# CacheCFI: Forward-edge Control-flow Integrity with Inline Branch-target Caches

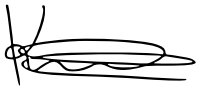
by
Yi Hao Tan
Sc.B., Brown University, 2025

A Thesis submitted in partial fulfillment of the requirements for Honors
in the Department of Computer Science at Brown University

Providence, Rhode Island
April 2025

This thesis by Yi Hao Tan is accepted in its present form by
the Department of Computer Science as satisfying the research requirement
for the awardment of Honors.

Date 04/30/2025



Vasilis Kemerlis, Advisor

Date _____

Nick DeMarinis, Reader

# Acknowledgements

I would like to thank Vasilis Kemerlis, my thesis advisor, for supporting me throughout the research process. After generously allowing me to join the Secure Systems Lab at Brown in my junior year, Vasilis has been incredibly supportive throughout my research journey, providing me with guidance and enabling me to grow immensely as a researcher.

I would also like to thank Nick DeMarinis for agreeing to be the reader of this thesis and for being a listening ear.

Finally, I would like to thank Alexander Gaidis, without whom this thesis would not be possible. Not only does this thesis build on Alex's prior work, but Alex has also acted as my primary mentor, providing me with guidance to overcome numerous difficulties faced throughout this entire thesis journey.

# Contents

# List of Tables

# List of Figures

Abstract of “CacheCFI: Forward-edge Control-flow Integrity with Inline Branch-target Caches” by Yi Hao Tan, Sc.B., Brown University, April 2025.

Currently, the performance of software-only control-flow integrity (CFI) enforcement lags behind hardware-assisted CFI enforcement. To bridge this gap, we observe that indirect call targets are rarely changed during the execution of a program. As such, to determine if the runtime performance of CFI enforcement can be improved by caching a function pointer’s most recent value, this thesis presents the design, implementation, and evaluation of CacheCFI: a software-only CFI enforcement mechanism that uses a novel inline caching scheme to speed-up runtime CFI enforcement checks. This thesis studies the design of CacheCFI on the x86-64 architecture, and implements it using the LLVM toolchain and musl. To evaluate the runtime overhead of CacheCFI, a fine-grained type-based CFI policy was selected to be enforced by CacheCFI on the SPEC CPU2017 C Benchmarks. CacheCFI incurs a median runtime overhead of 3.01%, and the runtime overhead ranges between $\approx$ 0%–53.8%

# Chapter 1

# Introduction

In 2024, the White House (Office of the National Cyber Director) released a statement in support of Software Measurability and Memory Safety [1]. The corresponding report with this press release highlights the severity of memory safety errors, stating that: "(m)any of the major cybersecurity vulnerabilities over the past several decades were facilitated by memory safety vulnerabilities," and "for over 35 years, this same class of vulnerability has vexed the digital ecosystem" [2]. Software vendors like Google and Microsoft have both reported that approximately 70% of their security vulnerabilities can be attributed to memory safety issues [3, 4]. In short, memory safety errors have proven to be persistent, pervasive, and capable of disastrous consequences.

With a memory-safety error in hand, attackers can potentially control the execution flow of a program and achieve arbitrary code execution [5]. Specifically, with W^X protections in place [6], attackers are pushed to use code-reuse methods (e.g., ROP [20], JIT-ROP [7]) to achieve their goal of arbitrary code execution.

Initially proposed by Abadi et al. [8], control-flow integrity (CFI) is a defense that aims to mitigate the effects of control-flow hijacking attacks by restricting the possible targets of indirect control-flow transfers to a smaller set of allowed destinations. Over the years there have been many other CFI schemes proposed, including both hardware-assisted schemes [9, 30, 33–35] and software-only schemes [8, 10, 19, 29, 32].

However, hardware-assisted CFI schemes suffer from compatibility issues. By definition, they depend on the respective hardware features that they are built upon, thus creating a problem with both older machines, where such hardware features are limited or non-existent, as well as newer machines where such hardware features are discontinued.

In contrast, while software-only schemes typically excel at compatibility, they often suffer from poor performance without the speedup given by hardware. For example, one of the most popular software-only CFI schemes, Clang-CFI, reports a runtime overhead of 1%–8.7% when enforcing a fine-grained, type-based policy that only covers forward-edge transfers on SPEC CPU2006

benchmarks [10]. In contrast, FineIBT, a hardware-assisted CFI scheme, is able to achieve negligible runtime overheads when enforcing a fine-grained, type-based policy that only covers forward-edge transfers on both SPEC CPU2017 benchmarks ($\approx$ 0%–1.94%) and real-world applications ($\approx$ 0%–1.92%) [9]. In other words, hardware-assisted schemes significantly outperform software-only schemes.

To find ways to bridge this gap in performance, we start by considering the behavior of function pointers. Namely, we hypothesize that function pointers should not change frequently. Since function pointers are often used to create interfaces, once an interface is initialized, it should no longer be changed frequently. To test this hypothesis, we measured the number of times the indirect call target was updated at an indirect call site for 4 real-world applications: MariaDB [24], nginx [22], Redis [23], and SQLite [25]. We observed that the value of function pointers rarely change during a program's execution (Chapter 4, Fig. 4.2).

Thus, this thesis presents the design, implementation, and evaluation of a software-only CFI enforcement scheme, named CacheCFI, in order to answer the following research question: **Using the observation that indirect call targets are changed infrequently, can the runtime performance of CFI enforcement be improved by caching a function pointer's most recent value?**

CacheCFI aims to improve in two key areas. First, performance. CacheCFI implements a novel caching scheme that only compares a call target with a single cached value instead of doing a range-based comparison [10]. Hence, CacheCFI aims to reduce the performance overhead from CFI enforcement by reducing the number of comparisons performed. Second, compatibility: CacheCFI is designed to be a software-only CFI enforcement scheme, and as such, is not dependent on architecture-specific hardware features (e.g., Intel's Indirect Branch Tracking [11], Intel's Memory Protection Extensions (MPX) [16]). In addition, CacheCFI is designed to be policy agnostic—it can enforce any CFI policy.

In summary, this thesis makes the following novel contributions:

- We present the design of CacheCFI, a software-only CFI enforcement scheme that capitalizes on the observation that after an indirect call target is initialized, it is rarely modified again. CacheCFI caches a function pointer's most recent value in order to improve the runtime overhead associated with CFI enforcement.

- We present a prototype implementation of CacheCFI as a set of modifications to LLVM (v15.0.7) and musl's dynamic linker (v1.2.5) on the x86-64 architecture.

- We evaluate the runtime overhead of CacheCFI by enforcing a fine-grained, type-based CFI policy on the SPEC CPU2017 C benchmarks. CacheCFI incurs a median runtime overhead of 3.01%, and the runtime overhead ranges between $\approx$ 0%–53.8%

# Chapter 2

# Background

## 2.1 Memory Safety

Memory unsafe languages such as C and C++ are vulnerable to memory errors [18] that can give an attacker the ability to corrupt or leak memory in a victim program's address space. With a memory safety error in hand, an attacker can then attempt to gain arbitrary code execution capabilities [5].

In the past, arbitrary code execution could be achieved via means of code injection [50]. However, with non-executable memory [12] and the enforcement of a W^X [6] policy, code injection is now significantly harder for an attacker to perform. Instead, attackers now utilize code reuse attacks (e.g., ROP [20], JOP [36, 37], JIT-ROP [7]).

In a code reuse attack, an attacker first finds pieces of benign code within a victim program called "gadgets." The attacker then hijacks the control flow of this victim program by stitching these gadgets together, then corrupting control data (e.g., code pointers) such that these gadgets are executed. Hence, the attacker can effectively achieve arbitrary code execution while bypassing W^X protections as the attacker does not need to inject new executable code.

## 2.2 Control-flow Integrity

To defend against code reuse attacks, CFI was proposed. Introduced by Abadi et al. [8], CFI is a defense designed to mitigate control-flow hijacking by limiting the possible locations that a given indirect control-flow transfer can target to a set of valid (i.e., legal) targets. In other words, while an attacker can still modify code pointers in a CFI-protected program, they are only allowed to redirect control flow to valid locations as determined by a CFI policy. In short, CFI prevents a code reuse attack by significantly decreasing the number of ways an attacker can stitch gadgets together to execute their desired code.

