, they are often overly permissive in practice. Prior research has demonstrated that such policies can be circumvented by sophisticated control-flow attacks [10], [11]. To address these limitations, CFI mechanisms have evolved to incorporate run-time execution state in order to further restrict valid indirect control-flow targets. Early dynamic CFI solutions [45], [46] refined equivalence classes (ECs) using the calling context at each indirect call site, while subsequent approaches [25], [28], [47] combined points-to analysis with dynamic context to increase precision.

Despite their increased strictness, these techniques introduce complexity in both design and implementation, which can lead to practical challenges. As shown in [48], such complexity may cause false positives, where legitimate targets are incorrectly blocked, compatibility issues with complex applications, and even an expanded attack surface due to the added instrumentation code.

A prominent example of a dynamic CFI implementation is πCFI [29], which enforces a *per-input* CFI policy to exclude function pointers that are never instantiated at run time from the set of allowable targets. In Listing 1, for instance, only one assignment in `fdevent_init()` executes, depending on server configuration, thus activating only one target function. Consequently, if line 19 executes, the indirect call at line 32 will only be permitted to call `fdevent_poll_event_set()`. πCFI achieves this by dynamically activating targets upon first assignment (e.g., in line 19). Nonetheless, while this leads to more precise and correct ECs, adjusting them safely at run time requires pervasive inline instrumentation to operate software-fault isolation (SFI), which introduces complexity and nontrivial performance overhead.

2.2. Offloading Security Checks

While strong CFI policies offer substantial security benefits, they often impose increased run-time overhead due to additional instrumentation. To reduce this overhead, and particularly the latency introduced by the added instrumentation, prior work [31], [32], [49] has explored offloading expensive security checks to external monitors running in parallel with the application, either on spare cores or specialized hardware.

Focusing specifically on CFI, two classes of approaches avoid the need for new hardware by leveraging existing processor features originally designed for debugging and tracing: *Execution tracing* and *Software-Driven Logging* (*SDL*).

Execution tracing, supported by Intel Processor Trace [37] (PT) and Arm CoreSight [50], has been used by prior work [33], [38] to offload CFI enforcement. These mechanisms capture fine-grained control-flow data with minimal impact on performance by logging it to OS-managed memory buffers. Monitors decode these traces in real time to reconstruct execution paths and apply CFI checks. However, due to the high volume of data produced, these approaches typically require either hardware accelerators or multiple decoding threads, increasing resource usage.

SDL, as introduced in SideCar [36], takes a different approach by explicitly logging only security-relevant metadata via enhanced debug features such as the System Trace Macrocell (STM) on Arm and the `PTWRITE` instruction on Intel processors. Applications are lightly instrumented to transmit this metadata securely to the processor, which then routes it to kernel-managed buffers accessible by the monitor. Although SideCar still requires instrumentation, it is lightweight and selective, producing substantially less data than execution tracing. This enables more efficient parallel monitoring with lower overhead and reduced resource demands.

2.3. Offloading Dynamic CFI with SDL

Dynamic CFI mechanisms, such as πCFI [29], enforce stricter and more context-sensitive security policies, but incur increased run-time overhead. We observe that software-driven logging (SDL), is particularly well-suited for accelerating the expensive inline operations, required by such schemes. Although additional metadata, such as the activation of function

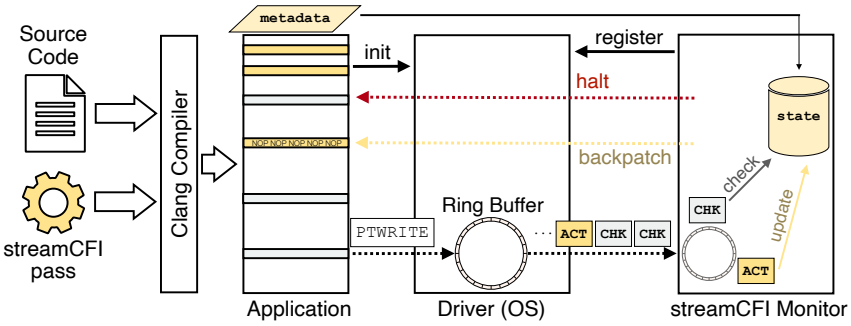


Figure 1: StreamCFI architecture, extending SideCFI (base components shown in gray) over SideCar. StreamCFI additions for per-input CFI (highlighted in yellow in the diagram) include its dedicated compiler pass, `ACTIVATE` messages, and an enhanced monitor. The monitor uses a unified hash table for activation status and type information, enabling a combined per-input and type-based CFI policy.

pointers in πCFI, must be transmitted to the monitor, these updates can be processed asynchronously and in parallel, without introducing latency on the application's execution path. Crucially, offloading this logic to the monitor eliminates the need for SFI on the application side, enabling a simpler and more efficient design. As a result, SDL provides an effective and practical foundation for enforcing high-precision, dynamic CFI policies with reduced overhead.

3. Threat Model

We consider an adversary exploiting memory vulnerabilities, such as buffer overflows and use-after-free errors, to hijack the control flow of user-space C/C++ applications. Our focus is on securing forward-edge control-flow transfers (i.e., indirect function calls) through dynamic Control-Flow Integrity (CFI) enforcement. We assume that backward-edge transfers (e.g., function returns) are protected by mechanisms like shadow stacks or hardware-assisted features such as Intel's Control-flow Enforcement Technology (CET) [51], which provide robust return address protection.

The system operates on an x86 platform with standard security measures, including non-executable memory (enforcing a W^X policy), Address Space Layout Randomization (ASLR) [13], and stack-smashing protection. These defenses are complementary to our approach. We also assume that the hardware supports Intel Processor Trace (PT) and specifically the `PTWRITE` instruction. Furthermore, the operating system is assumed to protect these trace streams from tampering, ensuring the integrity of the logged data. While the adversary may have capabilities such as arbitrary memory reads and writes, we assume they cannot compromise the hardware or the operating system.

4. StreamCFI

StreamCFI is a decoupled CFI enforcement solution that enhances a type-based policy with per-input checks, resulting in a dynamic CFI policy with low runtime overhead. As depicted in Figure 1, StreamCFI's design revolves around three key components: (i) an application analyzed and instrumented by a compiler pass, (ii) a privileged driver for secure communication, and (iii) a dedicated monitor. The compiler instrumentation emits lightweight SDL messages for security-relevant events: `ACT` messages (Figure 1) signal function pointer activations when function addresses are taken, while `CHK` messages request the monitor to verify both activation status and type compatibility at indirect call sites. The instrumentation relies on the `PTWRITE` instruction to efficiently emit these messages and stream them to the monitor for CFI policy enforcement. Building on the SideCar platform, StreamCFI implements an efficient (asynchronous) enforcement model. Upon detecting a policy violation, the monitor promptly halts the application (the 'halt' signal in Figure 1) to prevent further malicious activity.

