TAKING BACK CONTROL: CLOSING THE GAP BETWEEN C/C++ AND MACHINE SEMANTICS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Nathan H. Burow

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

This dissertation would not have been possible without my father Craig who through hiking and cycling taught be to test my limits and the value of perserverance, my mother Rebecca without whose wit and intelligence I would not have been capable of this endeavor, my brother and fellow “mountain-vert” Jackson whose companionship during escapes to the mountains of Colorado helped preserve my sanity during graduate school, and my uncle David who introduced me to programming and thereby set my feet on the path to a PhD.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Mathias Payer for his invaluable guidance and mentorship during my PhD; my fellow members of the HexHive group as well as external co-authors who all provided insight and contributions to my projects that made completing the PhD possible; the original members of the CTF team at Purdue: Scott Carr, Gregory Essertel, Daniele Midi, and Craig West whose shared love of security was a constant inspriation; and last but certainly not least, my great friends Jim and Pam Lembke, for whose fellowship and support I will be forever greatful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ABI	Application Binary Interface
ASLR	Address Space Layout Randomization
CFI	Control-Flow Integrity
CUP	Comprehensive User-space Protection for C/C++
CVE	Common Vulnerability and Exposure
CWE	Common Weakness Enumeration
DEP	Data Execution Prevention
MPK	Memory Protection Keys
MPX	Memory Protection Extensions
OTI	Object Type Integrity
ROP	Return Oriented Programming
TOCTTOU	Time of Check to Time of Use
UaF	Use After Free

ABSTRACT

Burow, Nathan H. PhD, Purdue University, December 2018. Taking Back Control: Closing the Gap Between C/C++ and Machine Semantics. Major Professor: Mathias Payer.

Control-flow hijacking attacks allow adversaries to take over seemingly benign software, e.g., a web browser, and cause it to perform malicious actions, i.e., grant attackers a shell on a system. Such control-flow hijacking attacks exploit a gap between high level language semantics and the machine language that they are compiled to. In particular, systems software such as web browsers and servers are implemented in C/C++ which provide *no* runtime safety guarantees, leaving memory and type safety exclusively to programmers. Compilers are ideally situated to perform the required analysis and close the semantic gap between C/C++ and machine languages by adding instrumentation to enforce full or partial memory safety.

In unprotected C/C++, adversaries *must* be assumed to be able to control to the contents of *any* writeable memory location (arbitrary writes), and to read the contents of any readable memory location (arbitrary reads). Defenses against such attacks range from enforcing full memory safety to protecting only select information, normally code pointers to prevent control-flow hijacking attacks. We advance the state of the art for control-flow hijacking defenses by improving the enforcement of full memory safety, as well as partial memory safety schemes for protecting code pointers.

We demonstrate a novel mechanism for enforcing full memory safety, which denies attackers both arbitrary reads and arbitrary writes at half the performance overhead of the prior state of the art mechanism. Our mechanism relies on a novel metadata scheme for maintaining bounds information about memory objects. Further, we maintain the application binary interface (ABI), support all C/C++ language features, and are mature enough to protect all of user space, and in particular libc.

Backwards control-flow transfers, i.e., returns, are a common target for attackers. In particular, return-oriented-programming (ROP) is a code-reuse attack technique built around corrupting return addresses. Shadow stacks prevent ROP attacks by providing partial memory safety for programs, namely integrity protecting the return address. We provide a full taxonomy of shadow stack designs, including two previously unexplored designs, and demonstrate that with compiler support shadow stacks can be deployed in practice. Further we examine the state of hardware support for integrity protected memory regions within a process' address space.

Control-Flow Integrity (CFI) is a popular technique for securing forward edges, e.g., indirect function calls, from being used for control-flow hijacking attacks. CFI is a form of partial memory safety that provides weak integrity for function pointers by restricting them to a statically determined set of values based on the program's control-flow graph. We survey existing techniques, and quantify the protection they provide on a per callsite basis. Building off this work, we propose a new security policy, Object Type Integrity, which provides full integrity protection for virtual table pointers on a per object basis for C++ polymorphic objects.

1 INTRODUCTION

Systems programming languages such as C/C++ are used to implement widely deployed, security critical applications such as web servers and browsers. The language semantics dictate how and when memory can be accessed, particularly via pointers. Per the language specifications, pointers can only be dereferenced when in bounds of a currently allocated object. Programs written in C/C++ are compiled to machine languages, such as x86, which enforce no such requirements, allowing access to any location in the program's address space. Consequently, there is a semantic gap between high level (C/C++) and machine (x86) languages, which results from C/C++ not enforcing memory or type safety. Memory and type safety can be enforced through compile and runtime support, or left entirely to the programmer as C/C++ do.

Attackers exploit the semantic gap between high level and machine languages, i.e., no memory of type safety, to corrupt program state and execute arbitrary code of their choosing. Such exploits take the form of memory safety violations, where an attacker abuses faulty logic by a programmer to write to unintended memory. In particular, over the last 20 years, control-flow hijacking attacks wherein attackers directly overwrite code pointers [1, 2, 3] have been the primary attack vector [4]. While the details of these attacks have evolved from code-injection attacks [1] to code-reuse attacks [2, 3, 5, 6, 7] in the face of deployed defenses [8, 9, 10, 11], attackers have a significant advantage in the control-flow hijacking arms race: defenses have to be perfect while attacks only require one flaw. Defenses that are strong enough to prevent control-flow hijacking attacks in general [12, 13] have proven to be prohibitively expensive to deploy.

Control-flow hijacking attacks have a significant impact on the security of deployed software. Exploits of memory corruptions remain common [4, 14, 15, 16], and lead to control-flow hijacks. The root cause of these attacks is C/C++ relying on the programmer to enforce memory and type safety. Over the last five years, there have been over 2,000 code execution Common Vulnerabilities and Exposures (CVEs), almost 7,000 CVEs categorized as "buffer errors", and 600 use after free (UaF) CVEs. As this continuing stream of memory corruption CVEs shows, programmer added checks to enforce memory safety are often inadequate. Control-flow hijacking attacks can be classified based on the type of code pointers they target: *backward edge*, i.e., return addresses or *forward edge* pointers, such as function pointers or virtual table pointers.

Control-flow hijacking attacks that target backward edges, e.g., code-reuse attacks that target return addresses known as Return Oriented Programming (ROP), are a significant problem in practice. In the last year, Google's Project Zero has published exploits against Android libraries, trusted execution environments, and Windows device drivers [17, 18, 19, 20, 21]. These exploits use arbitrary write primitives to overwrite return addresses, leading to privilege execution in the form of arbitrary execution in user space or root privileges.

Control-flow hijacking attacks that target forward edges [3, 6] are particularly dangerous for C++ applications. C++ relies on indirect control-flow transfers to implement dynamic dispatch, a key element of polymorphism in object oriented languages like C++. The major web browsers (Firefox, Chrome, Safari, and Internet Explorer/Edge) are written primarily in C++, which does not enforce type safety and is prone to use after free (UaF) and other memory safety errors. Attackers now increasingly use type safety and UaF vulnerabilities to corrupt the browser's memory and hijack the program's control flow, frequently by redirecting dynamic dispatch [22].

Compiler's are ideally situated to prevent these attacks. As compiler's translate high level languages to machine code, they have access to the full semantic information of the program. Consequently, my thesis is that:

> `Compilers can close the semantic gap between C/C++ and machine languages that attackers exploit to perform control-flow hijacking attacks by leveraging static analysis to add runtime instrumentation that enforces full or partial memory safety.`

Full memory safety would virtually eliminate control-flow hijacking attacks by preventing attackers from corrupting memory (some interesting type safety corner cases still arise), but has prohibitive performance overhead, on the order of 100%. We advance the state of the art for full memory safety, but also examine partial memory safety defenses. These defenses protect a subset of program data they deem critical, usually code pointers. Partial memory safety defenses thus allow attackers to perform advanced Data Oriented Programming [23] attacks. However, they prevent the far more common case of directly overwriting code pointers to perform control-flow hijacking attacks, and are practical to deploy today.

To improve the performance of full memory safety, we introduce a new hybrid metadata scheme which is capable of storing and using *per object* metadata for the stack, libc, heap, and globals. Our metadata is precise and does not require altering the program's memory layout. Additionally, we introduce a new way to check bounds that leverages hardware to increase our check's performance. Further, we present a novel use of escape analysis to reduce the number of instrumented stack allocations without loss of protection. This reduction allows scaling our mechanism to include all user-space data. By modifying all pointers so that dereferences fail by default, we guarantee that every pointer dereference is checked. To aid deployment, we successfully handle all system libraries, including libc, and are the first memory sanitizer to do so. Consequently, we guarantee that all user-space pointers are always protected, even after passing them across the kernel boundary. We have no false positives or false negatives on the Juliet test suite, and are twice as fast as the state of the art full memory safety mechanism, SoftBound+CETS [12, 13], with 126% overhead vs 245% on SPEC CPU2006 benchmarks where SoftBound+CETS runs.

Despite our advancements, full memory safety remains prohibitively expensive to deploy in practice. We therefore explore more performant partial memory safety scheme. One such partial memory safety scheme is to integrity protect return addresses, a favorite target for attackers, through shadow stacks. To improve the state of shadow stack design, we conduct a detailed survey of the design space. Our design study includes two novel designs, and considers five shadow stack mechanisms in total. We fully explore the trade-offs of these designs in terms of performance, compatibility, and security. We consider the impact of high level design decisions on runtime, memory overhead, and support for threading, stack unwinding, and unprotected code. Further, we propose novel optimizations for shadow stack implementations.

The most popular form of partial memory safety is Control-Flow Integrity (CFI) [11], which provides partial memory safety by restricting function pointers to a statically determined set of values. Function pointers are lazily verified when they are used for indirect calls, not when written, under CFI. This results in lower performance overhead almost all of CFI's work is done at compile time: at runtime only membership checks to verify the function pointer is in the statically determined set are required. We systematize the different CFI mechanisms (where "mechanism" captures both the analysis and enforcement aspects of an implementation) and compare them against metrics for security and performance. By introducing metrics for these areas, our analysis allows the objective comparison of different CFI mechanisms both on an absolute level and relatively against other mechanisms. This in turn allows potential users to assess the trade-offs of individual CFI mechanisms and choose the one that is best suited to their use case. Further, our systematization provides a more meaningful way to classify CFI mechanism than the ill-defined and inconsistently used "coarse" and "fine" grained classification.

Building off our CFI systematization, we propose a new defense policy, Object Type Integrity (OTI) which guarantees that an object's type cannot be modified by an adversary, i.e., integrity protecting it, thereby guaranteeing the correctness of dynamic dispatch on that object. Integrity protecting the object's type, which is recorded in memory, is effectively a form of partial memory safety. By protecting object types through partial memory safety, OTI mitigates code-reuse attacks by preventing key application control flow data from being corrupted. OTI tracks the assigned type for every object at runtime. Consequently, when the object's type is used for a dynamic dispatch, OTI can verify that the type is not corrupted. Further, OTI requires that each object has a known type, thus preventing the attacker from injecting objects [3], and using them for dynamic dispatch. OTI distinguishes itself from CFI by intervening earlier in the attack, i.e., when the virtual table pointer is written rather than only when it is used, and by being fully precise, instead of relying on target sets. The two can be deployed together, achieving even greater security, as they mitigate different stages of code-reuse attacks that utilize dynamic dispatch.

Contributions

We present novel techniques for mitigating control-flow hijacking attacks. The work presented in this dissertation has all been peer reviewed and published, with the exception of the shadow stack work which is still under review. In particular, our memory safety mechanism, Comprehensive User-space Protection (CUP) [24] appeared at AsiaCCS '18, our shadow stack design and optimization work is under review at ASPLOS '19, our CFI survey [25] was published in the ACM CSUR journal January '17, and our OTI paper [26] appeared at NDSS '18. To summarize, the core contributions of these papers are:

- Novel metadata scheme that enables 2x the performance for full memory safety that supports all user space software, including libc
- Novel shadow stack designs and optimizations, including our recommendation for deployment, Shadesmar, which has 3.65% performance overhead and full support for all C/C++ paradigms
- An evaluation of existing hardware acceleration techniques for partial memory safety, including a description of an ideal mechanism
- A survey and quantitative security comparison of CFI mechanisms
- Object Type Integrity, a novel security policy which allows only the programmer intended target for C++ virtual dispatch

2 CUP: COMPREHENSIVE USER-SPACE PROTECTION FOR C/C++

Despite extensive research into memory safety techniques, only a few mechanisms have been deployed, and exploits of memory corruptions remain common [4, 14, 15, 16]. These attacks rely on the fact that C/C++ require the programmer to manually enforce spatial safety (bounds checks) and temporal safety (lifetime checks). As the continuing stream of memory corruption Common Vulnerabilities and Exposures (CVEs) shows, these programmer added checks are often inadequate. Many of these bugs are in network facing code such as browsers, e.g., CVE-2016-5270, CVE-2016-5210, and servers, e.g., CVE-2014-0226, CVE-2014-0133, allowing attackers to illicitly gain arbitrary code execution on remote systems. Consequently, a memory safety sanitizer that *comprehensively* protects user-space is necessary to find and fix these bugs.

To correctly address memory safety in user-space, there are four main requirements. *Precision* addresses spatial safety by requiring that exact bounds are maintained for all allocations. *Object Awareness* prevents temporal errors by tracking whether the pointed-to object is currently allocated or not. These two requirements are sufficient to enforce memory safety. Adding *Comprehensive Coverage* expands this protection to all of user space by requiring that all data on the stack, heap and globals be protected. *Comprehensive Coverage* implies that all code must be instrumented with the sanitizer, including system libraries like libc. A sanitizer that meets these three requirements is powerful enough to find all memory corruption vulnerabilities in user-space programs. To be useful, such a sanitizer must also be practical. Requiring *Exactness* — no false positives and minimal false negatives — ensures that bugs reported by a sanitizer are real, and that all spatial and most temporal violations are found. We discuss these challenges in Section 5.3.

The research community has come up with many approaches that attempt to address memory safety. State of the art defenses can be divided into two categories: probabilistic and deterministic. Probabilistic defenses [27, 28, 29] can be bypassed by a sophisticated attacker, but provide lower overhead. Deterministic defenses [12, 13, 30, 31] cannot be bypassed but come with significant overhead. Modern defenses respect the Application Binary Interface (ABI), but may alter the memory layout of the program by, e.g., injecting "red zones" [27] or imposing additional alignment constraints [30].

Modern defenses still do not fully address the requirement we identify for a memory safety solution. *Comprehensive Coverage* is largely an open problem, with prior work mostly ignoring the stack and neglecting support for system libraries like libc. Low-Fat Pointers [30] is the only work to protect the stack — providing only spatial safety. No existing work provides spatial and temporal safety comprehensively for all user-space data (stack, heap, globals) and code (program code, libc, libraries). Doing so requires a large amount of additional metadata to protect the extra allocations (Section 2.1.1), which existing schemes are unable to handle. Further, existing tools' overhead per check does not allow them to scale to handle the additional memory surface of the stack and libc. *Comprehensive Coverage* weaknesses are exacerbated by the fact that existing works [12, 27, 30] can miss a check without consequence as the execution continues normally. If a pointer is not instrumented, e.g., for hand written assembly code, external uninstrumented code, or pointers passed from the kernel to user-space, these existing schemes continue without raising a fault. In contrast, a defense mechanism that protects all of user space can require that all pointer dereferences fail by default. The failure then alerts the programmer to the issue, allowing her to annotate the inline assembly code, kernel system call, or recompile the uninstrumented code. By failing by default, a mechanism with *Comprehensive Coverage* that executes a program also guarantees that all pointer dereferences were checked.

Libc is the most commonly used third party library, and a particularly critical part of user-space to protect. It is prone to memory errors, notably the `mem*` and `str*` family of functions (e.g., `memcpy` or `strcpy`). SoftBound, for example, uses intrinsics for some of the functions to catch memory safety issues in their use. Memory errors in libc are not limited to these functions, however as shown by, e.g., GHOST [15], a stack overflow in `getaddrinfo` [16], and a one byte corruption in glibc's `malloc` which led to ASLR breaks and arbitrary execution [32]. Consequently, the *entire* libc needs to be protected, not just certain interfaces. Libc is also the primary interface with the kernel. This is a natural boundary to unprotected code for a user-space defense mechanism. Given its prevalence, vulnerability, and boundary to unprotected code, supporting libc gives strong evidence that a defense mechanism is robust enough to protect all of user-space.

Exactness shows how well a memory safety solution protects against vulnerabilities in practice. The U.S. National Institute of Standards and Technology (NIST) maintains the Juliet test suite. Juliet consists of thousands of examples of bugs, grouped by class from the Common Weakness Enumeration (CWE). Juliet reveals that existing, open source memory safety solutions [12, 13, 27] have both false positives and false negatives (Section 2.4.2).

CUP satisfies all four requirements for a powerful, usable memory sanitizer. We introduce a new hybrid metadata scheme which is capable of storing and using *per object* metadata for the stack, libc, heap, and globals. Our metadata is precise and does not require altering the program's memory layout. Additionally, we introduce a new way to check bounds that leverages hardware to increase our check's performance. Hybrid metadata allows us to meet the *Precision* and *Object Awareness* requirements. CUP presents a novel use of escape analysis to reduce the amount of instrumented stack allocations without loss of protection. This reduction allows scaling our mechanism to include all user-space data, satisfying the *Comprehensive Coverage* requirement. By modifying all pointers so that dereferences fail by default, CUP guarantees that every pointer dereference is checked. Further, CUP successfully handles all system libraries, including libc, the first memory sanitizer to do so. Consequently, we guarantee that all user-space pointers are always protected, even after passing them across the kernel boundary. Our evaluation on Juliet (Section 2.4.2) shows that we have no false positives or negatives, considerably advancing the state of the art for *Exactness*. Further, this precision is achieved with *half* the overhead of SoftBound+CETS, the state of the art for full memory safety, on SPEC CPU2006 benchmarks where both run (CUP runs a super set of the benchmarks that SoftBound+CETS does).

Contributions

We present the following contributions:

- A new hybrid metadata scheme capable of tracking runtime information for all object allocations, and show how it can be applied to memory safety.
- The first sanitizer to fully protect user-space, including libc
- By design, CUP guarantees that every pointer dereference is proven safe statically or checked at run time. Missing checks halt execution.
- A new static analysis for determining what stack variables require active protection, and present a local protection scheme for non-escaping stack variables
- Evaluation of a CUP prototype that, using our hybrid metadata model, results in (i) no false positives and no false negatives on the NIST Juliet C/C++ test suite and (ii) reasonably low overhead (in line with other sanitizers).

2.1 Challenges and Background

Our requirements for *Precision* and *Object Awareness* are designed to enforce spatial and temporal memory safety, which we define here and then use to introduce the notion of a capability ID.

Spatial Vulnerabilities, also known as bounds-safety violations are over/under-flows of an object. Over/under-flows occur when a pointer is incremented/decremented beyond the bounds of the object that it is currently associated with. Even if the out-of-bounds pointer points to another valid object, it does not have the capability to access that object, and the operation results in a spatial memory safety violation. However, this violation is *only* triggered on a dereference of an out-of-bounds pointer. The C standard specifically allows out-of-bounds pointers to exist.

Temporal Vulnerabilities, or lifetime-safety violations, occur when the object that a pointer's capability refers to is no longer allocated and that pointer is dereferenced. For stack objects, this is because the stack frame of the object is no longer valid (the function it was created in returned); for heap objects, this happens as a result of a free. These errors do not *necessarily* cause segmentation faults (accesses to unmapped memory), because the memory may have been reallocated to a new object. Similarly, we cannot simply track what memory is currently allocated, because the object at a particular address can change, which still results in a temporal safety violation. Temporal bugs are at the heart of many recent exploits, e.g., for Google Chrome or Mozilla Firefox as shown in the pwn2own contests [33].

Violating either type of memory safety can be formulated as a capability violation. In our terminology, an object is a discrete memory area, created by an allocation regardless of location (stack, heap, data, or bss under the Linux ELF format). A capability identifies a specific object, along with information about its bounds and allocation status. Pointers retain a capability ID that identifies the capability of the object that was most recently assigned — either directly from the allocation or indirectly by aliasing another pointer [34]. Capabilities form a contract, upon dereference: (i) the pointer must be in bounds, and (ii) the referenced object must still be allocated. Violating the terms of this contract leads to spatial or temporal memory safety errors respectively.

2.1.1 Comprehensive Coverage Challenges

Understanding the scope of the challenge presented by *Comprehensive Coverage* is critical to understanding CUP's design. To illustrate this challenge, we show how intensively programs use different logical regions of memory. While the operating system presents applications with a contiguous virtual memory address space, that address space is partitioned into three logical groups for data: global, heap, and stack spaces.

Stack allocations account for almost all (99.9%) of memory allocations in SPEC CPU2006 (see Table 2.1), and are not fully handled by existing work Section 5.7. This measurement *includes* allocations made in libc. While memory usage in SPEC CPU2006 may not be representative of deployed applications, it is pervasively used to compare new

Table 2.1.: Allocation distribution in SPEC CPU2006.

Memory Type	Allocations	
global	3,900	0.00%
heap	425,433	0.07%
stack	621,043,816	99.90%

defense techniques. Defenses that only handle heap objects thus gain a significant advantage by protecting 1,000x fewer objects.

Stack allocations matter in practice as well. The latest data from van der Veen, et. al. [14, 35] show that stack-based vulnerabilities are responsible for an average of over 15% of memory related CVEs annually since tracking began in November 2002. By comparison, heap-based vulnerabilities account for an average of 25% of memory related CVEs over the same time period. Given the stack's exploitability and prevalence, which stresses memory safety designs, protecting it is a key design challenge for memory safety solutions.

2.2 Design

CUP provides precise, complete spatial memory safety and stochastic temporal memory safety by protecting all program data, including libc (and any other library). Safety is enforced, for all program data, by dynamically maintaining information about the size and allocation status of all objects that are vulnerable to memory safety errors. This information is recorded through our novel hybrid metadata scheme (Section 2.2.1). A compiler-based instrumentation pass is used to add code that records and checks metadata at runtime (Section 2.2.2). We provide a detailed argument for why our instrumentation guarantees memory safety in Section 2.2.3.

A powerful usable memory sanitizer must comply with the following requirements:

1. *Precision*. The solution must enforce exact object bounds, ideally without changing the memory layout (i.e., spatial safety).

2. *Object Awareness*. The solution must remember the allocation state of any object accessed through pointers (i.e., temporal safety).

3. *Comprehensive Coverage*. The solution must fully protect a program's user-space memory including the stack, heap, and globals, requiring instrumentation and analysis of all code, including system libraries such as libc (i.e., completeness).

4. *Exactness*. The solution must have no false positives, and any false negatives must be the result of implementation limitations, not design limitations (i.e., usefulness).

These requirements drive the design of CUP. Fully complying with the *Precision* and *Object Awareness* requirements requires creating metadata for all allocated objects. While it is possible [30, 36] to do alignment based spatial checks without metadata, these schemes lose precision, alter memory layout, and cannot support *Object Awareness*. *Object Awareness* for temporal checks requires metadata to lookup whether the object is still valid [13] or to find all pointers associated with an object and mark them invalid upon deallocations [31]. Consequently, CUP is a metadata based sanitizer.

CUP provides *Comprehensive Coverage*, and in particular protects globals, the heap, and stack by instrumenting all code, including libc. Our hybrid metadata scheme scales to handle the required number of allocations (Section 2.1.1), and our bounds check leverages the x86_64 architecture (Section 2.3.2) to perform the required volume of checks quickly enough to be usable. Additionally, our compiler pass is robust enough to handle libc (Section 2.3.3), making CUP the first memory sanitizer to do so. Protecting libc allows CUP to increase coverage to all user space code, reducing the TCB to the kernel and the CUP runtime library. Further, by protecting libc, CUP demonstrates that it can correctly handle the kernel interface, and guarantee that all user-space pointers are always protected.

Exactness is achieved, and *Comprehensive Coverage* verified, by *failing closed*, making a missed check equivalent to a failed check. We modify the initial pointer returned by object allocation (Section 2.2.2), and our modification marks it illegal for dereference. This modification propagates through aliasing and all other operations naturally. Consequently, we *must* check all uses of the pointer for the program to execute correctly, enabled by our comprehensive coverage of program data, and support of libc. Such an approach results in optimal precision, and requires novel design decisions as shown in Section 5.4, but removes all false negatives. False positives are prevented by maintaining accurate metadata, and having it propagate automatically.

2.2.1 Hybrid Metadata Scheme

To provide *Precision*, *Object Awareness*, and *Comprehensive Coverage*, we introduce a new hybrid metadata scheme that lets us embed a capability ID in a pointer without changing its bit width. This capability ID ties a pointer to the capability metadata for its underlying object. *Precision* is provided by the metadata containing exact bounds for every object, and by not rearranging the memory layout. *Object Awareness* results from having a unique metadata entry for each capability ID.

Providing *Comprehensive Coverage* requires assigning a capability ID to all vulnerable objects in order to associate their pointers with the object's capability metadata. However, the capability ID space is fundamentally limited by the width of pointers. To address this limit, we allow capability IDs to be reused. Consequently, our capability ID space only needs to support the maximum number of *simultaneously* allocated objects. This allows CUP to *comprehensively cover* globals, the heap, and stack for all allocations in long running applications.

Our metadata scheme that draws inspiration from both fat-pointers and disjoint metadata (and is thus a "hybrid" of the two) for 64-bit architectures. We conceptually reinterpret the pointer as a structure with two fields, as illustrated by Listing 2.1. The first field contains the pointer's capability ID. The second field stores the offset into the object. This does not change the size of the pointer, thus maintaining the ABI. Further, when pointers are assigned, the capability ID automatically transfers to the assigned pointer without further instrumentation. Enriching pointers in this manner causes unchecked dereferences to fail by default (see Section 2.3.1), verifying at runtime that all necessary checks have been performed.

Hybrid metadata rewrites pointers to include the capability ID of their underlying object and current offset, creating *enriched* pointers. The size of the offset field limits the size of supported object allocations. The tradeoffs of the field sizes and our implementation decisions are discussed in Section 2.3.1. While we use it for memory safety, this design allows access to arbitrary metadata, and could be applied for, e.g., type safety, or any property that requires runtime information about object allocations.

An application's address space is not affected by our hybrid metadata scheme. All available bytes (2^{48} in 64-bit architectures) remain usable. We simply reinterpret what a pointer means, but the pointer can conceptually be stored anywhere within an application's address space.

Capability IDs in our hybrid-metadata scheme are indexes into a metadata table. Each entry in this table is a tuple of the *base* and *end* addresses for the memory object, required for spatial safety checks. Each object that is currently allocated has an entry in the table, leading to a memory overhead of 16 bytes per allocated object. Note that we do not require per-pointer metadata due to our hybrid scheme. To reduce the number of required IDs to the number of *concurrently* active objects, we allow capability IDs to be reused. Allowing ID reuse thus allows us to protect long running programs, as our limit is on concurrent pointers, not total allocations supported. The security impact of ID reuse is evaluated in Section 5.5.

The metadata table provides strong probabilistic *Object Awareness*. For a temporal safety violation to go undetected, two conditions must hold. First, the capability ID must have been reused. Second, the accessed memory must be within the bounds of the new object. Current heap grooming techniques [37, 38] already require a large number of allocations to manipulate heap state. Adding the requirement that the same capability ID also be used makes temporal violations harder. Section 5.5 contains other suggestions to further increase the difficulty.

We are aware of two memory safety concerns for hybrid metadata: (i) arithmetic overflows from the offset to the capability ID, and (ii) protecting the metadata table. The first concern is addressed by operating on the two fields of the pointer separately. By treating them like separate variables — while maintaining them as one entity in memory — we

```
struct pointer_fields {
  int32 enriched : 1;
  int32 id : 31;
  int32 offset;
}

union enriched_ptr {
  struct pointer_fields capability;
  void *ptr;
}
```

Listing 2.1: Enriched pointer

prevent under/over flows from the offset field modifying the capability ID. The second concern is not relevant for CUP— if all memory accesses are checked, then the metadata table cannot be modified through a memory violation. As CUP is designed to protect all of user space, including libraries like libc, *all* memory accesses are actually checked.

2.2.2 Static Analysis

Our static analysis identifies when objects are allocated or deallocated, and when pointers are dereferenced through an intra-procedural analysis. All pointers passed inter-procedurally are instrumented using our metadata scheme, including all heap allocations (which are manually identified, see Section 2.3.3).

Our analysis divides protected stack allocations into (i) escaping and (ii) non-escaping allocations. An allocation does not escape if the following holds: (i) it does not have any aliases, (ii) it is not assigned to the location referenced by a pointer passed in as a function argument, (iii) it is not assigned to a global variable, (iv) it is not passed to a sub-function (our analysis is intra-procedural excluding inlining), and (v) is not returned from the function. For those that escape, we use our usual metadata scheme so that the bounds information can be looked up in other functions. For those that do not escape, we use an alternate instrumentation scheme.

The optimized instrumentation for non-escaping stack variables creates local variables with base and bounds information. Since these allocations are *only* used within the body of the function, we use local variables for checks instead of looking up the bounds in the metadata table. This reduces pressure on our capability IDs, helping us to achieve *Comprehensive Coverage*.

All other allocation sites requiring metadata are instrumented to assign the object the next capability ID and to create metadata (recording its precise *base* and *end* addresses) — returning an enriched pointer. We create metadata at allocation because it is the only time that we are guaranteed to know the size of the object.

Identifying deallocations for objects is straightforward. Global objects are never deallocated over the lifetime of the program. Heap objects are explicitly deallocated by, e.g., `free()` or `delete`. Stack objects are implicitly deallocated when their allocating function returns. Deallocations are instrumented to mark associated metadata invalid and to reclaim the capability ID.

Pointer dereferences are found by traversing the use-def chain of identified pointers. Dereferences are analyzed intra-procedurally, so we include pointers from function arguments (including variadic arguments) and pointers returned by called functions in the set of allocations for this analysis. We instrument dereferences with a bounds check. Note that the bounds check implicitly checks that the pointer's capability ID identifies the correct object. See Section 2.2.3 for a discussion of the safety guarantees.