A CFI policy is normally represented by the call graph (CG) of a given program and it dictates which control flow transfers are considered valid. For example, a CFI policy `A` could dictate that a

given indirect call site should only have two valid edges (control-flow transfers), `foo` and `bar`, while another CFI policy `B` could dictate that the same indirect call site should only have one valid edge, `foo`. At runtime, each control-flow transfer is verified by an enforcement mechanism. The key goal of this CFI enforcement mechanism is to ensure that the control-flow transfer adheres to the CFI policy. The following subsections outline the properties that typically differentiate CFI schemes.

### Control-flow Transfer Types

There exist two types of indirect control-flow transfers: forward-edge and backward-edge. In x86(-64), indirect forward-edge control-flow transfers occur via the `call` and `jmp` instructions, while indirect backward-edge control-flow transfers occur via a `ret` instruction. Different CFI schemes choose to protect different control flow transfers, ranging from schemes that only cover forward-edge transfers [9, 32, 41, 46, 55], to schemes that only cover backward-edge transfers [30, 43], and schemes that cover both [8, 19, 21, 28, 31, 33, 35, 38, 39, 49, 51]. While prior work has shown that it is important for CFI schemes to protect both forward- and backward- edge transfers [15], recent work typically focus on improving forward-edge protection [9, 32, 46, 55] as shadow stacks [56] have been demonstrated to be an efficient solution for backward-edge protection.

### Source Code Access

With access to source code, additional analysis can be performed to extract finer-grained CFI policies to achieve more precision when identifying valid call targets for each indirect call site. For example, points-to analysis [38–40], multi-layer type analysis [10, 32, 46] and class hierarchy analysis [41, 42]. However, some CFI implementations choose to generate their policy and perform runtime enforcement with only access to the binary [19, 28, 33, 49] to achieve compatibility in cases where source code is typically unavailable. For example, commercial off-the-shelf binaries with private code bases, and legacy binaries where the source code has been lost to time.

In addition to the generated policy, access to source code also affects how code is instrumented to introduce CFI enforcement checks. With access to source code, a compiler pass can be used to insert CFI enforcement checks [9, 10, 41]. With only access to the binary, CFI enforcement checks are inserted by directly modifying the binary using binary rewriting techniques [19, 28, 33, 49].

### CFI Enforcement Type

CFI enforcement is necessary in order to ensure that runtime control-flow transfers are valid. To achieve this, CFI enforcement schemes can either utilize hardware features or rely entirely on software.

Hardware-assisted CFI enforcement [9, 30, 33–35, 47, 48] utilize the hardware features provided by vendors in their respective CFI schemes. For example, PathArmor [33] uses Intel's 16 LBR registers to perform both forward- and backward-edge CFI enforcement; FineIBT [9] uses Intel CET [11] to perform forward-edge CFI enforcement; PT-CFI [30] uses Intel Processor Trace to construct a

shadow stack for backward-edge CFI enforcement; CFA+ [47] uses the ARM BTI security extension to perform both forward- and backward-edge CFI enforcement.

Even without relying on hardware features, software-only CFI enforcement [8, 10, 19, 29, 32, 51] schemes can still work. For example, kCFI [51] uses a label-based approach where code is instrumented to insert inlined "tags" that verify indirect branch targets; Clang-CFI [10] instruments code to insert range-based checks at each indirect call site, verifying an indirect call target against the set of valid call targets at said indirect call site.

While the use of hardware features could result in performance gains, hardware-assisted CFI enforcement schemes are, by definition, dependent on the respective hardware features that they are built upon, thus limiting the scope of their implementation. Specifically, they can be incompatible with both older machines where such hardware features are limited or non-existent, as well as newer machines where such hardware features are no longer available. For example, origin-sensitive CFI [48] leveraged Intel's MPX, but MPX has been discontinued for 10th generation and newer processors [16]. As such, there is still a need for software-only CFI implementations [8,10,19,29,32,51] for compatibility purposes.

**CFI Policy Granularity**

The chosen CFI policy determines the valid locations each indirect call site can target. In other words, a stricter CFI policy means fewer valid control flow transfers for a given call site and thus a reduction in attack surface.

Initial CFI policies were more coarse grained, e.g., restricting `call/jmp` instructions to any function whose address is taken [28], and arity-based policies [19] restricting call targets according to the number of arguments passed. However, such coarse-grained CFI policies leave a significant attack surface, which has been shown to be bypassable [27]. The underlying issue for such coarse-grained CFI schemes is a significant over-approximation in the set of allowed targets for indirect control-flow transfers, leaving a larger attack surface for an attacker to build an exploit from.

In response, more fine-grained CFI policies have been proposed, such as points-to based polices [38–40] which restrict call targets by limiting the set of memory locations (determined at compile-time using program variables) that a given pointer can point to; type-based policies [10, 32], which restrict call targets using function pointer types; and context-sensitive policies [21, 33] that verify the path of control-flow transfers leading up to a given point. Such fine-grained CFI policies enforce a more precise set of valid control flow transfers at indirect call sites, thereby shrinking the possible attack surface.

As a further response to these fine-grained CFI policies, an attack technique named "control flow bending" [15] was proposed. Control flow bending attempts to bypass CFI by exploiting the over-approximation of valid call targets in order to redirect control flow to valid but unintended call targets, thus enabling the attacker to gain arbitrary code execution capabilities from a single arbitrary memory write primitive, as in a ROP-style [20] attack. However, as this attack technique relies on the over-approximation of valid call targets, its effectiveness is largely dependent on the CFI

policy implemented. Hence, CFI is still an effective protection. Specifically, with CFI protections in play, an attacker must first find gadgets that are valid call targets according to the CFI policy [15], and with the right CFI policy, the attacker can still struggle to find useful gadgets for a control flow bending style attack.

# Chapter 3

# Threat Model

## 3.1 Adversarial Capabilities

This thesis assumes an attacker whose aim is to hijack the control flow of a program [5] and achieve arbitrary code execution. The attacker is able to exploit any number of spatial/temporal memory safety errors in the program and chain them together at will so as to get arbitrary read and write primitives that can tamper with code pointers. In other words, for as many times as desired, at any arbitrary point in the program's execution, the attacker can read any readable memory and write their desired data to any writable memory location in the program's address space.

## 3.2 Hardening Assumptions

This thesis assumes that non-executable memory [12] and W^X protections [6] are in place. In other words, the attacker is limited to only code-reuse style attacks [7] in order to achieve their goal of arbitrary code execution via control flow hijacking. Beyond this, other common hardening measures (e.g., ASLR [44], stack canaries [45]) are orthogonal to CacheCFI and can be freely included or excluded. In addition, side-channel attacks [53] and hardware vulnerabilities (e.g., speculative execution attacks [54]) are considered outside the scope of this thesis.

# Chapter 4

# Motivation

Enforcing a CFI policy requires runtime checks, which naturally incur a performance overhead. For example, a popular software-only CFI scheme, Clang-CFI, reports a runtime overhead of 1%–8.7% when enforcing a fine-grained, type-based policy that only covers forward-edge transfers on SPEC CPU2006 benchmarks [10]. In contrast, FineIBT, a hardware-assisted CFI scheme, is able to achieve negligible runtime overheads when enforcing a fine-grained type policy that only covers forward-edge transfers on both SPEC CPU2017 benchmarks (≈ 0%–1.94%) and real-world applications (≈ 0%–1.92%) [9].

In other words, there is a noticeable gap in the runtime overhead when comparing hardware-assisted CFI enforcement and software-only CFI enforcement. As such, there is room for performance improvements of software-only CFI enforcement schemes. Looking at software-only CFI enforcement schemes, range-based checks are commonly used [10]. However, depending on the runtime behavior of indirect calls, requiring a check against all valid call targets could be heavy handed. To determine this, we measure how often indirect call targets change during execution.