For the sake of simplicity, the following sections first detail a baseline design for StreamCFI's components. Later, we describe key optimizations—such as backpatching instrumentation for activated targets and efficient message encoding—that further reduce the performance impact and make StreamCFI a practical solution for robust, dynamic CFI.

4.1. Driver

The StreamCFI driver provides a dynamic CFI run-time service to applications. Any application wishing to use it must first issue an initialization (*init*) request to the driver, which activates tracing in the hardware for the cores running the application's threads. The driver is also responsible for setting up the communication channel between the application and the monitor. Specifically, it allocates a ring buffer in the kernel for each processor core. These buffers are then used by the hardware to log SDL messages. Each buffer is memory-mapped into the monitor's address space, enabling direct access to logged messages. The driver also tracks dynamic library loading and unloading events, in order to store metadata such as the library path and base address, accessible via the library's handle.

The driver is policy-agnostic, i.e., it is unaware of the specific CFI policy or SDL message semantics. Its role is to facilitate reliable communication and prevent message loss, including slowing down application

threads when buffers are near capacity. In addition, the driver handles privileged operations on behalf of the monitor, such as terminating (*halting*) the application upon a violation.

4.2. Compiler Pass

StreamCFI's compiler pass instruments the program to emit SDL messages in response to security-sensitive operations. These messages are processed by the monitor to enforce CFI policies. Below, we describe the program operations that are instrumented and the type of message used for each operation.

Indirect Function Calls. Indirect function calls are the primary targets for (forward-edge) CFI enforcement. StreamCFI instruments each indirect call site, such as the one shown at line 32 of Listing 1, to emit a `CHECK(typeID, addr)` message. This message encodes both the dynamic address of the indirect call target (`addr`) and a unique, static identifier (`typeID`) representing the expected function type. For instance, the call at line 32 has the signature `int(fdevents *, fdnode *, int)`, which determines its `typeID`. The message is emitted right before the indirect call executes.

C++ Virtual Calls. C++ virtual calls represent a common class of indirect calls where the actual target is resolved based on the run-time type of an object. These calls are typically dispatched via virtual tables (vTables), which store function pointers corresponding to virtual methods. Each object contains a vtable pointer (vptr), which the compiler uses to resolve the function at a fixed offset. Similar to other indirect calls, StreamCFI instruments virtual calls by emitting a `CHECK(typeID, addr)` message. `addr` corresponds to the value of the vptr being loaded, while `typeID` is derived from the object's static class type. This is sufficient to validate the subsequent call target.

Activation of Function Pointers. StreamCFI tracks function pointer activation dynamically, assuming that the program computes a usable pointer, such as at lines 19 and 22 of Listing 1, before the function can be invoked via an indirect call. Every time a function address is copied or loaded into a variable, StreamCFI emits an `ACTIVATE(addr)` message. This message informs the monitor that the function at address `addr` is now a valid target for future indirect calls. In our example, the function pointers are assigned to `ev->event_set` within the body of function. However, assignments can also occur during the initialization of global variables. For each function pointer initialized globally, StreamCFI emits a corresponding `ACTIVATE` message at load time. This ensures that all such global function pointers are activated before the application begins execution. Note that, although function addresses are symbolic during compilation, they resolve to constant values in the final binary after linking. These addresses may also be adjusted at load time by the operating system's loader.

Activation of C++ vTables. StreamCFI enforces the security of virtual calls by checking that the vtable being used is correct. To this end, StreamCFI also tracks vtable activation. During object construction, StreamCFI instruments class constructors to emit an `ACTIVATE(addr)` message, where `addr` is the address of the corresponding vtable. This message enables the monitor to validate subsequent virtual calls by ensuring that only methods from activated vTables are considered valid targets. This strategy ties call validity directly to the instantiation of objects, preserving both precision and compatibility with polymorphic behavior.

Dynamic Library Events. StreamCFI monitors dynamic library loading and unloading events to ensure that indirect control transfers to and from shared libraries are handled securely at run time. In UNIX-like systems, libraries are managed through the following standard interfaces:

```
void *handle = dlopen(const char* path, int mode);
int r = dlclose(void *handle);
```

StreamCFI intercepts these interfaces and augments their behavior by emitting dedicated messages to the monitor:

- `DLOPEN(handle)`: Informs the monitor that a new library, identified by its `handle`, has been loaded into the process's address space. The monitor can then use this `handle` to retrieve further details about the library (such as its path and base load address) from the driver.
- `DLCLOSE(handle)`: Signals that a library previously loaded with `dlopen()` has been unloaded. This enables the monitor to invalidate and clean up associated metadata, ensuring that functions from the unloaded library can no longer be used as valid targets for indirect calls.

By tracking these dynamic events, StreamCFI maintains precise and up-to-date control over valid call targets across the application's full execution lifecycle, even in highly modular or plugin-based software.

CFI Metadata – typemap. StreamCFI's compiler pass generates a *typemap* file for each output binary, whether an application or a library, containing the static metadata required by the monitor to enforce the CFI policy. This metadata includes the set of all functions that may serve as indirect call targets, along with their corresponding `typeID`s. It also records the symbols of all class vTables that may be used in virtual calls, each annotated with its own `typeID`. Prior to stripping symbol information from the final binary, StreamCFI replaces symbol references in the typemap with their relative offsets within the binary's image. This transformation ensures that the monitor can resolve targets efficiently at load time, without relying on external debug symbols.

4.3. Monitor

The StreamCFI monitor is a user-space process responsible for enforcing the CFI policy at run

time. Upon initialization, it registers with the kernel driver and remains idle until an application requests StreamCFI's services. When such a request occurs, the driver notifies the monitor and maps the corresponding ring buffer and associated metadata (e.g., the pointer to the last written location from the processor) into the monitor's address space. In response, the monitor receives information about the main application binary and its libraries. It then loads the associated *typemap* file for the application and (if available) its libraries. To compute absolute addresses, the monitor combines these offsets with the base load addresses of the binaries, which are supplied by the driver. This process yields the actual memory addresses of all control-flow targets, which are stored in a *lookup table*. Each entry in the table includes the target's `typeID` and an activation status flag.