CUP also inserts instrumentation to handle int to pointer casts. These are commonly inserted by LLVM during optimization, and have matching pointer to int casts in the same function. In this case, and any others where we can identify a matching pointer to int cast, we restore the original capability ID to the pointer. To date, we have not found a case where an int is cast to a pointer without a matching pointer to int cast.

2.2.3 Memory Safety Guarantees

We discuss how CUP guarantees spatial memory safety and probabilistically provides temporal safety. We assume that all code is instrumented and capability IDs are protected against arithmetic overflow (as proposed).

For code that we instrument, we keep a capability ID (and thus metadata) for every memory object that can be accessed via a pointer. This subset is sufficient to enforce spatial memory safety. Objects that are not accessed via pointers are guaranteed to be safe by the compiler (if you are reading an `int`, it will always emit instructions to read the correct 4 bytes from memory).

Pointers can be used to read or write arbitrary memory. Further, the address that they reference is often determined dynamically. Thus, pointers require dynamic checks at runtime for memory safety guarantees. As defined in Section 2.1, memory objects define capabilities for pointers. These capabilities include the size and validity of the object. We only create capabilities when objects are allocated at runtime. Objects can change size due to, e.g., `realloc()` calls, in which case we update our metadata appropriately by changing *base* and *end* to the new values (Section 2.3.2). Thus, we always have correct metadata for every object that has been created since the start of execution. The metadata for objects that have not been created yet is invalid by default.

Pointers can receive values in five ways. First, pointers can be directly assigned from the memory allocation, e.g., through a call to `malloc()`. We have instrumented all allocations to return instrumented pointers. Second, they can receive the address of an existing object, via the `&` operator. We treat this as a special case of object allocation and instrument it. Third, pointers can be assigned to the value of another pointer. As all existing pointers have been instrumented under the first two scenarios, this case is covered as well. Fourth, pointers can be assigned the result of pointer arithmetic. This is handled naturally, with our separate loads preventing overflows into the capability ID.

The fifth scenario is a cast from an int to a pointer. This is exceedingly rare in well written user-space code. However, the compiler frequently inserts these operations in optimized code. As a result, we have to allow these operations. We assume that all ints casted to pointers were previously pointers, and thus instrumented.

Variadic arguments are also given a capability ID when passed to variadic functions. This ensures that variadic functions can only read the arguments explicitly given, and, since all pointers passed to functions are individually given their own capability ID, all writes (e.g., when `%n` is used in the format string given to `printf`) are equally protected.

So far we have established that all pointers are enriched with capabilities that accurately reflect the state of the underlying memory object. Memory safety violations occur when pointers are dereferenced [4]. We instrument every dereference to check the pointers capability and ensure that the dereference is valid. Because each pointer has a capability and each capability is up-to-date this ensures full memory safety.

A program is memory safe before any pointer dereference happens. We have shown that each type of pointer dereference is protected. Consequently, if our checks are correct, every pointer dereference is valid. Thus, showing memory safety for the program depends on showing that our checks are correct.

Spatial safety requires a simple bounds check. The correctness of this check depends only on the validity of the bounds used. We have shown that the capability ID associated with a pointer, and the metadata that ID references are always correct, which in turn implies that the spatial safety check is correct.

Temporal safety in our system requires that either: i) the capability ID has not been reused, or ii) the pointer not be in bounds for the new capability ID. CUP allows capability ID reuse, so our temporal guarantees depend on how difficult exploiting ID reuse is. To successfully reuse an ID, the attacker needs to dictate the location of the new object assigned to the ID that she wants to reuse. Randomized memory allocators such as DieHard [28, 29] make controlling the memory location of a particular allocation extremely difficult. Randomizing ID reuse adds another independent variable for an attacker to control. As an example, with 20 bits of entropy from the allocator and 10 bits of entropy from randomized IDs, a total of 2^{30} allocations would be required to defeat our temporal protection.

2.3 Implementation

We implemented CUP on top of LLVM version 4.0.0-rc1. Our compiler pass is ≈2,500 LoC (lines of code), the runtime is another ≈300 LoC for ≈2,800 LoC total. The line count excludes modifications to our libc, which required only light annotations (Section 2.3.3). Our pass runs after all optimizations, so that our instrumentation does not prevent compiler optimizations. This also reduces the total amount of memory locations that must be protected, reducing capability ID pressure.

Here we discuss the technical details of how we implemented CUP in accordance with our design (Section 5.3). We first discuss how our hybrid metadata scheme is implemented. Next we present how we find the sets of allocations and dereferences required by our design. With the metadata implementation in mind, we then show how we instrument allocations and dereferences. With these details established, we discuss the modifications required to libc for it to work with CUP.

2.3.1 Metadata Implementation

Our metadata scheme consists of four elements: (i) a table of information, (ii) a book-keeping entry for the next entry to use in that table, (iii) a free list (encoded in the table) that enables us to reuse entries in the table, and (iv) how to divide the 64 bits in a pointer between the capability ID and offset in our enriched pointers (Section 2.2.1). Our metadata table is maintained as a global pointer to a `mmap`'d region of memory. Similarly, the next entry in that table is a global variable known as *next_entry*.

By `mmap`'ing our metadata table, we allow the kernel to lazily allocate pages, limiting our effective memory overhead. Further, our ID reuse scheme reduces fragmentation of our metadata since it will always reuse a capability ID before allocating a new one. This also helps improve the locality of our metadata lookups, reducing cache pressure. Alternative reuse schemes with better temporal security are discussed in Section 5.5.

To implement our capability ID reuse scheme, we update *next_entry* using our free list. The first entry in our metadata table is reserved (Section 2.3.2), so *next_entry* is initially one. The free list is encoded in the *base* fields of each free entry in the table. These are all initialized to zero. When an entry is free'd, the *base* field is set to the *offset* to the next available table entry. When we add a metadata entry, *next_entry* is incremented, and the

```
uint_32 next_entry = 1;

//This is done inline, functions are illustrative
void *on_allocation(size_t base, size_t end){
    size_t offset = table[next_entry].base;
    table[next_entry].base = base;
    table[next_entry].end = end;
    uint_32 ret = 0x80000000 & next_entry;
    next_entry = next_entry + offset + 1;
    return (void *)(ret << 32);
}
void on_deallocation(int id){
    table[id].base = next_entry - id - 1;
    table[id].end = 0;
    next_entry = id;
}
```

Listing 2.2: Free list

offset is added. When an object is deallocated, we have to update the *base* field for its corresponding capability ID (*ID*) to maintain the free list correctly. This requires calculating the *offset* to the next free entry. C code illustrating these operations is in Listing 2.2.

The final implementation decision for our metadata scheme is how to divide the 64 bits of the pointer between the capability ID and offset. We use the high order 32 bits to store our enriched flag and capability ID (Listing 2.1). This leaves the low order 32 bits for the offset. Limiting the offset to 32 bits does limit individual object size to 4GB under our current design (with up to 2^{31} such allocations). However, hardware naturally supports 32-bit manipulations, improving the performance of our implementation. Further, having a 31-bit capability ID space is crucial for protecting the entire application (Section 2.1.1). The enriched bit plays two roles. First, it causes all dereferences of enriched pointers to fail. Consequently, the fact that a program runs guarantees that all pointer dereferences were correctly checked. Second, it enables us to support compatibility with unprotected code, by allowing the correct dereference of unenriched pointers by means of a mask that voids our pointer reconstruction. Unprotected code is not supported by default, however the mechanics are described here, and the implications of enabling unprotected code are discussed in Section 5.5.

Note that no matter how a 64-bit pointer is divided, no limits are placed on the application address space. The entire base pointer is stored in the metadata, and so the offset can reach any part of the address space. The only limit is the size of an individual object, which is restricted by the space available for the offset in the enriched pointer.

With a minimal allocation size of 8 bytes, a 31-bit ID allows for at least 16 GB of allocated memory. In practice, much more memory can be allocated as objects are usually larger than 8 bytes. When fully allocated, our metadata table uses `2GB * sizeof(struct Metadata)`, see Listing 2.3. Note that CUP only allocates pages for ID's that are actually used.

2.3.2 Compiler Pass

Our LLVM compiler pass operates in two phases: (i) analysis, and (ii) instrumentation. As per our design, the analysis phase first determines a set of code points that require us to add code to perform our runtime checks, and the instrumentation phases adds these checks. These checks have been optimized to let the hardware detect bounds violations rather than doing comparisons in software. Listing 2.3 has a running example for stack objects.

Analysis Implementation

The first task of our pass is to find the set of object allocations that we must protect to guarantee spatial safety Section 5.3. Heap-based allocations via `malloc` are found by our instrumented musl libc (see Section 2.3.3). Stack-based allocations are found by examining `alloca` instructions in the LLVM Intermediate Representation (IR). These are used to allocate all stack local variables. We only target allocations which can be indirectly accessed via, e.g., pointers. In practice, this means that we need to protect arrays and address-taken variables on the stack, all others are accessed via fixed offsets from the frame pointer. Arrays are trivially found by checking the type of `alloca` instruction as LLVM's type system for their IR includes arrays. LLVM's IR has no notion of the `&` operator. However, clang (the C/C++ front end) inserts markers — `llvm.lifetime.start` — which we use to identify stack allocations that have their address taken.

```
struct Metadata{
    size_t base;
    size_t end;
}

struct Metadata *table;

//This is done inline, functions are illustrative
static inline size_t check_bounds(size_t base, size_t end, size_t check)
   {
    //High order bit is 0 if check passes, 1 otherwise
    size_t valid = (check - base) | (end - (check + size));
    valid = valid & 0x8000000000000000;
    return valid;
}
static inline void *check(void *ptr, uint_32 size){
    size_t tmp = (size_t)ptr;
    //Mask Supports compatibility mode, otherwise omitted
    size_t mask = ptr >> 63;
    uint_32 id = (tmp >> 32) & 0x7fffffff;
    id = id & mask;
    size_t base = table[id].base;
    size_t end = table[id].end;
    size_t check = base + (uint_32)ptr;
    return (void *)(check_bounds(base, end, check) & check);
}
void set(int *x, int val){
    //Size of 4 inferred by compiler for int type
    *(check(x, 4)) = val;
}
//example of dereferencing an escaping and a local stack array
int main(void){
    int escapes[5];
    escapes = on_allocation(escapes, escapes+5*4);
    set(escapes[2], 10);

    int local[5];
    size_t local_base = local;
    size_t local_end = local+5*4;
    *(local & check_bounds(local_base, local_end, local+2*4)) = 10;

    on_deallocation(escapes);
}
```

Listing 2.3: Instrumentation example

CUP also protects global variables. We only need to protect global arrays (non-pointer globals are inherently memory safe). Global arrays present a challenge for our instrumentation scheme. CUP relies on changing pointer values. Unfortunately in C/C++ it is illegal to assign to a global array once it has been allocated. This means that we cannot change the pointer's value. To address this challenge, we create a new global pointer to the first element in the array, and instrument that pointer. We then replace all uses of the global array with our new pointer that can be manipulated as described above. The new pointer must be initialized at runtime, once the address of the global array it replaces has been determined. To do this, we add a new global constructor that initializes our globals. The constructor is given priority such that it runs before any code that relies on our globals.

Instrumentation

Our compiler is required to add two types of instrumentation to the code: (i) allocation, and (ii) dereference. Allocation instrumentation is responsible for assigning a capability ID to the resulting pointer, creating metadata for it, and returning the enriched pointer. A subcategory of allocation instrumentation is handling deallocations — where metadata must be invalidated and the free list updated. Dereference instrumentation is responsible for performing our bounds check, and returning a pointer that can be dereferenced. While our runtime library provides functions for the functionality described in this section (for use in manual annotations), all of our compiler added checks are done inline. Listing 2.2 and Listing 2.3 contain examples for these operations for stack based allocations.

Allocation Instrumentation CUP requires us to rewrite the pointer for every allocation that we identify as potentially unsafe. Our rewritten or "enriched" pointer contains the assigned capability ID, has the enriched bit set, and the lower 32 bits (which encode the distance from the *base* pointer) are set to 0. All uses of the original pointer are then replaced with the new, instrumented pointer. At allocation time, we use the *next_entry* global variable as the capability ID, and then update *next_entry* as described in Section 2.3.1. See the `escapes` variable in `main()` in Listing 2.3.

Note that this creates a pointer which cannot be dereferenced on x86_64, which requires that the high order 16 bits all be 1 (kernel-space) or 0 (user-space). As a result, any dereference without a check will cause a hardware fault. Consequently, for any program that runs correctly we can guarantee that all pointers to protected allocations are checked on dereference. This is in contrast to other schemes [12] that fail open, i.e., silently continue, when a check is missed, sacrificing precision. In LLVM-IR allocations that need protection are relatively easy to identify. However, only a subset of `load` and `store` instructions actually correspond to dereferences that need to be checked. By failing closed, we can aggressively limit the `loads` and `stores` that are checked, while still knowing that all dereferences are being checked at runtime.

When an object is deallocated, we insert code to update the free list in our metadata table as per Section 2.3.1 and Listing 2.2. Further, we mark the *end* address 0 to invalidate the entry.

Dereference Instrumentation To dereference a pointer, two things need to be done. First, we need to recreate the unenriched pointer. Then, using the metadata from the enriched pointer's capability ID, we need to make sure that the unenriched pointer is in bounds.

To create the unenriched pointer, we first retrieve the high order bit (which indicates whether the pointer is enriched or not). When supporting unprotected code, we create a 64-bit mask with the value of this bit. We then extract the capability ID, and `AND` it with this mask. If the pointer was *not* enriched, this yields an ID of 0 and otherwise preserves the capability ID. This step is skipped by default, as we assume all code is protected. We then lookup the *base* pointer for the capability ID, and add the offset to it. See `check` in Listing 2.3.

In the case where the pointer was not enriched, we lookup the reserved entry 0 in our metadata table. This entry has *base* and *end* values that reflect all of user-space (`0` to `0x7fffffffffff`). Thus, performing our unenrichment on an unenriched pointer has no effect, and our spatial check below simply sandboxes it in user-space.

Our spatial check performs the requisite lower and upper bounds check. Note, however, that on the upper bound we have to adjust for the number of bytes being read or written. This adjustment adds significantly to our improved precision against other mechanisms (Section 2.4.2). To illustrate its importance consider the following. An `int` pointer is being dereferenced, meaning the last byte used is the pointer base + 4 bytes - 1, while for a `char` pointer, the last byte used is the pointer base + 1 byte - 1. Equation 2.1 shows our bounds check formula, the size of the pointer dereferenced is `element_size`.

$$base \leq ptr + element\_size - 1 < base + length \tag{2.1}$$

Hardware Enforcement The check in Equation 2.1 could naively be implemented with comparison instructions and a jump — resulting in additional overhead. We propose a novel way to leverage hardware to perform the check for us. We observe that the difference between the adjusted pointer and the *base* address should always be greater than or equal to 0. Similarly, the *end* address minus the adjusted pointer should always be greater than or equal to 0. Consequently, the high order bit in the differences should always be 0 (x86_64 with two's complement arithmetic). We `OR` these two differences together, mask off the low order 63 bits, and then `OR` the result into the unenriched pointer. If it passed the check, this changes nothing. If it failed the check, it results in a invalid pointer, causing a segmentation fault when dereferenced. Listing 2.3 shows this optimized check in `check_bounds()`.

2.3.3 LIBC

Libc is the foundation of nearly every C program and therefore linked with nearly every executable. Unfortunately, many of its most popular functions such as the `str*` and `mem*` functions are highly prone to memory safety errors. They make assumptions about program state (e.g., null terminated strings, buffer sizes) and rely on them without checking that they hold. Prior work [12, 27, 30] assumes that libc is part of the trusted code base (TCB).

In contrast, CUP removes libc, including kernel invoking functions like `malloc` and `free`, from the TCB by instrumenting libc with our compiler. The majority of the instrumentation is automatic, with few exceptions such as the memory allocator, system calls, and functions implemented in assembly code. Dealing correctly with all of these cases demonstrates the CUP is robust enough to protect all of user space for real world code, and does not contain hidden design flaws that would prevent its adoption as a development tool. The most mature libc implementation that we are aware of that compiles with Clang-4.0.0-rc1 is musl [39]. Our instrumented musl libc is part of CUP.

Dynamic Memory Allocator

The dynamic memory allocator is responsible for requesting memory for the process from the kernel, and returning it. To do so efficiently, most allocators — including musl — modify user requests. In particular, musl rounds up the number of bytes requested. Further, musl maintains metadata inline on the heap in the form of headers before each allocated segment. Consequently, the allocator's view of memory is different than that of the program.

To compensate for this difference, we manually instrumented musl's allocator. We ensure that pointers returned to the application are instrumented with the programmer specified length, not the rounded length. Doing so requires two annotations in `malloc` and `free` to adjust the pointer's capability to allow for the headers and rounded length when the pointer is used by the allocator. Doing so allows us to prevent vulnerabilities that rely on corrupting heap metadata [32], while still removing the allocator from the trusted computing base by instrumenting its internals.

An interesting corner case is `realloc()`. By design it changes the *end* address. Additionally, it can change the *base* address if it was forced to move the allocation to find sufficient room. We manually intervene in both cases (one annotation per) to keep our metadata table up-to-date.

Kernel Boundary

User-space code interacts with the kernel through the system call API. For CUP to correctly protect all user-space code, this boundary must be handled so that pointers passed to the kernel are unenriched, and pointers returned from the kernel are enriched. Almost all system calls are made through libc. Consequently, to show that CUP correctly handles system calls, we instrument them in musl. System calls are made through a dedicated API containing inline assembly in musl. We initially instrumented this API to ensure that no enriched pointers are passed to the kernel. Unfortunately, this is insufficient as structs containing pointers are passed to the kernel. Consequently, we required more context than the narrow system call API provided. This forced us to find the actual system call sites and add one annotation per pointer contained in the struct to unenrich these pointers. When pointers are returned from the kernel through, e.g., `mmap` or `brk`, they are annotated to instrument them with their known size. To avoid this (limited) manual annotation effort, future work could define the system call boundary in, e.g., a list that specifies function names, and the instrumentation to be added.

The same level of effort would be required for any boundary to uninstrumented code. As library boundaries are well defined, we consider this level of effort reasonable, particularly for developers, who have intimate knowledge of their code bases.

Assembly Functions

musl implements many of the `mem*`, `str*` functions in assembly for x86. As a result, clang is unaware of these functions as they are directly assembled and linked in. We therefore manually instrumented these functions. Our manual effort was approximately ten assembly instructions per function. These instructions call our runtime support library, while preserving register state after our intervention. Setting up these calls would be trivial for any developer writing inline assembly. Supporting them automatically would require a binary only tool, with a corresponding overall loss of precision. Consequently, CUP accepts this level of manual effort.

Other Libc Concerns

`printf()` can be used to modify memory via the `%n` format. Similarly, attackers commonly pass format strings that cause memory beyond the supplied arguments to be read, resulting in memory leaks that are used to bypass, e.g., ASLR. Both of these attacks rely on using pointer's outside of their assigned capability, and so are prevented by CUP.

Libc, and for C++ libc++, support non standard control flow via `setjmp`, `longjmp`, and exceptions. `setjmp` and `longjmp` are implemented in assembly, and rely on pointers. Consequently, we manually added instructions to call our runtime library to unenrich the pointers. This required the same level of effort as the other assembly function in libc. Exception handling relies on libc++, and in particular libunwind for stack unwinding. Recompiling these with CUP causes exceptions to be handled correctly.

Manual Annotation Effort

Any pointer passed to uninstrumented code must be unenriched, and similarly, any pointer returned from uninstrumented code should be enriched. The programming effort needed to perform these actions is one line of code for enrichment (`on_allocation`) and one to two lines of code for unenrichment (`check`) for every pointer going to and returning from the uninstrumented code. Two lines of code are needed for unenrichment when the unprotected code can access more than one byte, e.g., `read`. In this case, the programmer must check that both the start of the buffer and the end of the buffer (as determined by the number of bytes the caller intends to access) are in bounds. As an example of the cumulative level of effort required, we manually modified 37 lines of code in musl, including the assembly functions discussed above. The annotations we added to libc places `malloc` and `free` in the TCB.

2.4 Evaluation

We evaluate CUP along two axes. First, we show that its performance is competitive with existing sanitizers - and slightly better on benchmarks where both CUP and the existing tools run. Further, our performance is markedly better than other tools that provide both *Precision* and *Object Awareness*, given that CUP provides *Comprehensive Coverage* (we do more checks in the same amount of time). Second, we demonstrate our *Exactness*, showing that we have no false positives or false negatives on the NIST Juliet test suite. The robustness

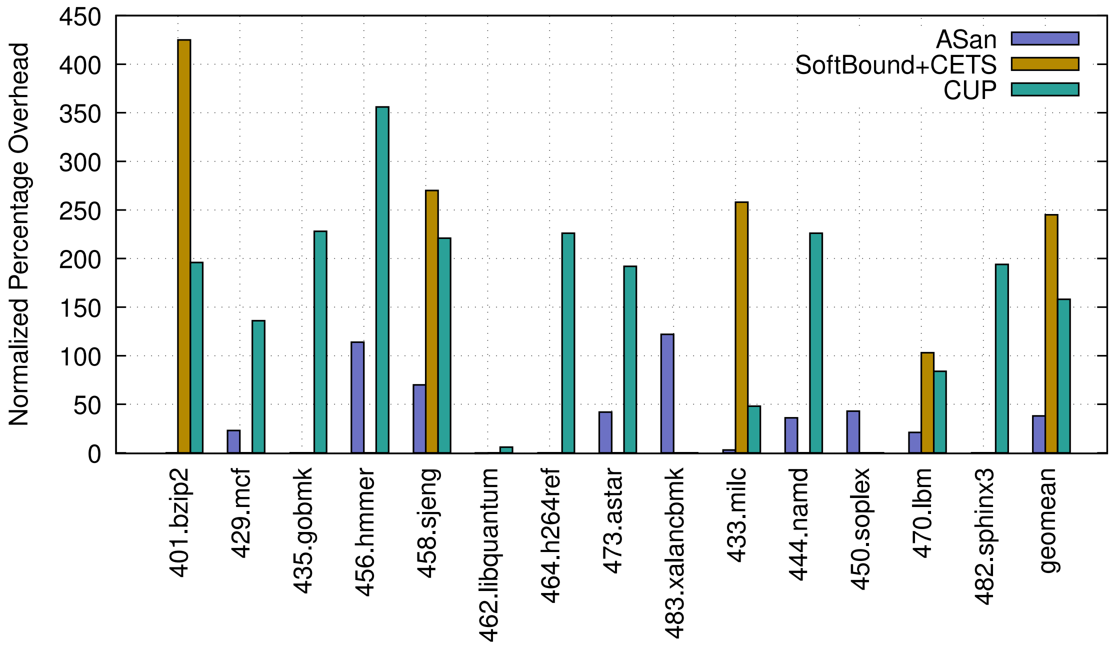


Figure 2.1.: Performance overhead as percentage normalized to baseline using musl. Benchmarks missing entries indicate the mechanism failed to run. Geometric means are calculated over benchmarks where the mechanism ran, and are not directly comparable.

of CUP and its ability to cope with the full spectrum of corner cases found in real code is demonstrated by supporting libc, making CUP the first sanitizer to do so.

All experiments were run on Ubuntu 14.04 with a 3.40GHz Intel Core i7-6700 CPU and 16GB of memory. For SoftBound+CETS we used the latest version [1]. For AddressSanitizer [27], we used the version based on LLVM-4.0.0-rc1.

2.4.1 Performance

Performance is an important requirement for any usable sanitizer. We evaluate the performance of CUP with musl. For comparison, we also measure the performance overhead of similar sanitizers, using glibc as the baseline. Musl has effectively the same performance as glibc on SPEC CPU2006, with minor variations on different test cases. ASan is the closest

[1] commit 9a9c09f04e16f2d1ef3a906fd138a7b89df44996

open source related work that is compatible with LLVM-4.0.0-rc1. SoftBound+CETS is open source, but relies on LLVM-3.4 and can only run a small subset of the SPEC CPU2006 benchmarks. Its performance results are reported here, but are not directly comparable. Low-Fat Pointers (developed concurrently with this work) was open sourced after the submission of this work.

Figure 2.1 summarizes the performance results for CUP. We have 158% overhead vs baseline, compared to 38% for ASan. Note that ASan only provides probabilistic safety, i.e., if a pointer is incremented past guard zones then violations are not detected compared to CUP which catches all spatial memory safety violations by tracking the validity and bounds of pointers. The geometric means in the figure are not directly comparable, as they are calculated over different sets of benchmarks (those where the mechanism actually ran). On benchmarks where both run, CUP has half the overhead of SoftBound+CETS's (126% vs. 245%). Further, compared to these existing tools, we offer stronger, more precise coverage, while being able to run more SPEC CPU2006 benchmarks. Note that both ASan and SoftBound+CETS do not instrument the libc standard library. As we show in Section 2.4.2, CUP has 10x the coverage of SoftBound+CETS, for less overhead. Low-Fat Pointers [30] reports 16% to 62% overhead depending on their optimization level. They achieve this by using clever alignment tricks to avoid metadata look ups. While resulting in low performance overhead, this approach has two limitations: (i) they round allocations up to the nearest power of two, losing precision for bounds checks, and (ii) their design cannot support temporal checks which require metadata.

2.4.2 Juliet Suite

NIST maintains the Juliet test suite, a large collection of C source code that demonstrates common programming practices that lead to memory vulnerabilities, organized by Common Weakness Enumeration (CWE) numbers [40]. Every example comes with two versions: one that exhibits the bug and one that is patched. We extracted the subset of tests for heap and

Table 2.2.: Juliet suite results.

	False Neg.	False Pos.
CUP	0 (0%)	0 (0%)
SoftBound+CETS	1032 (25%)	12 (0.3%)
AddressSanitizer	315 (8%)	0 (0%)
Total Tests	4038	

stack buffers[2]. We compiled all sources with our pass, as well as with SoftBound+CETS and with ASan. Every patched test should execute normally. If a memory protection mechanism prematurely kills a patched test, we call it a false positive. Conversely, every buggy test should be stopped by the memory safety mechanism. All three memory protection mechanisms kill the process in case of a memory violation. If the process is not killed, we count it as false negative.

We found four architecture dependent bugs in the Juliet test suite. These tests attempt to allocate memory for a structure containing exactly two `ints`. However, erroneously, the examples use the size of a pointer to the two `int` structure when allocating memory. (e.g., `malloc(sizeof(TwoIntStruct*))`). On architectures which do not define pointers as twice the size of `ints` (including 32-bit x86), such a mistake would lead to a memory violation when reading or writing the second int. On the x86_64 architecture, though, the size of a pointer and the size of the two `int` structure are the same. These test cases show a type violation and not a memory safety violation for x86_64. Thus, while semantically incorrect, no true memory violation occurs.

Table 2.2 summarizes the results. Out of 4,038 tests that should not fail, we incur no false positives. ASan and SoftBound+CETS show a 0% and 0.3% respectively. We produce no false negatives, while ASan produces an 8% false negative rate, and SoftBound+CETS has a 25% false negative rate.

Most of the false positives that SoftBound+CETS registers comes from variations in how the `alloca()` function call is handled. `alloca()` is compiler dependent [41]. The failing examples use `alloca` which is wrapped around a static function. SoftBound+CETS

[2]The tests that match the following regular expression: CWE(121|122)+((?!socket).)*(\.c)$

uses clang 3.4 as the underlying compiler, and CUP uses clang 4.0. Consequently, SoftBound+CETS handles the examples differently, and sees the memory from `alloca` as invalid, while CUP does not. However, there were four false positives that involve `memcpy`, which SoftBound+CETS claims to handle. An implementation bug is likely the cause of these failing tests, as the Juliet tests cover a wide spectrum of corner cases. While SoftBound+CETS theoretically has no false positives, this issue highlights the importance of implementation.

ASan and SoftBound+CETS higher false negative rate results from their incomplete *coverage*. In particular, many of the Juliet examples involve calling libc functions to copy strings and buffers (e.g., `strcpy` and `memcpy`). Neither ASan nor SoftBound+CETS are able to protect against unsafe libc calls. Our instrumentation of libc (Section 2.3.3) allows us to properly detect memory violations in these calls. Further, our adjustment for read / write size (Section 2.3.2) allows us to catch additional memory safety violations.

Given its support for separate compilation, and its theoretical soundness, SoftBound+CETS should be able to properly protect libc. However, as shown in Section 2.1.1, supporting all data across the entirety of user space significantly stresses the design of memory safety mechanisms. Our annotations in libc Section 2.3.3 allow us to protect all of user space. SoftBound+CETS has yet to prove that it can do so.

Listing 2.4 provides a representative Juliet test that CUP properly handles, and which ASan handles incorrectly[3]. The code allocates two 10 byte buffers on the stack, reads input from the user, and starts writing user input to one of the buffers. ASan protects the stack by surrounding stack objects with poison zones. However, depending on the value of of `pos`, an integer overflow bug allows an attacker to skip over the poison zone, and to write to an address higher in the stack. CUP enforces strict boundaries on a per-object basis, so the write at line 9 would be blocked.

[3]Adapted from https://github.com/ewimberley/AdvancedMemoryChallenges/blob/master/4.cpp

```
int main() {
    int pos = 0;
    char data = 0;
    scanf("%d", &pos);
    char buff2[10] = "abc";
    char buff[10];
    while(pos < 10 && data != '\n') {
        scanf("%c", &data);
        buff[pos] = data;
        printf("%d\n", pos);
        pos++;
    }
}
```

Listing 2.4: Example of code ASan fails to protect

2.5 Discussion

We discuss several design aspects of CUP, how we handle specific corner cases, potential for optimization, and further extensions.

Reducing instrumentation Prior work on reducing the amount of required runtime checks has focused on type systems. CCured [42] infers three types of pointers: safe, sequential, and wild. Safe pointers are statically proven to stay in bounds. Sequential pointers are only incremented (or decremented) — e.g., iterating over an array in a loop. All remaining pointers are classified as wild. Leveraging CCured-style type systems to further optimize memory safety solutions is left as an open problem.