Looking at four real-world programs, the vast majority of indirect call sites only update the indirect call target once (77.6%–89.9%, Fig. 4.2). In other words, we observe that after a function pointer is initialized, it is rarely modified again. For example, consider the implementation of `__clock_gettime` in Fig. 4.1. The address of the `vdso` implementation of `clock_gettime` is stored in `f` (Fig. 4.1, lines 6–7) if it exists. Since the virtual address of the `vdso` implementation of `clock_gettime` is initialized when the program is linked and never modified after, the indirect call target stored in `f` should not change during the runtime of the program.

In cases like this, it is inefficient to always require a check of a function pointer against each value in the entire set of valid targets until a match is found as in other CFI schemes (e.g., Clang-CFI [10]). Instead, by caching the indirect call target accordingly, a check against the entire set of valid targets is now effectively reduced to a single check against the last valid call target in most (i.e., 77.6%–89.9%, Fig. 4.2) cases.

Thus, this thesis seeks to answer the following research question: **Using this observation that indirect call targets are changed infrequently, can the runtime performance of CFI**

**enforcement be improved by caching a function pointer's most recent value?**

```
1  int __clock_gettime(clockid_t clk, struct timespec *ts)
2  {
3          int r;
4
5  #ifdef VDSO_CGT_SYM
6          int (*f)(clockid_t, struct timespec *) =
7                  (int (*)(clockid_t, struct timespec *))vdso_func;
8          if (f) {
9                  r = f(clk, ts);
10                 if (!r) return r;
11                 if (r == -EINVAL) return __syscall_ret(r);
12                 /* Fall through on errors other than EINVAL. Some buggy
13                  * vdso implementations return ENOSYS for clocks they
14                  * can't handle, rather than making the syscall. This
15                  * also handles the case where cgt_init fails to find
16                  * a vdso function to use. */
17         }
18 #endif
19 ...
20 }
```

Figure 4.1: code snippet of an indirect call in musl libc `__clock_gettime`. The indirect call target `__clock_gettime` will be set to the `vdso`'s implementation of `clock_gettime` if it exists

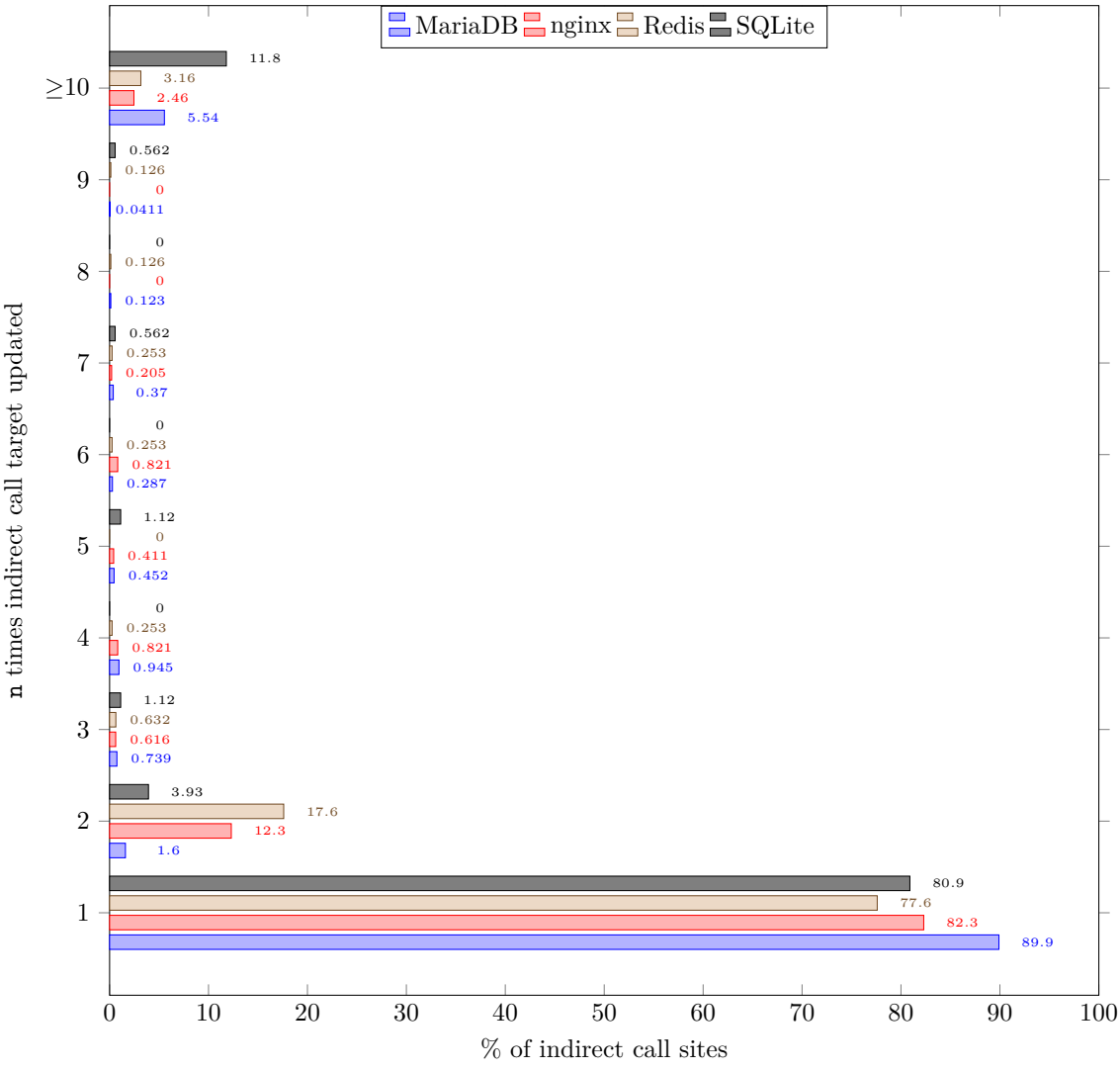



Figure 4.2: Indirect Call Target Update Percentage.
This graph shows the percentage of indirect call sites where the indirect call target was updated n times during the program's runtime

# Chapter 5

# Design

To answer this research question, we designed CacheCFI. CacheCFI is a software-only CFI enforcement scheme that performs forward-edge, control-flow transfer checks. The main idea behind CacheCFI is to develop a novel, inline caching scheme for targets of (indirect) control-flow transfers to reduce the runtime overhead associated with CFI enforcement checks. From a high-level overview, this caching scheme is implemented using *cache lines* in software (Section 5.2). These *cache lines* are stored in a custom `.cfi.cache` (Section 5.3) ELF section with read-only permissions. When the cached call target is updated, any updates to these *cache lines* are done using a secure procedure; otherwise, the *cache lines* remain immutable to avoid corruption.

In addition, CacheCFI is designed to be policy agnostic, such that its control-flow enforcement mechanism is independent of the CFI policy being enforced. As such, CacheCFI can effectively enforce a wide variety of CFI policies, ranging from coarse-grained [19, 28] to single-target CFI policies [38]. Note that while the design of CacheCFI is shown for the x86-64 architecture, it is possible to implement CacheCFI in other architectures too.

## 5.1 CFI Policy Representation

From a high-level, CacheCFI represents CFI policies as mappings from indirect call sites to sets of valid call targets. A policy item is a mapping from a single indirect call site to a set of valid call targets corresponding to this call site (i.e., `policy_item_i:icall_site_i` $\mapsto$ {`valid call targets at i`}). For example, consider a file with two indirect call sites `icall_site_a` and `icall_site_b`. `icall_site_a` has two valid call targets `foo` and `bar` while `icall_site_b` has one valid call target `baz` according to CFI policy `A`. The policy items are thus `policy_item_a:icall_site_a` $\mapsto$ {`foo, bar`} and `policy_item_b:icall_site_b` $\mapsto$ {`baz`} respectively. The CFI policy for this file is thus represented as {`policy_item_a, policy_item_b`}.

This representation enables CacheCFI to be policy agnostic and support any granularity of CFI policy by simply adjusting the set of allowed targets at each indirect call site according to the chosen CFI policy, from coarse-grained policies all the way down to single target policies. For example, if

some CFI policy `B` instead has three valid call targets `foo`, `bar`, and `baz` for `icall_site_a`, CacheCFI can easily represent it as `policy_item_a:icall_site_a` $\mapsto$ {`foo, bar, baz`} instead.