Once initialized, the monitor begins processing messages from the ring buffer. Messages are decoded and used to update or check the monitor's internal state. Specifically, the following actions are performed for each message type:

- `CHECK(typeID, addr)`: The monitor validates `addr` against its lookup table. An application halt is triggered via the driver if `addr` is unknown/nonactivated or its stored `typeID` mismatches the received `typeID`.
- `ACTIVATE(addr)`: The monitor locates the matching entry in the lookup table and marks it as active.
- `DLOPEN(handle)`: Upon receiving a library's `handle`, the monitor queries the driver to retrieve the library's path and base load address. Using the path, it attempts to load the library's *typemap*. If successful (indicating a StreamCFI-instrumented library), the monitor computes absolute target addresses from the typemap offsets and the base address, populates its lookup table with these targets (address, typeID, initial activation status), and associates them with the `handle`. If no typemap is found, the library is considered uninstrumented; the monitor records its address range (obtained via the driver) and exempts control transfers involving this range from CFI checks to ensure compatibility.
- `DLCLOSE(handle)`: The monitor uses the `handle` to identify all target entries in its lookup table that were associated with the library being unloaded. These entries are then invalidated or removed, ensuring that functions from the unloaded library are no longer considered valid targets for future indirect calls.

4.4. Optimizations

Eliminate Redundant Activations. Previous work on inline per-input CFI policies [29] has shown that repeatedly activating the same targets, such as when new objects of a class are instantiated, can incur unnecessary overhead. To eliminate this redundancy, StreamCFI employs a backpatching optimization that disables further activations for targets that have already been logged. Specifically, once the monitor processes an `ACTIVATE` message for a given target, it instructs the system to overwrite the corresponding instrumentation site with no-operation (NOP) instructions, preventing future redundant activations.

To support this optimization, we extend the `ACTIVATE` message format to include the current program counter (i.e., `ACTIVATE(addr, pc)`). Upon receiving such a message, the monitor additionally issues a *backpatch* request to the driver, providing the associated program counter (`pc`). The driver, operating with kernel privileges, safely rewrites the instrumented `ACTIVATE` instructions at the specified location with NOPs.

This optimization is both efficient and secure: since the driver is trusted and executes within the kernel, it can modify the application's memory without violating W^X (write XOR execute) memory protection policies or relying on software fault isolation (SFI). As a result, backpatching reduces both message volume and run-time overhead without compromising enforcement precision. To ensure code rewriting is atomic with respect to application execution, we temporarily pause the application during the backpatching process.

Static Global Activations. By default, StreamCFI activates all targets used in the initialization of global variables at startup. This can be costly for short-lived applications or for those with many global function pointers and can result in significant bus contention at startup. As an optimization, we treat such assignments as static events known at compile time. The targets are collected during compilation and stored in the typemap of the binary. When the monitor reads the metadata, it proceeds to update its lookup table by marking the corresponding targets as activated. The instrumentation pass is also updated and stops injecting `ACTIVATE` messages for these targets.

Payload Reduction. While `PTWRITE` can support 64-bit payloads for encoding full target addresses and type IDs, such lengthy payloads result in significant bus contention. As an optimization, StreamCFI reduces this overhead by compressing `ACTIVATE` and `CHECK` message payloads to 32 bits. Less frequent messages, such as those for dynamic library events, retain larger payloads.

The 32-bit message format uses the top bit as an opcode (i.e., to distinguish between `ACTIVATE` and `CHECK` messages) and the remaining 31 bits to encode the target address. To guarantee that 31 bits are sufficient to encode any function address, we pre-link the application binary and all libraries within the first 2GB of the address space. This strategy ensures that target addresses fit safely within 31 bits without the need for lossy encoding. We use `libshrink` [52] to pre-link all binaries and libraries, enabling the monitor to reliably perform address matching without runtime address translation.

For `CHECK` messages, the type ID is omitted from the payload. Instead, each instrumented indirect call site is associated at compile time with its expected function pointer type ID. At runtime, a mechanism (see Section 5) enables the monitor to recover the correct type ID for each `PTWRITE`, allowing full CFI

validation without transmitting the type ID in every message.

5. Implementation

We implemented StreamCFI for Linux Intel x86-64 processors supporting Intel Processor Trace (PT) with the `PTWRITE` instruction (SDL support). We built our prototype on top of the SideCar framework [36], which also provides a static CFI policy tool, named SideCFI.

Compiler Pass. We implemented a new compiler pass (∼600 LoC) for the LLVM 12.0.1 toolchain that identifies address-taken function pointers and inserts instrumentation to emit `ACTIVATE` messages. We extend the existing SideCFI pass to handle `CHECK` messages, generate the typemap file, and manage `dlopen` events. Our optimizations rely on Intel PT's Follow-Up Packet (FUP) feature, which records the instruction pointer (IP) of specific events in the processor's trace. By enabling FUPonPTW, the driver configures PT to emit a FUP after each `PTWRITE`, allowing the monitor to recover the precise address of the instrumentation site that generated an SDL message.

This capability enables two key optimizations. First, for payload reduction, the compiler emits a `nopl` instruction encoding the type ID immediately after each `PTWRITE` for a `CHECK` message; at runtime, the monitor uses the FUP to locate the matching `nopl` instruction and extract the type ID, eliminating the need to transmit the ID in the message. Second, for backpatching, the monitor uses the FUP from an `ACTIVATE` message to identify and request the driver to NOP out the corresponding `PTWRITE`, eliminating redundant activations.

Driver. We extended the SideCar driver (∼200 LoC) to support StreamCFI's optimizations. Above all, we enabled Intel PT's FUPonPTW (FUP on `PTWRITE`) capability for traced applications, as this is essential for the monitor to efficiently locate instrumentation sites.

For the backpatching optimization, we also implemented support for a new `ioctl` call. When the monitor requests a backpatch at a specific program counter (obtained via FUP, as enabled by the compiler's instrumentation strategy), the driver suspends the target application, temporarily marks the relevant code page as writable, replaces the `PTWRITE` instruction with NOPs, and restores original page protections.

Monitor. We extended the SideCFI monitor (+ ∼600 LoC) to support StreamCFI's dynamic CFI policy. Our implementation processes `ACTIVATE` messages by marking targets as active in its hash table-based lookup structure. For the backpatching optimization, upon processing an `ACTIVATE` message, the monitor uses the associated FUP to get the `PTWRITE` instruction's program counter and then issues the backpatch `ioctl` to the driver.

When processing compressed `CHECK` messages, our monitor utilizes the associated FUP to locate the `PTWRITE` instruction and, subsequently, the following `nopl` instruction (placed there by the compiler pass). It then extracts the 14-bit type ID from the `nopl` instruction to perform the type check, effectively reconstructing the full check information without transmitting the type ID in the SDL payload.