Optimization through LTO CUP does not depend on Link Time Optimization (LTO). However, LTO makes it possible to inline functions across source files, and generally flattens code. Inlining would increase the effectiveness of our stack optimization and further reduce the amount of instrumented stack variables, reducing the number of IDs that a program consumes. Reducing the IDs a program uses reduces the overhead for ID management and resources used by CUP.

Arithmetic overflow Our hybrid metadata scheme stores the capability ID as part of the pointer. Pointer arithmetic can potentially modify the ID, allowing an adversary to chose the metadata the pointer is checked against. To prevent this attack vector, the upper and lower 32 bits of the pointer are loaded separately. The compiler enforces that any arithmetic operations only happen on the lower 32 bits, which contain the pointer's offset. This protects our capability ID from manipulation by an adversary.

Stronger temporal protection As discussed in Section 2.2.1, it is possible (albeit difficult) for an attacker to perform a temporal attack on software instrumented with CUP. If CUP were deployed as an active defense, the difficulty of a successful temporal attack could be increased by utilizing a randomized memory allocator like DieHard [28] or DieHarder [29]. Such allocators randomize heap allocations, making heap grooming [37, 38] much more difficult. Beyond getting the addresses to line up, the increased number of required allocations makes matching the capability IDs even harder. This renders a successful use-after-free attack highly unlikely, with minimal additional overhead. Additionally, a "lock and key" scheme [13] can be added to our metadata. Alternatively, our metadata can be extended to include a DangNull [31]-style approach that records how many references are still pointing to an object and either explicitly invalidating those references or waiting until the last reference has been overwritten before reusing IDs.

Third-Party Code CUP supports linking with uninstrumented libraries. Enriched pointers are the same size as regular pointers, maintaining the ABI. Dereferencing enriched pointers in uninstrumented code results, by design, in a segmentation fault. A segmentation fault handler can, on demand, dereference individual pointers. As the memory allocator is instrumented, memory allocations in uninstrumented code will return enriched pointers. Stack allocations on the other hand will not be protected in uninstrumented code. While this option allows compatibility, it clearly results in high performance overhead. An alternate solution is to manually annotate pointers passed to uninstrumented code (one annotation per pointer) and sacrifice protection, while gaining performance. This solution, however, exposes our metadata table to corruption from uninstrumented code. All memory safety solutions share this vulnerability to unprotected code. CUP alleviates this situation by recompiling all user-space code, including the libc.

Memory errors in unprotected code While it is possible to use CUP with unprotected code, we cannot guarantee the safety of applications that use unprotected code. Any memory violation in unprotected code can lead to unsafe modifications to user-space applications. Since CUP has no knowledge of the behavior of unprotected code, it is up to the programmer to ensure that all relevant buffers are properly checked before they are used. In our instrumentation of musl, we have already done this (see Section 2.3.3) for kernel system calls.

Assembly code CUP automatically instruments all code written in high level languages; our analysis pass runs on LLVM IR. Our analysis does not (and cannot) instrument inline assembly and assembly files due to missing type information. We rely on the programmer to either instrument the assembly code accordingly or to fall back on supporting uninstrumented code as mentioned above.

Multithreading CUP does not protect against inter-thread races of updates to metadata (e.g., one thread frees an object while the other thread is dereferencing a pointer). We leave the design of a low-overhead metadata locking scheme as future work. Note that this limitation is shared with other sanitizers.

Code Pointers A substantial research effort has gone into protecting code pointers [11]. CUP does not need to directly protect code pointers, including function pointers, C++ virtual table pointers, and C++ virtual tables. Function pointers cannot be used to read or write memory themselves. Further, by enforcing memory safety, CUP guarantees that they point to the correct targets.

2.6 Related Work

Precision is required to enforce spatial memory safety (bounds checks). There is a class of memory safety solutions that only approximately enforce this property [27, 30, 36]. These solutions make use of techniques such as poisoned zones — detecting spatial violations within limits, or rounding allocation size — which causes the executed program to differ from the programmers intent, and results in challenges when trying to handle intra-array and intra-struct checks. By changing the memory layout and not enforcing *exact* bounds, these solutions are not faithful to the programmer's intent. SoftBound [12] is the existing solution which best satisfies this requirement, while lacking comprehensive coverage (see below).

Object-based memory checking [43, 44, 45] keeps track of metadata on a per-object basis. Since the meta-data is associated on a per object level, every pointer to the object shares the same metadata. If a pointer is incremented, it may suddenly point to another valid object and therefore be (illegally) assigned to that object. Object layout in memory is generally left unchanged, which increases ABI compatibility. However, pointer casts and pointers to subfield struct members are unhandled [46]. SAFECode [47] is an example for efficient object-based memory checking.

Recently, Intel has started to add memory safety extensions, called Intel MPX, to their processors, starting with the Skylake architecture [48]. These extensions add additional registers to store bounds information at runtime. While effective at detecting spatial memory violations, MPX is incapable of detecting temporal violations [49], and typecasts to integers are not protected. In addition, current implementations of MPX incur a large memory penalty of up to 4 times normal usage [50]. Other ISA extensions include Watchdog [51], WatchdogLite [52], HardBound [53], and Chuang et al. [54]. As a software only solution, CUP does not require extra hardware or ISA extensions.

Object Awareness is required to prevent temporal memory safety violations (lifetime errors). This property requires remembering for every pointer whether the object to which it is assigned is still allocated. AddressSanitizer [27] and Low-Fat Pointers [30] make no attempt to do this (and their metadata does not support this property), while SoftBound+CETS [13] enforces this property. AddressSanitizer and Low-Fat Pointers do not maintain metadata either per object or per pointer. *Object Awareness* requires either per object or per pointer metadata. Consequently, they fundamentally *cannot* enforce temporal safety because their (current) metadata *cannot* be object aware. CUP's contributions on top of SoftBound+CETS lie primarily in Comprehensive Coverage.

Temporal only detectors include DangNull [31] and Undangle [55] from Microsoft. DangNull automatically nullifies all pointers to an object when it is freed. Undangle uses an early detection technique to identify unsafe pointers when they are created, instead of being used. CUP only provides a probabilistic temporal defense, however, DangNull and Undangle lack any spatial protection. Other probabilistic approaches [28, 29] change the memory allocator to reduce the frequency with which memory is reallocated.

Comprehensive Coverage is required to fully protect the program. As shown in Section 2.1.1 stack objects are the overwhelming majority of allocations, and to this day a significant portion of memory safety Common Vulnerabilities and Exposures (CVE) are stack related. Our evaluation of SoftBound+CETS (Section 2.4.2) shows that it has poor coverage — missing many stack vulnerabilities. AddressSanitizer and Low-Fat Pointers do better. AddressSanitizer protects the stack through the use of poisoned zones, and, as illustrated in Section 2.4.2 cannot handle all invalid stack memory accesses. Additionally, neither of them supports compiling libc — leaving the window open for vulnerabilities such as GHOST [15]. Tripwires [56, 57, 58, 59] are a way to detect some spatial and temporal memory errors [27, 60]. Tripwires place a region of invalid memory around objects to avoid small stride overflows and underflows. Temporal violations are caught by registering memory freed as invalid, until reclaimed. Tripwires, however, miss long stride memory errors, and thus cannot be said to be completely secure.

The state-of-the-art C/C++ pointer-based memory safety scheme is SoftBound+CETS [46]. Other pointer-based schemes include CCured [42] and Cyclone [61]. CCured uses a fat pointer to store metadata, as well as programmer annotations for indicating safe casts. Unfortunately, fat pointers break the ABI, and programmer annotations can significantly increase developer time. Even with annotations, CCured fails to handle structure changes. Cyclone also uses a fat pointer scheme, but does not guarantee full memory safety.

Control-Flow Hijacking Defenses. Mechanisms like control-flow integrity (CFI) [11, 25] or Object Type Integrity [26] check if a code pointer has been modified to point to an illegal address before it is used, e.g., for an indirect function call. CFI mechanisms assume that memory safety violations can happen and only checks the integrity of code pointers. CFI is a low-overhead approach to protect programs against illicit uses of a corrupted pointer (without protecting the integrity of the code pointer itself) while memory safety protects against the pointer corruption in the first place.

2.7 Conclusion

We present CUP, a C/C++ memory safety mechanism that provides full user-space protection, including libc, and strong probabilistic temporal protection. It is the first such mechanism that satisfies all requirements for a complete memory safety solution, while incurring only modest performance overhead compared with the state-of-the-art. CUP is exact, object aware, comprehensive in its coverage, and precise. We fully protect all user-space memory, including the stack, which, despite being the largest source of pointers, remained largely unprotected. Finally, we produce zero false negatives and zero false positives in the NIST Juliet Vulnerability example suite, which represents a significant advancement over existing memory safety mechanisms.

Our prototype is available at `https://github.com/HexHive/CUP`.

3 SHINING LIGHT ON SHADOW STACKS

Arbitrary code execution exploits give an attacker fine-grained control over a system. Such exploits leverage software bugs to corrupt code pointers to hijack the control-flow of an application. Code pointers can be divided into two categories: *backward edge*, i.e., return addresses or *forward edge* pointers, such as function pointers or virtual table pointers. Control-Flow Integrity (CFI) [25, 62] protects forward edges, and is being deployed by Google [63] to protect Chrome and Android, and Microsoft [64] to protect Windows 10 and Edge. CFI assumes that backward edges are protected. However, stack canaries [8] are the strongest deployed backward edge protection, and are easily bypassed.

Control-flow hijacking attacks that target backward edges, e.g., ROP, are a significant problem in practice, and will only increase in frequency. In the last year, Google's Project Zero has published exploits against Android libraries, trusted execution environments, and Windows device drivers [17, 18, 19, 20, 21]. These exploits use arbitrary write primitives to overwrite return addresses, leading to privilege execution in the form of arbitrary execution in user space or root privileges. The widespread adoption of CFI increases the difficulty for attacks on forward edge code pointers. Consequently, attackers will increasingly focus on the easier target, backward edges.

C / C++ applications are fundamentally vulnerable to ROP style attacks for two reasons: (i) the languages provide neither memory nor type safety, and (ii) the implementation of the call-return abstraction relies on storing values in writeable memory. In the absence of memory or type safety, an attacker may corrupt *any* memory location that is writeable. Consider, for the sake of exposition, x86_64 machine code where the call-return abstraction is implemented by pushing the address of the next instruction in the caller function, i.e., the return address, onto the stack; the callee function then pops this address off the stack and sets the instruction pointer to that value to perform a return. As C / C++ are memory unsafe, attackers may modify return addresses on the stack to arbitrary values and perform code-reuse attacks such as ROP.

Mitigating ROP attacks requires guaranteeing the integrity of the return address used to reset the instruction pointer after a function executes. There are four principle attempts to do this: (i) stack canaries, (ii) back edge CFI, (iii) safe stacks, and (iv) shadow stacks. Stack Canaries [8] protect against sequential overwrites of a return address through, e.g., buffer overflows by inserting a magic value onto the stack after the return address, which is then checked before returns. However, canaries are not effective against arbitrary writes where, e.g., an attacker controls a pointer and can precisely overwrite memory. CFI computes a valid set of targets for indirect control-flow transfers, for returns this means any potential call site of the function. As shown by Control-Flow Bending [7], this is too imprecise to prevent control-flow hijacking attacks in the general case. Safe Stacks [65], move potentially unsafe stack variables to a separate stack, thereby protecting return addresses. However, Safe Stacks offer limited compatibility with unprotected code, so are unlikely to be deployed.

Shadow stacks [66, 67, 68] enforce stack integrity, protecting against stack pivot attacks and overwriting return addresses. Shadow stacks store the return address in a separate, isolated region of memory that is not accessible by the attacker. Upon returning, the integrity of the program return address is checked against the protected copy on the shadow stack. By protecting return addresses, shadow stacks enforce a one to one mapping between calls and returns, thereby preventing ROP. Two shadow stack designs have been proposed: compact shadow stacks [67], which rely on a separate shadow stack pointer, and parallel shadow stacks [66], which place the shadow stack at a constant offset to the original stack. These existing shadow stack designs suffer from a combination of poor performance, high memory overhead, and difficulty supporting C and C++ programming paradigms such as multi-threading and exception handling.

To improve the state of shadow stack design, we conduct a detailed survey of the design space. Our design study includes two novel designs, and considers five shadow stack mechanisms in total. We fully explore the trade-offs of these designs in terms of performance, compatibility, and security. We consider the impact of high level design decisions on runtime, memory overhead, and support for threading, stack unwinding, and unprotected code. Further, we propose novel optimizations for shadow stack implementations.

Beyond the design of the shadow stacks, we analyze the options for guaranteeing their integrity, including existing software solutions and two new ISA extensions. Unlike CFI, which relies on immutable metadata stored on read-only pages, shadow stacks, and other security mechanisms, require mutable metadata that must be integrity protected. Integrity protection is accomplished by isolating an area of the address space within a process, preventing attackers from modifying it. We discuss the limitations of existing hardware mechanisms for intra-process isolation, and propose a new primitive better suited for use by software security mechanisms.

Based on our design study, we propose Shadesmar, a new compact shadow stack mechanism that leverages a dedicated register for the shadow stack pointer and our optimizations for comparing the program and shadow return addresses. Protecting Phoronix and Apache highlights Shadesmar's deployability. We hope that our thorough evaluation on both common benchmarks and real world software will lead to the adoption and deployment of shadow stacks in practice, closing a significant loop-hole in modern software's protection against code-reuse attacks. Shadesmar will be open sourced upon acceptance to aid deployment of shadow stacks.

We present the following contributions: (i) Comprehensive evaluation of the shadow stack design space along the axes of performance, compatibility, and security; (ii) Performance evaluation of each shadow stack design, including sources of overhead, and our optimizations for x86; (iii) Comparative study of new ISA features that can be used to create integrity protected memory regions for any runtime mitigation, and a proposal for an intra-process isolation mechanism; (iv) Shadesmar a register-based compact performant, secure, and deployable shadow stack scheme and its evaluation.

3.1 Background

To enable security analysis of shadow stacks, we first establish our attacker model. Using this attacker model, we then discuss common attacks on the stack, e.g., ROP, which overwrite return addresses for interested readers. Knowledgeable readers may wish to move directly to our discussion of the shadow stack design space in Section 5.3.

3.1.1 Attacker Model

As is standard for defenses that aim to mitigate exploits, e.g., CFI and Shadow Stacks, rather than the underlying corruptions, e.g., memory or type safety, we assume an attacker with arbitrary memory read and write primitives. The attacker uses these arbitrary reads and writes to inject her payload, and then corrupts a code pointer to hijack the program's execution, executing her payload and exploiting the application. The adversary is constrained only by standard defenses: DEP [9] and ASLR [69]. We disable stack canaries [8] as they are strictly weaker than Shadow Stacks.

For the final step of the attack, corrupting a code pointer, we assume that the attacker only targets *backward edges*, i.e., return addresses of functions. Protection for *forward edges* is an orthogonal problem, covered by defenses such as CFI [25, 62]. Attacking forward edge control flow is therefor out of scope for this paper. Also out of scope are data-only attacks, i.e., attacks that do not corrupt code pointers.

3.1.2 Attacks on the Stack

Attacks against stack integrity began with Aleph One's seminal work on stack smashing [1]. To this day, control-flow information on the stack remains an active battle ground in software security [4]. Code reuse attacks such as ROP and Stack Pivots are the latest iteration of this threat.

ROP [2] is a style of code-reuse attack that hijacks application control flow by overwriting return addresses on the stack. When the function returns, control is redirected to the attacker

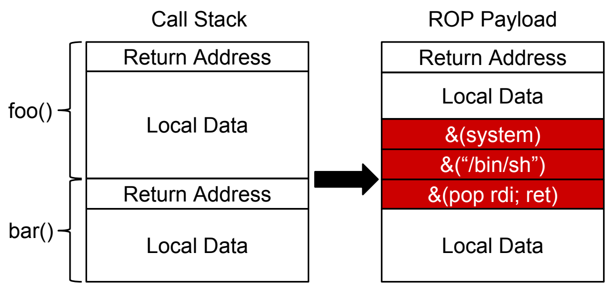


Figure 3.1.: ROP illustration

chosen address. Absent any hardening, return addresses on the stack can be modified to target any executable byte in the program. If a non-executable byte is targeted, attempting to execute that byte will lead to a fault, terminating the program. In practice, attackers target so called "gadgets", which are sequences of executable bytes ending in a return instruction that perform some useful computation for the attacker. The attacker's payload consists of a sequence of addresses of such gadgets that combined perform the desired computation, e.g., open a shell, or, in most real-world attacks map a memory page as executable and writable and `memcpy` target shellcode to that page before executing the injected shellcode.

Figure 3.1 illustrates a payload that executes `system()` to spawn a shell. When function `bar()` returns, the first gadget is executed. Returning to the first gadget moves the stack pointer to `&("/bin/sh")`, which is then `pop`d into `rdi`, and moving the stack pointer to `&(system)`. Consequently, the return in the first gadget calls `system("/bin/sh")`, opening a shell.

Stack Pivots are an emerging attack technique wherein the adversary controls the stack pointer, i.e., `rsp` on x86 architectures. Consequently, instead of having to selectively overwrite data on the stack, the attacker can move the stack frame to a region of memory she entirely controls, thereby making, e.g., ROP attacks significantly easier. This technique has also been used to bypass ASLR [70, 71]. While stack pivoting changes how the payload is delivered, code-reuse attacks utilizing it must still overwrite a code pointer. Consequently, for the purposes of shadow stacks and back edge defenses in general, stack pivoting is just a payload delivery variant of ROP.

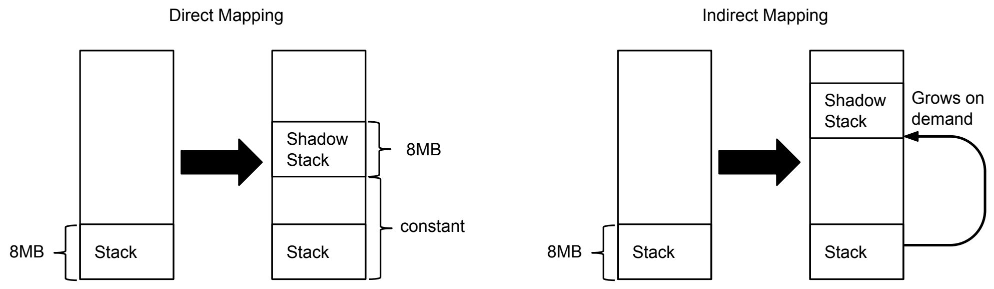


Figure 3.2.: Shadow stack designs – mapping options

3.2 Shadow Stack Design Space

For any shadow stack mechanism to be adopted in practice, it must be highly performant, compatible with existing code, and provide meaningful security. We analyze the performance of each shadow stack mechanism that we identify in terms of runtime, memory, and code size overhead qualitatively in this section, and quantitatively in our evaluation. Compatibility for shadow stacks means supporting C and C++ paradigms such as multi-threading and stack unwinding, as well as interfacing well with unprotected code. Security is dictated both by how a shadow stack mechanism validates the return address, and by any orthogonal technique the mechanism uses to guarantee the integrity of the shadow stack. See Section 3.4 for details on such integrity mechanisms.

Shadow stack mechanisms are defined by how they map from the program stack to the shadow stack, illustrated in Figure 3.2. This includes the type of mapping, as well as how the mapping is encoded in the protected binary. We analyze five such mechanisms using the two types of shadow stack identified by the literature: compact [67] and parallel [66]. For compact shadow stacks we identify three ways to encode the mapping in the binary, and two such ways for parallel shadow stacks. Each of these mechanisms has unique performance and compatibility characteristics. All shadow stack mechanisms must adopt a policy on validating the return address. Traditionally, this has been to compare the shadow and program return addresses and only proceed if they match. We examine the security

Table 3.1.: Summary of performance overhead, memory overhead, and compatibility trade-offs between shadow stack mechanisms. ✓– supported; ✗– not supported; ✧ – implementation dependent

Mapping	Encoding	Performance	Memory	Compatibility		
				Threading	Stack Unwinding	Unprotected Code
Compact	Global Variable	Slow	Low	✗	✓	✓
	Segment	Medium	Low	✓	✓	✓
	Register	Fast	Low	✓	✓	✧
Parallel	Constant Offset	Fast	High	✗	✓	✓
	Register Offset	Medium	High	✧	✓	✓

impact of utilizing the shadow return address without a comparison and find it increases performance without impacting security.

3.2.1 Shadow Stack Mechanisms

Direct mappings schemes for parallel shadow stacks use the location of the return address on the program stack to directly find the corresponding entry on the shadow stack. The parallel shadow stack is as large as the program stack, and a simple offset maps from the program stack to the shadow stack. Consequently, the direct mapping trades memory overhead – twice the stack memory usage, for performance – a very simple shadow stack look up.

Indirect mapping schemes for compact shadow stacks maintain a shadow stack pointer, equivalent to the stack pointer used for the program stack. The shadow stack pointer points to the last entry on the shadow stack, exactly as the stack pointer does for the program stack. Maintaining a shadow stack pointer allows a compact shadow stack to allocate significantly less memory, as only room for the return address is required, instead of duplicating the program stack. Therefor, indirect mappings trade performance overhead – from using the shadow stack pointer, for reduced memory overhead – by only requiring a compact shadow stack.

In addition to the performance versus memory overhead trade-off, parallel and compact shadow stacks have different compatibility implications. If calls and returns were always perfectly matched, there would be no difference. However, the `setjmp` / `longjmp` functionality of C, which allows jumping multiple stack frames back up the stack, and the equivalent stack unwinding capability used by C++ for exception handling, both break the assumption of perfectly matched calls and returns. The direct shadow stack paradigm naturally handles these, as C / C++ adjust the stack accordingly, and then it uses the adjusted stack to find the appropriate shadow stack entry. The indirect shadow stack scheme on the other hand must know how many stack frames the program stack has been unwound to appropriately adjust its shadow stack pointer. Consequently, stack unwinding leads to

```
mov rax, [rsp]
mov [rsp+CONSTANT], rax
```

(a) Constant offset

```
mov rax, [rsp]
mov [rsp+r15], rax
```

(b) Offset in register

Figure 3.3.: Direct mapping shadow stack prologues. The epilogues execute the inverse.

additional overhead for indirect shadow stack mapping schemes, while having no affect on direct mapping schemes.

For each shadow stack mapping scheme, there are multiple possible mechanisms with different implications for performance and compatibility. In particular, we introduce the use of a register for the shadow stack pointer for compact shadow stacks, or the offset for parallel shadow stacks. Now that all 64 bit architectures have at least 16 general purpose registers, it is possible to dedicate a general purpose register to the shadow stack mechanism, unlike in 2001 when the original shadow stack proposal was made [67] and only eight general purpose registers were available on x86. We find that using a dedicated register allows compact shadow stack mappings to be as performant as parallel shadow stacks, and allows parallel shadow stacks to increase their compatibility with multi-threading while also being more secure.

A summary of our shadow stack mechanisms and their trade-offs for each design is shown in Table 3.1. Each row in the table represents a shadow stack mechanism that we evaluate. The table reports qualitative differences between them, we refer to the evaluation in Section 3.6.1 for quantitative measurements.

Parallel Shadow Stack Mechanisms

Parallel shadow stack mechanisms effectively use the stack pointer as the shadow stack pointer. The existing mechanism [66] places shadow stack entries at a constant offset from the program stack. This is very efficient, requiring no extra registers or memory access, and no instrumentation to maintain the shadow stack pointer. This performance benefit is offset by higher memory overhead, compatibility problems, and lower security. All parallel

```
mov     r10,  rcx
mov     r11,  rdx
mov     rax,  [rsp]
mov     rdx,  GLOBAL
mov     rcx,  [rdx]
mov     [rcx], rax
mov     [rcx], rsp
add     [rdx], 16
mov     rcx,  r10
mov     rdx,  r11
```

(a) Global variable

```
mov  rax,      [rsp]
mov  r10,      gs:[0]
mov  [r10],    rax
mov  [r10+8],  rsp
add  r10,      16
mov  gs:[0],   r10
```

(b) Segment

```
mov  rax,  [rsp]
mov  [r15],  rax
mov  [r15+8],  rsp
lea  r15,  [r15+16]
```

(c) Register

Figure 3.4.: Indirect mapping shadow stack prologues. Note - Epilogues are the inverse.

shadow stacks suffer from higher memory overhead, as they fundamentally require the program stack to be duplicated. The compatibility concerns arise from requiring a constant offset, which is limited to 32 bits for immediate operands in x86, from the program to the shadow stack from all threads, severely constraining the address space layout for programs with many threads, such as browsers. Hard-coding the offset in the binary is also a security hazard, as recovering the offset leaks the address of the shadow stack to adversaries.

To mitigate the compatibility and security concerns, we propose a new parallel shadow stack mechanism. Our parallel shadow stack mechanism encodes the offset in a dedicated register, see Figure 3.3, allowing the offset to the shadow stack to be determined at runtime. Further, the offset may vary from thread to thread as registers are thread local, and the offset can be set when the thread is created. This register is only updated once, when the offset is determined for the thread, and therefor adds no per function call overhead (unlike shadow stack pointers for compact shadow stacks).

Compact Shadow Stack Mechanisms

For compact shadow stack mechanisms, the key question is where to store the shadow stack pointer. This decision will not impact the memory overhead of the implementation, but does have performance and compatibility ramifications. The shadow stack pointer will be dereferenced twice in every function: once in the prologue to push the correct return address, and once in the epilogue to pop the shadow return address. Consequently, the speed of accessing the shadow stack pointer is critical for the performance of shadow stacks that are indirectly mapped. There are three locations to store a variable: in memory, in a segment, or in a register. We discuss and evaluate the performance and compatibility trade-offs of all three, and x86 code for each is shown in Figure 3.4.

Using a memory location, e.g., a global variable is the simplest solution, and we present it as a straw man. Accessing memory is orders of magnitude slower than accessing a value stored in a register. Even with caching, this effect is noticeable, see Figure 3.7. This slow down is aggravated by the need for an additional move instruction to load the location of the global variable into a register to access it – x86 does not support 64 bit immediate values. Further, changing memory access patterns can affect cache behavior, with unpredictable effects on the program's performance. An additional problem for this scheme is that the memory must be thread local to support multi-threaded programs. Consequently, a scheme that has better performance characteristics and is inherently thread local is desirable.

Segment registers, used by existing shadow stack mechanisms [67] to store the location of the shadow stack base, are an architectural feature left over from when physical memory was larger than the virtual address space. Segment registers are faster to access than memory, and are inherently thread local. Consequently, they improve performance significantly over using a memory location to store the shadow stack pointer, while also improving compatibility by supporting multi-threading. We point the segment register at the base of the shadow stack, and store the shadow stack pointer there. Accessing the shadow stack is thus double indirect, through the segment register and then the shadow stack pointer.

We propose a new compact shadow stack mechanism that uses a general purpose register to store the shadow stack pointer. General purpose registers provide the fastest possible option for storing the shadow stack pointer. The disadvantage of using a general purpose register is that it must be reserved for the shadow stack pointer, reducing by one the number of registers available for the compiler to use, and thereby increasing register pressure. Increased register pressure can reduce performance if it leads to additional register spills to the stack. Despite this potential overhead, our evaluation finds that this is the fastest shadow stack encoding, see Figure 3.7.

```
pop r10
;assuming r11 holds shadow RA
xor r11, r10
popcnt r11, r11
shl r11, 48
or r11, r10
;will fault if r11 != 0
jmp r11
```

(a) `Fault` epilogue

```
pop r10
;r11 holds shadow RA
xor r11, r10
popcnt r11, r11
;will fault if r11 != 0
mov r11b, [Last_Byte_of_Page+r11
    ]
jmp r10
```

(b) `LBP` Epilogue

Figure 3.5.: Shadow stack epilogue optimizations

3.2.2 Return Address Validation

Shadow stack mechanisms can ensure a valid return address in two ways: by either comparing the program and shadow return addresses, or by using the shadow return address. Comparing the shadow and program return addresses detects corruptions of the program return address immediately, and can halt execution. Immediate detection is useful during testing and debugging as it helps isolate the bug. In deployment, however, preventing control-flow hijacking attacks only requires that the corrupt program return address not be used. Checking the program return address is equivalent to a low entropy stack canary, possibly detecting sequential buffer overflows. Consequently, the shadow stack mechanism can simply use the return address on the shadow stack. Doing so fully mitigates control-flow hijacking attacks as the attacker controlled return address is not used and avoids the overhead of comparing the return addresses. Either policy provides the same security: an attacker cannot control the target address of a function return.

3.3 Shadow Stack Implementations

Each of the shadow stack mechanisms we evaluate is implemented as a backend compiler pass in LLVM 7.0.0, and shares some common implementation details. In particular, each shadow stack mechanism must instrument calls and returns to update its shadow stack and validate the return address before using it to transfer control. We show that the best way to accomplish this is to instrument function prologues and epilogues. Our implementations further include a small runtime library to set up the shadow stacks, and support stack unwinding for compact shadow stack schemes. Additionally, we introduce novel peep hole optimizations for x86 epilogues.

3.3.1 Instrumented Locations

Shadow stack mechanisms can instrument function calls either at the location of the call instruction or in the function prologue on the callee side. This instrumentation is responsible for pushing the return address to the shadow stack, and updating the shadow stack pointer for compact shadow stacks. Returns must be instrumented to pop from the shadow stack and validate the program return address in the function epilogue before the control-flow transfer to mitigate control-flow hijacking attacks. Code that can unwind stack frames, such as `longjmp` and C++'s exception handling mechanism, which uses `libunwind`, must also be instrumented to maintain the shadow stack pointer for compact shadow stacks. Failing to handle stack unwinding correctly can lead to false positives as the shadow and program stack are out of sync.