CacheCFI maintains this mapping in a shared library (i.e., the *policy library*) as a series of arrays: one for each policy item. At runtime, the policy items are relocated, and then the section is marked read-only to prevent an attacker from corrupting it.

## 5.2 Cache Line Representation

| Cached call target (8 bytes) | Update count (4 bytes) | Policy item size (4 bytes) | Pointer to policy item (8 bytes) | Indirect call site (8 bytes) |
|---|---|---|---|---|

Figure 5.1: cache line components

CacheCFI uses software-based *cache lines* to enable its caching scheme. A *cache line* has a size of 32-bytes to achieve good alignment and hardware cache behavior. Each *cache line* contains following components:

1. **Cached call target (8 bytes)**: A store of the most recent, valid indirect call target at a given call site, used to perform an inline CFI check.
2. **Update count (4 bytes)**: The number of times the cached call target has been updated (update count, Fig. 5.1), used to determine when to JIT a new verification area.
3. **Policy item size (4 bytes)**: The number of valid targets in the policy item, used to determine size of new verification area to be JIT-ed.
4. **Pointer to policy item (8 bytes)**: Pointer to a CFI policy item, used to verify the indirect call target in the slow path.
5. **Indirect call site (8 bytes)**: Address of the indirect call site, used to determine the return address for the JIT-ed verification area.

## 5.3 CacheCFI-instrumented Binaries

To store these *cache lines*, a new `.cfi.cache` section is introduced into a CacheCFI instrumented ELF binary. This section is hardened with RelRO protections [52] and contains all the *cache lines* for the given binary. In other words, at runtime, after *cache lines* are loaded into memory using the policy library, the dynamic linker resolves the addresses of each policy item and sets this region of memory to read-only.

## 5.4 Runtime CFI Enforcement Scheme

### 5.4.1 CacheCFI Instrumentation

```
1  /* Indirect call target stored in %rax */
2  /* Cached call target stored in 0x17eb(%rip) */
3  /* __cfi_cache_update stored in 0x19c0(%rip) */
4  180e: cmp    0x17eb(%rip),%rax
5  1815: jne    181f
6  1817: call   *%rax
7  ...
8  181f: mov    %rax,%r10
9  1822: lea    0x17d7(%rip),%r11 /* cache line */
10 1829: mov    0x19c0(%rip),%rax /* __cfi_cache_update */
11 1830: jmp    1817
```

Figure 5.2: Sample CacheCFI Instrumented Indirect Callsite

Fig. 5.2 shows an example of what a CacheCFI instrumented indirect call site looks like. The original indirect call instruction, `call *%rax`, is modified to insert runtime CFI enforcement such that: (1) the indirect call target stored in `%rax` is first checked for a match against a cached call target (*fast path*) (Fig. 5.2, line 3); and (2) if no match is found, the instrumentation calls a separate function which performs additional checks and updates the *cache line* accordingly (*slow path*) (Fig. 5.2, lines 7–10).

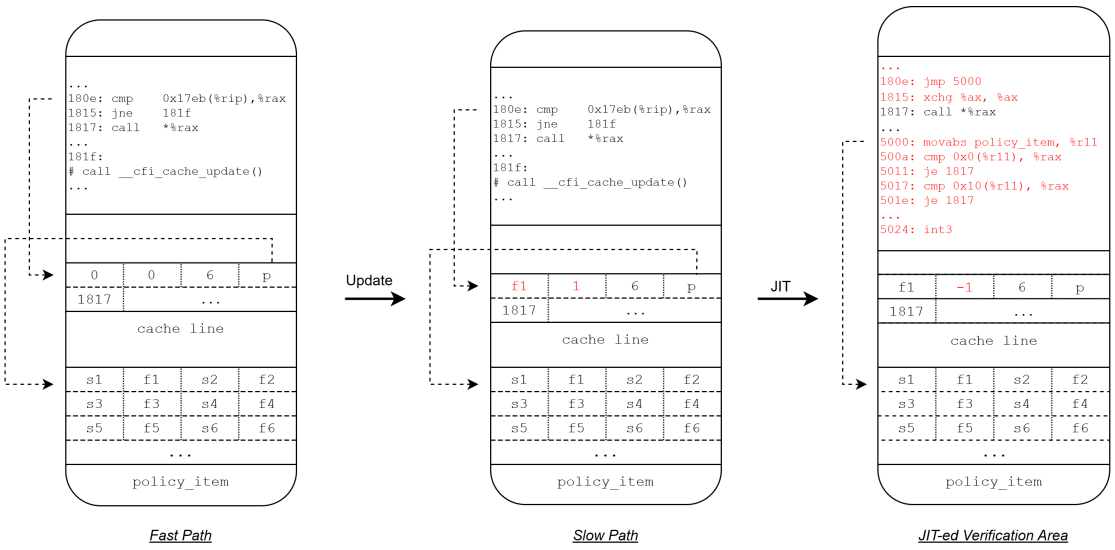



Figure 5.3: CacheCFI Instrumentation Overview. The update step caches the indirect call target and increments the update count. The JIT step JITs a new verification area at this indirect call site, which all future CFI enforcement checks will go through instead.

Zooming out to a high level overview of CFI enforcement in CacheCFI, the indirect call target is first checked against the cached call target (most recent valid call target) in the *cache line* (Fast Path, Fig. 5.3). If the target does not match, a longer lookup through all valid call targets in the policy item is performed, after which the cached call target is updated and the update count is incremented by one (Slow Path, Fig. 5.3). When the *cache line* is updated above a certain number of times, a new verification area is JIT-ed and future CFI checks go through this new verification area instead (JIT-ed Verification Area, Fig. 5.3).

**Fast Path**

At a CacheCFI instrumented indirect call site, the first CFI enforcement check is an inline check against a cached call target (Fig. 5.2, line 3). On a cache hit, the indirect call is executed as per normal (Fig. 5.2, line 5). On a cache miss, the *slow path* is executed instead (Fig. 5.2, line 4).

**Slow Path**

The *slow path* (Fig. 5.2, lines 8–11) is triggered when there is a cache miss (i.e., the indirect call target is different from the cached call target). In the *slow path*, a separate `__cfi_cache_update` function (Fig. 5.2, line 10) is called before the indirect call target. Before this function is called, the indirect call target is first stored in a separate register (`%r10` in the example, Fig. 5.2, line 8), and the register which originally stored the indirect call target is overwritten with the address of `__cfi_cache_update` function (`%rax` in the example, Fig. 5.2, line 9). Now, when the indirect call is executed (Fig. 5.2, line 6), `__cfi_cache_update` is called instead. The `__cfi_cache_update` function does the following:

1. **Preserve register values**: In order to ensure that register values remain the same when the indirect call is executed once the CFI enforcement check passes, register values are pushed onto the stack. This is important to ensure that this function will not affect normal program execution (assuming that the target is valid).

2. **Verify the call target**: The function performs a binary search through the set of valid call targets for the given indirect call site to find a match for `target`. In other words, the function verifies that `target` $\in$ {`valid targets`}. An error is raised when no match is found as that means that the CFI policy is violated.

3. **Claim the corresponding policy library lock**: To ensure that the updating of the cache line is thread-safe in the next step, the function claims a corresponding mutex lock for the binary containing the cache line. This lock will be released after the cached call target has been updated.

4. **Update cache line**: To ensure that only the cached call target and the update count is modified, the function updates the cache line as such:

(a) A copy of the original page (`page_original`) containing the cached call target, `page_copy`, is created with read-write permissions.

(b) The previously cached call target on `page_copy` is overwritten with the new indirect call target. The update count is also incremented by one on `page_copy`.

(c) `page_copy` is set to have read-only permissions.

(d) To prevent an attacker from arbitrarily modifying `page_copy` before (c), the contents of `page_copy` are compared with the `page_original` to ensure that the only difference is the updated cached call target and the update count.

(e) `page_copy` is remapped atop the `page_original`.