The monitor also handles dynamic library events, using the library handle from `DLOPEN` messages to query the driver for the library's path and base address, then loading the corresponding typemap if available, as described earlier.

6. Security Analysis

In this section, we evaluate the security of StreamCFI by testing its ability to block unauthorized control-flow transfers. We verify StreamCFI's correctness in detecting violations and also quantify the improvement over a static CFI policy (i.e., LLVM-CFI) by measuring the number of activated targets.

6.1. Correctness

To verify the correctness of StreamCFI, we first ran all SPEC CPU2017 INT benchmarks with the `ref` input set and StreamCFI enabled. All benchmarks completed successfully without crashes, hangs, or alerts raised by StreamCFI's monitor. We also confirmed that the output matched the expected reference results, ensuring that StreamCFI does not interfere with normal application behavior or program semantics. We used a patched SPEC CPU2017 [53] that ensures compatibility with type-based CFI.

To further validate correctness, we used the RIPE64 benchmark suite [39], which contains a wide range of memory corruption attacks. Since StreamCFI focuses on securing forward-edge control-flow transfers, we only selected the 1,350 forward-edge attack variants from the suite. We executed each attack with and without StreamCFI, and compared results against LLVM-CFI and SideCFI. All systems successfully detected the attacks, hence, StreamCFI provides equivalent protection while logging less metadata and enforcing a stricter policy than SideCFI.

6.2. Accuracy

A key goal of StreamCFI is to reduce valid indirect call targets, minimizing the attack surface against control-flow hijacks. Accuracy here means how precisely StreamCFI determines the intended target; higher accuracy implies fewer permissible targets per call site, limiting attackers. By refining this set at runtime based on program behavior, StreamCFI dynamically filters unused address-taken functions, achieving finer-grained enforcement than static or type-based CFI. This significantly reduces the dynamic control-flow graph, yielding a smaller, more defensible attack surface.

To evaluate attack surface reduction, we measured the percentage of address-taken functions activated

during execution relative to the total identified at compile time. For each SPEC CPU2017 INT benchmark using the `ref` input set, we executed every input and report the average number of activated targets across all runs. We also report the percentage of targets that were actually invoked during program execution, indicating the accuracy of the system in filtering out unused targets.

TABLE 1: Liveness of Function Targets for SPEC CPU2017 INT Benchmarks.

Benchmark	All Targets	Activated # (%)	Invoked # (%)
`perlbench`	2056	676 (32.86%)	271 (13.20%)
`gcc`	9747	3209 (32.92%)	1072 (11.00%)
`mcf`	38	2 (5.26%)	2 (5.26%)
`omnetpp`	2052	586 (28.55%)	464 (22.61%)
`xalancbmk`	2858	438 (15.33%)	289 (10.11%)
`x264`	430	215 (50.00%)	125 (29.15%)
`deepsjeng`	104	0 (0.00%)	0 (0.00%)
`leela`	28	4 (14.29%)	0 (0.00%)
`xz`	349	52 (15.04%)	16 (4.73%)
Average %	-	21.58%	10.67%

Table 1 presents the results for the SPEC CPU2017 INT benchmarks executed with the `ref` input set. The *activated* set refers to address-taken functions whose pointers were assigned at run time, while the *invoked* set reflects those that were actually called. Two benchmarks, `deepsjeng` and `leela`, had no invoked targets, with `leela` activating only four and `deepsjeng` activating none. In general, most benchmarks exhibit a low percentage of activated targets, with their average at 21.58% across all benchmarks. Some, like `mcf`, activate as few as 5.26% of their total address-taken targets. On the other hand, `x264` stands out with 50% of targets activated and 29.15% invoked, indicating a denser dynamic use of function pointers. The geometric mean for invoked targets is 11.34%, which is relatively close to the activation percentage. This indicates that most activated targets are indeed used during execution, suggesting a high level of precision in our enforcement policy. Overall, StreamCFI achieves a substantial reduction of the attack surface by filtering out approximately 80% of unused address-taken functions at runtime.

Figure 2 shows target activation over time for each SPEC CPU2017 INT benchmark (using the longest-running `ref` input; `deepsjeng` and `leela` excluded due to no invocations). Activation is heavily front-loaded: most targets activate within the first 10% of execution, reflecting common patterns of initializing function tables, callbacks, or vTables in early setup phases, as also reported by Niu et al. [29]. Post-initialization, these pointers are stable with few new activations. For instance, `x264_s` and `gcc_s` activate nearly all targets at startup. Even `xz_s`, with later activations near 90% execution time, shows only marginal changes post-initial spike. This confirms StreamCFI establishes a minimal attack surface early, enabling stronger, more stable CFI than static or type-based methods.

To further quantify the precision of StreamCFI

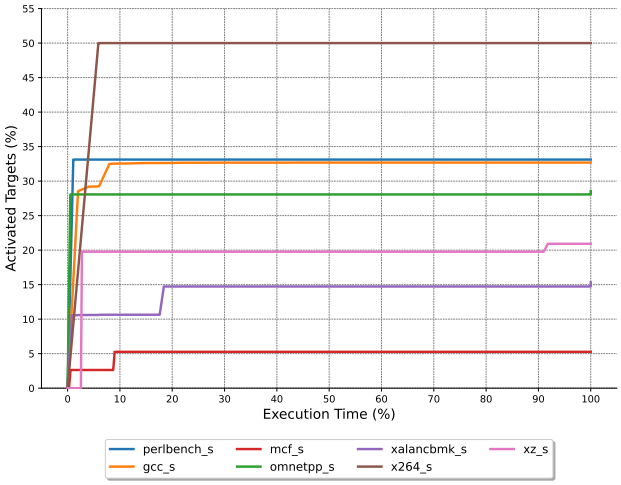


Figure 2: % of activated targets over the course of execution for SPEC CPU2017 INT.