The elegant solution for instrumenting calls is to place the protection in the function prologue. In this way, the *function* is protected, not particular call sites. The compiler does not have to distinguish between calls to protected and unprotected functions as it would if call sites were instrumented instead. The distinction must be made if call sites are instrumented to keep the shadow stack in sync for compact shadow stack where calls and returns must be perfectly matched. Instrumenting function prologues and epilogues maintains this symmetry naturally, as each will be executed for every function call. The only down side is that the return address is passed into the function on the stack. This allows a window of a single instruction where the attacker can modify the return address before we read it and store it on the shadow stack, effectively resulting in a time of check to time of use (TOCTTOU) opportunity. Given the extremely precise timing required, we do not believe this potential vulnerability to be exploitable. Further, resolving the TOCTTOU window requires instrumenting call sites, and thus distinguishing between protected and unprotected code, and so introduces a dependency on whole program analysis. We leave implementing shadow stacks as a whole program analysis as future work.

Our prologue and epilogue rely on the stack pointer to find the return address, and are therefor agnostic to optimizations that delete the stack frame base pointer. Once our epilogue has popped the return address, we do not read it again from memory, thereby preventing TOCTTOU attacks that modify the return address in memory between the time it is read for the shadow stack check and the time it is used by the return instruction. One consequence of this is that `ret` instructions become `pop` and `jmp` instructions. This single transformation accounts for approximately half of the shadow stack overhead, see Figure 3.8. Hardware solutions that avoid this overhead are discussed in Section 5.5.

Stack unwinding mechanisms such as `longjmp` and C++ exceptions require additional instrumentation for compact shadow stacks. parallel shadow stacks are unaffected as they do not require adjustment to track stack frames, i.e., they do not maintain a shadow stack pointer. For compact shadow stacks, we must be able to unwind to the correct point on the shadow stack as well. Simply matching return addresses does not suffice for this, as the same return address can show up multiple times in the call stack due to, e.g., recursive calls. To deal with this, our compact shadow stack implementations also push the stack pointer, i.e., `rsp`. The stack pointer and return address uniquely identify the stack frame to unwind to, allowing our mechanisms to support stack unwinding.

For the shadow stack mechanisms that use a register to encode the shadow stack mapping, ensuring compatibility with unprotected code constrains our selection of register. A callee saved register must be used, so that any unprotected code that is called will restore the shadow stack pointer, but only if it is clobbered, which helps performance. Our implementations use `r15` in practice.

3.3.2 Runtime Support

Our runtime library is responsible for allocating the shadow stack, and hooking `setjmp` and `longjmp`. We add a new function in the `pre_init` array that initializes the shadow stack for the main program thread. This function also initializes the shadow stack pointer for compact shadow stack mappings. In particular, for segment encodings it invokes the system call to assign the shadow stack to the segment register. `Setjmp` and `longjmp` are redirected to versions that are aware of our shadow stacks. These patched versions required less than 20 lines of assembly to modify.

For compact shadow stack mappings to support multi-threading and libunwind, we preload a small support library. It intercepts calls to `pthread_create` and `pthread_exit` to set up and tear down shadow stacks for additional threads. We use a patched version of libunwind, to which we added 20 lines of code for compatibility with our shadow stacks. These changes are minimal, and easily deployable by having, e.g., two version of the library on the system and a compiler flag to chose which one is linked in. If shadow stacks are universally used to harden libraries, no such additional support would be required. Consequently, we believe compact shadow stacks are readily deployable.

3.3.3 Shadow Stack Epilogue Optimizations

Traditionally, shadow stacks have relied on compare instructions to validate the shadow return address and program return address are equivalent. However, the compare and jump paradigm is relatively expensive, potentially leading to pipeline stalls even with branch prediction. Consequently, as an optimization, we explore two different methods to optimize this validation. Our optimizations rely on the insight that a full comparison is not required, only an equality test.

To replace the compare instruction of traditional shadow stack epilogues, we propose an `xor` of the program return address and shadow return address. This will result in `0` bits anywhere the two are identical, and `1`s elsewhere. x86 has an instruction, `popcnt`, that returns the number of bits set to `1`. Consequently, if the `popcnt` of the `xor` of the program return address and shadow return address is `0`, then the two are equivalent.

We leverage the MMU to compare the `popcnt` to zero as a side effect by creating a protection fault. We propose two different ways to do so: `fault` and last byte in page

```
mov eax, 0     ; Read Write
xor ecx, ecx
xor edx, edx
wrpkru

;protection is turned off
;write to shadow stack

mov eax, 8     ; Read Only
xor ecx, ecx
xor edx, edx
wrpkru
```

Figure 3.6.: MPK page permission toggling

(LBP), see the code in Figure 3.5. For fault, we note that the maximum value of the popcnt is 64, therefor fitting in six bits. By shifting this value left 48 and oring it into the return address, we create a general purpose fault for a non-canonical address form if its value is not zero, by setting one of the high order 16 bits to one in user space. This scheme abuses the fact that the high order 16 bits are currently unused, and may break if those bits are utilized in future processors. Alternately, the LBP scheme creates two pages in memory, the first of which is mapped read write, the second of which has no permissions. We then attempt to read from the first page at the address of the last valid byte, plus the popcnt value. If the popcnt value is zero, we read the last byte of the valid page, otherwise we read from the guard page, causing the MPU to return a fault. The trade-off between the two is that the fault scheme requires serialization in the processor, while the LBP scheme requires a memory access and the Memory Protection Unit (MPU). We show the performance of both schemes in Figure 3.9.

3.4 Hardware Integrity Mechanisms

Once a shadow stack design has been chosen, the shadow stack mechanism must guarantee the integrity of the shadow stack. How to guarantee the integrity of a protected region of memory is a problem faced not only by shadow stacks, but also by all mitigations that rely on writable runtime metadata. Integrity guarantees are best provided by hardware solutions, though software solutions exist and are covered here. Hardware solutions for integrity protecting part of the address space within a process should be evaluated on two metrics: their performance, and the number of supported concurrent code regions.

Existing hardware mechanisms take two different approaches to encoding access privileges to provide integrity protection: (i) in each thread's register file, providing per thread (thread centric) integrity, and (ii) in the individual instructions, so that access privileges are the same across all threads and depend only on the executed instruction (code centric). Note that thread centric solutions require additional instructions to change the register file, consequently, code centric solutions are (potentially) more performant as they operate in a single step, checking an instruction's permissions, instead of first toggling bits in the register file and then checking permissions. For code centric mechanisms, the ability to execute the instruction grants the necessary permissions while for thread centric mechanisms, the state of the register file determines the policy.

Assuming code integrity and a control-flow hijacking defense such as CFI, we prefer code centric solutions for their potential performance and flexibility. Unfortunately, no existing code centric solution is fully satisfactory in that they have excessive code size increases, lack performance, and are not as flexible as required, i.e., split memory into only two regions. Consequently, we call for a new ISA extension that is hardware-based for performance, supports multiple secure regions to be general purpose (e.g., to support multiple concurrent security monitors, each with its own protected region), and requires minimal code changes. We show how our proposed mechanism is a code centric adaptation of the state of the art thread centric mechanism, and thus is fully practical.

3.4.1 Thread Centric Solutions

Thread centric solutions operate by changing the permissions on the pages of the protected memory region. Adding write permissions elevates the thread's privileges, thereby creating a privileged region that is able to modify the protected memory region, i.e., the shadow stack. Removing the write permissions ends the privileged region. The traditional mechanism for doing this is the `mprotect` system call. Using `mprotect` is prohibitively expensive as it not only requires a context switch into the kernel, but a full page table walk to change the permissions on the indicated pages. In addition, `mprotect` enables write capabilities for all concurrent threads and not just for the thread writing the privileged data.

For hardware enforced privilege based mechanism, segmentation registers used to provide privilege based isolation for x86, where the segmentation register served to give an instruction access privileges to the protected region. For 64 bit architectures however, x86 no longer provides a hardware-enforced isolation mechanism with segmentation registers.

A new Intel ISA extension, Memory Protection Keys (MPK) aims to address this by providing a single, unprivileged instruction that can change the permissions of a group of pages on a per-thread basis. MPK works by assigning every page to one of sixteen keys. The new `wrpkru` instruction can then change the permissions for all the pages associated with a given key. A thread is associated a given key with which it can access all pages protected with that key. This approach elegantly solves the TOCTTOU problem of `mprotect` and allows per-thread protected regions.

The assembly to enforce privileged code regions using MPK is shown in Figure 3.6. Note that the `wrpkru` instructions requires `edx` and `ecx` to be set to 0. Intel did not disclose why the two registers are required to be 0, it may be for future extension of the `wrpkru` instruction to allow a full API to be developed. The System V calling convention, used by Linux, uses these registers to pass the third and fourth arguments to a function respectively. Consequently, for functions which take more than two arguments, it is necessary to preserve the original values of these registers, which is accomplished by moving their values to caller save registers, and then restoring them after the `wrpkru` instruction. Surprisingly, this scheme is slower than MPX which must instrument almost every memory write in the program, see Figure 3.10 for full results.

3.4.2 Code Centric Solutions

The most common code centric solution is information hiding, where a pointer to the protected region gives any instruction access privileges. Information hiding is attractive because it adds no additional overhead; however, it is the weakest option as the many attacks against ASLR and other information hiding schemes attest [72, 73, 74, 75, 76]. In essence, a memory leak, side channel, or simply an implementation bug may allow attackers to bypass randomization defenses. Consequently, we consider information hiding to provide minimal security for the shadow stack, and recommend against it.

Software Fault Isolation (SFI) [77, 78, 79] is a secure software solution for isolating intra-process address regions. Even the best SFI implementations [77] from industry still have 7% overhead just for the isolation, significantly more than is acceptable in total for a deployed security monitor. Additionally, the x86 ISA supports an address override prefix that limits addressable memory to 32 bits. This can be used to crudely separate the program's address space in a 4GB region for the process to access, leaving all other memory for the security monitor. 4GB of memory is insufficient for many modern applications however. Consequently, a more flexible hardware mechanism is required.

The Intel ISA extension Memory Protection Extension (MPX) provides a hardware mechanism that can be used to implement segmentation [26] in a flexible manner, though it can only split memory into two regions. MPX provides a bounds checking mechanism, with four new 128 bit registers to store the bounds, and two new primitives to perform the upper and lower bounds checks. MPX segmentation schemes divide writes into two categories, those that are privileged to write into the protected region, and all others. All non-privileged writes in the code are instrumented with a bounds check to ensure that they do not touch the privileged region. This approach is surprisingly performant, see Figure 3.10.

3.4.3 Privileged Move

Intel's MPK comes closest of all existing hardware mechanisms to meeting our requirements – it is a hardware based mechanism so should be performant, and supports 16 code regions within a process. However, while faster than rewriting page tables, MPK is still too expensive to execute for every function call, see Figure 3.10. Further, security monitors do not require a thread centric protection scheme. Rather, a code centric scheme with a single privileged move instruction would suffice. This instruction could take a one byte immediate specifying the region of memory it is allowed to write to. Unprivileged moves would be limited by default to the unprotected code region, allowing minimal changes. Privileged moves which encode their access permissions should be faster than toggling a thread control register as MPK does. Further, its implementation should be largely similar, relying on the same four bits in the page table that MPK does, and with the same checks. The difference being that instead of referencing a thread local state for permissions, the permissions would be encoded in the instruction proper.

Such a privileged move instruction would make an entire class of security policies that rely runtime metadata practical. Currently, protecting metadata at runtime is the bottleneck for many of these policies, covering areas as diverse as type safety [80], use after free protection [31], and partial memory safety for function pointers [65]. This hardware primitive would allow for the creation of flexible security policies in software that can change and adapt, such as shadow stacks. With the availability of such a primitive, the policies would be secure in practice, and make them deployable in adversarial environments, instead of only being useful for testing as they cannot withstand direct attacks.

Protection schemes that rely on new ISA extensions are unlikely to be immediately adopted by the wider community. However, analyzing them can show which hardware schemes are useful, hopefully paving the way for eventual broad deployment as happened with the DEP and the NX bit.

3.5 Discussion

Orthogonal to the main design, optimization, and protection points above there are interesting details around dealing with unprotected code, existing compiler optimizations with ramifications for shadow stacks, and forthcoming hardware extensions that we include here for completeness.

Unprotected Code. Unprotected code weakens the guarantees of shadow stack schemes, as they cannot prevent a control-flow hijacking attack in the unprotected region. Both parallel and compact shadow stack can be fully compatible with unprotected code regions. Parallel shadow stacks are completely oblivious to unprotected code as they do not require a shadow stack pointer. Compact shadow stack schemes fully support unprotected code as long as the shadow stack pointer is not clobbered. In particular, the register implementation of the compact shadow stack scheme is exposed to this. The register implementation can handle calls into unprotected code that return directly to protected code, as the register used is callee saved and thus restored before protected code runs again. However, if the unprotected region calls into protected code due to, e.g., a call back function to a sorting routine, the shadow stack pointer may have been clobbered causing the call back function to fail. Note that we anticipate that all code on a system is protected in practice.

Tail Call Optimizations. Tail calls allow call return pairs to be omitted by the compiler, when, for example, a function returns the value of another function call, or for recursive calls. In these cases, the same program return address can be used for the tail called function. However, new stack frames are required for the case where the call being optimized is the last instruction in an arbitrary function. The optimization simply saves instructions by omitting a call return pair by jumping directly to the callee, which can then use one return to exit itself and the caller. As a function can be both tail called and called normally, the full function prologue is executed even when the function has been tail called. To keep the shadow stack in sync, we execute the normal shadow stack epilogue before tail calls, though we omit the jump through the return address in these cases. Consequently, `fault` epilogues fall back to `LBP` for tail calls, as there is no `jmp` to modify.

Mobile Architectures. Beyond x86, ARM is in wide use for mobile and embedded devices. ARM uses the `link` register to store the return address for the current function, only pushing the return address to the stack when additional functions are called. Consequently, shadow stacks can instrument function prologues without a potential TOCTTOU window. Our analysis of the design space applies to other architectures while our epilogue optimizations are x86 specific because of the `popcnt` instruction. Of course, this instruction can be replaced with shift and or instructions. We leave the evaluation of an ARM implementation as future work.

Intel Control Enforcement Technology. Intel has released a preview document for a proposed new ISA extension called Control Enforcement Technology (CET) [81]. CET provides hardware support for shadow stacks, and checks on forward edge indirect control-flow transfers. CET modifies call instructions to push the return address to a hardware protected shadow stack as well as the program stack, and return instructions to compare the program and shadow return addresses, raising a fault if they are not equal. While this technology has great promise, no release date has been made public so it is unclear when / if it will become available. In the meantime, software solutions for hardening programs are required. Orthogonally, other architectures and legacy systems equally require protection.

3.6 Evaluation

We evaluate the five different shadow stack implementations from Table 3.1, and we examine the impact of our proposed epilogue optimizations. Orthogonally, we evaluate the cost of providing deterministic integrity protection for the shadow stack. Based on this evaluation, we recommend a shadow stack mechanism, Shadesmar, for broad use. To show Shadesmar's practicality, we present two case studies: Phoronix and the Apache web server. The Phoronix benchmarks are common use cases for widely used, real-world applications, and Apache is the most popular web server. Consequently, these case studies show Shadesmar is ready for deployment. All of our evaluation is done on an Intel(R) Xeon(R) Bronze 3106 CPU at 1.7GHz, with 48GB memory, running Debian-9.3.0. We compile software at O2 and for SPEC CPU2006 we use the default configuration with three reportable runs on the ref dataset.

3.6.1 Shadow Stack Evaluation

For each of the five different shadow stack designs, we first evaluate their performance on SPEC CPU2006. For the existing shadow stack designs identified in Section 5.3, we ported the implementations to LLVM 7.0.0 to control for performance effects from compiler improvements. For these experiments, we used the traditional `cmp`-based epilogue, and

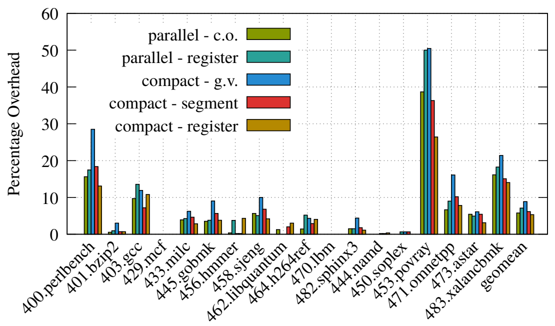


Figure 3.7.: Design comparison

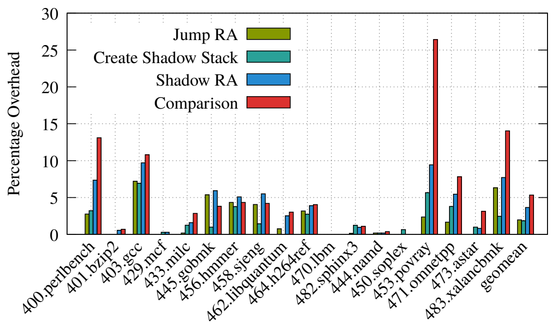


Figure 3.8.: Overhead breakdown for a compact register configuration

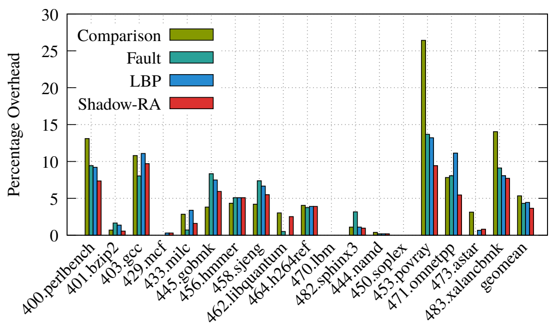


Figure 3.9.: Epilogue micro-optimizations

information hiding to protect the shadow stack. The results are in Figure 3.7. Note that the parallel shadow stack constant offset implementation and the compact shadow stack register implementation are within measurement noise of each other at 5.78% overhead and 5.33% respectively. This removes the performance justification for parallel shadow stacks greater

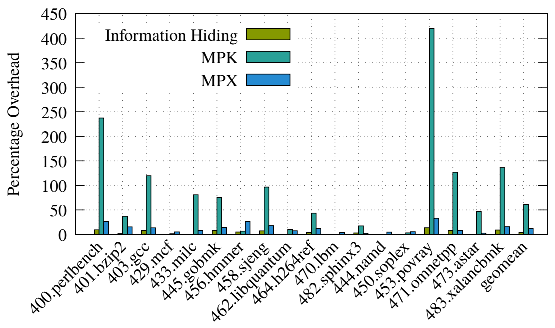


Figure 3.10.: Integrity protection overhead

memory use, if a dedicated register is used for the shadow stack pointer. The compact and parallel shadow stacks have effectively the same code size impact as well, 15.57% and 14.88% respectively. Consequently, we recommend compact shadow stacks.

Figure 3.9 shows the overheads for the compact shadow stack register implementation with our different epilogue optimizations. The traditional `cmp`-based epilogue has 5.33% overhead, 25% more than our optimized epilogues at 4.31% for the `fault` epilogue, and 4.44% for the `LBP` epilogue. Further, the `cmp` epilogue has significant outliers on perlbench, povray, and Xalancbmk. Consequently, we believe our epilogue optimizations are highly effective as they not only reduce overhead but also reduce its variation. As the `fault`-based epilogue is faster (albeit marginally) and does not require additional changes to the address space, we recommend it for vulnerability discovery settings, e.g., software testing and fuzzing. Using the shadow return address without any comparison as discussed in Section 3.2.2 results in 3.65% overhead, and is our recommendation for deployment.

We break down the sources of overhead within the compact shadow stack register implementation in Figure 3.8. Changing the `ret` instruction to a `pop; jmp` sequence has 1.97% overhead (the overhead is likely due to the loss of the CPU's return value prediction). Maintaining the shadow stack but leaving the normal return instruction has 1.85% overhead. If the epilogue jumps through the shadow stack return address, there is 3.65% overhead, effectively the sum of the return instruction transformation and maintaining the shadow stack, as expected. Our experiment highlights an opportunity for architectural improvement:

moving the return stack buffer to the shadow stack would recover most of the overhead and, due to the compact design and fixed layout of the shadow stack, could simplify the management of that buffer and possibly improve performance.

Our last experiment on SPEC CPU2006 evaluates the overhead of our three different shadow stack integrity mechanisms. For these experiments, we used a compact shadow stack with the register implementation and the `fault`-based epilogue. The results are in Figure 3.10. As expected, the information hiding scheme is the fastest with 4.31% overhead. The MPX-based, code centric, isolation scheme was the next fastest with 12.12% overhead on average. The MPK thread centric, isolation scheme had 61.18% overhead. Our finding is in line with [82] which finds that adding a permission switch to a direct call increases the number of cycles for the call from 8 to 69. Consequently, we conclude that MPK is serializing execution, and was not intended for hot path use. MPX has a code size increase of 41.67% vs 21.24% for MPK. Neither the MPX nor MPK overhead numbers are acceptable for a deployed mechanism, highlighting the need for our proposed privi leged move instruciton, as per Section 3.4.3.

Table 3.2.: Phoronix benchmark results

Benchmark	Overhead	Deviation
sqlite	8.94%	0.22%
flac	1.19%	0.85%
MP3	1.47%	0.28%
wavpack	0.35%	0.15%
crafty	0.84%	0.15%
hmmer	0.28%	0.42%
LZMA	0.84%	0.29%
apache	-2.05%	0.40%
minion-graceful	1.18%	0.16%
minion-quasigroup	3.39%	0.13%

Table 3.3.: Apache throughput reduction

File Size	Simultaneous Connections		
	1	4	8
70K - HTML	6.21%	0.63%	-0.40%
1.4M - Image	1.13%	0.45%	-0.31%

3.6.2 Shadesmar Case Studies

We recommend Shadesmar: a compact, register based shadow stack that directly uses the shadow RA, and relies on information hiding to protect the shadow stack for immediate deployment based on our SPEC CPU2006 analysis. Note that information hiding still significantly raises the bar for attackers by requiring an information leak, and a write to a region of memory with only one pointer into it (the shadow stack pointer) to bypass Shadesmar. Shadesmar exclusively keeps the shadow stack pointer in a register, making leaking the location of the shadow stack difficult.

To demonstrate the usefulness of Shadesmar for real software, we run benchmarks from the Phoronix test suite for typical desktop user experiences, and benchmark the throughput of the Apache webserver for server deployments. For all case studies, Shadesmar has minimal performance impact while greatly increasing security by removing *backward edge* control-flow transfers from the attack surface. In particular, this shows that on modern 64 bit architectures with 16 general purpose registers, dedicating one general purpose register to a security mechanism is acceptable in practice.

Phoronix. We run ten benchmarks from Phoronix with workloads including databases, audio encoding, data compression, chess, protein sequencing, and their version of Apache. These workloads are representative of common workloads for user space computation. The results are in Table 3.2. For eight of the ten benchmarks, the overhead is less than 2%; for five benchmarks overhead is within 1%; and it is within measurement noise of zero for two benchmarks. Consequently, we believe that Shadesmar is performant enough to be deployed in desktop computing environments, and that users would not notice any slow down.

Apache. To evaluate Shadesmar in server settings, we benchmarked the throughput of Apache with Shadesmar instrumentation. For this experiment, we used two different files, a

70KB HTML file and a 1.4MB image file, representative of the average size of webpages in 2016 [83]. The overhead drops with the number of connections, and file size, and is non-existent for eight concurrent connections, as shown in Table 3.3. This demonstrates that Shadesmar has no impact on the performance of IO bound applications like servers.

3.7 Related Work

Code-Reuse Attack Surface. Code-reuse attacks as an attack vector began with the original ROP attack [2]. Since then, the research community has worked to fully understand the scope of this attack vector. Follow on work established that any indirect control-flow transfer could be used for code-reuse attacks, not just returns [6, 84]. JIT-ROP [71] showed how just in time compiled code, like JavaScript, can be abused for code-reuse attacks. Counterfeit Object Oriented Programming (COOP) [3] specialized code reuse attacks for C++ programs, while PIROP [70] shows how to perform ROP in the face of ASLR. Control Jujustu [85] and Control Flow Bending [7] showed that CFI defenses cannot prevent code-reuse attacks in general. Newton [86] provides a framework for analyzing code-reuse defenses' security.

Control-Flow Integrity. CFI [62] mitigates forward edge code-reuse attacks. CFI mechanisms work by using static analysis to create an over approximation of the control-flow graph (CFG), and then enforce at runtime that all transitions must be within the statically computed CFG. After the initial proposal, follow on research has removed the need for whole program analysis [87, 88], and specialized CFI to use additional information in C++ programs when protecting virtual calls [89, 90]. To improve the precision of the CFG construction underlying CFI, more advanced static analysis techniques have been proposed [91]. Alternately, dynamic analysis-based approaches that leverage execution history [92], or analyze execution history on a separate core [93] significantly increase the precision of CFI, and thereby the security it provides. See [25] for a survey of CFI techniques.

Alternatives to CFI for forward edge protection have been proposed. Code Pointer Integrity (CPI) [65] isolates and protects code pointers, thereby keeping them from being corrupted. CPI included a proposal for Safe Stacks which rely on a precise escape analysis for stack variables, and other inter-procedural analysis to divide the stack into two new stacks: a safe stack with the return address, and variables that cannot be accessed through pointers, and an unsafe stack. Safe stacks have significant compatibility problems, particularly with unprotected code and without full program analysis the conservative analysis ends up allocating a large number of unsafe stack frames, resulting in unnecessary overhead. CFIXX [26] provides object type integrity by protecting the virtual table pointers of C++ objects, thereby precisely protecting virtual dispatch.

Shadow Stacks. Prior work on shadow stacks is split between binary translation solutions [68, 94, 95, 96, 97] and compiler-based solutions [66, 67, 98, 99, 100]. The binary solutions employ static binary rewriting to add trampolines to the shadow stack instrumentation, and may enforce additional policies such as CFI, or utilize Intel's Process Trace (PT) feature and an additional core to analyze the process trace [96]. The compiler-based solutions come in three flavors: those that only attempt to prevent stack pivots [98, 99], an attempt to remove all ROP gadgets from the binary [100], and finally full shadow stacks [66, 67], which offer the strongest security. Shadesmar builds on full shadow stacks and introduces a dedicated shadow stack register to improve performance for compact shadow stacks, and compatibility by fully supporting stack unwinding. We also introduce hardware mechanisms to integrity protect the shadow stack.

3.8 Conclusion

With the increasing deployment of CFI to protect against forward-edge attacks, backward-edge defenses are required to fully mitigate control-flow hijack attacks. We conduct a qualitative and quantitative study of the design space of shadow stacks along performance, compatibility, and security dimensions and propose Shadesmar, a register-based, performant, secure, and deployable shadow stack mechanism that is compatible with all required C/C++

paradigms. Our case studies on Apache, where we had no performance impact for real work loads, and Phoronix where we had less than 2% overhead for 8 of the 10 benchmarks show Shadesmar's deployability. Orthogonally, we show that no existing HW mechanisms is usable in practice for intra-process address space isolation, and propose a new code-centric mechanism to fit this need for general security monitors that require mutable metadata. Our evaluation shows that Shadesmar is practical, prevents backward-edge attacks, and, together with CFI, will stop control-flow hijacking.

4 CONTROL-FLOW INTEGRITY: PRECISION, SECURITY, AND PERFORMANCE

Systems programming languages such as C and C++ give programmers a high degree of freedom to optimize and control how their code uses available resources. While this facilitates the construction of highly efficient programs, requiring the programmer to manually manage memory and observe typing rules leads to security vulnerabilities in practice. Memory corruptions, such as buffer overflows, are routinely exploited by attackers. Despite significant research into exploit mitigations, very few of these mitigations have entered practice [4]. The combination of three such defenses, (i) Address Space Layout Randomization (ASLR) [10], (ii) stack canaries [101], and (iii) Data Execution Prevention (DEP) [9] protects against *code-injection attacks,* but are unable to fully prevent *code-reuse attacks*. Modern exploits use Return-Oriented Programming (ROP) or variants thereof to bypass currently deployed defenses and divert the control flow to a malicious payload. Common objectives of such payloads include arbitrary code execution, privilege escalation, and exfiltration of sensitive information.

The goal of Control-Flow Integrity (CFI) [102] is to restrict the set of possible control-flow transfers to those that are strictly required for correct program execution. This prevents code-reuse techniques such as ROP from working because they would cause the program to execute control-flow transfers which are illegal under CFI. Conceptually, most CFI mechanisms follow a two-phase process. An *analysis* phase constructs the Control-Flow Graph (CFG) which approximates the set of legitimate control-flow transfers. This CFG is then used at runtime by an *enforcement* component to ensure that all executed branches correspond to edges in the CFG.

During the analysis phase, the CFG is computed by analyzing either the source code or binary of a given program. In either case, the limitations of static program analysis lead to an over-approximation of the control-flow transfers that can actually take place at runtime. This over-approximation limits the security of the enforced CFI policy because some non-essential edges are included in the CFG.

The enforcement phase ensures that control-flow transfers which are potentially controlled by an attacker, i.e., those whose targets are computed at runtime, such as indirect branches and return instructions, correspond to edges in the CFG produced by the analysis phase. These targets are commonly divided into forward edges such as indirect calls, and backward edges like return instructions (so called because they return control back to the calling function). All CFI mechanisms protect forward edges, but some do not handle backward edges. Assuming code is static and immutable [1], CFI can be enforced by instrumenting existing indirect control-flow transfers at compile time through a modified compiler, ahead of time through static binary rewriting, or during execution through dynamic binary translation. The types of indirect transfers that are subject to such validation and the number of valid targets per branch varies greatly between different CFI defenses. These differences have a major impact on the security and performance of the CFI mechanism.

CFI does not seek to prevent memory corruption, which is the root cause of most vulnerabilities in C and C++ code. While mechanisms that enforce spatial [12] and temporal [13] memory safety eliminate memory corruption (and thereby control-flow hijacking attacks), existing mechanisms are considered prohibitively expensive. In contrast, CFI defenses offer reasonably low overheads while making it substantially harder for attackers to gain arbitrary code execution in vulnerable programs. Moreover, CFI requires few changes to existing source code which allows complex software to be protected in a mostly automatic fashion. While the idea of restricting branch instructions based on target sets predates CFI [104, 105, 106], Abadi et al.'s seminal paper [102] was the first formal description of CFI with an accompanying implementation. Since this paper was published over a decade ago, the research community has proposed a large number of variations of the original idea.