5. **Return via tail-call to indirect call target**: Since the indirect call target is valid, the CacheCFI instrumentation needs to ensure that the indirect call target is executed accordingly. All register values stored on the stack are restored and normal execution is resumed via a tail-call to the indirect call target in the register where it is stored (`%r10` in the example, Fig. 5.2). A tail-call is used to prevent the indirect call from returning to `__cfi_cache_update`. Instead, the indirect call will return to the original indirect call site (Fig. 5.2, line 7). As such, CacheCFI's CFI enforcement check does not interfere normal program execution.

### CFI Enforcement Walkthrough

There are three possible cases that can happen under this CFI enforcement scheme. Using the example in Fig. 5.2, we will walk through each of them. Let the cache line contain two valid call targets `foo` and `bar`, and the cached call target be `foo`.

Fast path check succeeds: Let `%rax` store the address of `foo` (i.e., `&foo`).

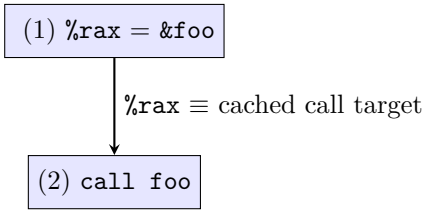



Figure 5.4: Fast Path Check Succeeds Walkthrough

Since `%rax` ≡ `&foo` ((1), Fig. 5.4), the inline check succeeds and `foo` is called ((2), Fig. 5.4).

Fast path check fails: `%rax` currently stores the address of `bar`.

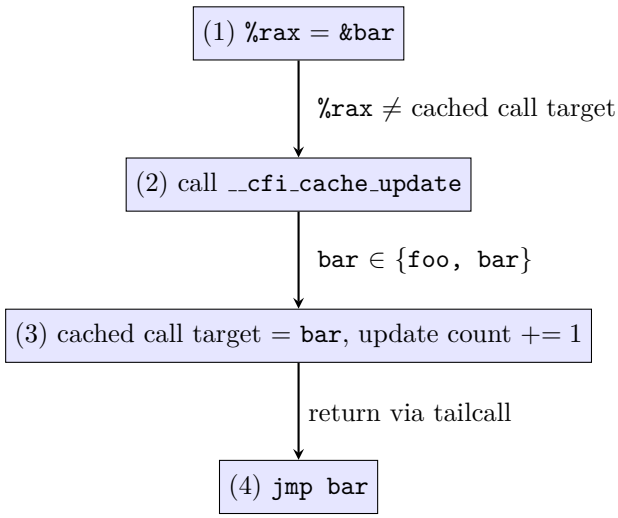



Figure 5.5: Fast Path Check Fails Walkthrough

Since `%rax` $\neq$ `foo` ((1), Fig. 5.5) the *fast path* check fails and the `jne` instruction (Fig. 5.2, line 5) jumps to the *slow path* instead ((2), Fig. 5.5). In the *slow path*, since `bar` $\in$ {`foo, bar`}, the cached call target is updated to `bar` and the update count is incremented by one ((3), Fig. 5.5). Finally, execution returns to `bar` via a tail-call ((4), Fig. 5.5).

Invalid indirect call target: `%rax` currently stores the address of `evil`.

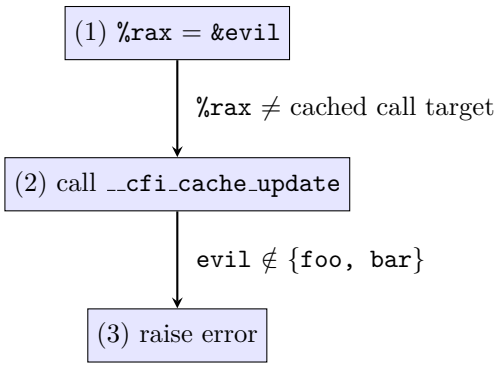



Figure 5.6: Invalid Indirect Call Target Walkthrough

The steps leading up to the call to `__cfi_cache_update` are identical to the previous case. However, since `evil` $\notin$ {`foo, bar`}, the indirect call target verification fails and an error is raised instead ((3), Fig. 5.6).

## 5.5 Optimizations

### 5.5.1 Verification Area JIT-ing

In the case where the indirect call target alternates between addresses at an indirect call site, it means that the *slow path* is always triggered and the cache needs to be continually updated. For example, consider the `sift` function in musl libc (Fig. 5.7). An indirect call is made depending on the function (`cmp`) passed in by the caller (Fig. 5.7, line 1). Thus, a program that calls `sift` multiple times with different values of `cmp` will result in the indirect call target frequently changing. This would lead to significant overhead from repeatedly running the *slow path* to update the cached call target at this indirect call site.

```
1   static void sift(unsigned char *head, size_t width,
2                    cmpfun cmp, void *arg, int pshift, size_t lp[])
3  {
4           unsigned char *rt, *lf;
5           unsigned char *ar[14 * sizeof(size_t) + 1];
6           int i = 1;
7
8           ar[0] = head;
9           while(pshift > 1) {
10                  rt = head - width;
11                  lf = head - width - lp[pshift - 2];
12
13                  if(cmp(ar[0], lf, arg) >= 0 && cmp(ar[0], rt, arg) >= 0) {
14                          break;
15                  }
16                  if(cmp(lf, rt, arg) >= 0) {
17                          ar[i++] = lf;
18                          head = lf;
19                          pshift -= 1;
20                  } else {
21                          ar[i++] = rt;
22                          head = rt;
23                          pshift -= 2;
24                  }
25          }
26          cycle(width, ar, i);
27  }
```

Figure 5.7: musl libc sift

As such, by JIT-ing a new verification area, new checks can be added to act as a faster `if/else` check to avoid the need to continually update the cache. Specifically, a JIT-ed indirect call site means that the fast path (Fig. 5.9) is replaced by a check against each valid call target in the cache line instead (Fig. 5.10). However, note that the vast majority (77.6%–89.9%, Fig. 4.2) of indirect call sites only change their call target once. As such, JIT-ing a new verification area should only

happen when this behavior of constantly changing indirect call target is detected to avoid premature loss of the performance gains from caching.

```
/* Indirect call target stored in %rax */
/* Policy array stored in %r11 */

/* JIT-ed Verification Area */
0x555559555000:      movabs 0x7ffff7eea020, %r11
0x55555955500a:      cmp    0x10(%r11), %rax
0x555559555011:      je     0x5555555558c7 /* <indirect call site> */
0x555559555017:      cmp    0x0(%r11), %rax
0x55555955501e:      je     0x5555555558c7 /* <indirect call site> */
0x555559555024:      int3

/* Indirect call site */
...
0x5555555558c7:      call   *%rax
```

Figure 5.8: Sample CacheCFI JIT-ed Verification Area

```
0x00005555555558be:      cmp    0x173b(%rip), %rax
0x00005555555558c5:      jne    0x5555555558e3
0x00005555555558c7:      call   *%rax
```

Figure 5.9: CacheCFI Fast Path pre-JIT

```
/* Newly JIT-ed verification area at 0x555559555000 */
0x00005555555558be:      jmp    0x555559555000
0x00005555555558c5:      xchg   %ax, %ax
0x00005555555558c7:      call   *%rax
```

Figure 5.10: CacheCFI Fast Path post-JIT

The JIT-ing of a new verification area is implemented as an additional step in the *slow path* (Section 5.4.1) as such:

Claim the corresponding binary lock → *JIT verification area* → Update cached call target.

The verification area is JIT-ed according to the following steps:

1. **Check number of times cached call target has been updated**: JIT-ing a new verification area incurs two sources of overhead, namely: (1) the JIT-ing of the new verification area itself, and (2) `if/else` checks against the set of valid call targets instead of a single check against the cached call target. Thus, JIT-ing does not always result in an improvement in performance. As such, to ensure that a verification area is only JIT-ed when the behavior of call targets changing multiple times is observed, the number of times the cached call target has been updated is tracked in the cache line (Fig. 5.1). Only when this count has exceeded some threshold is a new verification area JIT-ed.

2. **JIT-ing a new verification area**: Memory is first allocated to create the verification area. In the new verification area, the indirect call target is checked against each valid call target (Fig. 5.8, lines 5–9). If a match is found, the indirect call target is executed (Fig. 5.8, line 11). When no match is found, an interrupt is raised indicating that the CFI policy has been violated (Fig. 5.8, line 10).