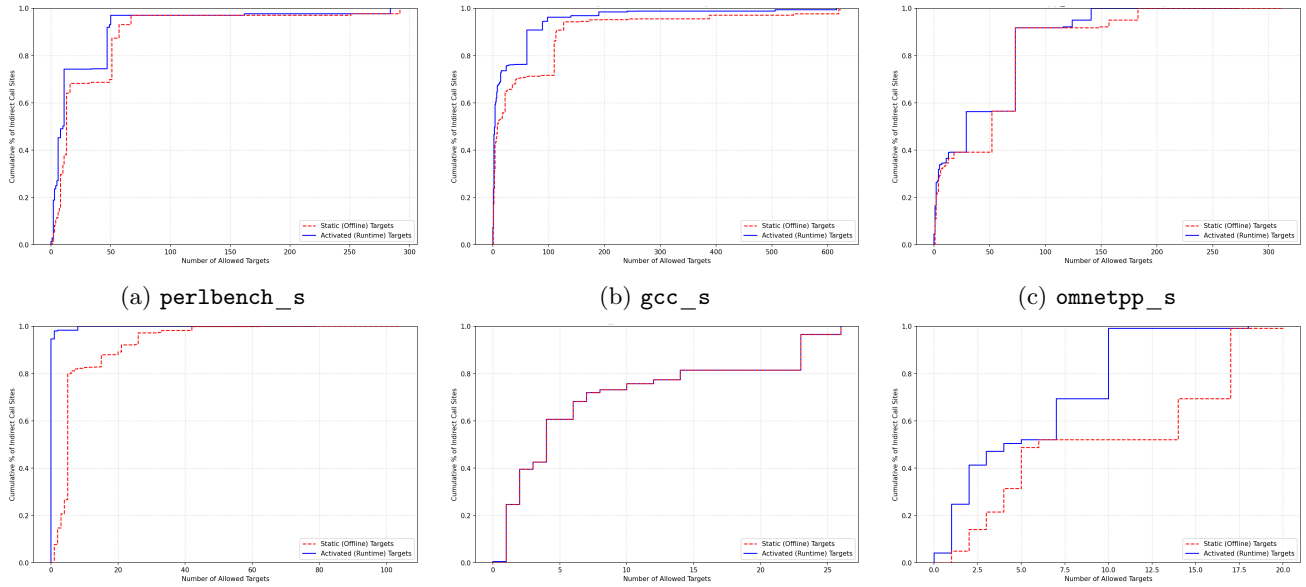
at the granularity of individual call sites, we analyzed the distribution of allowed targets per call site. During compilation, we identified all indirect call sites and their Type IDs, correlating them with the set of statically allowed targets. We then excluded call sites that were never invoked during the execution of *all* inputs, creating a *worst-case* per-call-site CDF where each call site is assigned the maximum number of activated targets observed across all inputs.

Figure 3 illustrates these CDFs for six benchmarks (we excluded `deepsjeng_s` and `leela_s` as they invoke no indirect targets, and `mcf_s` due to its negligible number of targets). The x-axis shows the number of allowed targets, and the y-axis shows the cumulative percentage of call sites. The gap between the static baseline (red line) and StreamCFI (blue line) represents the reduction in attack surface. A curve closer to the top-left corner indicates a stricter policy, where a larger percentage of call sites are restricted to fewer targets. For instance, in `xz_s`, the static policy allows hundreds of targets for many call sites, whereas StreamCFI shifts the distribution significantly to the left, restricting the majority of call sites to a much smaller set of activated targets. This per-call-site analysis confirms that StreamCFI not only reduces the global set of active functions but also effectively minimizes the legal targets for individual indirect calls.

In Fig. 3, `x264_s` constitutes an exception, showing minimal divergence between the static and dynamic policies, because in this case we only consider call sites that executed at least once. This indicates that for the specific Type IDs utilized by the hot indirect call sites in `x264_s`, nearly all compatible functions are activated during execution. The unactivated targets belong to other Type IDs associated with call sites not triggered by the benchmark's inputs.

6.2.1. Detection Latency

Due to StreamCFI's asynchronous monitoring, a brief window exists between violation and termination. We quantify this latency using a microbenchmark on a 24-core Intel i9-13900K that timestamps messages in



(a) `perlbench_s` (b) `gcc_s` (c) `omnetpp_s`

(d) `xalancbmk_s` (e) `x264_s` (f) `xz_s`

Figure 3: CDF of allowed targets per call site. The red line represents the static Type-based CFI baseline, while the blue line shows StreamCFI's reduction in allowed targets by filtering non-activated functions. A shift to the left indicates a tighter control-flow policy.

the monitored application and in the security monitor upon receipt. Measurements across varying message volumes yield latencies between 1.9 and 2.4 milliseconds, consistent with prior work [36]. The monitor executes in a separate privileged domain, so a compromised application cannot tamper with its state or suppress an alert. Nevertheless, asynchronous enforcement is not equivalent to synchronous prevention: a violating control transfer may transiently execute for up to 2.4 ms before the application is halted. In many realistic post-compromise scenarios [54], this window is too short to establish durable effects, and critical system modifications (e.g., filesystem writes or process creation) can either be rolled back or delayed until monitor acknowledgment. Still, this is a real limitation: if an attacker already has a staged, reliable payload whose effect can be completed entirely within that window (e.g., a one-shot privilege-escalation step), StreamCFI may detect the violation only after the immediate effect has occurred. Eliminating this residual gap would require fully synchronous mediation of such operations or offloading the monitor to a more isolated specialized coprocessor.

7. Performance Evaluation

Testbed. All experiments were conducted on a workstation equipped with a 24-core Intel i9-13900K (5.20 GHz), 128 GB DDR4 RAM, and a Samsung SSD 980 PRO 1 TB NVMe drive, running Debian 11. We evaluated StreamCFI using the integer SPEC CPU2017 benchmarks with the `ref` input set, compiled at `-O3`. Unless otherwise noted, hyper-threading was disabled to reduce measurement variance. For fairness, we ran StreamCFI and πCFI under identical hardware and software configurations. Since πCFI relies on deprecated LLVM 3.5 tooling and legacy libraries, we built it on the same machine within a Docker container running Ubuntu 14.04.2, matching the setup described in the original paper, but executed the resulting binaries directly on the host system to avoid virtualization overhead. Additionally, we modified the πCFI source code to enable only forward-edge protection, *excluding* its original back-edge protection mechanisms, which are not relevant to our evaluation and would have artificially inflated performance overhead.

7.1. Single-threaded Tasks

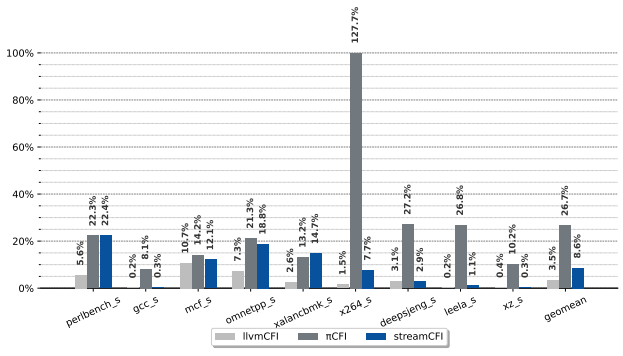


Figure 4: Overhead relative to execution without CFI for SPECspeed CPU2017 Integer under LLVM-CFI, πCFI, and StreamCFI.