[1] DEP marks code pages as executable and readable by default. Programs may subsequently change permissions to make code pages writable using platform-specific APIs such as `mprotect`. Mitigations such as PaX MPROTECT, SELinux [103], and the `ProcessDynamicCodePolicy` Windows API restrict how page permissions can be changed to prevent code injection and modification.

More recently, CFI implementations have been integrated into production-quality compilers, tools, and operating systems.

Current CFI mechanisms can be compared along two major axes: performance and security. In the scientific literature, performance overhead is usually measured through the SPEC CPU2006 benchmarks. Unfortunately, sometimes only a subset of the benchmarks is used for evaluation. To evaluate security, many authors have used the Average Indirect target Reduction (AIR) [107] metric that counts the overall reduction of targets for any indirect control-flow transfer.

Current evaluation techniques do not adequately distinguish among CFI mechanisms along these axes. Performance measurements are all in the same range, between 0% and 20% across different benchmarks with only slight variations for the same benchmark. Since the benchmarks are evaluated on different machines with different compilers and software versions, these numbers are close to the margin of measurement error. On the security axis, AIR is not a desirable metric for two reasons. First, all CFI mechanisms report similar AIR numbers (a $> 99\%$ reduction of branch targets), which makes AIR unfit to compare individual CFI mechanisms against each other. Second, even a large reduction of targets often leaves enough targets for an attacker to achieve the desired goals [108, 109, 110], making AIR unable to evaluate security of CFI mechanisms on an absolute scale.

We systematize the different CFI mechanisms (where "mechanism" captures both the analysis and enforcement aspects of an implementation) and compare them against metrics for security and performance. By introducing metrics for these areas, our analysis allows the objective comparison of different CFI mechanisms both on an absolute level and relatively against other mechanisms. This in turn allows potential users to assess the trade-offs of individual CFI mechanisms and choose the one that is best suited to their use case. Further, our systematization provides a more meaningful way to classify CFI mechanism than the ill-defined and inconsistently used "coarse" and "fine" grained classification.

To evaluate the security of CFI mechanisms we follow a *comprehensive* approach, classifying them according to a *qualitative* and a *quantitative* analysis. In the qualitative security discussion we compare the strengths of individual solutions on a conceptual level by evaluating the CFI policy of each mechanism along several axes: (i) precision in the forward direction, (ii) precision in the backward direction, (iii) supported control-flow transfer types according to the source programming language, and (iv) reported performance. In the quantitative evaluation, we measure the target sets generated by each CFI mechanism for the SPEC CPU2006 benchmarks. The precision and security guarantees of a CFI mechanism depend on the *precision* of target sets used at runtime, i.e., across all control-flow transfers, how many superfluous targets are reachable through an individual control-flow transfer. We compute these target sets for all available CFI mechanisms and compare the ranked sizes of the sets against each other. This methodology lets us compare the actual sets used for the integrity checks of one mechanism against other mechanisms. In addition, we collect all indirect control-flow targets used for the individual SPEC CPU2006 benchmarks and use these sets as a lower bound on the set of required targets. We use this lower bound to compute how close a mechanism is to an *ideal* CFI mechanism. An ideal CFI mechanism is one where the enforced CFG's edges exactly correspond to the executed branches.

As a second metric, we evaluate the performance impact of open-sourced, compiler-based CFI mechanisms. In their corresponding publications, each mechanism was evaluated on different hardware, different libraries, and different operating systems, using either the full or a partial set of SPEC CPU2006 benchmarks. We cannot port all evaluated CFI mechanisms to the same baseline compiler. Therefore, we measure the overhead of each mechanism relative to the compiler it was integrated into. This apples-to-apples comparison highlights which SPEC CPU2006 benchmarks are most useful when evaluating CFI.

The paper is structured as follows. We first give a detailed background of the theory underlying the analysis phase of CFI mechanisms. This allows us to then qualitatively compare the different mechanisms on the precision of their analysis. We then quantify this comparison with a novel metric. This is followed by our performance results for the different implementation. Finally, we highlight best practices and future research directions for the CFI community identified during our evaluation of the different mechanisms, and conclude.

Overall, we present the following contributions:

1. a systematization of CFI mechanisms with a focus on discussing the major different CFI mechanisms and their respective trade-offs,
2. a taxonomy for classifying the underlying analysis of a CFI mechanism,
3. presentation of both a qualitative and quantitative security metric and the evaluation of existing CFI mechanisms along these metrics, and
4. a detailed performance study of existing CFI mechanisms.

4.1 Foundational Concepts

We first introduce CFI and discuss the two components of most CFI mechanisms: (i) the *analysis* that defines the CFG (which inherently limits the precision that can be achieved) and (ii) the runtime instrumentation that *enforces* the generated CFG. Secondly, we classify and systematize different types of control-flow transfers and how they are used in programming languages. Finally, we briefly discuss the CFG precision achievable with different types of static analysis.

4.1.1 Control-Flow Integrity

CFI is a policy that restricts the execution flow of a program at runtime to a predetermined CFG by validating indirect control-flow transfers. On the machine level, indirect control-flow transfers may target any executable address of mapped memory, but in the source language (C, C++, or Objective-C) the targets are restricted to valid language constructs such as functions, methods and switch statement cases. Since the aforementioned languages rely on manual memory management, it is left to the programmer to ensure that non-control data accesses do not interfere with accesses to control data such that programs execute legitimate control flows. Absent any security policy, an attacker can therefore exploit memory corruption to redirect the control-flow to an arbitrary memory location, which

```
   void foo(int a){
 2     return;
   }
 4 void bar(int a){
       return;
 6 }
   void baz(void){
 8     int a = input();
       void (*fptr)(int);
10     if(a){
         fptr = foo;
12       fptr();
       } else {
14       fptr = bar;
         fptr();
16     }
   }
18
```

Figure 4.1.: Simplified example of over approximation in static analysis.

is called control-flow hijacking. CFI closes the gap between machine and source code semantics by restricting the allowed control-flow transfers to a smaller set of target locations. This smaller set is determined per indirect control-flow location. Note that languages providing complete memory and type safety generally do not need to be protected by CFI. However, many of these "safe" languages rely on virtual machines and libraries written in C or C++ that will benefit from CFI protection.

Most CFI mechanisms determine the set of valid targets for each indirect control-flow transfer by computing the CFG of the program. The security guarantees of a CFI mechanism depend on the precision of the CFG it constructs. The CFG cannot be perfectly precise for non-trivial programs. Because the CFG is statically determined, there is always some over-approximation due to imprecision of the static analysis. An equivalence class is the set of valid targets for a given indirect control-flow transfer. Throughout the following, we reference Figure 4.1. Assuming an analysis based on function types or a flow-insensitive analysis, both `foo()` and `bar()` end up in the same equivalence class. Thus, at line 12 and line 15 either function can be called. However, from the source code we can tell that at line 12 only `foo()` should be called, and at line 15 only `bar()` should be called. While this specific problem can be addressed with a flow-sensitive analysis, all known static program analysis techniques are subject to some over-approximation.

Once the CFI mechanism has computed an approximate CFG, it has to enforce its security policy. We first note that CFI does not have to enforce constraints for control-flows due to direct branches because their targets are immune to memory corruption thanks to DEP. Instead, it focuses on attacker-corruptible branches such as indirect calls, jumps, and returns. In particular, it must protect control-flow transfers that allow runtime-dependent, targets such as `void (*fptr)(int)` in Figure 4.1. These targets are stored in either a register or a memory location depending on the compiler and the exact source code. The indirection such targets provide allows flexibility as, e.g., the target of a function may depend on a call-back that is passed from another module. Another example of indirect control-flow transfers is return instructions that read the return address from the stack. Without such an indirection, a function would have to explicitly enumerate all possible callers and check to which location to return to based on an explicit comparison.

For indirect call sites, the CFI enforcement component validates target addresses before they are used in an indirect control-flow transfer. This approach detects code pointers (including return addresses) that were modified by an attacker – if the attacker's chosen target is not a member of the statically determined set.

4.1.2 Classification of Control-Flow Transfers

Control-flow transfers can broadly be separated into two categories: (i) *forward* and (ii) *backward*. Forward control-flow transfers are those that move control to a new location inside a program. When a program returns control to a prior location, we call this a backward control-flow[2].

[2]Note the ambiguity of a backward edge in machine code (i.e., a backward jump to an earlier memory location) which is different from a backward control-flow transfer as used in CFI.

A CPU's instruction-set architecture (ISA) usually offers two forward control-flow transfer instructions: call and jump. Both of these are either direct or indirect, resulting in four different types of forward control-flow:

- *direct jump*: is a jump to a constant, statically determined target address. Most local control-flow, such as loops or if-then-else cascaded statements, use direct jumps to manage control.
- *direct call*: is a call to a constant, statically determined target address. Static function calls, for example, use direct call instructions.
- *indirect jump*: is a jump to a computed, i.e., dynamically determined target address. Examples for indirect jumps are switch-case statements using a dispatch table, Procedure Linkage Tables (PLT), as well as the threaded code interpreter dispatch optimization [111, 112, 113].
- *indirect call*: is a call to a computed, i.e., dynamically determined target address. The following three examples are relevant in practice:

  Function pointers are often used to emulate object-oriented method dispatch in classical record data structures, such as C `structs`, or for passing callbacks to other functions.

  vtable dispatch is the preferred way to implement dynamic dispatch to C++ methods. A C++ object keeps a pointer to its *vtable*, a table containing pointers to all virtual methods of its dynamic type. A method call, therefore, requires (i) dereferencing the vtable pointer, (ii) computing table index using the method offset determined by the object's static type, and (iii) an indirect call instruction to the table entry referenced in the previous step. In the presence of multiple inheritance, or multiple dispatch, dynamic dispatch is slightly more complicated.

  Smalltalk-style `send`-method dispatch that requires a dynamic type look-up. Such a dynamic dispatch using a `send`-method in Smalltak, Objective-C, or JavaScript requires walking the class hierarchy (or the prototype chain in JavaScript) and selecting the first method with a matching identifier. This procedure is required for all method calls and therefore impacts performance negatively. Note that, e.g., Objective-C uses a lookup cache to reduce the overhead.

We note that jump instructions can also be either conditional or unconditional. For the purposes of this paper this distinction is irrelevant.

All common ISAs support backward and forward indirect control-flow transfers. For example, the x86 ISA supports backward control-flow transfers using just one instruction: return, or just `ret`. A return instruction is the symmetric counterpart of a call instruction, and a compiler emits function prologues and epilogues to form such pairs. A call instruction pushes the address of the immediately following instruction onto the native machine stack. A return instruction pops the address off the native machine stack and updates the CPU's instruction pointer to point to this address. Notice that a return instruction is conceptually similar to an indirect jump instruction, since the return address is unknown a priori. Furthermore, compilers are emitting call-return pairs by *convention* that hardware usually does not enforce. By modifying return addresses on the stack, an attacker can "return" to all addresses in a program, the foundation of return-oriented programming [2, 5, 84].

Control-flow transfers can become more complicated in the presence of exceptions. Exception handling complicates control-flows locally, i.e., within a function, for example by moving control from a try-block into a catch-block. Global exception-triggered control-flow manipulation, i.e., interprocedural control-flows, require unwinding stack frames on the current stack until a matching exception handler is found.

Other control-flow related issues that CFI mechanisms should (but not always do) address are: (i) separate compilation, (ii) dynamic linking, and (iii) compiling libraries. These present challenges because the entire CFG may not be known at compile time. This problem can be solved by relying on LTO, or dynamically constructing the combined CFG. Finally, keep in mind that, in general, not all control-flow transfers can be recovered from a binary.

Summing up, our classification scheme of control-flow transfers is as follows:

- **CF.1**: backward control-flow,
- **CF.2**: forward control-flow using direct jumps,
- **CF.3**: forward control-flow using direct calls,
- **CF.4**: forward control-flow using indirect jumps,
- **CF.5**: forward control-flow using indirect calls supporting function pointers,
- **CF.6**: forward control-flow using indirect calls supporting vtables,
- **CF.7**: forward control-flow using indirect calls supporting Smalltalk-style method dispatch,
- **CF.8**: complex control-flow to support exception handling,
- **CF.9**: control-flow supporting language features such as dynamic linking, separate compilation, etc.

According to this classification, the C programming language uses control-flow transfers 1–5, 8 (for setjmp/longjmp) and 9, whereas the C++ programming language allows all control-flow transfers except no. 7.

4.1.3 Classification of Static Analysis Precision

As we saw in Section 4.1.1, the security guarantees of a CFI mechanism ultimately depend on the precision of the CFG that it computes. This precision is in turn determined by the type of static analysis used. For the purposes of this paper, the following classification summarizes prior work to determine forward control-flow transfer analysis precision. In order of increasing static analysis precision (SAP), our classifications are:

- **SAP.F.0**: No forward branch validation
- **SAP.F.1a**: ad-hoc algorithms and heuristics
- **SAP.F.1b**: context- and flow-insensitive analysis

- **SAP.F.1c**: labeling equivalence classes
- **SAP.F.2**: class-hierarchy analysis
- **SAP.F.3**: rapid-type analysis
- **SAP.F.4a**: flow-sensitive analysis
- **SAP.F.4b**: context-sensitive analysis
- **SAP.F.5**: context- and flow-sensitive analysis
- **SAP.F.6**: dynamic analysis (optimistic)

The following classification summarizes prior work to determine backward control-flow transfer analysis precision:

- **SAP.B.0**: No backward branch validation
- **SAP.B.1**: Labeling equivalence classes
- **SAP.B.2**: Shadow stack

Note that there is well established and vast prior work in static analysis that goes well beyond the scope of this paper [114]. The goal of our systematization is merely to summarize the most relevant aspects and use them to shed more light on the precision aspects of CFI.

4.1.4 Nomenclature and Taxonomy

Prior work on CFI usually classifies mechanisms into fine-grained and coarse-grained. Over time, however, these terms have been used to describe different systems with varying granularity and have, therefore, become overloaded and imprecise. In addition, prior work only uses a rough separation into forward and backward control-flow transfers without considering sub types or precision. We hope that the classifications here will allow a more precise and consistent definition of the precision of CFI mechanisms underlying analysis, and will encourage the CFI community to use the most precise techniques available from the static analysis literature.

4.2 Security

In this section we present a security analysis of existing CFI implementations. Drawing on the foundational knowledge in Section 4.1, we present a qualitative analysis of the theoretical security of different CFI mechanisms based on the policies that they implement. We then give a quantitative evaluation of a selection of CFI implementations. Finally, we survey previous security evaluations and known attacks against CFI.

4.2.1 Qualitative Security Guarantees

Our qualitative analysis of prior work and proposed CFI implementations relies on the classifications of the previous section (cf. Section 4.1) to provide a higher resolution view of precision and security. Figure 4.2 summarizes our findings among four dimensions based on the author's reported results and analysis techniques. Figure 4.3 presents our verified results for open source LLVM-based implementations that we have selected. Further, it adds a quantitative argument based on our work in Section 4.2.2.

In Figure 4.2 the axes and values were calculated as follows. Note that (i) the scale of each axis varies based on the number of data points required and (ii) weaker/slower always scores lower and stronger/faster higher. Therefore, the area of the spider plot roughly estimates the security/precision of a given mechanism:

- CF: supported control-flow transfers, assigned based on our classification scheme in Section 4.1.2;
- RP: reported performance numbers. Performance is quantified on a scale of 1-10 by taking the arctangent of reported runtime overhead and normalizing for high granularity near the median overhead. An implementation with no overhead receives a full score of 10, and one with about 35% or greater overhead receives a minimum score of 1.
- SAP.F: static-analysis precision of forward control-flows, assigned based on our classification in Section 4.1.3; and

- SAP.B: static-analysis precision of backward control-flows, assigned based on our classification in Section 4.1.3.

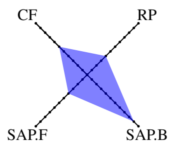
The shown CFI implementations are ordered chronologically by publication year, and the colors indicate whether a CFI implementation works on the binary-level (blue), relies on source-code (green), or uses other mechanisms (red), such as hardware implementations.

Our classification and categorization efforts for reported performance were hindered by methodological variances in benchmarking. Experiments were conducted on different machines, different operating systems, and also different or incomplete benchmark suites. Classifying and categorizing static analysis precision was impeded by the high level, imprecise descriptions of the implemented static analysis by various authors. Both of these impediments, naturally, are sources of imprecision in our evaluation.

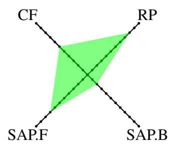
Comprehensive protection through CFI requires the validation of both forward and backward branches. This requirement means that the reported performance impact for forward-only approaches (i.e., SafeDispatch, T-VIP, VTV, IFCC, vfGuard, and VTint) is restricted to partial protection. The performance impact for backward control-flows must be considered as well, when comparing these mechanisms to others with full protection.

CFI mechanisms satisfying SAP.B.2, i.e., using a shadow stack to obtain high precision for backward control-flows are: original CFI [102], MoCFI [117], HAFIX [126, 137], and Lockdown [136]. PathArmor emulates a shadow stack through validating the last-branch register (LBR).

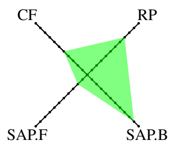
Increasing the precision of static analysis techniques that validate whether any given control-flow transfer corresponds to an edge in the CFG decreases the performance of the CFI mechanism. Most implementations choose to combine precise results of static analysis into an equivalence class. Each such equivalence class receives a unique identifier, often referred to as a label, which the CFI enforcement component validates at runtime. By not using a shadow stack, or any other comparable high-precision backward control-flow transfer validation mechanism, even high precision forward control-flow transfer static analysis becomes imprecise due to labeling. The explanation for this loss in precision is straightforward: to validate a control-flow transfer, all callers of a function need to carry the



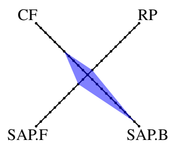
(a) Orig. CFI [102]



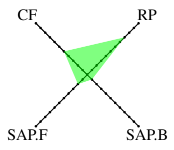
(b) Hypersafe [115]



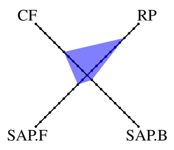
(c) CF-Locking [116]



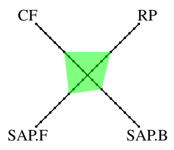
(d) MoCFI [117]



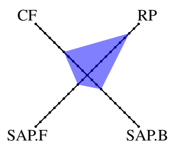
(e) MIP [118]



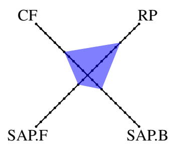
(f) kBouncer [119]



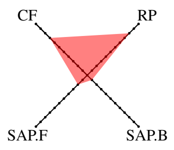
(g) CF-Restrictor [120]



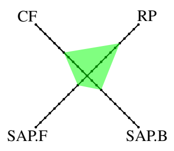
(h) CCFIR [121]



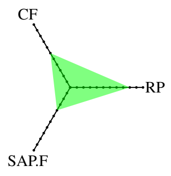
(i) bin-CFI [107]



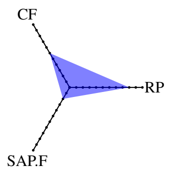
(j) ROPecker [122]



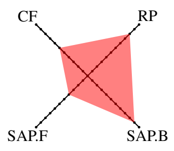
(k) KCoFI [123]



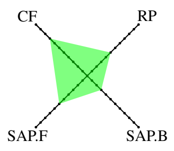
(l) SafeDispatch [124]



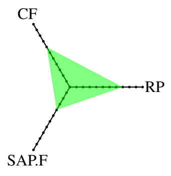
(m) T-VIP [125]



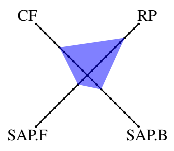
(n) HW-asst. CFI [126]



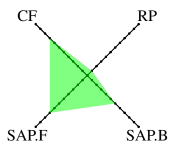
(o) RockJIT [127]



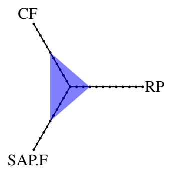
(p) VTV [128]



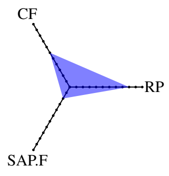
(q) O-CFI [129]



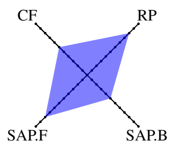
(r) C-CFI [130]



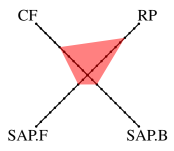
(s) vfGuard [131]



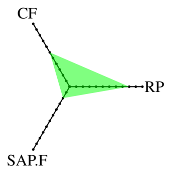
(t) VTint [132]



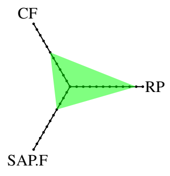
(u) PathArmor [133]



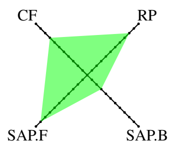
(v) CFIGuard [134]



(w) CFGuard [64]

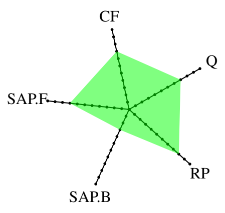


(x) VTI [90]

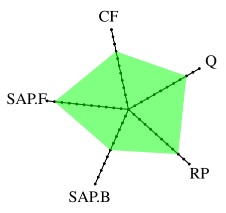


(y) Kernel CFI [135]

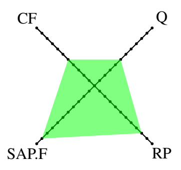
Figure 4.2.: CFI implementation comparison: supported control-flows (CF), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B). Backward (SAP.B) is omitted for mechanisms that do not support back edges. Color coding of CFI implementations: binary are blue, source-based are green, others red.



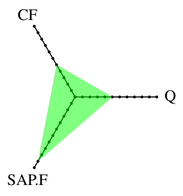
(a) MCFI
[87]



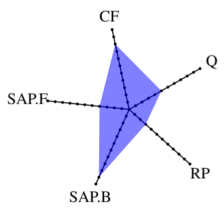
(b) πCFI
[88]



(c) IFCC
[128]



(d) LLVM-CFI-3.7
(2015)



(e) Lockdown
[136]

Figure 4.3.: Quantitative comparison: control-flows (CF), quantitative security (Q), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B).

same label. Labeling, consequently, is a substantial source of imprecision (see Section 4.2.2 for more details). The notable exception in this case is πCFI, which uses dynamic information, to activate pre-determined edges, dynamically enabling high-resolution, precise control-flow graph (somewhat analogous to dynamic points-to sets [138]. Borrowing a term from information-flow control [139], πCFI can, however, suffer from *label creep* by accumulating too many labels from the static CFG.

CFI implementations introducing imprecision via labeling are: the original CFI paper [102], control-flow locking [116], CF-restrictor [120], CCFIR [121], MCFI [87], KCoFI [123], and RockJIT [127].

According to the criteria established in analyzing points-to precision, we find that at the time of this writing, πCFI [88] offers the highest precision due to leveraging dynamic points-to information. πCFI's predecessors, RockJIT [127] and MCFI [87], already offered a high precision due to the use of context-sensitivity in the form of types. Ideal PathArmor also scores well when subject to our evaluation: high-precision in both directions, forward and backward, but is hampered by limited hardware resources (LBR size) and restricting protection to the main executable (i.e., trusting libraries). Lockdown [136] offers high precision on the backward edges but derives its equivalence classes from the number of libraries used in an application and is therefore inherently limited in the precision of the forward edges. IFCC [128] offers variable static analysis granularity. On the one hand, IFCC describes a Full mode that uses type information, similar to πCFI and its predecessors. On the other hand, IFCC mentions less precise modes, such as using a single set for all destinations, and separating by function arity. With the exception of Hypersafe [115], all other evaluated CFI implementations with supporting academic publications offer lower precision of varying degrees, at most as precise as SAP.F.3.

4.2.2 Quantitative Security Guarantees

Quantitatively assessing how much security a CFI mechanism provides is challenging as attacks are often program dependent and different implementations might allow different attacks to succeed. So far, the only existing quantitative measure of the security of a CFI implementation is Average Indirect Target Reduction (AIR). Unfortunately, AIR is known to be a weak proxy for security [128]. A more meaningful metric must focus on the number of targets (i.e., number of equivalence classes) available to an attacker. Furthermore, it should recognize that smaller classes are more secure, because they provide less attack surface. Thus, an implementation with a small number of large equivalence classes is more vulnerable than an implementation with a large number of small equivalence classes.

One possible metric is the product of the number of equivalence classes (EC) and the inverse of the size of the largest class (LC), see Equation 4.1. Larger products indicate a more secure mechanism as the product increases with the number of equivalence classes and decreases with the size of the largest class. More equivalence classes means that each class is smaller, and thus provides less attack surface to an adversary. Controlling for the size of the largest class attempts to control for outliers, e.g., one very large and thus vulnerable class and many smaller ones. A more sophisticated version would also consider the usability and functionality of the sets. Usability considers whether or not they are located on an attacker accessible "hot" path, and if so how many times they are used. Functionality evaluates the quality of the sets, whether or not they include "dangerous" functions like mprotect. A large equivalence class that is pointed to by many indirect calls on the hot path poses a higher risk because it is more accessible to the attacker.

$$EC * \frac{1}{LC} = QuantitativeSecurity \tag{4.1}$$

This metric is not perfect, but it allows a meaningful direct comparison of the security and precision of different CFI mechanisms, which AIR does not. The gold standard would be adversarial analysis. However, this currently requires a human to perform the analysis on a per-program basis. This leads to a large number of methodological issues: how many analysts, which programs and inputs, how to combine the results, etc. Such a study is beyond the scope of this work, which instead uses our proposed metric which can be measured programatically.

This section measures the number and sizes of sets to allow a meaningful, direct comparison of the security provided by different implementations. Moreover, we report the dynamically observed number of sets and their sizes. This quantifies the maximum achievable precision from the implementations' CFG analysis, and shows how over-approximate they were for a given execution of the program.

Implementations

We evaluate four compiler-based, open-source CFI mechanisms IFCC, LLVM-CFI, MCFI, and πCFI. For IFCC and MCFI we also evaluated the different analysis techniques available in the implementation. Note that we evaluate two different versions of LLVM-CFI, the first release in LLVM 3.7 and the second, highly modified version in LLVM 3.9. In addition to the compiler-based solutions, we also evaluate Lockdown, which is a binary-based CFI implementation.

MCFI and πCFI already have a built-in reporting mechanism. For the other mechanisms we extend the instrumentation pass and report the number and size of the produced target sets. We then used the implementations to compile, and for πCFI run, the SPEC CPU2006 benchmarks to produce the data we report here. πCFI must be run because it does dynamic target activation. This does tie our results to the ref data set for SPEC CPU2006, because as with any dynamic analysis the results will depend on the input.

IFCC[3] comes with four different CFG analysis techniques: *single*, *arity*, *simplified*, and *full*. *Single* creates only one equivalence class for the entire program, resulting in the weakest possible CFI policy. *Arity* groups functions into equivalence classes based on their number of arguments. *Simplified* improves on this by recognizing three types of arguments: composite, integer, or function pointer. *Full* considers the precise return type and types of each argument. We expect full to yield the largest number of equivalence classes with the smallest sizes, as it performs the most exact distribution of targets.

[3]Note that the IFCC patch was pulled by the authors and will be replaced by LLVM-CFI.

Both MCFI and πCFI rely on the same underlying static analysis. The authors claim that disabling tail calls is the single most important precision enhancement for their CFG analysis [140]. We measure the impact of this option on our metric. MCFI and πCFI are also unique in that their policy and enforcement mechanisms consider backward edges as well as forward edges. When comparing to other implementations, we only consider forward edges. This ensures direct comparability for the number and size of sets. The results for backward edges are presented as separate entries in the figures.

As of LLVM 3.7, LLVM-CFI could not be directly compared to the other CFI implementations because its policy was strictly more limited. Instead of considering all forward, or all forward and backward edges, LLVM-CFI 3.7 focused on virtual calls and ensures that virtual, and non-virtual calls are performed on objects of the correct dynamic type. As of LLVM 3.9, LLVM-CFI has added support for all indirect calls. Despite these differences, we show the full results for both LLVM-CFI implementations in all tables and graphs.

Lockdown is a CFI implementation that operates on compiled binaries and supports the instrumentation of dynamically loaded code. To protect backward edges, Lockdown enforces a shadow stack. For the forward edge, it instruments libraries at runtime, creating one equivalence class per library. Consequently, the set size numbers are of the greatest interest for Lockdown. Lockdown's precision depends on symbol information, allowing indirect calls anywhere in a particular library if it is stripped. Therefore, we only report the set sizes for non-stripped libraries where Lockdown is more precise.

To collect the data for our lower bound, we wrote an LLVM pass. This pass instruments the program to collect and report the source line for each indirect call, the number of different targets for each indirect call, and the number of times each of those targets was used. This data is collected at runtime. Consequently, it represents only a subset of all possible indirect calls and targets that are required for the sample input to run. As such, we use it to present a lower bound on the number of equivalence sets (i.e. unique indirect call sites) and size of those sets (i.e. the number of different locations called by that site).

Results

We conducted three different quantitative evaluations in line with our proposed metric for evaluating the overall security of a CFI mechanism and our lower bound. For IFCC, LLVM-CFI (3.7 and 3.9), and MCFI it is sufficient to compile the SPEC CPU2006 benchmarks as they do not dynamically change their equivalence classes. πCFI uses dynamic information, so we had to run the SPEC CPU2006 benchmarks. Similarly, Lockdown is a binary CFI implementation that only operates at run time. We highlight the most interesting results in Figure 4.3.