3. **Update fast path to jump to JIT-ed verification area**: The fast path is updated to now jump to the newly JIT-ed verification area instead. To ensure that the number of bytes used for this is the same as the original fast path (Fig. 5.9, lines 1–2), padded bytes are inserted accordingly (Fig. 5.10, line 3).

   Note that modifying the fast path in the CacheCFI instrumentation means writing to memory pages with read and execute permissions. In order to ensure that W^X protections [6] hold, the same secure-update method is used as in Section 5.4.1. In other words, a copy of the page(s) (Fig. 5.9) containing the corresponding fast path check is first modified, then set to have read-only permissions, after which this copy (Fig. 5.10) is finally remapped atop the original pages(s) with appropriate permissions (read and execute in this case).

### 5.5.2 Page-based Binary Locking

When updating a cache line in a binary, a naive approach to ensure thread-safety would be to claim a mutex lock for the entire binary. However, this incurs a significant performance overhead when multiple threads need to modify different cache lines within the same shared binary file as it means that no two threads can update cache lines at the same time. Hence, to reduce the performance overhead, this locking scheme can be replaced by a page-based one that only locks the page of memory that contains the cache line to be modified.

To achieve the goal of page-based locking per binary, a corresponding mutex needs to be allocated per page of memory within the `.cfi.cache` section in a binary file, thus creating an array of mutexes.

At runtime, by resolving the offset to the segment containing cache lines from the `.cfi.cache` section, the CacheCFI instrumentation can then index into this array of mutexes and claim the lock for the corresponding page of memory.

**Extracting size and offset of `.cfi.cache` section at runtime**: In order to allocate the correct number of mutexes and index into the newly created mutex array, two pieces of information are needed at runtime. Namely, the offset and size of the `.cfi.cache` section. However, ELF sections are not present at runtime, which means that this information would not be available. To resolve this problem, two new symbols are created at compile time which store the size and offset of the `.cfi.cache` section respectively. With this change, at runtime, the offset and size of the `.cfi.cache` section can be extracted by finding these symbols within the dynamic table entries.

Under the page-based binary locking scheme, thread-safety is achieved as a thread needs to claim this lock before modifying a cache line on a given binary. As such, the race condition where two threads update the same cache line a once is prevented. In addition, multiple threads can modify cache lines (on different pages) on the same binary file concurrently, thus reducing the performance overhead from CacheCFI's runtime CFI enforcement.

# Chapter 6

# Implementation

CacheCFI is implemented as a set of modifications to (≈ 600 C++ LOC) LLVM (v15.0.7) and (≈ 1000 C LOC) musl's dynamic linker (v1.2.5) on the x86-64 architecture. Specifically, a custom CacheCFI LLVM backend pass modifies all indirect call sites to insert the CacheCFI instrumentation, thus generating a CacheCFI instrumented binary. Note that CacheCFI is implemented as an LLVM backend pass to ensure that the CacheCFI instrumented code will not be modified by any optimizations from further LLVM passes.

At runtime, in addition to the CacheCFI instrumented binary and any library files used, the modified musl dynamic linker also loads in the policy library so as to initialize all the cache lines and policy items for this executable. The *slow path* is also implemented as part of the modifications made to the musl dynamic linker.

## 6.1 Support For Callee-saved Registers

In the *slow path*, a separate `__cfi_cache_update` function is invoked to perform additional runtime checks before the indirect call is invoked (Fig. 5.2). According to the x86 calling convention, the state of callee-saved registers needs to be preserved across function calls [26]. However, consider the case where the indirect call target is stored in a callee-saved register. In the *slow path*, the indirect call target will be overwritten with the address of `__cfi_cache_update` (Fig. 5.2, line 10). As such, this value needs to be restored before `__cfi_cache_update` returns in the *slow path* in order to adhere to the x86 calling convention.

To achieve this, variants of `__cfi_cache_update_*` are added for each callee-saved register so as to ensure that the indirect call target is restored to the register (Fig. 6.1, line 32) before `__cfi_cache_update_*` returns via the tail-call through `%r10` (Fig. 6.1, line 34).

```
00000000000fec20 <__cfi_cache_update_rbx>:
/* Save register values on the stack except %rbx */
fec21:   push   %rcx
fec22:   push   %rdx
fec23:   push   %rsi
fec24:   push   %rdi
fec25:   push   %r8
fec27:   push   %r9
fec29:   push   %r12
fec2b:   push   %r13
fec2d:   push   %r14
fec2f:   push   %r15

/* Verify target and update cached call target */
fec31:   mov    %r10,%rdi
fec34:   mov    %r11,%rsi
fec37:   call   fdd80 <__cficache_update2>

/* Restore register values stored on the stack */
fec3c:   pop    %r15
fec3e:   pop    %r14
fec40:   pop    %r13
fec42:   pop    %r12
fec44:   pop    %r9
fec46    pop    %r8
fec48:   pop    %rdi
fec49:   pop    %rsi
fec4a:   pop    %rdx
fec4b:   pop    %rcx

/* Restore indirect call target to %rbx */
fec4c:  mov    %r10,%rbx
/* return via tail call through %r10 */
fec4f:  jmp    *%r10

```

Figure 6.1: Sample callee-saved register __cfi_cache_update_* variant

## 6.2 Resolving Register Use Conflicts For Indirect Calls

In the *slow path*, CacheCFI calls `__cfi_cache_update` via the register which originally stored the indirect call target, and uses the `%r10` and `%r11` registers to store the indirect call target and cache line respectively (Fig. 6.1, lines 15–16) in order to perform further target verification and updating of the cached call target. The `%r10` and `%r11` registers are used as they are "temporary register(s)." [26] However, a conflict in register use occurs when the indirect call target is stored in `%r10` or `%r11`. By overwriting the register value with `__cfi_cache_update`, it would mean that either the indirect call target (`%r10` case) or cache line (`%r11` case) is lost.

To resolve this, two separate variants of `__cfi_cache_update_*` are created for each case where the indirect call target is stored in `%r10` or `%r11`. The *slow path* now uses `%rax` to temporarily store the indirect call target/cache line (Fig. 6.2) instead of `%r10/%r11` before calling the respective variant of `__cfi_cache_update_*`. This variant of `__cfi_cache_update_*` overwrites `r10/r11` with the value stored in `%rax` (Fig. 6.3) before performing any further runtime CFI checks, thereby preventing the indirect call target/cache line from being lost. `%rax` is chosen as it is a caller-saved register designated for use as a "temporary register." [26] As such, the value in `%rax` can be freely overwritten before the indirect call.

```
1  /* Indirect call target stored in %r10 */
2  /* Cached call target stored in 0x1cb94(%rip) */
3  302ec5: cmp    0x1cb94(%rip),%r10
4  302ecc: jne    303297
5  302ed2: call   *%r10
6  ...
7  303297: mov    %r10,%rax
8  30329a: lea    0x1c7bf(%rip),%r11 /* cache line */
9  3032a1: mov    0x1f790(%rip),%r10 /*__cfi_cache_update_r10*/
10 3032a8: jmp    302ed2
```

Figure 6.2: Slow path for `%r10`

```
00000000000fec20 <__cfi_cache_update_r10>:
/* Restore value of target %r10 */
fec1d:   mov    %rax, %r10
/* Save register values on the stack */
...
/* Verify target and update cached call target */
fec31:   mov    %r10,%rdi
fec34:   mov    %r11,%rsi
fec37:   call   fdd80 <__cficache_update2>

/* Restore register values stored on the stack */
...
/* return via tail call through %r10 */
fec4c:  jmp    *%r10
```

Figure 6.3: `__cfi_cache_update_r10` variant

In addition, a separate variant of the JIT-ed verification area is created when the indirect call target is stored in `%r11` in order to prevent conflicts with `%r11` being used to store both the indirect call target and the cache line by using `%r10` to store the policy array instead (Fig. 6.4).

```
/* Indirect call target stored in %r11 */
/* Policy array stored in %r10 */
0x5555596752bb:      cmp    0x0(%r10), %r11
0x5555596752c2:      je     0x555555675cca /* <indirect call site> */
0x5555596752c8:      int3
```

Figure 6.4: `%r11` Variant For JIT-ed Verification Area

## 6.3 Thread-safe JIT-ing of Verification Areas

```
180e: cmp    <cached call target>,%rax
1815: jne    <slow path>
1817: call   *%rax
...
190e: cmp    <cached call target>,%rbx
1915: jne    <slow path>
1917: call   *%rbx
```

Figure 6.5: Multiple Indirect calls on same page

New verification areas are JIT-ed at the page level. However, multiple different indirect call sites can exist on the same page of memory which can cause a race condition when JIT-ing a new verification area. Consider the example in Fig. 6.5, where there are two indirect call sites, which store the target in `%rax` and `%rbx` respectively. When thread `A` is JIT-ing a new verification area for

the `%rax` indirect call site, and thread `B` is doing so for the `%rbx` indirect call site at the same time, the following chain of events will lead to a race condition where the updated instrumentation at the indirect call site is erroneously overwritten.