Figure 4 shows the overhead of LLVM-CFI, πCFI, and StreamCFI relative to an unprotected baseline on SPECspeed CPU2017 Integer benchmarks, which measure how fast single-threaded tasks complete. πCFI performs worst, with a geometric mean overhead of 26.67% and particularly high overhead in `x264_s`

(127.74%) and `deepsjeng_s` (27.17%). This confirms πCFI's inlined checks and coarse-grained sandboxing scale poorly for modern indirect-call-heavy workloads, despite its optimizations.

In contrast, StreamCFI incurs 8.63% geometric mean overhead, compared to LLVM-CFI's 3.45%. Despite enforcing a strict policy by combining type-based and per-input CFI, and handling both activation and invocation events through SDL, StreamCFI's overhead remains consistently low across most benchmarks, staying at 0.31% in `xz_s` and `gcc_s`, and 1.05% in `leela_s`. The main outliers are `perlbench_s` (22.43%) and `omnetpp_s` (18.75%), where the density of indirect control transfers amplifies the cost of logging. These results demonstrate that StreamCFI offers a strong balance between security and performance, achieving significantly tighter control-flow enforcement than prior work with only modest overhead. While LLVM-CFI achieves the lowest overhead in this experiment, it enforces only a static type-based policy; StreamCFI remains comparatively close while providing added security by reducing the set of allowed targets at run time.

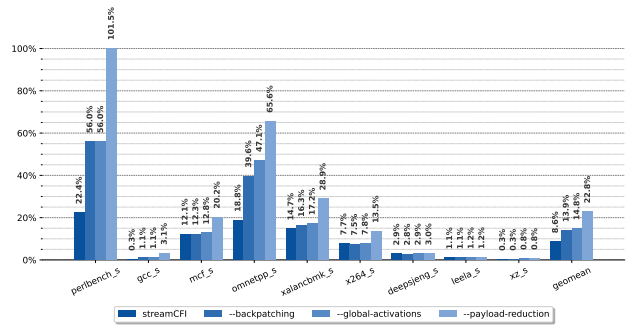


Figure 5: Overhead impact of individual StreamCFI optimizations across SPECspeed CPU2017 Integer. The leftmost bar shows StreamCFI with all optimizations enabled. Each subsequent bar removes one optimization, isolating its cost. The rightmost bar reflects the overhead with all optimizations disabled.

Impact of Optimizations. We also quantified the contribution of each optimization we have introduced by disabling them one by one, and measuring the performance impact on the same benchmark. In order, we disable backpatching, static global activations, and 32-bit payloads for `CHECK` messages. Figure 5 shows that the fully optimized configuration incurs 8.63% geometric mean overhead, while selectively disabling backpatching or static global activations increases it to 13.89% and 14.82%, respectively. Removing the 32-bit payload compression has the largest single impact, increasing the geometric mean overhead to 22.81%; the effect is especially pronounced on `perlbench_s` (101.54%) and `omnetpp_s` (65.59%), where the density of indirect calls turns every extra bus word into measurable stall time. Taken together, these results confirm that StreamCFI's selective logging strategy—small packets, one-off activations, and on-demand FUP metadata—strikes the best balance between trace richness and run-time cost. As also observed by πCFI, repeated activations are common enough that eliminating them provides a measurable benefit; in our evaluation, disabling backpatching increases geometric mean overhead from 8.63% to 13.89%.

7.2. Parallel Tasks

In this experiment, we evaluated StreamCFI's performance under severe resource contention when executing parallel workloads. Our objective was to saturate all available performance cores to 100% utilization, thereby eliminating spare resources for the monitor thread, which must share the same cores as the protected application. To achieve this, we employed the SPECrate CPU2017 Integer throughput benchmark suite, which spawns one process per available core; higher scores indicate greater throughput. Note that SPECrate CPU2017 Integer uses different input sets than the SPECspeed CPU2017 Integer variant evaluated in the previous section, rendering direct performance comparisons infeasible.

We evaluated two configurations: Hyper-Threading (HT) enabled (16 logical threads) and HT disabled (8 physical cores). Because the Intel i9-13900K features a hybrid architecture with efficiency cores that lack support for lossless Intel PT tracing, we restricted our evaluation to the performance cores only. We pinned the StreamCFI monitor thread to the same core set as the benchmark processes (cores 0–7 for HT-off, 0–15 for HT-on), deliberately inducing contention for CPU cycles between the monitor and the workload it protects. We measured overhead against an unprotected baseline compiled with LTO enabled. For comparison, we evaluated πCFI under identical conditions. Note that the `gcc` benchmark is excluded from the graph due to a crash in the πCFI runtime.

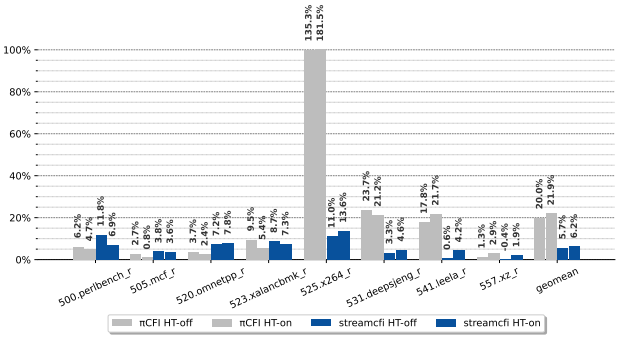


Figure 6: Overhead relative to execution without CFI on SPECrate CPU2017 Integer under CPU saturation (HT-off vs. HT-on).

Figure 6 presents the results. Even under full core saturation, StreamCFI demonstrates robust performance. With HT enabled, StreamCFI incurs only 6.18% geometric mean overhead, significantly outperforming πCFI (21.88%). Similarly, with HT disabled, StreamCFI maintains 5.67% geometric mean overhead, compared to πCFI's 20.01%. StreamCFI performs slightly better with HT-off because Intel PT hardware is shared between two logical threads on

the same physical core; enabling HT introduces a small amount of contention for the trace generation hardware itself. While πCFI suffers severe degradation in benchmarks like `525.x264_r` (135.35% overhead with HT-off and 181.52% with HT-on) and `531.deepsjeng_r` (23.72% and 21.17%), StreamCFI remains consistent across the suite, including 10.99% and 13.56% overhead on `525.x264_r`. These results indicate that StreamCFI's design remains efficient even when the monitor must share execution resources with the protected application.