Figure 4.4 shows the number of equivalence classes for the five CFI implementations that we evaluated, as well as their sub-configurations. As advertised, IFCC *Single* only creates one equivalence class. This IFCC mode offers the least precision of any implementation measured. The other IFCC analysis modes only had a noticeable impact for perlbench and soplex. Indeed, on the sjeng benchmark all four analysis modes produced only one equivalence class.

On forward edges, MCFI and πCFI are more precise than IFCC in all cases except for perlbench where they are equivalent. LLVM-CFI 3.9 is more precise than IFCC while being less precise than MCFI. MCFI and πCFI are the only implementations to consider backward edges, so no comparison with other mechanisms is possible on backward edge precision. Relative to each other, πCFI's dynamic information decreases the number of equivalence classes available to the attacker by 21.6%. The authors of MCFI and πCFI recommend disabling tail calls to improve CFG precision. This only impacts the number of sets that they create for backward edges, not forward edges. As such this compiler flag does not impact most CFI implementations, which rely on a shadow stack for backward edge security.

LLVM-CFI 3.7 creates a number of equivalence classes equal to the number of classes used in the C++ benchmarks. Recall that it only provides support for a subset of indirect control-flow transfer types. However, we present the results in Figure 4.4 and Figure 4.5 to show the relative cost of protecting vtables in C++ relative to protecting all indirect call sites.

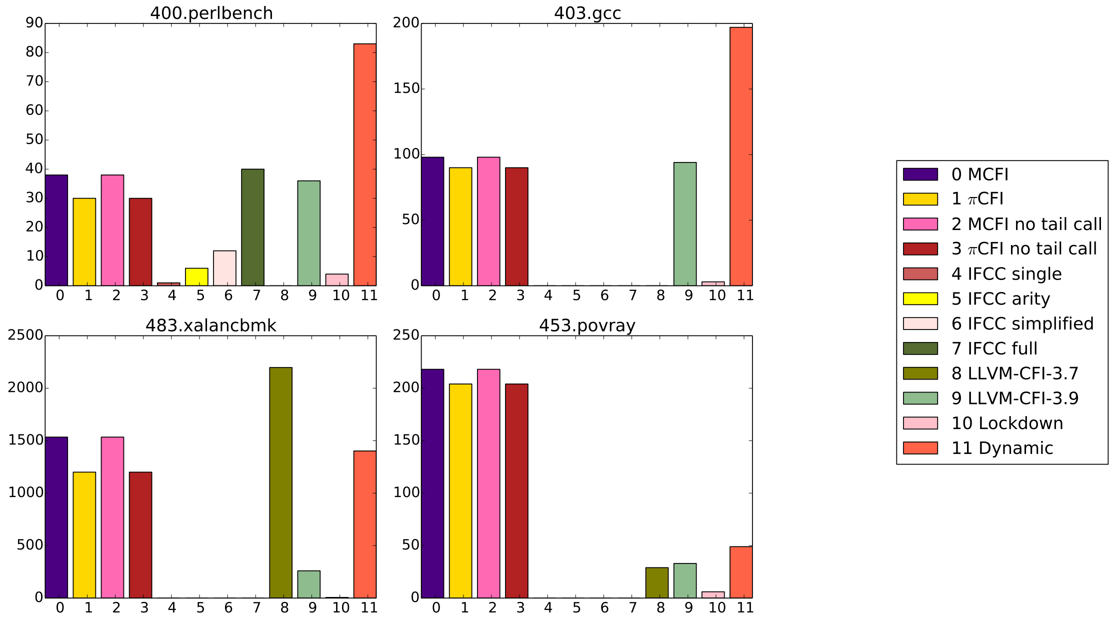


Figure 4.4.: Total number of forward-edge equivalence classes when running SPEC CPU2006 (higher is better).

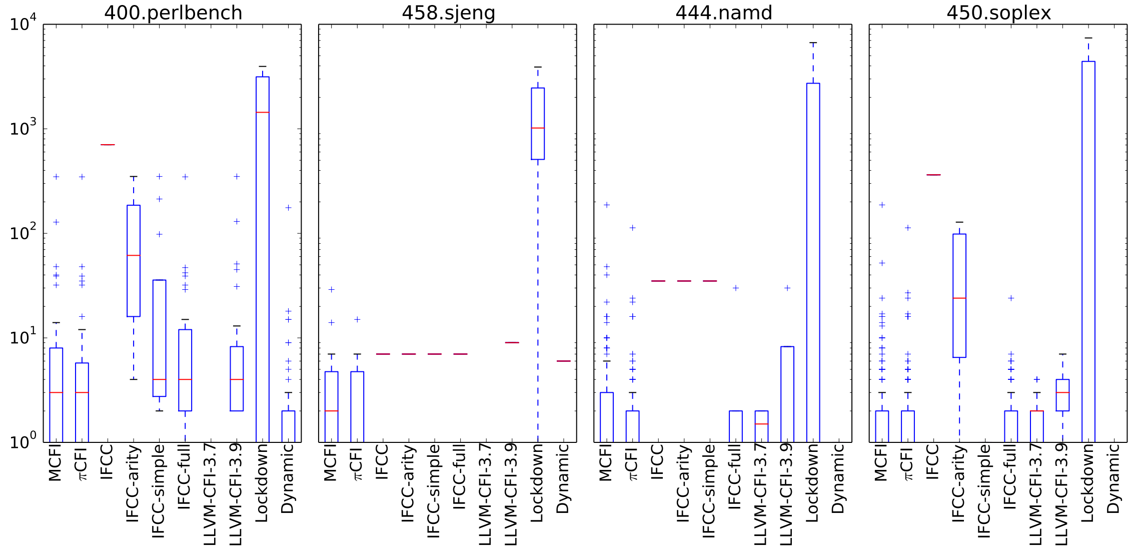


Figure 4.5.: Whisker plot of equivalence class sizes for different mechanisms when running SPEC CPU2006. (Smaller is Better)

We quantify the set sizes for each of the four implementations in Figure 4.5. We show box and whisker graphs of the set sizes for each implementation. The red line is the median set size and a smaller median set size indicates more secure mechanisms. The blue box extends from the 25th percentile to the 75th, smaller boxes indicate a tight grouping around the median. An implementation might have a low median, but large boxes indicate that there are still some large equivalence classes for an attacker to target. The top whisker extends from the top of the box for 150% of the size of the box. Data points beyond the whiskers are considered outliers and indicate large sets. This plot format allows an intuitive understanding of the security of the distribution of equivalence class sizes. Lower medians and smaller boxes are better. Any data points above the top of the whisker show very large, outlier equivalence classes that provide a large attack surface for an adversary.

Note that IFCC only creates a single equivalence class for xalancbmk and namd (except for the Full configuration on namd which is more precise). Entries with just a single equivalence class are reported as only a median. IFCC data points allow us to rank the different analysis methods, based on the results for benchmarks where they actually impacted set size: perlbench and soplex. In increasing order of precision (least precise to most precise) they are: *single*, *arity*, *simplified*, and *full*. This does not necessarily mean that the more precise analysis methods are more secure, however. For perlbench the more precise methods have outliers at the same level as the median for the least precise (i.e., *single*) analysis. For soplex the outliers are not as bad, but the *full* outlier is the same size as the median for *arity*. While increasing the precision of the underlying CFG analysis increases the overall security, edge cases can cause the incremental gains to be much smaller than anticipated.

The MCFI forward-edge data points highlight this. The MCFI median is always smaller than the IFCC median. However, for all the benchmarks where both ran, the MCFI outliers are greater than or equal to the largest IFCC set. From a quantitative perspective, we can only confirm that MCFI is at least as secure as IFCC. The effect of the outlying large sets on relative security remains an open question, though it seems likely that they provide opportunities for an attacker.

LLVM-CFI 3.9 presents an interesting compromise, as it has fewer outliers. However, it also has, on average, a greater median set size. Given the open question of the importance of the outliers, LLVM-CFI 3.9 could well be more secure in practice.

LLVM-CFI 3.7 - which only protects virtual tables - sets do not have extreme outliers. Additionally, Figure 4.5 shows that the equivalence classes that are created have a low variance, as seen by the more compact whisker plots that lack the large number of outliers present for other techniques. As such, LLVM-CFI 3.7 does not suffer from the edge cases that effect more general analyzes.

Lockdown consistently has the largest set sizes, as expected because it only creates one equivalence class per library and the SPEC CPU2006 benchmarks are optimized to reduce the amount of external library calls. These sets are up to an order of magnitude larger than compiler techniques. However, Lockdown isolates faults into libraries as each library has its independent set of targets compared to a single set of targets for other binary-only approaches like CCFIR and bin-CFI.

The lower bound numbers were measured dynamically, and as such encapsulate a subset of the actual equivalence sets in the static program. Further, each such set is at most the size of the static set. Our lower bound thus provides a proxy for an ideal CFI implementation in that it is perfectly precise for each run. However, all of the IFCC variations report fewer equivalence classes than our dynamic bound.

The whisker plots for our dynamic lower bound in Figure 4.5 show that some of the SPEC CPU2006 benchmarks inherently have outliers in their set sizes. For perlbench, gcc, gobmk, h264ref, omnetpp, and xalancbmk our dynamic lower bound and the static set sizes from the compiler-based implementations all have a significant number of outliers. This provides quantitative backing to the intuition that some code is more amenable to protection by CFI. Evaluating what coding styles and practices make code more or less amenable to CFI is out of scope here, but would make for interesting future work.

Note that for namd and soplex in Figure 4.5 there is no visible data for our dynamic lower bound because all the sets had a single element. This means the median size is one which is too low to be visible. For all other mechanisms no visible data means the mechanism was incompatible with the benchmark.

4.2.3 Previous Security Evaluations and Attacks

Evaluating the security of a CFI implementation is challenging because exploits are program dependent and simple metrics do not cover the security of a mechanism. The Average Indirect target Reduction (AIR) metric [107] captures the average reduction of allowed targets, following the idea that an attack is less likely if fewer targets are available. This metric and variants were then used to measure new CFI implementations, generally reporting high numbers of more than 99%. Such high numbers give the illusion of relatively high security but, e.g., if a binary has 1.8 MB of executable code (the size of the glibc on Ubuntu 14.04), then an AIR value of 99.9% still allows 1,841 targets, likely enough for an arbitrary attack. A similar alternative metric to evaluate CFI effectiveness is the gadget reduction metric [87]. Unfortunately, these simple relative metrics give, at best, an intuition for security and we argue that a more rigorous metric is needed.

A first set of attacks against CFI implementations targeted *coarse-grained* CFI that only had 1-3 equivalence classes [108, 109, 110]. These attacks show that equivalence classes with a large number of targets allow an attacker to execute code and system calls, especially if return instructions are allowed to return to any call site.

Counterfeit Object Oriented Programming (COOP) [3] introduced the idea that whole C++ methods can be used as gadgets to implement Turing-complete computation. Virtual calls in C++ are a specific type of indirect function calls that are dispatched via vtables, which are arrays of function pointers. COOP shows that an attacker can construct counterfeit objects and, by reusing existing vtables, perform arbitrary computations. This attack shows that indirect calls requiring another level-of-indirection (e.g., through a vtable) must have additional checks that consider the types at the language level for the security check as well.

Control Jujutsu [141] extends the existing attacks to so-called fine-grained CFI by leveraging the imprecision of points-to analysis. This work shows that common software engineering practices like modularity (e.g., supporting plugins and refactoring) force points-to analysis to merge several equivalence classes. This imprecision results in target sets that are large enough for arbitrary computation.

Control-Flow Bending [142] goes one step further and shows that attacks against ideal CFI are possible. Ideal CFI assumes that a precise CFG is available that is not achievable in practice, i.e., if any edge would be removed then the program would fail. Even in this configuration attacks are likely possible if no shadow stack is used, and sometimes possible even if a shadow stack is used.

Several attacks target data structures used by CFI mechanisms. StackDefiler [143] leverages the fact that many CFI mechanisms implement the enforcement as a compiler transformation. Due to this high-level implementation and the fact that the optimization infrastructure of the compiler is unaware of the security aspects, an optimization might choose to spill registers that hold sensitive CFI data to the stack where it can be modified by an attack [144]. Any CFI mechanism will rely on some runtime data structures that are sometimes writeable (e.g., when MCFI loads new libraries and merges existing sets). Missing the Point [145] shows that ASLR might not be enough to hide this secret data from an adversary.

4.3 Performance

While the security properties of CFI (or the lack thereof for some mechanisms) have received most scrutiny in the academic literature, performance characteristics play a large part in determining which CFI mechanisms are likely to see adoption and which are not. Szekeres et al. [4] surveyed mitigations against memory corruption and found that mitigations with more than 10% overhead do not tend to see widespread adoption in production environments and that overheads below 5% are desired by industry practitioners.

Comparing the performance characteristics of CFI mechanisms is a non-trivial undertaking. Differences in the underlying hardware, operating system, as well as implementation and benchmarking choices prevents apples-to-apples comparison between the performance overheads reported in the literature. For this reason, we take a two-pronged approach in our performance survey: for a number of publicly available CFI mechanisms, we measure performance directly on the same hardware platform and, whenever possible, on the same operating system, and benchmark suite. Additionally, we tabulate and compare the performance results reported in the literature.

We focus on the aggregate cost of CFI enforcement. For a detailed survey of the performance cost of protecting backward edges from callees to callers we refer to the recent, comprehensive survey by [66].

4.3.1 Measured CFI Performance

Selection Criteria It is infeasible to replicate the reported performance overheads for all major CFI mechanisms. Many implementations are not publicly available or require substantial modification to run on modern versions of Linux or Windows. We therefore focus on recent, publicly available, compiler-based CFI mechanisms.

Several compiler-based CFI mechanisms share a common lineage. LLVM-CFI, for instance, improves upon IFCC, πCFI improves upon MCFI, and VTI is an improved version of SafeDispatch. In those cases, we opted to measure the latest available version and rely on reported performance numbers for older versions.

Method Most authors use the SPEC CPU2006 benchmarks to report the overhead of their CFI mechanism. We follow this trend in our own replication study. All benchmarks were compiled using the `-O2` optimization level. The benchmarking system was a Dell PowerEdge T620 dual processor server having 64GiB of main memory and two Intel Xeon E5-2660 CPUs running at 2.20 GHz. To reduce benchmarking noise, we ran the tests on an otherwise idle system and disabled all dynamic frequency and voltage scaling features. Whenever possible, we benchmark the implementations under 64-bit Ubuntu Linux 14.04.2 LTS. The CFI mechanisms were baselined against the compiler they were implemented on top of: VTV on GCC 4.9, LLVM-CFI on LLVM 3.7 and 3.9, VTI on LLVM 3.7, MCFI on LLVM 3.5, πCFI on LLVM 3.5. Since CFGuard is part of Microsoft Visual C++ Compiler, MSVC, we used MSVC 19 to compile and run SPEC CPU2006 on a pristine 64-bit Windows 10 installation. We report the geometric mean overhead averaged over three benchmark runs using the reference inputs in Table 4.1.

Some of the CFI mechanisms we benchmark required link-time optimization, LTO, which allows the compiler to analyze and optimize across compilation units. LLVM-CFI and VTI both require LTO, so for these mechanisms, we report overheads relative to a baseline SPEC CPU2006 run that also had LTO enabled. The increased optimization scope enabled by LTO can allow the compiler to perform additional optimizations such as de-virtualization to lower the cost of CFI enforcement. On the other hand, LLVM's LTO is less practical than traditional, separate compilation, e.g., when compiling large, complex code bases. To measure the πCFI mechanism, we applied the author's patches[4] for 7 of the SPEC CPU2006 benchmarks to remove coding constructs that are not handled by πCFI's control-flow graph analysis [87]. Likewise, the authors of VTI provided a patch for the xalancbmk benchmark. It updates code that casts an object instance to its sibling class, which can cause a CFI violation. We found these patches for hmmer, povray, and xalancbmk to also be necessary for LLVM-CFI 3.9, which otherwise reports a CFI violation on these benchmarks. VTI was run in interleaved vtable mode which provides the best performance according to its authors [90].

Results Our performance experiments show that recent, compiler-based CFI mechanisms have mean overheads in the low single digit range. Such low overhead is well within the threshold for adoption specified by [4] of 5%. This dispenses with the concern that CFI enforcement is too costly in practice compared to alternative mitigations including those based on randomization [146]. Indeed, mechanisms such as CFGuard, LLVM-CFI, and VTV are implemented in widely-used compilers, offering some level of CFI enforcement to practitioners.

We expect CFI mechanisms that are limited to virtual method calls—VTV, VTI, LLVM-CFI 3.7— to have lower mean overheads than those that also protect indirect function

[4]The patches are available at: `https://github.com/mcfi/MCFI/tree/master/spec2006`.

Table 4.1.: Measured and reported CFI performance overhead (%) on the SPEC CPU2006 benchmarks. The programming language of each benchmark is indicated in parenthesis: C(C), C++(+), Fortran(F). CF in a cell indicates we were unable to build and run the benchmark with CFI enabled. Blank cells mean that no results were reported by the original authors or that we did not attempt to run the benchmark. Cells with bold fonts indicate 10% or more overhead, ntc stands for no tail calls.

Benchmark	Measured Performance							Reported Performance									
	VTV	LLVM-CFI		VTI	CFGuard	πCFI	πCFI	VTV	VTI	πCFI	IFCC	MCFI	PathArmor	Lockdown	C-CFI	ROPecker	bin-CFI
Version		3.7	3.9														
Options		LTO	LTO	LTO			ntc		LTO		LTO						
400.perlbench(C)			2.4			8.2	5.3			5.0	1.9	5.0	**15.0**	**150.0**		5.0	**12.0**
401.bzip2(C)			-0.7		-0.3	1.2	0.8			1.0		1.0	0.0	8.0	5.0	0.0	-9.0
403.gcc(C)			CF			6.1	**10.5**			4.5		4.5	9.0	**50.0**		3.0	4.5
429.mcf(C)			3.6		0.5	4.0	1.8			4.0		4.0	1.0	2.0	**10.0**	1.0	0.0
445.gobmk(C)			0.2		-0.2	**11.4**	**11.8**			7.5		7.0	0.0	**43.0**	**50.0**	1.0	**15.0**
456.hmmer(C)			0.1		0.7	0.1	-0.1			0.0		0.0	1.0	3.0	**10.0**	0.0	-0.5
458.sjeng(C)			1.6		3.4	8.4	**11.9**			5.0		5.0	0.0	**80.0**	**40.0**	0.0	-2.5
464.h264ref(C)			5.3		5.4	7.9	8.3			6.0		6.0	1.0	**43.0**	**45.0**	1.0	**28.0**
462.libquantum(C)			-6.9			-3.0	-1.0			-0.3		0.0	3.0	5.0	**10.0**	0.0	-0.5
471.omnetpp(+)	5.8	-1.9	CF	CF	3.8	6.7	**18.8**	8.0	1.2	5.0	-1.2	5.0				2.0	**45.0**
473.astar(+)	3.6	-0.3	0.9	1.6	0.1	2.0	2.9	2.4	0.1	4.0	-0.2	3.5		**17.0**	**75.0**	0.0	**14.0**
483.xalancbmk(+)	**24.0**	7.1	7.2	3.7	5.5	**10.3**	**17.6**	**19.2**	1.4	7.0	3.1	7.0		**118.0**	**170.0**	**15.0**	
410.bwaves(F)														1.0			
416.gamess(F)														**11.0**			
433.milc(C)			0.2		2.0	-1.7	1.4			2.0		2.0	4.0	8.0			2.5
434.zeusmp(F)														0.0			
435.gromacs(C,F)														1.0			
436.cactusADM(C,F)														0.0			
437.leslie3d(F)														1.0			
444.namd(+)	-0.1	-0.2	0.1	-0.3	0.1	-0.3	-0.5			-0.5	-0.2	-0.5		3.0			-2.0
447.dealII(+)	0.7	CF	7.9	CF	-0.1	5.3	4.4			4.5	-2.2	4.5					
450.soplex(+)	0.5	0.5	-0.3	-0.6	2.3	-0.7	0.9		-0.7	-4.0	-1.7	-4.0		**12.0**			3.5
453.povray(+)	-0.6	1.5	8.9	2.0	**10.8**	**11.3**	**17.4**			**10.5**	0.2	**10.0**		**90.0**			**37.0**
454.calculix(C,F)														3.0			
459.gemsFDTD(F)														7.0			
465.tonto(F)														**19.0**			
470.lbm(C)			-0.2		4.2	-0.2	-0.5			1.0		1.0	0.0	2.0			-2.5
482.sphinx3(C)			-0.8		-0.1	0.7	2.4			1.5		1.5	3.0	8.0			0.5
Geo Mean	4.6	1.1	4.4	1.3	2.3	4.0	5.8	9.6	0.5	3.2	-0.3	2.9	3.0	**20.0**	**45.0**	2.6	8.5

calls such as IFCC. The return protection mechanism used by MCFI should introduce additional overhead, and πCFI's runtime policy ought to result in a further marginal increase in overhead. In practice, our results show that LLVM-CFI 3.7 and VTI are the fastest, followed by CFGuard, πCFI, and VTV. The reported numbers for IFCC when run in *single* mode show that it achieves -0.3%, likely due to cache effects. Although our measured overheads are not directly comparable with those reported by the authors of the seminal CFI paper, we find that researchers have managed to improve the precision while lowering the cost[5] of enforcement as the result of a decade worth of research into CFI enforcement.

The geometric mean overheads do not tell the whole story, however. It is important to look closer at the performance impact on benchmarks that execute a high number of indirect branches. Protecting the xalancbmk, omnetpp, and povray C++ benchmarks with CFI generally incurs substantial overheads. All benchmarked CFI mechanisms had above-average overheads on xalancbmk. LLVM-CFI and VTV, which take virtual call semantics into account, were particularly affected. On the other hand, xalancbmk highlights the merits of the recent virtual table interleaving mechanism of VTI which has a relatively low 3.7% overhead (vs. 1.4% reported) on this challenging benchmark.

Although povray is written in C++, it makes few virtual method calls [132]. However, it performs a large number of indirect calls. The CFI mechanisms which protect indirect calls—πCFI, and CFGuard—all incur high performance overheads on povray. Sjeng and h264ref also include a high number of indirect calls which again result in non-negligible overheads particularly when using πCFI with tail calls disabled to improve CFG precision. The hmmer, namd, and bzip2 benchmarks on the other hand show very little overhead as they do not execute a high number of forward indirect branches of any kind. Therefore these benchmarks are of little value when comparing the performance of various CFI mechanisms.

Overall, our measurements generally match those reported in the literature. The authors of VTV [128] only report overheads for the three SPEC CPU2006 benchmarks that were impacted the most. Our measurements confirm the authors' claim that the runtimes of the other C++ benchmarks are virtually unaffected. The leftmost πCFI column should

[5]Non-CFI related hardware improvements, such as better branch prediction [147], also help to reduce performance overhead.

be compared to the reported column for πCFI. We measured overheads higher than those reported by Niu and Tan. Both gobmk and xalancbmk show markedly higher performance overheads in our experiments; we believe this is in part explained by the fact that Niu and Tan used a newer Intel Xeon processor having an improved branch predictor [147] and higher clock speeds (3.4 vs 2.2 GHz).

We ran πCFI in both normal mode and with tail calls disabled. The geometric mean overhead increased by 1.9% with tail calls disabled. Disabling tail calls in turn increases the number of equivalence classes on each benchmark Figure 4.4. This is a classic example of the performance/security precision trade-off when designing CFI mechanisms. Implementers can choose the most precise policy within their performance target. CFGuard offers the most efficient protection of forward indirect branches whereas πCFI offers higher security at slightly higher cost.

4.3.2 Reported CFI Performance

The right-hand side of Table 4.1 lists reported overheads on SPEC CPU2006 for CFI mechanisms that we do not measure. IFCC is the first CFI mechanism implemented in LLVM which was later replaced by LLVM-CFI. MCFI is the precursor to πCFI. PathArmor is a recent CFI mechanism that uses dynamic binary rewriting and a hardware feature, the Last Branch Record (LBR) [148] register, that traces the 16 most recently executed indirect control-flow transfers. Lockdown is a pure dynamic binary translation approach to CFI that includes precise enforcement of returns using a shadow stack. C-CFI is a compiler-based approach which stores a cryptographically-secure hash-based message authentication code, HMAC, next to each pointer. Checking the HMAC of a pointer before indirect branches avoids a static points-to analysis to generate a CFG. ROPecker is a CFI mechanism that uses a combination of offline analysis, traces recorded by the LBR register, and emulation in an attempt to detect ROP attacks. Finally, the bin-CFI approach uses static binary rewriting like the original CFI mechanism; bin-CFI is notable for its ability to protect stripped, position-independent ELF binaries that do not contain relocation information.

The reported overheads match our measurements: xalancbmk and povray impose the highest overheads—up to 15% for ROPecker, which otherwise exhibits low overheads, and 1.7x for C-CFI. The interpreter benchmark, perlbench, executes a high number of indirect branches, which leads to high overheads, particularly for Lockdown, PathArmor, and bin-CFI.

Looking at CFI mechanisms that do not require re-compilation—PathArmor, Lockdown, ROPecker, and bin-CFI we see that the mechanisms that only check the contents of the LBR before system calls (PathArmor and ROPecker) report lower mean overheads than approaches that comprehensively instrument indirect branches (Lockdown and bin-CFI) in existing binaries. More broadly, comparing compiler-based mechanisms with binary-level mechanisms, we see that compiler-based approaches are typically as efficient as the binary-level mechanisms that trace control flows using the LBR although compiler-based mechanisms do not limit protection to a short window of recently executed branches. More comprehensive binary-level mechanisms, Lockdown and bin-CFI generally have higher overheads than compiler-based equivalents. On the other hand, Lockdown shows the advantage of binary translation: almost any program can be analyzed and protected, independent from the compiler and source code. Also note that Lockdown incurs additional overhead for its shadow stack, while none of the other mechanisms in Table 4.1 have a shadow stack.

Although we cannot directly compare the reported overheads of bin-CFI with our measured overheads for CFGuard, the mechanisms enforce CFI policies of roughly similar precision (compare Figure 4.2i and Figure 4.2w). CFGuard, however, has a substantially lower performance overhead. This is not surprising given that compilers operate on a high-level program representation that is more amenable to static program analysis and optimization of the CFI instrumentation. On the other hand, compiler-based CFI mechanisms are not strictly faster than binary-level mechanisms, C-CFI has the highest reported overheads by far although it is implemented in the LLVM compiler.

Table 4.2 surveys CFI approaches that do not report overheads using the SPEC CPU2006 benchmarks like the majority of recent CFI mechanisms do. Some authors, use an older version of the SPEC benchmarks [102, 129] whereas others evaluate performance using, e.g., web browsers [121, 124], or web servers [136, 149]. Although it is valuable to quantify overheads of CFI enforcement on more modern and realistic programs, it remains helpful to include the overheads for SPEC CPU2006 benchmarks.

Table 4.2.: CFI performance overhead (%) reported from previous publications. A label of ^{C} indicates we computed the geometric mean overhead over the listed benchmarks, otherwise it is the published average.

	Benchmarks	Overhead
ROPGuard [150]	PCMark Vantage, NovaBench, 3DMark06, Peacekeeper, Sunspider, SuperPI 16M	0.5%
SafeDispatch [124]	Octane, Kraken, Sunspider, Balls, linelayout, HTML5	2.0%
CCFIR [121]	SPEC2kINT, SPEC2kFP, SPEC2k6INT	^{C} 2.1%
kBouncer [119]	wmplayer, Internet Explorer, Adobe Reader	^{C} 4.0%
OCFI [129]	SPEC2k	4.7%
CFIMon [149]	httpd, Exim, Wu-ftpd, Memcached	6.1%
Original CFI [102]	SPEC2k	16.0%

4.3.3 Discussion

As Table 4.1 shows, authors working in the area of CFI seem to agree to evaluate their mechanisms using the SPEC CPU2006 benchmarks. There is, however, less agreement on whether to include both the integer and floating point subsets. The authors of Lockdown report the most complete set of benchmark results covering both integer and floating point benchmarks and the authors of bin-CFI, πCFI, and MCFI include most of the integer benchmarks and a subset of the floating point ones. The authors of VTV and IFCC only report subsets of integer and floating point benchmarks where their solutions introduce non-negligible overheads. Except for CFI mechanisms focused on a particular type of control flows such as virtual method calls, authors should strive to report overheads on the full suite of SPEC CPU2006 benchmarks. In case there is insufficient time to evaluate a CFI mechanism on all benchmarks, we strongly encourage authors to focus on the ones that are challenging to protect with low overheads. These include perlbench, gcc, gobmk, sjeng, omnetpp, povray, and xalancbmk. Additionally, it is desirable to supplement SPEC CPU2006 measurements with measurements for large, frequently targeted applications such as web browsers and web servers.

Although “traditional” CFI mechanisms (e.g., those that check indirect branch targets using a pre-computed CFG) can be implemented most efficiently in a compiler, this does not automatically make such solutions superior to binary-level CFI mechanisms. The advantages of the latter type of approaches include, most prominently, the ability to work directly on stripped binaries when the corresponding source is unavailable. This allows CFI enforcement to be applied independently of the code producer and therefore puts the performance/security trade off in the hands of the end-users or system administrators. Moreover, binary-level solutions naturally operate on the level of entire program modules irrespective of the source language, compiler, and compilation mode that was used to generate the code. Implementers of compiler-based CFI solutions on the other hand must spend additional effort to support separate compilation or require LTO operation which, in some instances, lowers the usability of the CFI mechanism [4].

4.4 Cross-cutting Concerns

This section discusses CFI enforcement mechanisms, presents calls to action identified by our study for the CFI community, and identifies current frontiers in CFI research.

4.4.1 Enforcement Mechanisms

The CFI precursor Program Shepherding [104] was built on top of a dynamic optimization engine, RIO. For CFI like security policies, Program Shepherding effects the way RIO links basic blocks together on indirect calls. They improve the performance overhead of this approach by maintaining traces, or sequences of basic blocks, in which they only have to check that the indirect branch target is the same.