1. `A` and `B` both create a copy of this page, `page_original`
2. `A` and `B` JIT a verification area for their respective indirect call sites, `JIT_A` and `JIT_B`. They each write the `jmp` instruction to `&JIT_A`/ `&JIT_B` (i.e., Fig. 5.10, line 2) on their copies of `page_original` to get `page_A` and `page_B` respectively.
3. `A` remaps `page_original` with `page_A`.
4. `B` remaps `page_A` with `page_B`. However, `page_B` does not contain the `jmp` to `JIT_A` as `B` created a copy of `page_original` instead of `page_A`.

Hence, in order to prevent this race condition from occurring, a page-based locking system is used to synchronize write access to executable pages in the binary. In other words, using the previous example, `A` needs to claim a mutex lock in order to create a copy of the to-be JIT-ed page and `A` can only release the mutex lock once `A` is done with the entire process of JIT-ing a new verification area. Now, before `B` can JIT a new verification area for an indirect call site on the same page, `B` must wait until `A` is done with the JIT-ing process before `B` can create a new copy of this page. As such, `B` will no longer erroneously overwrite the `A`'s modification to the instrumentation at the `%rax` indirect call site.

**Implementation**: Similar to the page-based library locking (Section 5.5.2), an array of mutexes are allocated for the executable segment in the binary. At runtime, by resolving the offset to the executable segment (Fig. 6.6, lines 4–7), the CacheCFI instrumentation can then index into this array of mutexes and claim the lock for the corresponding page of memory (Fig. 6.6, line 9).

```
1  hidden void jit_claim_lock(uint8_t *addr)
2  {
3      /* Get the DSO that corresponds to the address passed in. */
4      struct dso *dso = addr2dso((size_t)addr);
5
6      /* Get index associated with address for page locking */
7      size_t p_num = PAGE_ALIGN((size_t)addr - dso->exe_base);
8
9      pthread_mutex_lock(&dso->jit_locked_pages[p_num]);
10     return;
11 }
```

Figure 6.6: Page-based Locking Code (JIT)

# Chapter 7

# Evaluation

Since one of the main goals of CacheCFI is to determine whether caching a function pointer's most recent value can improve the runtime performance of CFI enforcement, it is necessary to measure the performance overhead of CacheCFI. We do so by running CacheCFI against the SPEC CPU 2017 C benchmarks.

## 7.1 Evaluation Environment

Experiments were performed on an x86-64 host with a 16-core Intel Xeon W-2145 3.70GHz CPU and 64GB of DDR4 RAM. The host runs Debian GNU using Linux v6.1.0-28-amd64 kernel. To reduce noise and improve reproducibility, the CPU was configured to run at a fixed frequency range of 3.00GHz, with dynamic voltage and frequency scaling (Intel Turbo Boost, Intel SpeedStep) disabled. Each benchmark in the SPEC CPU 2017 C benchmark suite was run for 5 iterations each with size=ref using a single thread. In addition, musl (v1.2.5) libc was used for all benchmarks to be consistent with CacheCFI.

### 7.1.1 CFI Policy

A fine-grained, type-based CFI policy (identical to Clang-CFI [10]) was used when evaluating the runtime overhead of CacheCFI. In order to generate this policy, function types are extracted at compile time and processed to create the policy library. To support this, the CacheCFI LLVM pass has to extract function types.

However, the CacheCFI LLVM pass runs at the machine instruction (MI) level. Compilation in LLVM is performed by passing LLVM intermediate representation (IR) through multiple passes. Importantly, a Code Generator pass is used to lower LLVM IR to LLVM machine instructions (MI) [13]. While LLVM IR contains type information, at the MI level, type information is absent. As such, to propagate types down to the MI level, the Code Generator pass is modified. Specifically, type information is propagated down each step in the LLVM Code Generator pass (Fig. 7.1) in order

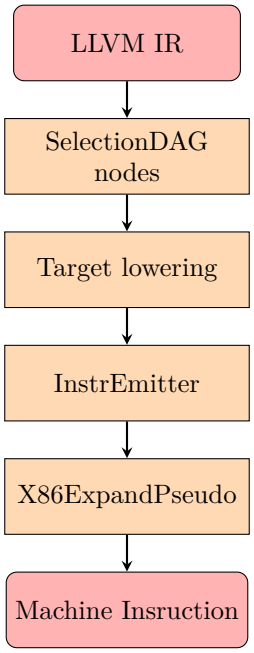



Figure 7.1: Overview of LLVM Code Generator Pass

to ensure that type information will still be available at the MI level. Now, function types can be extracted by CacheCFI LLVM pass to create the type-based policy.

## 7.2 Performance

The performance overhead of CacheCFI was evaluated using the C benchmarks from the SPEC CPU2017 SPECspeed integer and floating point suites [14]. Table 7.1 shows the runtimes (rounded to 3 s.f) and the corresponding overhead compared to the baseline (un-instrumented) binary for each SPEC CPU2017 C benchmark. The runtime overhead ranges between $\approx 0\%$ – 53.8% and the median overhead is 3.01%. `600.perlbench` seems to be an outlier, with a slowdown of 53.8%. While future work is required to determine the specific source of the slowdown, we suspect that it could be due to indirect call targets in `600.perlbench` frequently changing, incurring a large source of overhead from repeatedly updating cache lines.

| **Benchmark** | **Base runtime/s** | **CacheCFI runtime/s** |
|---|---|---|
| 600.perlbench | 689.0 | 1060.0 (53.8%) |
| 602.gcc | 630.0 | 631.0 (0.159%) |
| 605.mcf | 980.0 | 1080.0 (10.2%) |
| 619.lbm | 1010.0 | 1020.0 (0.990%) |
| 625.x264 | 199.0 | 209.0 (5.03%) |
| 638.imagick | 4830.0 | 4760.0 ($\approx 0\%$) |
| 644.nab | 2368.0 | 2504.0 (5.74%) |
| 657.xz | 2350.0 | 2350.0 ($\approx 0\%$) |

Table 7.1: SPEC CPU2017 C Benchmarks Results

# Chapter 8

# Future Work

## 8.1 Runtime Overhead Improvement

As can be seen from the runtime results of benchmarking CacheCFI against the SPEC CPU2017 C benchmarks, there is a significant slowdown. As such, more work needs to be done in order to determine what causes the runtime overhead, and how to decrease it. Specifically, the following possible sources of runtime overhead in the implementation of CacheCFI need to be measured: First, the dynamic linking process. Second, the updating of a cached call target value at an indirect call site. Third, the JIT-ing of a new verification area. Last, the overhead introduced from going through a JIT-ed verification area. The performance can thus be improved by modifying the implementation of CacheCFI according to the measured runtime overhead.

## 8.2 Effectiveness and Compatibility Evaluation

In addition to performance, the effectiveness (in terms of security) and compatibility of CacheCFI also needs be evaluated. The ConFIRM CFI benchmarking suite [17] is a benchmarking suite designed to assess the effectiveness and compatibility of CFI schemes. ConFIRM achieves this by introducing 20 CFI compatibility metrics, and providing a corresponding set of tests. As such, by running CacheCFI against this set of tests in the ConFIRM CFI benchmarking suite, we can effectively determine its effectiveness and compatibility.