7.3. Code Size Overhead

A key challenge in developing StreamCFI was mitigating binary bloat introduced by LLVM's `sanitizer_common` library. Despite StreamCFI's minimal instrumentation and lightweight runtime, early builds had a memory footprint comparable to LLVM-CFI, primarily due to unused `__sanitizer` symbols and implicit dependencies on `sanitizer_common`. To eliminate this overhead, we compiled with `-fno-sanitize-link-runtime`, which excluded the sanitizer runtime. Additionally, we replaced all `sanitizer_common` utilities (e.g., `Printf`, `Die`) with standard libc functions (`fprintf`, `exit`), and ensured inclusion of our runtime by linking with `-Wl,--whole-archive`. To avoid intercepting `dlopen`/`dlclose`, we replaced sanitizer interceptors with `dlsym(RTLD_NEXT, ...)` and `pthread_once` for safe initialization. These changes fully decouple StreamCFI from `sanitizer_common`, significantly reducing binary size while preserving core functionality.

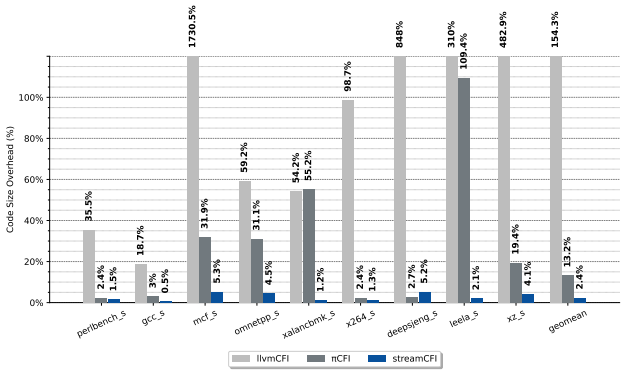


Figure 7: Code size overhead of StreamCFI compared to LLVM-CFI and πCFI.

Figure 7 reports binary size overhead relative to an LTO baseline on SPEC CPU2017 INT. LLVM-CFI shows significant overhead (154.35% geometric mean), with extremes like `mcf_s` (1730.50%) and `deepsjeng_s` (848.04%), largely from heavy instrumentation and `sanitizer_common` linkage, prominent in small binaries. For `leela_s` and `deepsjeng_s` (no indirect calls), LLVM-CFI still adds unnecessary instrumentation due to `sanitizer_common` dependencies, bloating the binary. In contrast, StreamCFI achieves a drastically lower 2.41% geometric-mean overhead via lean instrumentation and full sanitizer runtime decoupling, despite building on LLVM-CFI. StreamCFI also outperforms πCFI (13.16% geometric-mean overhead). Runtime memory overhead is minimal since StreamCFI maintains no data structures within the application, instead forwarding all metadata to the external monitor.

8. Limitations

First, our approach builds upon SideCar and SideCFI, and thus depends on LLVM and Link-Time Optimization (LTO) to perform code instrumentation. This design choice implies that StreamCFI requires access to source code and does not currently support precompiled binaries or libraries. Additionally, StreamCFI focuses solely on securing forward-edge indirect control transfers (ICTs) and does not offer protection for return addresses. Nevertheless, well-established mechanisms such as shadow stacks [55], ASLR [13], and Intel CET [51] can complement StreamCFI for backward-edge protection. Alternatively, a SideCar-based monitor like SideStack could be integrated to provide full forward and backward-edge CFI enforcement.

A limitation of dynamic target activation is that long-running programs might eventually activate most address-taken functions, reducing dynamic CFI benefits. Future work could augment StreamCFI with richer runtime information (e.g., call-stack history) for higher precision, or investigate target deactivation for functions unused for extended periods. Though requiring complex dynamic analysis [56], deactivating or isolating infrequently used/syscall-heavy functions could yield security gains.

StreamCFI's 32-bit payload optimization imposes a practical constraint: fitting full function addresses requires prelinking binaries/libraries into a low memory range (sub-2GB) using libshrink. This suffices for evaluated applications but may limit applicability for larger ones like Chrome. Mitigation strategies include: (i) dynamic address masking, (ii) NOP-padding functions for realignment, or (iii) collision-resistant 32-bit function hashes. For instance, masking the lower 4 address bits (assuming 16-byte alignment) increases the range but risks attacks on untracked bits; requiring targets' lower 4 bits to be zero mitigates this. Hashing, even if collision-resistant, risks redirection to nearby gadgets on collision. Supporting very large applications without strict prelinking is an open challenge for future research.

9. Related Work

Early dynamic CFI approaches include PathArmor [45], which compared recent branch histories from LBR against static references. However, its precision is limited by the short history buffer. PittyPat [46] improved precision by leveraging PT's detailed execution tracing but incurred significant decoding overhead. πCFI [29] introduced per-input activation of indirect call targets, significantly reducing equivalence classes by dynamically enabling targets only upon first assignment, employing runtime backpatching for efficiency.

μCFI [47] further tightened restrictions by enforcing a unique code target property through dynamic points-to analysis integrated with PT traces, though facing scalability challenges due to extensive trace data. OS-CFI [25] leveraged pointer-origin tracking to limit valid targets based on their assignment history but had partial coverage due to its focus on explicit assignments. ECCUT [28] addressed these limitations by adding complete field sensitivity, accurately tracking function pointers within complex structures while optimizing overhead through selective offset tracking.

MLTA [57], KELP [27], and SMLTA [26] refined target sets by incorporating multi-layer type information, analyzing not only function pointers but also their enclosing composite types, significantly enhancing enforcement precision at the cost of increased analysis complexity.

While these approaches achieve high precision, their runtime overhead has motivated exploration of decoupled enforcement mechanisms. Software-Driven Logging (SDL) offers a complementary approach to decoupled runtime enforcement. SideCar [36] leveraged SDL by emitting targeted `PTWRITE` metadata, offloading expensive type-based CFI checks to an external monitor. Similarly, DTrace [34] employed an emulated `PTWRITE` mechanism for fine-grained data integrity checks. MemGaze [35] further explored SDL's potential by using `PTWRITE` to efficiently log memory access patterns and analyze load-level data reuse, significantly reducing overhead compared to traditional memory tracing techniques. While these works demonstrate SDL's versatility, none have utilized it to implement dynamic CFI policies such as per-input CFI. In contrast, our approach demonstrates SDL's capability for efficient, high-precision CFI enforcement, blending dynamic activation policies with selective, low-overhead logging.