Many CFI papers follow the ID-based scheme presented by Abadi et. al [102]. This scheme assigns a label to each indirect control flow transfer, and to each potential target in the program. Before the transfer, they insert instrumentation to insure that the label of the control flow transfer matches the label of the destination.

Recent work from Google [63, 128] and Microsoft [64] has moved beyond the ID-based schemes to optimized set checks. These rely on aligning metadata such that pointer transformations can be performed quickly before indirect jumps. These transformations guarantee that the indirect jump target is valid.

Hardware-Supported Enforcement Modern processors offer several hardware security-oriented features. Data Execution Prevention is a classical example of how a simple hardware feature can eliminate an entire class of attacks. Many processors also support AES encryption, random number generation, secure enclaves, and array bounds checking via instruction set extensions.

Researchers have explored architectural support for CFI enforcement [126, 137, 151, 152] with the goal of lowering performance overheads. A particular advantage of these solutions is that backward edges can be protected by a fully-isolated shadow stack with an average overhead of just 2% for protection of forward and backward edges. This stands in contrast to the average overheads for software-based shadow stacks which range from 3 to 14% according to **(author?)** [66].

There have also been efforts to repurpose existing hardware mechanisms to implement CFI [119, 122, 133, 134]. **(author?)** [119] were first to demonstrate a CFI mechanism using the 16-entry LBR branch trace facility of Intel x86 processors. The key idea in their kBouncer solution is to check the control flow path that led up to a potentially dangerous system call by inspecting the LBR; a heuristic was used to distinguish execution traces induced by ROP chains from legitimate execution traces. ROPecker by **(author?)** [122] subsequently extended LBR-based CFI enforcement to also emulate what code would execute past the system call. While these approaches offer negligible overheads and do not require recompilation of existing code, subsequent research showed that carefully crafted ROP attacks can bypass both of these mechanisms [108, 109, 110]. The CFIGuard mechanism [134] uses the LBR feature in conjunction with hardware performance counters to heuristically detect ROP attacks. [149] used the branch trace store, which records control-flow transfers to a buffer in memory, rather than the LBR for CFI enforcement. **(author?)** [130]'s C-CFI uses the Intel AES-NI instruction set to compute cryptographically-enforced hash-based message authentication codes, HMACs, for pointers stored in attacker-observable memory. By verifying HMACs before pointers are used, C-CFI prevents control-flow hijacking. **(author?)** [129] leverage Intel's MPX instruction set extension by re-casting the problem of CFI enforcement as a bounds checking problem over a randomized CFG.

Most recently, Intel announced hardware support for CFI in future x86 processors [153]. Intel Control-flow Enforcement Technology (CET) adds two new instructions, ENDBR32 and ENDBR64, for forward edge protection. Under CET, the target of any indirect jump or indirect call must be a ENDBR instruction. This provides coarse-grained protection where any of the possible indirect targets are allowed at every indirect control-flow transfer. There is only one equivalence class which contains every ENDBR instruction in the program. For backward edges, CET provides a new Shadow Stack Pointer (SSP) register which is exclusively manipulated by new shadow stack instructions. Memory used by the shadow stack resides in virtual memory and is protected with page permissions. In summary, CET provides precise backward edge protection using a shadow stack, but forward edge protection is imprecise because there is only one possible label for destinations.

4.4.2 Open Problems

As seen in Section 4.2.1 most existing CFI implementations use ad hoc, imprecise analysis techniques when constructing their CFG. This unnecessarily weakens these mechanisms, as seen in Section 4.2.2. All future work in CFI should use flow-sensitive and context-sensitive analysis for forward edges, SAP.F.5 from Section 4.1.3. On backward edges, we recommend shadow stacks as they have negligible overhead and are more precise than any possible static analysis. In this same vein, a study of real world applications that identifies coding practices that lead to large equivalence classes would be immensely helpful. This could lead to coding best practices that dramatically increase the security provided by CFI.

Quantifying the incremental security provided by CFI, or any other security mechanism, is an open problem. However, a large adversarial analysis study would provide additional insight into the security provided by CFI. Further, it is likely that CFI could be adapted as a result of such a study to make attacks more difficult.

4.4.3 Research Frontiers

Recent trends in CFI research target improving CFI in directions beyond new analysis or enforcement algorithms. Some approaches have sought to increase CFI protection coverage to include just-in-time code and operating system kernels. Others leverage advances in hardware to improve performance or enable new enforcement strategies. We discuss these research directions in the CFI landscape which cross-cut the traditional categories of performance and security.

Protecting Operating System Kernels. In monolithic kernels, all kernel software is running at the same privilege levels and any memory corruption can be fatal for security. A kernel is vastly different from a user-space application as it is directly exposed to the underlying hardware and an attacker in that space has access to privileged instructions that may change interrupts, page table structures, page table permissions, or privileged data structures. KCoFI [123] introduces a first CFI policy for commodity operating systems and considers these specific problems. The CFI mechanism is fairly coarse-grained: any indirect function call may target any valid functions and returns may target any call site (instead of executable bytes). Xinyang Ge et al. [135] introduce a precise CFI policy inference mechanism by leveraging common function pointer usage patterns in kernel code (SAP.F.4b on the forward edge and SAP.B.1 on the backward edge).

Protecting Just-in-time Compiled Code. Like other defenses, it is important that CFI is deployed comprehensively since adversaries only have to find a single unprotected indirect branch to compromise the entire process. Some applications contain just-in-time, JIT, compilers that dynamically emit machine code for managed languages such as Java and JavaScript. Niu et al. [127] presented RockJIT, a CFI mechanism that specifically targets the additional attack surface exposed by JIT compilers. RockJIT faces two challenges unique to dynamically-generated code: (i) the code heap used by JIT compilers is usually simultaneously writable and executable to allow important optimizations such as inline caching [154] and on-stack replacement, (ii) computing the control-flow graphs for dynamic languages during execution without imposing substantial performance overheads. RockJIT solves the first challenge by replacing the original heap with a shadow code heap which is readable and writable but not executable and by introducing a sandboxed code heap which is readable and executable, but not writable. To avoid increased memory consumption, RockJIT maps the sandboxed code heap and the shadow heap to the same physical memory pages with different permissions. RockJIT addresses the second challenge by both (i) modifying the JIT compiler to emit meta-data about indirect branches in the generated code and (ii) enforcing a coarse-grained CFI policy on JITed code which avoids the need for static analysis. The authors argue that a less precise CFI policy for JITed code is acceptable as long as both (i) the host application is protected by a more precise policy and (ii) JIT-compiled code prevents adversaries from making system calls. In the Edge browser, Microsoft has updated the JIT compilers for JavaScript and Flash to instrument generated calls and to inform CFGuard of new control-flow targets through calls to `SetProcessValidCallTargets` [155, 156, 157].

Protecting Interpreters. Control-flow integrity for interpreters faces similar challenges as just-in-time compilers. Interpreters are widely deployed, e.g., two major web browsers, Internet Explorer and Safari, rely on mixed-mode execution models that interpret code until it becomes “hot” enough for just-in-time compilation [158], and some Desktop software, too, is interpreted, e.g., Dropbox’s client is implemented in Python. We have already described the “worst-case” interpreters pose to CFI from a security perspective: even if the interpreter’s code is protected by CFI, its actual functionality is determined by a program in data memory. This separation has two important implications: (i) static analysis for an interpreter dispatch routine will result in an over-approximation, and (ii) it enables non-control data attacks through manipulating program source code in writeable data memory prior to JIT compilation.

Interpreters are inherently dynamic, which on the one hand means, CFI for interpreters could rely on precise dynamic points-to information, but on the other hand also indicates problems to build a complete control-flow graph for such programs. Dynamically executing strings as code (`eval`) further complicates this. Any CFI mechanism for interpreters needs to address this challenge.

Protecting Method Dispatch in Object-Oriented Languages. In C/C++ method calls use vtables, which contain addresses to methods, to dynamically bind methods according to the dynamic type of an object. This mechanism is, however, not the only possible way to implement dynamic binding. Predating C++, for example, is Smalltalk-style method dispatch, which influenced the method dispatch mechanisms in other languages, such as Objective-C and JavaScript. In Smalltalk, all method calls are resolved using a dedicated function called `send`. This `send` function takes two parameters: (i) the object (also called the receiver of the method call), and (ii) the method name. Using these parameters, the `send` method determines, at call-time, which method to actually invoke. In general, the determination of which methods are eligible call targets, and which methods cannot be invoked for certain objects and classes cannot be computed statically. Moreover, since objects and classes are both data, manipulation of data to hijack control-flow suffices to influence the method dispatch for malicious intent. While Pewny and Holz [120] propose a mechanism for Objective-C send-like dispatch, the generalisation to Smalltalk-style dispatch remains unsolved.

4.5 Conclusions

Control-flow integrity substantially raises the bar against attacks that exploit memory corruption vulnerabilities to execute arbitrary code. In the decade since its inception, researchers have made major advances and explored a great number of materially different mechanisms and implementation choices. Comparing and evaluating these mechanisms is non-trivial and most authors only provide ad-hoc security and performance evaluations. A prerequisite to any systematic evaluation is a set of well-defined metrics. In this paper, we have proposed metrics to qualitatively (based on the underlying analysis) and quantitatively (based on a practical evaluation) assess the security benefits of a representative sample of CFI mechanisms. Additionally, we have evaluated the performance trade-offs and have surveyed cross-cutting concerns and their impacts on the applicability of CFI.

Our systematization serves as an entry point and guide to the now voluminous and diverse literature on control-flow integrity. Most importantly, we capture the current state of the art in terms of precision and performance. We report large variations in the forward and backward edge precision for the evaluated mechanisms with corresponding performance overhead: higher precision results in (slightly) higher performance overhead.

We hope that our unified nomenclature will gradually displace the ill-defined qualitative distinction between “fine-grained” and “coarse-grained” labels that authors apply inconsistently across publications. Our metrics provide the necessary guidance and data to compare CFI implementations in a more nuanced way. This helps software developers and compiler writers gain appreciation for the performance/security trade-off between different CFI mechanisms. For the security community, this work provides a map of what has been done, and highlights fertile grounds for future research. Beyond metrics, our unified nomenclature allows clear distinctions of mechanisms. These metrics, if adopted, are useful to evaluate and describe future improvements to CFI.

5 CFIXX: OBJECT TYPE INTEGRITY FOR C++

Web browsers are among the most commonly used and most complex applications running on today's systems. Browsers are responsible for correctly processing arbitrary input in the form of web pages, including large and complex web applications with a threat model that includes an adversary with (restricted) code execution through JavaScript. Consequently, browsers are some of the most commonly attacked programs. The major web browsers (Firefox, Chrome, Safari, and Internet Explorer/Edge) are written primarily in C++, which does not enforce type safety and is prone to use after free (UaF) and other memory safety errors. Attackers now increasingly use type safety and UaF vulnerabilities to hijack the program's control flow, frequently by redirecting dynamic dispatch [22]. This is a form of code-reuse, and indeed Return Oriented Programming (ROP) [5, 6] attacks have been specialized for C++ [3]. To the best of our knowledge, no existing work fully explores how vulnerable C++'s dynamic dispatch is to these attacks, or is capable of stopping the full spectrum of attacks on dynamic dispatch.

C++ uses dynamic dispatch to implement polymorphism. A key feature of polymorphism is that objects which inherit from a base class can be up cast to that base class, while still retaining their own implementations of any *virtual* methods. Put another way, an object's underlying type — the class it was allocated as — determines the behavior of dynamic dispatch. Consequently, one call site on a base class can have different behavior depending on the type of the object at runtime. Dynamic dispatch is used to determine which implementation of the virtual method should be invoked. C++ relies on virtual table pointers to implement dynamic dispatch. Virtual table pointers tie an object to its underlying type and thus determine the correct target for the dispatch. For dispatch, *Object Type Integrity* (OTI) requires that the correct object type is used (the one assigned by the constructor when the object was allocated). Dynamic dispatch leverages the virtual table pointer to identify the type of an object, OTI therefore requires integrity of the virtual table pointer.

Violations of OTI are possible because virtual table pointers are fundamentally required to be in writeable memory. Each polymorphic object (i.e., an object with virtual methods) needs a virtual table pointer, and can exist anywhere in memory. Consequently, the virtual table pointer has to be included in the memory representation of the object. As C++ has neither memory nor type safety, having virtual table pointers in the objects exposes them to corruption. Once virtual table pointers have been corrupted — and OTI violated — the attacker has successfully corrupted control flow information. The attacker then forces that object to be used for dynamic dispatch, hijacking the control flow of the program. Given control over the program's control flow, she can mount a code reuse attack, leading to arbitrary execution.

Observe that code reuse attacks operate in three stages: (i) an initial memory or type safety violation, (ii) corruption of control data, and (iii) the control-flow hijack [4]. The current state-of-the art for efficiently mitigating code-reuse attacks is Control-Flow Integrity (CFI) [11, 87, 89, 90], which mitigates the third step of the attack by limiting the control flow to paths that are valid in the program's control flow graph. CFI mechanisms specialized to deal with C++ dynamic dispatch [89, 90] leverage the class hierarchy when computing the set of valid targets for virtual dispatch sites, increasing precision. Despite the precision added by class hierarchy information, the target set can be large enough for the attacker to find sufficient gadgets to achieve her desired goals [3, 7, 25, 85]. Each overridden virtual function increases imprecision, as all implementations must be in the same target set. For large C++ applications, such as web browsers, these sets are surprisingly large in practice. See Section 5.2.3 for details.

CFI's weakness — over-approximate target sets — is not fundamental to the problem of preventing dynamic dispatch from being used for code-reuse attacks. At runtime there is only one correct target for any virtual call. The correct target is dictated by the *dynamic* type of the object used to make the virtual call. The alias analysis problem prevents static analysis from determining the dynamic type of the object at the virtual call site. Indeed, any case where static analysis can determine the exact type, as opposed to a set of types, should be devirtualized as an optimization. Consequently, a security mechanism must leverage runtime information to correctly track an object's type and secure dynamic dispatch on that object, as opposed to the static, compile time information that CFI policies rely on.

We propose a new defense policy, Object Type Integrity (OTI) which guarantees that an object's type cannot be modified by an adversary, i.e., integrity protecting it, thereby guaranteeing the correctness of dynamic dispatch on that object. OTI thus mitigates the second step of a code-reuse attack by preventing key application control flow data from being corrupted. OTI tracks the assigned type for every object at runtime. Consequently, when the object's type is used for a dynamic dispatch, OTI can verify that the type is uncorrupted. Further, OTI requires that each object has a known type, thus preventing the attacker from injecting objects [3], and using them for dynamic dispatch. OTI distinguishes itself from CFI by intervening one stage earlier in the attack, and by being fully precise, instead of relying on target sets. The two can be deployed together, as discussed in Section 5.2.5, achieving even greater security, as they mitigate different stages of code-reuse attacks that utilize dynamic dispatch.

We present CFIXX, a C++ defense mechanism that ensures the integrity of virtual table pointers, thereby enforcing OTI. CFIXX is a practical, deployable defense that removes an entire class of control-flow hijacking targets. On Chromium, it has negligible overhead on JavaScript benchmarks (2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream), unnoticeable to users. For this low cost, it guarantees that every object used for dynamic dispatch and casts has the correct type, removing a significant attack surface for C++ applications. On the CPU bound SPEC CPU2006 benchmarks, CFIXX has 4.98% overhead. This is slightly higher than the best CFI mechanisms, which is to be expected since CFIXX intervenes at runtime, and provides greater security. CFIXX has been used to recompile libc++, and so protects all of user space, leaving no vulnerabilities due to unprotected code. Further, CFIXX shows how to efficiently use the new Intel Memory Protection Extensions (MPX) to integrity protect arbitrary regions of memory, and applies this technique to virtual table pointers.

To support further development and replication of our results, our prototype OTI enforcement mechanism, CFIXX, is open source: `https://github.com/HexHive/CFIXX`.

Our contributions are:

1. A new defense policy, Object Type Integrity (OTI), which mitigates all known attacks on dynamic dispatch.
2. A defense mechanism that enforces OTI, CFIXX, and an evaluation of the prototype on SPEC CPU2006 — including libc++, and Chromium.
3. A demonstration of how to efficiently use MPX to integrity protect arbitrary regions of memory.

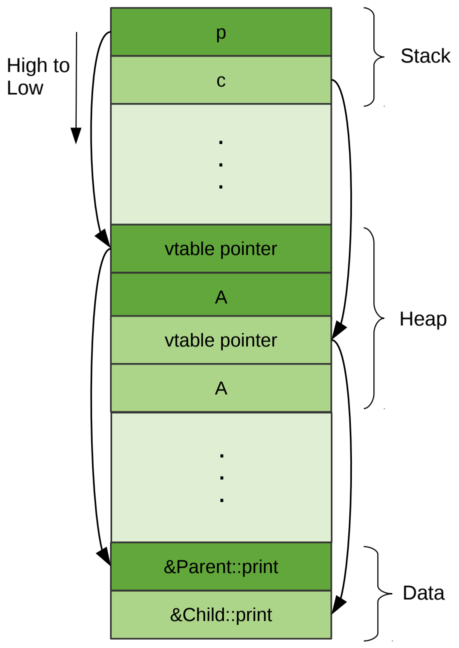
5.1 C++ Dynamic Dispatch

Dynamic dispatch is a key part of polymorphism in C++, allowing classes to *override* implementations of *virtual* functions that they inherit. Figure 5.1 contains a simple code example illustrating virtual functions, overriding implementations, and the associated memory layout. C++ implements dynamic dispatch by maintaining a mapping from each object to its underlying type, and thus the true implementation of the object's virtual functions. At each virtual call site, two things occur: (i) the appropriate function is determined from the object's virtual table, and (ii) an indirect call to that function is made. The correctness of dynamic dispatch thus depends on the integrity of the mapping from an object to its underlying type, i.e., Object Type Integrity.

The object type mapping at the core of dynamic dispatch is implemented by creating a *virtual table* for each polymorphic class. The virtual table consists of function pointers to the correct implementation of each of the class's virtual functions. The compiler populates the virtual tables with the correct function pointers, and is responsible for managing the virtual function name to virtual table index mapping. The virtual tables are fixed at compile time, and mapped read only at runtime. Each object of a class is given a *virtual table pointer* which points to the virtual table for the class. Consequently, the virtual table assigned to an

```cpp
#include <iostream>

class Parent {
  public:
    int A = 0;
    virtual void print(void) {
      cout << "Parent" << endl;
    }
};

class Child : public Parent {
  public:
    void print(void) override {
      cout << "Child" << endl;
    }
};

void virtualDispatch(Parent *p)
{
  p->print();
}

int main(void){
  Parent *p = new Parent();
  Child *c = new Child();

  virtualDispatch(p);
  virtualDispatch(c);
}
```

(a) C++ polymorphic function



(b) Memory layout

Figure 5.1.: Dynamic dispatch illustrated.

object can be thought of as encoding the correct dynamic class of the object. An assignment to the virtual table pointer is added by the compiler in each of the class's constructors.

Virtual function calls are compiled to look ups in the object's virtual table. The virtual table is located by using the virtual table pointer, and the function pointer is retrieved (recall that the compiler manages the virtual function name to index mapping). This function pointer is then called, completing the dispatch.

Figure 5.1 illustrates how this process works. Figure 5.1a shows a small C++ program with two classes, `Parent` and `Child`. Both implement a virtual function, `print()`, which prints out the class name. The memory layout for an object of each class is demonstrated in Figure 5.1b. Note that the virtual table pointer is the first field in the object, as required by the ABI, and is initialized by the object's constructor. To see how the virtual table pointer and virtual tables are used for dynamic dispatch, consider the virtual call at line 20 in Figure 5.1a. The `print()` function is the only virtual function, and so is at index zero in the virtual table. To find the correct function pointer, indirection through the virtual table pointer is used. This compiles to: `p->vtp[0]()` for `print()`. Regardless of whether `p` is a `Parent` (line 27) or `Child` (line 28) object, this lookup finds the correct implementation of `print()`.

The description of dynamic dispatch above holds for single inheritance. C++ allows multiple inheritance, which complicates the implementation of dynamic dispatch, however the same concepts hold. Multiple inheritance can result in objects having multiple virtual tables. CFIXX handles multiple inheritance, see Section 5.3.2 for details.

5.1.1 Constructors and Destructors

In C++, when a new object is created, a special method known as a *Constructor* is called. This method initializes the object, including assigning its virtual table pointer, or pointers in the case of multiple inheritance. When a class is part of an inheritance chain, the constructor for the base class is called first, and then the constructor for its descendant, this continues recursively until the class of the object is reached. These objects are all created at the same memory location, with the result that the virtual table pointer is overwritten for each class in the inheritance chain (for use by the intermediate class's constructor). Analogously, *Destructors* are called when an object is destroyed. The only difference is that the inheritance chain is traversed in the opposite order (from the object's destructor to the base class's destructor). Multiple inheritance complicates the class traversal for constructors and destructors, the details are again ABI specific and not needed here.

5.2 Object Type Integrity

Object Type Integrity (OTI) is a new security policy which focuses on C++ objects and, at its core, prevents an attacker from changing an object's type, or creating new, synthetic, object types. By protecting objects' types, OTI prevents attackers from hijacking the application's control flow during dynamic dispatch. OTI and Control-Flow Integrity (CFI) thus share a goal — mitigating control-flow hijacking and thus code-reuse attacks. However, OTI's approach is orthogonal, and complementary to, CFI policies as it intervenes earlier in the attack, integrity protecting object types instead of limiting control flow transfers. Additionally, by guaranteeing an object's type, OTI can also protect dynamic casts, which CFI is unable to do. To understand the security added by OTI, we provide background on code-reuse attacks in C++, and define both an attacker model and what is meant by control and data attacks. With these definitions, we discuss the limitations of CFI as a policy in securing forward control flow transfers (e.g. virtual calls), and how OTI mitigates these vulnerabilities. In particular, OTI restricts each forward control flow transfer to one target at runtime, instead of a set of targets. As OTI and CFI are orthogonal, they can be deployed together. We show that using both OTI and CFI together provides significantly stronger security than using either in isolation.

5.2.1 Code-Reuse Attacks

Code-reuse attacks are designed to bypass deployed defenses: Data Execution Prevention (DEP) [9], stack canaries [8], and Address Space Layout Randomization (ASLR) [10, 69]. DEP prevents code injection, so code-reuse attacks instead leverage existing executable code. Randomization techniques such as ASLR are bypassed by leaking information or using side-channels [72, 76]. Stack canaries [8] prevent sequential overwrites of return addresses, but can be bypassed by corrupting a pointer, and then using the corrupted pointer to write to arbitrary locations.

Code-reuse attacks operate in three steps: (i) an initial memory corruption, (ii) corrupting control data, and (iii) using the corrupted value to hijack a program's control flow. As C++ enforces neither memory nor type safety, any such violation can serve to perform the initial corruption. UaF vulnerabilities in C++ programs are increasingly popular with attackers, and are currently one of the most common form of memory safety violation for C++ applications [159]. Type safety violations can be used to attack a program's control flow through type confusion attacks, e.g., CVE-2017-2095, CVE-2017-2415, CVE-2017-5023, and Counterfeit Object Oriented Programming (COOP) [3]. Type confusion attacks cast an object to an illegal type where the underlying object is often of different size. This allows the adversary to access unintended memory and, e.g., overwrite virtual table pointers. This is typically done through an attacker forced illegal downcast (e.g., to a sibling class). A more advanced form of type safety violation are COOP attacks. COOP creates (interleaved) objects that use attacker chosen (including synthetic) virtual tables, and thus have invalid types. COOP uses these invalidly typed objects to achieve arbitrary execution. Both type confusion attacks and COOP violate OTI by assigning an object a different type after allocation, or creating a synthetic type.

C++ supports dynamic casts, which execute a runtime check to test if the cast object is compatible with the target type. Dynamic casts leverage C++'s Run Time Type Information (RTTI) to verify that a cast between two classes is valid at runtime. RTTI is associated with virtual tables in C++. Consequently, dynamic casts rely on objects having the correct virtual table pointer to access valid RTTI and correctly verify the type casts. Type confusion attacks either use an object of a different type at a static cast site or an object with a modified vtable pointer at a dynamic cast site. In both cases a cast that should fail completes and an object of the wrong type is used, enabling the type confusion attack.

5.2.2 Attacker Model

We assume a strong attacker with arbitrary read and write capabilities. The attacker cannot modify or inject code due to DEP [9]. Her arbitrary write capability can be used to perform control or data attacks. Control attacks are a subset of data attacks, where the data being overwritten directly determines control flow, such as virtual table pointers. A gray area exists between control and data attacks where data (such as `this` pointers) that points to control flow data is modified. Attacks against data that points to control flow data (at any level of indirection) are in scope. Other data attacks are out of scope.

The attacker's goal is a code-reuse attack, which requires hijacking control flow. To hijack control flow, the attacker must corrupt control data, or change the control data that is used. There are two fundamental types of code pointers that are writeable at run time — and thus vulnerable to corruption — in C++ programs: (i) return addresses and (ii) virtual table pointersṄote that backwards control flow transfers (e.g., returns) can be secured by shadows stacks or CET. These defenses can be deployed together with OTI and CFI. Consequently, we only consider attacks which target forward edges, meaning dynamic dispatch through virtual calls for C++ objects. There are four avenues of attack on C++ virtual calls: modifying an existing virtual table, injecting a new virtual table, substituting an existing virtual table, or creating a fake object. Current compilers map virtual tables to read only memory, preventing attackers from modifying existing virtual tables. Injecting a new virtual table is possible, as is substituting an existing virtual table. The final step of either attack is to overwrite the virtual table pointer of an existing object with a pointer to the attacker chosen virtual table. COOP [3] relies on creating "fake" objects, which do not have constructors in the program's source, and so have no official type — virtual table pointer — assigned. All three attacks center on corrupting an object's type via its virtual table pointer, and are included in our attacker model.

A more subtle form of attack involves substituting one valid object for another at a virtual call site. This is accomplished by, e.g., overwriting object pointers in memory. Such attacks that use indirection (by modifying a pointer to a code pointer, instead of the code pointer directly) are also considered in our model.

5.2.3 Limitations of CFI – Web Browser Case Study

The CFI policy operates in two phases: analysis and enforcement. At compile time, the analysis phase determines the allowed set of targets for each virtual call site. Since this set is determined *statically*, the dynamic type of the underlying object is unknown. Consequently, the set must include, at a minimum, the implementation of the virtual function in the static class of the object, and any implementation in a descendant class that overrides it. Each such override provides a target that an attacker can divert control flow to without violating the security policy. More coarse grained defenses that do not leverage C++ class hierarchies rely on matching function prototypes (with various degrees of precision), resulting in even larger target sets. Function prototype analysis is also used for C style function pointers. In principle, these sets could be further limited by control and/or data flow analysis. The enforcement component performs a set check at runtime, to verify that the target of the virtual call is in the allowed set. Consequently, even though the check is performed dynamically, it is relying on *static* information — the compile time target sets — for its security properties.

To gain a sense of just how susceptible browsers are to attacks within the minimum allowed target sets for CFI, we quantify the size of minimum target set for virtual calls in Chromium and Firefox. To do so, for each virtual function we count the number of overriding methods. With this information, and the static class of the object used, we can determine the size of the target set at each virtual call site.

In Chromium, we found a total of 13,834 virtual functions, nearly half of which (6,671) have more than one implementation. Out of the 4,679 virtual functions in Firefox, 1,867 (40%) have more than one implementation. 5,828 of the virtual calls in Chromium have more than one call site (including
`blink::ExceptionState::rethrowV8Exception` which has 1,559 call sites), and 2,188 virtual functions in Firefox have more than one call site. Taking this data in its totality, it is clear that modern browsers have a large remaining exploitable surface that must be protected even if CFI is deployed. Table 5.1 shows the number of virtual functions in Chrome and Firefox with a given number of implementations. The "Max" row is the maximum number of implementations for any virtual function. Each of these virtual functions can potentially be abused without violating the CFI policy, highlighting the severity of the problem.

The size of the target sets represents a genuine security risk in light of the prevalence of virtual calls. We instrumented Chromium to count the prevalence of indirect calls through function pointers and virtual calls. Under a normal browsing workload, 42% of indirect calls calls were virtual. This is in line with the results reported by VTrust [89], which found 41% of indirect calls were virtual for Firefox. Of these indirect but non virtual calls, over 50% are due to Chrome's memory management wrapper. Chrome's wrapper intercepts,

Table 5.1.: Number and percent of virtual functions with multiple implementations. Max is the maximum number of implementations for any virtual function.

Impl Count	Chromium	Firefox
2	3,260 (23.57%)	1,065 (22.76%)
3	1,556 (11.25%)	400 (8.55%)
4	723 (5.23%)	143 (3.06%)
5	705 (5.10%)	88 (1.88%)
$[6-9]$	349 (2.52%)	124 (2.65%)
$[10-19]$	63 (0.46%)	45 (0.96%)
≥ 20	15 (0.11%)	2 (0.04%)
Max	78	107

e.g., `malloc, operator new`, dispatching them to its chosen allocator, tcmalloc for Linux[1].

5.2.4 OTI Policy

Object Type Integrity guarantees that each C++ object has the correct type, e.g., virtual table pointer, and further that this type was assigned by a valid constructor, thereby preventing "fake" objects. Guaranteeing that every object has the correct type in turn guarantees the correctness of C++ dynamic dispatch and cast on the object (though not necessarily that the correct object is used, see Section 5.2.5). Every virtual call or dynamic cast site has only one *correct* object type at runtime, determined by the dynamic type of the object, determined through the virtual table associated with the object.

OTI enforces that the correct virtual pointer be used for each object during dynamic dispatch and cast, thereby ensuring that the correct virtual table — and thus object type — is used. Further, in order to establish the correct type for each object, OTI requires that every object's type be assigned by compiler generated code in the object's constructor. OTI can be thought of as a fully context and flow sensitive analysis, that takes advantage of runtime information to perfectly resolve the correct virtual table for every object.

[1]See `https://chromium.googlesource.com/chromium/src/base/+/master/allocator/` for the implementation of Chromium's memory management wrapper.