## 8.3 Benchmarking against Real-world Applications

The SPEC CPU2017 benchmarks are designed to focus on compute intensive performance [14]. However, this may not provide a full picture on what the potential overhead of CacheCFI is when run against actual applications. Hence, to determine this, CacheCFI should also be benchmarked against real-world applications such as nginx [22], Redis [23], MariaDB [24], and SQLite [25].

## 8.4 Memory Overhead Improvement

In addition to lowering the runtime overhead, the memory overhead of CacheCFI can also be decreased. For example, the mutex array used for thread-safe JIT-ing allocates a mutex for every single page that contains indirect calls. However, this can be excessive as not all pages contain indirect calls. Instead, the memory overhead for this can be lowered by only allocating mutexes for pages that contain indirect calls or using a linked list to track which pages are being JIT-ed.

# Chapter 9

# Conclusion

This thesis observes that indirect call are targets are rarely changed throughout the runtime of a program. Using this observation, this thesis presents the design, implementation, and evaluation of a software-only CFI enforcement scheme named CacheCFI in order to determine whether caching a function pointer's most recent value can improve the runtime performance of CFI enforcement. CacheCFI is implemented as a set of modifications to the LLVM toolchain and musl's dynamic linker. In addition, CacheCFI is evaluated by enforcing a fine-grained type policy on the SPEC CPU 2017 C benchmarks, and incurs a median runtime overhead of 3.01%, and the runtime overhead ranges between $\approx 0\%$ – 53.8%. Future work is still required to investigate the source of the overhead as well as to measure the effectiveness and compatibility of CacheCFI.

# Bibliography


[1] The White House. 2024. Statements of Support for Software Measurability and Memory Safety [Press Release]. https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/memory-safety-statements-of-support/

[2] The White House. 2024. BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[3] The Chromium Projects. 2025. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/

[4] MSRC Team. 2019. A proactive approach to more secure code. Microsoft Blog. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[5] Aleph One. 1996. Smashing The Stack For Fun And Profit. Phrack Magazine 7, 49 (1996). https://phrack.org/issues/49/14.html

[6] OpenBSD. 2003. i386 W^X. https://marc.info/?l=openbsd-misc&m=105056000801065

[7] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In IEEE Symposium on Security and Privacy (S&P). 574–588. https://doi.org/10.1109/SP.2013.45

[8] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In Proceedings of the 12th ACM conference on Computer and communications security (CCS '05). Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[9] 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. 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID). Hong Kong, HK, October 2023. https://cs.brown.edu/~vpk/papers/fineibt.raid23.pdf

[10] C. Tice, T. Roeder, P. Collingbourne, Stephen Checkoway, Ú. Erlingsson, L. Lozano and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. 23rd USENIX Security Symposium (USENIX Security 14). San Diego, CA. August, 2014. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf

[11] J. Corbet. 2022. Indirect branch tracking for Intel CPUs. LWN.net. https://lwn.net/Articles/889475/

[12] LWN.net. 2004. x86 NX support. https://lwn.net/Articles/87814/

[13] A. Sampson. 2015. LLVM for Grad Students. https://www.cs.cornell.edu/~asampson/blog/llvm.html

[14] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation Compute Benchmark. In ACM/SPEC International Conference on Performance Engineering (ICPE). 41–42. https://doi.org/10.1145/3185768.3185771

[15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In USENIX Security Symposium (SEC). 161–176. https://dl.acm.org/doi/10.5555/2831143.2831154

[16] Intel Corporation. 2019. Control-flow Enforcement Technology Specification. https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf

[17] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. 2019. CONFIRM: Evaluating Compatibility and Relevance of Control-Flow Integrity Protections for Modern Software. In USENIX Security Symposium (SEC). 1805–1821. https://dl.acm.org/doi/10.5555/3361338.3361463

[18] Victor van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. 2012. Memory Errors: The Past, the Present, and the Future. In International Symposium on Research in Attacks, Intrusions and Defenses (RAID). 86–106.

[19] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In IEEE Symposium on Security and Privacy (S&P). 934–953.

[20] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return into-libc Without Function Calls (on the x86). In ACM Conference on Computer and Communications Security (CCS). 552–561.

[21] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In USENIX Security Symposium (SEC). 131–148. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-ding.pdf

[22] 2025. nginx. https://nginx.org.

[23] 2025. Redis. https://redis.io.

[24] 2025. MariaDB. https://mariadb.com.

[25] 2025. SQLite. https://www.sqlite.org.

[26] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. AMD64 ABI. 2025. System V Application Binary Interface - AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models). https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build

[27] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14). USENIX Association, USA, 401–416.

[28] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In USENIX Security Symposium (SEC). 337–352.

[29] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In IEEE Symposium on Security and Privacy (S&P). 559–573.

[30] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection using Intel Processor Trace. In ACM Conference on Data and Application Security and Privacy (CODASPY). 173–184

[31] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In ACM Conference on Computer and Communications Security (CCS). 941–951.

[32] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In ACM Conference on Computer and Communications Security (CCS). 1867–1881.

[33] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). Association for Computing Machinery, New York, NY, USA, 927–940. https://doi.org/10.1145/2810103.2813673

[34] Jiliang Zhang, Binhang Qi, Zheng Qin, and Gang Qu. 2018. HCIC: Hardware Assisted Control-Flow Integrity Checking. IEEE Internet of Things Journal 6, 1 (2018), 458–471.

[35] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. 2019. Control-Flow Integrity for Real-Time Embedded Systems. In Euromicro Conference on Real-Time Systems (ECRTS).

[36] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump Oriented Programming: A New Class of Code-Reuse Attack. In ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS). 30–40.

[37] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming without Returns. In ACM Conference on Computer and Communications Security (CCS). 559–572.

[38] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In ACM Conference on Computer and Communications Security (CCS). 1470–1486.

[39] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In ACM Conference on Computer and Communications Security (CCS). 914–926

[40] Wenhao Wang, Xiaoyang Xu, and Kevin W Hamlen. 2017. Object Flow Integrity. In ACM Conference on Computer and Communications Security (CCS). 1909–1924.

[41] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls. In Network and Distributed System Security Symposium (NDSS)

[42] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. ShrinkWrap: VTable Protection without Loose Ends. In Annual Computer Security Applications Conference (ACSAC). 341–350.

[43] Jun Zhang, Rui Hou, Junfeng Fan, Ke Liu, Lixin Zhang, and Sally A McKee. 2017. RAGuard: A Hardware Based Mechanism for Backward-Edge Control-Flow Integrity. In ACM International Conference on Computing Frontiers (CF). 27–34.

[44] Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In Workshop on Hot Topics in Operating Systems (HotOS). 67–72

[45] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In USENIX Security Symposium (SEC), Vol. 98. 63–78.

[46] Tianrou Xia, Hong Hu, and Dinghao Wu. 2024. DEEPTYPE: refining indirect call targets with strong multi-layer type analysis. In Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24). USENIX Association, USA, Article 329, 5877–5894.

[47] Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu. 2024. On bridging the gap between control flow integrity and attestation schemes. In Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24). USENIX Association, USA, Article 371, 6633–6650.

[48] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. 2019. Adaptive Call-Site Sensitive Control Flow Integrity. In IEEE European Symposium on Security and Privacy (EuroS&P). 95–110.

[49] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. $\tau$CFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In International Symposium on Research in Attacks, Intrusions and Defenses (RAID)

[50] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering Code-Injection Attacks With Instruction-Set Randomization. In ACM Conference on Computer and Communications Security (CCS). 272–280.

[51] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. DROP THE ROP: Fine-grained Control-flow Integrity for the Linux Kernel. Black Hat Asia (BHASIA) (2017).

[52] Red Hat Blog. Huzaifa Sidhpurwala. 2019. Security Technologies: RELRO. `https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocationread-only-relro`.

[53] Daniel Gruss. 2018. Software-based Microarchitectural Attacks. (2018).

[54] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In IEEE Symposium on Security and Privacy (S&P). 1–19

[55] Dinghao Liu, Shouling Ji, Kangjie Lu, and Qinming He. 2024. Improving indirect-call analysis in LLVM with type and data-flow co-analysis. In Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24). USENIX Association, USA, Article 330, 5895–5912.

[56] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In IEEE Symposium on Security and Privacy (S&P). 985–999.