10. Conclusion

We presented StreamCFI, a decoupled dynamic CFI mechanism that combines per-input and type-based policies using Software-Driven Logging (SDL) and Intel PT's `PTWRITE` instruction to efficiently log function pointer activations and invocations. StreamCFI enforces stricter control-flow integrity than traditional static CFI, reducing the attack surface by approximately 80% and reducing geometric-mean overhead from 26.67% to 8.63% compared to πCFI on SPECspeed CPU2017 Integer. Moreover, our design dramatically lowers code size overhead to a 2.41% geometric mean, compared to LLVM-CFI's 154.35% and πCFI's 13.16%, through lean instrumentation and decoupling from the sanitizer runtime. StreamCFI also remains close to LLVM-CFI, which incurs 3.45% geometric-mean overhead, while reducing the set of allowed targets at run time. Optimizations such as 32-bit payload compression and backpatching enable efficient, precise enforcement without inline overhead, as validated by RIPE64 testing. These results establish StreamCFI as a practical and deployable solution that advances the state of the art in CFI enforcement on commodity hardware.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported by DARPA under award D21AP10116-00. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or DARPA.

References

[1] V. Van der Veen, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012, pp. 86–106.

[2] A. One, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[3] LWN.net, "x86 NX support," Jun. 2004. [Online]. Available: https://lwn.net/Articles/87814/

[4] OpenBSD, "i386 W^X," Apr. 2003. [Online]. Available: https://marc.info/?l=openbsd-misc&m=105056000801065

[5] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[6] N. Carlini and D. Wagner, "{ROP} is still dangerous: Breaking modern defenses," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 385–399.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A new class of code-reuse attack," in *Proceedings of the ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011, pp. 30–40.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 559–572.

[9] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-Oriented Programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 745–762.

[10] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 575–589.

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity," in *Proceedings of the USENIX Security Symposium*, 2015, pp. 161–176.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks," in *Proceedings of the USENIX Security Symposium*, Jan. 1998.

[13] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[14] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining entropy-based information hiding (and what to do about it)," in *Proceedings of the USENIX Security Symposium*, August 2016, pp. 105–119.

[15] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020, pp. 1–16.

[16] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus, "Bypassing memory safety mechanisms through speculative control flow hijacks," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 633–649.

[17] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020, pp. 1871–1885.

[18] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005, p. 340–353.

[19] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[20] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[21] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the USENIX Security Symposium*, Aug. 2013, pp. 337–352.

[22] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. of the USENIX Security Symposium*, 2014, pp. 941–955.

[23] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert, "Analyzing control flow integrity with llvm-cfi," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 584–597.

[24] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 901–913.

[25] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium*. Santa Clara, CA, USA: USENIX Association, 2019.

[26] T. Xia, H. Hu, and D. Wu, "{DEEPTYPE}: Refining indirect call targets with strong multi-layer type analysis," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5877–5894.

[27] Y. Cai, Y. Jin, and C. Zhang, "Unleashing the power of type-based call graph construction by using regional pointer information," in *33nd USENIX Security Symposium (USENIX Security 24)*, 2024.

[28] H. Xiang, Z. Cheng, J. Li, J. Ma, and K. Lu, "Boosting practical control-flow integrity with complete field sensitivity and origin awareness," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Salt Lake City, UT, USA: ACM, 2024.

[29] B. Niu and G. Tan, "Per-input control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 914–926.

[30] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *2008 International Symposium on Computer Architecture*, 2008, pp. 377–388.

[31] S. Ainsworth and T. M. Jones, "The guardian council: Parallel programmable hardware security," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1277–1293.

[32] D. D. Chen, W. S. Lim, M. Bakhshalipour, P. B. Gibbons, J. C. Hoe, and B. Parno, "HerQules: Securing programs via hardware-enforced message queues," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, p. 773–788.

[33] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.

[34] X. Wang, F. Huang, and H. Chen, "Dtrace: Fine-grained and efficient data integrity checking with hardware instruction tracing," *Cybersecurity*, vol. 2, no. 1, pp. 1–15, 2019.

[35] O. O. Kilic, N. R. Tallent, Y. Suriyakumar, C. Xie, A. Marquez, and S. Eranian, "Memgaze: Rapid and effective load-level memory trace analysis," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 484–495.

[36] K. Kleftogiorgos, P. Zielinski, S. Huang, J. Xu, and G. Portokalidis, "SideCar: Leveraging debugging extensions in commodity processors to secure software," in *Proceedings of the 40th Annual Computer Security Applications Conference*, 2024.

[37] James Reinders (Intel), "Processor tracing," Sep 2013, https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[38] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge control flow violation detection using Intel Processor Trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.

[39] Hubert Rosier, "RIPE64. National University of Singapore." https://github.com/hrosier/ripe64, 2019.

[40] M. Ammar, A. Caulfield, and I. De Oliveira Nunes, " SoK: Integrity, Attestation, and Auditing of Program Execution ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 3255–3272. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00077

[41] B. Niu and G. Tan, "Monitor integrity protection with space efficiency and separate compilation," in *ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 199–210.

[42] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones." in *NDSS*, vol. 26, 2012, pp. 27–40.

[43] J. Pewny and T. Holz, "Control-flow restrictor: Compiler-based cfi for ios," in *Annual Computer Security Applications Conference (ACSAC)*, 2013, pp. 309–318.

[44] The Clang Team, "Control flow integrity," Clang 21.0.0git documentation – https://clang.llvm.org/docs/ControlFlowIntegrity.html, May 2025.

[45] V. van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2015.

[46] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of Path-Sensitive control security," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 131–148.

[47] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2018, pp. 1470–1486.

[48] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1821–1835.

[49] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry, "Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 25–35.

[50] ARM, "Better trace for better software," https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Better_Trace_for_Better_Software_-_CoreSight_STM_with_LTTng_-_19th_October_2010.pdf, 2010.

[51] I. Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[52] K. Koning, "ShrinkAddrSpace," https://github.com/vusec/libshrink, 2021.

[53] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "FineIBT: Fine-grain control-flow enforcement with indirect branch tracking," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Oct. 2023.

[54] CrowdStrike, "2024 Global Threat Report: Executive Summary," https://www.crowdstrike.com/wp-content/uploads/2024/02/crowdstrike-2024-global-threat-report-executive-summary.pdf, 2024.

[55] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2015, p. 555–566.

[56] V. Lakshmi Rajagopalan, K. Kleftogiorgos, E. Göktaş, J. Xu, and G. Portokalidis, "Syspart: Automated temporal system call filtering for binaries," *arXiv e-prints*, pp. arXiv–2309, 2023.

[57] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

Appendix A. Data Availability

The source code for StreamCFI is publicly available at: https://github.com/kleftog/streamcfi.git