OTI mitigates all classes of attacks that can be used to subvert forward control flow transfers for any given object, and all type confusion attacks that target dynamic casts. UaF attacks can be mitigated by invalidating an object's virtual table pointer when its memory is deallocated. This provides only partial safety, as reallocating the memory can assign a new, legitimate virtual table pointer, but it does prevent dynamic dispatch on a free'd object before the memory is re-allocated. Type confusion attacks, or other memory corruptions, that attack control flow by overwriting function / virtual table pointers, are naturally stopped by OTI. Attacks that manufacture "fake" objects, such as Counterfeit Object Oriented Programming, are prevented. These "fake" objects do not have recognized types under OTI. Consequently, our check at the call site fails due to lack of type information for the object, preventing the attack.

Guaranteeing OTI, i.e., the integrity of virtual table pointers, can be used to enforce different security policies. OTI always protects the integrity of virtual table pointers, removing adversary access to them. The OTI policy requires that any mechanism that implements it dispatch on the protected virtual table pointers. By requiring dispatch on protected pointers, the OTI policy already stops control-flow hijack attacks. Implementations can extend the OTI policy by using the virtual table pointer in attacker controllable memory as a canary. Such a policy extension would compare the canary to the protected virtual table pointer. If the canary and the protected pointer are different, a policy violation is reported.

5.2.5 Combined Security of OTI and CFI

OTI distinguishes itself from CFI by utilizing *runtime* information to fully protect dynamic dispatch from direct overwrites of virtual table pointers. Such overwrites correspond to control attacks in our attacker model. Recall, however, that data pointers can be used indirectly to change control flow by changing, e.g., which object gets used for dynamic dispatch. OTI on its own cannot mitigate such attacks, just as CFI on its own cannot fully mitigate direct attacks on control data. However, CFI can partially mitigate such indirect attacks on control data because it creates an absolute set of valid targets, thereby limiting the valid object substitutions that can be made. Consequently, OTI is best used complementary with CFI where, fully stopping attacks against code pointers earlier than CFI, and orthogonally leveraging (the potentially imprecise) CFI target sets for any remaining attacks that modify data pointers.

5.3 Design

CFIXX enforces OTI by identifying legitimate (compiler inserted) virtual table pointer assignments. These are recorded as the ground truth type of the objects. CFIXX then protects the integrity of these assignments, ensuring that an attacker cannot modify them. The integrity protection is accomplished by recording the original virtual table pointer assignment in a protected metadata table. Dynamic dispatch is then modified to lookup the virtual table pointer for the object in our metadata table instead of in the object itself. Dynamic casts could be protected in the same way, but are not part of the current prototype. CFIXX requires recompilation of the entire program, including libraries (though a compatibility mode with weaker security guarantees is discussed in Section 5.5). By protecting all of user-space, CFIXX is guaranteed to see the ground truth type for every legitimate object. Consequently, dynamic dispatch on objects without metadata is detected, and the OTI requirement that every object's type be assigned by compiler generated code in a constructor is enforced.

Enforcing OTI, and guaranteeing the integrity of dynamic dispatch per object only requires protecting a small subset of a program's data. Dynamic dispatch only occurs for virtual methods of polymorphic objects, meaning CFIXX only needs to protect the virtual tables and virtual table pointers of these objects. Virtual tables are mapped to read only memory by current compilers. Consequently, the integrity of the virtual table's themselves is already guaranteed. This leaves only the virtual table pointers in polymorphic objects that need to be integrity protected by CFIXX.

To see how OTI secures dynamic dispatch, consider dynamic dispatch from an attacker's perspective. She wants to change the target of a virtual call, and has an arbitrary read and write per our attacker model. She cannot modify a virtual table, because they are mapped read only. Consequently, she must either change a virtual table pointer in an existing object, or create a “fake” object with a virtual table pointer of her choice. OTI prevents her from executing either attack. She cannot change an existing virtual table pointer, because that would change the object’s type, violating OTI. Any object she injects will not have a ground truth type, and thus also violates OTI. A subtle attacker might change the object used at a virtual call site through data-only attacks (i.e., changing object pointers to alternate objects). The effects of such data-only attacks can be limited through CFI.

5.3.1 Integrity for Virtual Table Pointers

Guaranteeing OTI is accomplished by protecting the integrity of virtual table pointers, which fundamentally requires allowing only *legitimate* writes to virtual table pointers. Determining what is and is not a legitimate write is a difficult problem in general due to aliasing. However, the problem is simplified for virtual table pointers because only constructors can *legitimately* write virtual table pointers in C++. Any other write is invalid. OTI is guaranteed if a legitimate write of a virtual table pointer is the reaching definition at a virtual call site. Full memory safety would accomplish this, however it is a strictly stronger property than is required. Instead, CFIXX guarantees that only legitimately written virtual table pointers can be used for dynamic dispatch.

CFIXX guarantees virtual table pointer integrity through a two part enforcement mechanism which utilizes a metadata table: (i) recording the correct virtual table pointer in the metadata table, and (ii) using our recorded virtual table pointer for dynamic dispatch on the object. The metadata table is conceptually a key-value store. The object, identified by the `this` pointer serves as the key. The value stored is the correct virtual table pointer for the object. For (i), we note that the initial virtual table pointer assignment to the object cannot be tampered with under DEP as it is in compiler emitted code, and the virtual table pointer is a fixed constant. Consequently, we record the virtual table pointer assigned to the object in our metadata table during this initial assignment. Part (ii) is achieved by looking up the correct virtual table pointer for the object in our metadata table during dynamic dispatch, instead of using the (possibly attacker corrupted) virtual table pointer in the object.

By default, our protection mechanism requires that every object used for a virtual call has an entry in the metadata table. If there is no metadata entry, an attack is detected and execution is terminated. Only objects whose constructors CFIXX did not compile do not have metadata table entries. CFIXX can miss constructors for three reasons: deliberately unprotected code module (e.g., third party shared library), an object created outside of C++ semantics (see Section 5.5), or an attacker injected object (such as "fake" objects created by COOP). Differentiating between an object from an unprotected module and an attacker injected object is a key challenge for maintaining soundness while still supporting non-protected libraries.

A "compatibility mode" that addresses the challenges of executing with third-party libraries is described in Section 5.5. CFIXX supports separate compilation and shared libraries. Consequently, there is no technical reason to require a compatibility mode. However, its availability does make it possible for CFIXX to be incrementally deployed, hopefully encouraging its adoption. The rest of this paper assumes that all code is protected by CFIXX.

5.3.2 Multiple Inheritance

C++ supports multiple inheritance by allowing objects to have multiple virtual tables. This complicates our metadata scheme, as we can no longer use the `this` pointer (the pointer to the base of the object) as our metadata key, as an object can now require multiple metadata entries. Instead, we use the location of each virtual table pointer in the object as the key. Using the virtual table pointer's address as the key allows multiple values per object in the metadata table, thereby supporting multiple virtual tables and multiple inheritance. CFIXX is ABI agnostic, and relies on the compiler to determine which virtual table for a particular object should be used at a given call-site. Once the compiler has made this decision, CFIXX uses the integrity protected virtual table pointer corresponding to the location in the object.

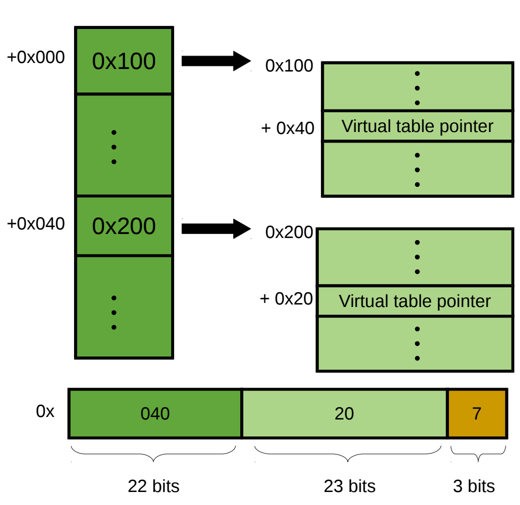
5.4 Implementation

CFIXX is implemented on top of LLVM 3.9.1, and has three components. The first part is creating, and protecting, the data structure for our metadata. Secondly, we have to create metadata entries each time the compiler assigns a virtual table to a polymorphic object. These compiler generated assignments are the set of all *valid* writes to an object's virtual table pointer. Finally, we alter the dynamic dispatch mechanism to leverage our recorded virtual table pointer for an object, instead of the virtual table pointer contained in the object. The current prototype of CFIXX supports 64-bit x86_64 systems with the Itanium ABI.

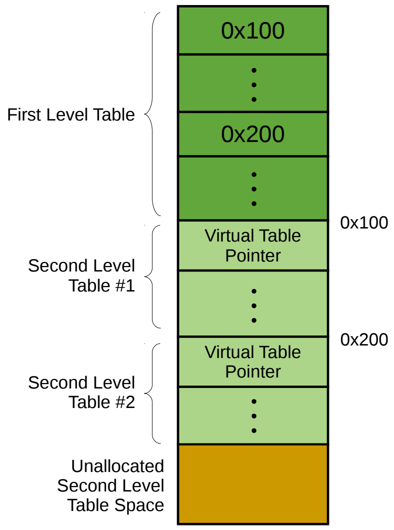
5.4.1 Metadata Data Structure

Our metadata is stored in a two level lookup table, see Figure 5.2a. To find the correct entry, the pointer is split into two parts. The high order bits are the index in the top level table, `040` in the figure. Entries in the top level table are pointers to the second level tables. The low order bits, `20` in the example, are the index in the second level table, which stores the correct virtual table pointer for the object.

Our decision to use a two level lookup table is based on the allocation patterns of polymorphic objects. We found relatively little entropy in the high order bits of object addresses, and insufficient entropy in the middle bits to justify a three level table (see Section 5.6.2). Consequently, we use a two level table, which requires one less indirection and so has better performance without increasing the size of our metadata table. While SPEC CPU2006 was used for our design study, the results reflect that objects are either heap or stack allocated and tend to be grouped in memory. The same metadata table structure was used for Chromium, and is not "tuned" for SPEC CPU2006. x86_64 uses 48 bits for virtual addresses. Of these, we chose to use the high order 22 bits for the top level table, and the next 23 bits for the



(a) Logical structure of metadata



(b) CFIXX Metadata memory layout

Figure 5.2.: Metadata design illustrated. Figure 5.2a shows how a pointer is broken into two indexes into the top level and second level table. The entry for the top level table points to the second level table. Figure 5.2b shows how all the tables are laid out in the protected data region. The top level table is allocated first, then all second level tables are allocated on demand.

second level tables. All polymorphic objects are at least eight bytes large (for the virtual table pointer), so we can disregard the low order 3 bits.

Our metadata tables are memory mapped, and large enough for the top level table, and a fixed number of second level tables, as shown in Figure 5.2b. Note that mapped pages are not touched (and thus actually allocated by the OS) until they are needed, so we only pay the memory overhead for touched areas of instantiated tables. Contiguously allocating the metadata tables allows us to only integrity protect one memory region. The location of this region can be determined either at runtime, allowing maximum flexibility for integration with randomization defenses such as ASLR, or at a fixed location, a performance optimization that removes an indirection on each metadata access to lookup the table's location.

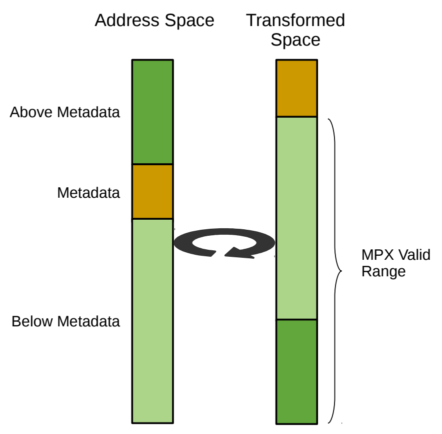


Figure 5.3.: Address space rotation for MPX checks

5.4.2 Metadata Protection

Our attacker model assumes a strong attacker who can perform arbitrary reads and writes. Consequently, we must protect our metadata from modification by an adversary. The lowest overhead option is information hiding through randomization. However, this approach is comparatively easy to defeat [72, 76]. Further, it has a performance cost because it requires an additional redirection at runtime to find the metadata table. This leaves three general approaches: (i) masking, (ii) using MPX to prevent unauthorized writes to our metadata, or (iii) leveraging Intel's forthcoming Memory Protection Keys (PKEYS) to make the metadata read only except when we need to write it (doing this via `mprotect` is prohibitively expensive). Masking is well known from software-based fault isolation, and simply masks each pointer before it is dereferenced, e.g., through an `and` instruction.

The MPX hardware extension is designed to provide hardware support for verifying that a pointer is in bounds for memory object accesses. To do so, MPX introduces four new 128 bit registers for storing upper and lower bounds (each register stores two 64 bit pointers — one for the upper bound and one for the lower bound). Additional instructions are provided to make bounds, load and store to the bounds registers, and perform upper and lower bounds checks.

We leverage MPX bounds checks to prevent unauthorized writes to our metadata, by dividing memory writes into two categories: those added by CFIXX to maintain the metadata, and all other writes. The first category does not require bounds checks. For an attacker to use such writes maliciously, she would have to have already broken CFIXX, or hijacked control through an orthogonal attack. CFIXX adds MPX bounds checks to the second category — all writes that do not touch metadata. Note that reads do not need be checked as CFIXX only requires integrity. There is no need to keep the virtual table pointers confidential. MPX checks ensure that a memory address is within a given range. There is no way to check that an address is not in a given range, or that either the upper or lower bounds check passes, but not both. Consequently, one set of bounds is needed for all non-metadata writes, e.g, all legal memory for the writes needs to be treated as a single, contiguous array. However, our metadata table is mapped somewhere in the middle of the address space, leaving a valid region both below and above our table.

CFIXX uses a novel technique to adapt MPX to perform the required check, illustrated in Figure 5.3. Our exclusive checks are performed on a rotated version of the address space. In our rotation, the metadata table is at the top of the address space. This allows us to treat everything below it as the valid range. In the rotated address space, the last valid byte of our metadata table is at $2^{64} - 1$. Consequently, the rotation can be accomplished by adding the appropriate offset to a given address. Addresses that are above the metadata table in the normal address space naturally wrap around to the bottom due to arithmetic overflow. The translation is implemented as a `lea, mov, lea`. The first `lea` calculates the address that is being written to, the `mov` loads the offset, and the second `lea` applies it. Note that `lea` is side effect free and does not effect the flag register. CFIXX uses one MPX bounds register, with the bounds set to `0x0` and the rotated base of our metadata table. Consequently, after rotation CFIXX performs an MPX upper bound check `bndcu`. This works because the base of our metadata table is the upper limit of valid addresses in the translated address space.

Our rotation offers two key advantages: (i) the protected region can be anywhere in the address space — unlike for masking, and (ii) it requires half as many bounds checks as a naïve MPX implementation. Masking requires the metadata table to be at the top of the address space. To see why, consider what happens when high order, non-masked bits are changed. For each combination of such bits, a hole is created by the mask for the protected, lower order bits in the address. Consequently, many disparate regions of memory would be protected. Applying MPX to integrity protect a memory region without our rotation is possible. However, MPX only provides inclusive checks, i.e., that an address is in bounds. To provide integrity, the opposite is desired, i.e., that a write is *not* in the protected region. This is not easily accomplished, as it requires checking if the address is in either the area below the protected region, and if that check fails, if it is in the area above the protected area. A violation must only be signalled if both checks fail. MPX does not support this gracefully. However, our rotation provides one valid region for any legal write, which naturally fits the MPX paradigm, and only requires one bounds check.

Intel has announced, but not yet shipped in production, a new hardware feature called Memory Protection Keys (PKEYS). These will allow a process to switch a page (or group of pages) between RW and R with one register write. This feature can naturally be used to protect our metadata. The pages would be set R anytime CFIXX is not performing a write to them. Since CFIXX is implemented in the compiler, it can ensure that only authorized locations use this feature. To maliciously use these locations, an attacker would have to already have diverted the program's control flow. As for MPX, this can only happen if CFIXX has already been bypassed — rendering it irrelevant, or the attacker is utilizing an orthogonal attack. Consequently, PKEYS can be used to protect the metadata. We anticipate that this will have significantly less overhead. Only metadata writes would be protected, instead of all writes. Further, the protection would only require two additional instructions per metadata write — to toggle the metadata to RW and back to R, instead of four instructions per data write (three to rotate the address space, and one to perform the bounds check). Metadata writes are dwarfed by the number of data writes, significantly reducing the number of checks required by the PKEYS protection scheme.

```
//Constructor
Foo::Foo(){
  this.vtablePointer = 0x200;
}
//Virtual Call
bar(Foo *f){
  f->vtablePointer[1]();
}
```

(a) Before CFIXX

```
//Constructor
Foo::Foo(){
  this.vtablePointer = 0x200;
  secondLevel = metadata[this>>26];
  if (secondLevel == NULL)
    secondLevel = allocate2ndlvl(this);
  secondLevel[(this>>3)&((1UL<<23)-1UL)] =
    this.vtablePointer;
}
//Virtual Call
bar(Foo *f){
  secondLevel = metadata[this>>26];
  vtablePointer = secondLevel[(this>>3)&((1UL<<23)-1UL)];
  vtablePointer[1]();
}
```

(b) After CFIXX

Figure 5.4.: Virtual table pointer initialization and dynamic dispatch before and after CFIXX

5.4.3 Runtime Instrumentation

With this metadata scheme in mind, the next questions are how and when metadata is created and referenced. Recall from Section 5.3 that there is only one valid write of an object's virtual table pointer. This is when we want to create metadata for the object. As described in Section 5.1, virtual table pointers are used during dynamic dispatch for virtual calls. CFIXX instruments the virtual calls to use the virtual table pointer from our metadata, as is illustrated with C pseudo code in Figure 5.4. Figure 5.4a shows the C equivalent of the original code generated by the compiler for an object constructor, and then virtual code. Figure 5.4b shows the same code after CFIXX's instrumentation.

Virtual tables are assigned to objects by the object's constructor. The compiler implicitly adds the virtual table pointer field and writes the correct value to it in the constructor. This holds for all constructors in C++11 (and the relevant subset for older versions of C++):

default, programmer specified, copy, move. Clang-3.9.1 has a common sub-routine (`CodeGenFunction::InitializeVTablePointer`) that initializes the virtual table pointer for all constructors. CFIXX modifies this sub-routine to also emit instructions to create metadata for the object. The virtual table pointer is still written into the object as well for backward compatibility with unprotected code and to make it possible for CFIXX to prevent *and* detect attacks.

CFIXX's current prototype supports virtual dispatch via the Itanium ABI. In clang-3.9.1, the Itanium ABI relies on `CodeGenFunction::GetVTablePtrCXX` to retrieve the virtual table pointer that is then used for dynamic dispatch. CFIXX reimplements this function to read the virtual table pointer from our metadata instead of the object. As the virtual tables are unchanged, no further changes to dynamic dispatch are required for it to succeed. The detection and logging modes mentioned in Section 5.3 are simple to implement, though not evaluated here. Implementing them requires adding a comparison of the virtual table pointer in the object to the virtual table pointer retrieved from memory. If they are different, an attack would be detected, or a violation logged. This extra `if` check, and potential policy action, would add overhead, and does not affect the security provided by CFIXX, and so was omitted here.

5.5 Discussion

Here we elaborate on possible extensions to CFIXX, and highlight some edge cases and low level implementation challenges such as metadata allocation, non-standard objects, leveraging run time type information, compatibility modes, and devirtualization.

Metadata Allocation CFIXX requires its metadata tables to be allocated before any object is allocated and has its constructor called. The most reliable way to enforce early metadata allocation is to add our metadata allocation function to the `.preinit-array` section of the binary. This `ELF` section is supported by the gold linker on both Linux and FreeBSD (as used in our evaluation).

Use After Free Detection The OTI policy can, in principle, provide UaF protection. However, the current CFIXX implementation does not invalidate metadata when objects are deallocated. There are a few challenges to doing so. First, there is no straightforward way to instrument destructors as polymorphic objects can have trivial destructors, which are omitted. Adding destructors would break the C++ ABI. Consequently, we would have to instrument deallocations directly. This means `free` for heap objects, and for stack objects we would have to handle stack frame deallocation explicitly.

Non-Standard Objects It is possible to create an object without calling its constructor. This can be accomplished by copying an existing object with, e.g., `memcpy()`. Such code violates the C++ standard[2], which requires using move or copy constructors, but does often work in practice. We found one instance of this behavior in Chromium, see Section 5.6.3. Such code should be refactored to use move or copy constructors, as appropriate. CFIXX does not support objects created in this non-standard way, requiring programmer intervention to refactor the code. Note that such behavior is rare, we only found one instance in total, namely in the large Chromium code base and we refactored the code, adding three lines (a call to our metadata create function, and a forward declaration of that function). An alternative solution would refactor the code to use a move constructor.

Run Time Type Information RTTI is used to validate dynamic casts at runtime. To do so, the compiler emits objects that encapsulate an object's type, and can be used to traverse the type hierarchy. The RTTI objects contain virtual pointers and are emitted at compile time. Since RTTI objects are statically created by the compiler, no constructor is ever called for them. To handle these objects, CFIXX adds a new function to the `ELF .ctor` array that creates metadata for every RTTI object.

Compatibility Mode To support non-protected libraries, CFIXX can be extended to provide a compatibility mode. In this mode, the virtual table pointer in the object's memory is used if we do not have a metadata entry for the object. To secure this mode against

[2]See, e.g., section 12.8 of the C++14 standard which specifies how objects can be copied or moved.

attacker injected objects, such as created by COOP attacks, this mode is only enabled for virtual call sites where the static type of the object is known to be defined in a third party library. This limits COOP attacks to creating synthetic objects from unprotected classes. As these classes are unprotected, this does not affect the soundness of CFIXX. Note that, if all code is protected through CFIXX then the compatibility mode is disabled, as done for our experiments.

Devirtualization An important optimization for C++ is devirtualization — replacing a virtual call with a direct call. As CFIXX changes the dynamic dispatch mechanism, we investigated the impact on devirtualization. Currently, enabling CFIXX causes about 10% more virtual calls in SPEC CPU2006. We are aware of no fundamental reason why CFIXX should prevent devirtualization and are working on making the optimization passes in the LLVM middle-end aware of CFIXX.

C Style Indirect Function Calls CFIXX can be adapted to protect C style indirect function calls. There are two challenges: rogue function pointer writes and designing invoke semantics for indirect function calls. First, writes to function pointers are more difficult to detect than writes of virtual table pointers. Additionally, function pointers can be assigned from other variables, leading to the alias analysis problem. Consequently, possible writes of a function pointer would have to be traced throughout the program to correctly maintain our metadata, like SoftBound [12] did for memory objects. Alternately, a different metadata scheme could be used. These difficulties arise because we can no longer rely on the semantics of move and copy constructors. The second challenge would be to refactor indirect function calls. Currently, no lookup is performed. This would have to be changed so that the correct function pointer is looked up in our metadata and then called. Designing an efficient system to address both these challenges is left as future work.

Metadata Protection There are alternatives to protecting the metadata beyond our current MPX implementation, and the forthcoming PKEYS. An alternate MPX implementation is possible, where the metadata table is allocated at the top of user space, eliminating the need

Table 5.2.: Exploits caught by vtable protection mechanisms. VTI is Virtual Table Interleaving. See Section 5.6.1 for explanations of exploit types. † inferred from published description.

Exploit Type	LLVM CFI	CFIXX	VTrust / VTI†	CPS†
FakeVT	✓	✓	✓	✓
FakeVT-sig	✓	✓	✓	✓
VTxchg	✓	✓	✓	✓
VTxchg-hier	✗	✓	✗	✓
COOP	✗	✓	✗	✗

to rotate the address space for checks. Allocating the metadata table at the top of user space would require moving the stack down. The principled way to move the stack down would be to change the fixed stack start address (to which ASLR then adds randomness) in the kernel, to guarantee that there would always be room above the stack. Moving the stack via a kernel modification would have no impact on the soundness of CFIXX, and would provide only a performance improvement. Further, it would make CFIXX more difficult to deploy. Consequently, we chose not to implement a kernel modification. Another alternative to using MPX would be storing the metadata in an SGX enclave. However, SGX provides much greater protection than we require. Consequently, we chose not to evaluate it for CFIXX.

5.6 Evaluation

CFIXX is evaluated along two dimensions: security and performance. We show that CFIXX stops more attacks than any existing control-flow hijacking mitigation. On the performance axis, we present our design study of metadata implementation alternatives, and our performance results both on SPEC CPU2006 (4.98%) and on Chromium, using JavaScript benchmarks (2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream). SPEC CPU2006 was run on a machine with a 3.40 GHz Intel Core i7-6700 CPU and 16GB of RAM, under Ubuntu 16.04. Chromium was run on a 4.0 GHz Intel Core i7-6700K with 32GB of RAM, under FreeBSD 11.

5.6.1 Exploit Coverage

We created a suite of C++ tests[3] demonstrating various possible scenarios of vtable pointer overwrites. The scenarios we tested for are the following, listed in order of mitigation difficulty:

1. *FakeVT:* Inject a fake virtual table with a pointer to a function that does not share the same signature as the function in the original virtual table.

2. *FakeVT-sig:* Inject a fake virtual table with a pointer to a function that shares the same function signature (same name, number of arguments, types of arguments, and order of arguments) as the function in the original virtual table.

3. *VTxchg:* Overwrite a virtual table pointer with a virtual table pointer of a class outside of the original class hierarchy (i.e., change `Child`'s vtable pointer with the vtable pointer of `Stranger`, and `Child` and `Stranger` are unrelated).

4. *VTxchg-hier:* Overwrite a virtual table pointer with the virtual table pointer of a class within the same class hierarchy (i.e., change `Child1`'s vtable pointer with `Child2`'s and both `Child1` and `Child2` are both direct children of `Parent`).

5. *COOP:* Create a fake object of a class (via, e.g., `malloc`) without calling the class' constructor, and call a virtual function of the fake object. This is the basic COOP attack.

Table 5.2 summarizes the results of running the code from our suite with LLVM CFI and CFIXX. We could not evaluate against the state-of-the-art in virtual table protection, VTrust [89] and Virtual Table Interleaving [90], because the source was not yet made available from the authors. Based on our analysis of the papers, we expect that VTrust and Virtual Table Interleaving would detect 1 – 3. They cannot detect 4 and 5 as a valid virtual table pointer is used in the wrong context. The VTrust paper discusses mitigating COOP, but assumes that the control-flow hijack attack violates the class hierarchy constrained

[3]Source available at `https://github.com/HexHive/CFIXX`.

set of valid functions. This assumption does not necessarily hold — particularly for web browsers which have large class hierarchies, as shown in Section 5.2.3. The latest Code Pointer Separation [65] code was not able to compile our code suite. However, based on the description in the paper, we expect that CPS mitigates exploit types 1 – 4, but would not protect against exploit type 5.

5.6.2 Metadata Design

An efficient implementation of the metadata table is a key component of CFIXX. The metadata table must support an object at every (8 byte aligned) virtual address. At the same time, CFIXX seeks to minimize memory overhead. The classic solution to this situation is a multilevel lookup up table, e.g., as used for virtual memory in the page table. We initially experimented with a three level lookup table. To examine how program's use memory, we logged the address of every polymorphic object created by xalancbmk and omnetpp from the SPEC CPU2006 benchmark suite. We then post processed the log to determine the metadata table that would be built under different lookup schemes.

Table 5.3.: Three level metadata design study. Level Sizes are in bits. Sum is 45 — the 48 bits used for virtual addresses less the low order 3 bits (minimum object size is 8 bytes). #(%) is the average number of entries (density of entries). Density is $\frac{entries}{2^{levelsize}}$

L1, L2, L3	L1 #(%)	L2 #(%)	L3 #(%)
16, 16, 13	2 (0.00%)	2,562 (3.91%)	648 (0.99%)
8, 16, 21	2 (0.78%)	14 (0.02%)	123,032 (0.73%)
8, 8, 29	2 (0.78%)	1 (0.39%)	1,660,932 (0.04%)

Table 5.4.: Two level metadata design study. Level Sizes are in bits. Sum is 45 — the 48 bits used for virtual addresses less the low order 3 bits (minimum object size is 8 bytes). #(%) is the average number of entries (density of entries). Density is $\frac{size}{2^{levelsize}}$

L1, L2	L1 #(%)	L2 #(%)
18, 27	2 (0.00%)	1,660,932 (0.15%)
20, 25	3 (0.00%)	1,107,288 (0.41%)
22, 23	8 (0.00%)	415,233 (0.62%)

Table 5.3 shows the results for a three level metadata table under different configurations. Note the extremely low usage rates for all levels. The top level (L1) only had two entries - corresponding to the stack and the heap. The middle level (L2) was also effectively empty. As each level of indirection on metadata reads and writes cost performance, we eliminated the intermediate level.

The remaining design question regards the size of the first and second levels in our two level lookup table. Our experiments on this, shown in Table 5.4, indicate that using the high order 22 bits is ideal. After 22 bits, density levels off and then declines, increasing the virtual address space footprint of our metadata.

When considering memory overhead for our scheme, it is important to remember that we are predominantly using virtual address space. The vast majority of the underlying pages in any table remains untouched, and so not actually allocated by the kernel. This minimizes our impact on physical memory usage and performance. In the worst case, if every single memory page of the application holds at least one virtual object, the memory of the application doubles. In practice, we measured a 79% increase in memory usage. It should in principle be possible to design a more efficient metadata scheme, as we only need to store eight bytes per polymorphic object.

5.6.3 Performance

We evaluated CFIXX on the SPEC CPU2006 compiler benchmarks, as well as on Chromium. The SPEC CPU2006 benchmarks are standard for evaluating compiler based security mechanisms, and included for comparison purposes. Chromium is the open source version of the popular Chrome browser. Web browsers are the most common network facing applications, and so the usability of Chromium with CFIXX is an important indicator of the ability of CFIXX to be deployed in practice. For this evaluation, we used a fixed metadata table location.

Figure 5.5 shows the performance of the C++ benchmarks in SPEC CPU2006 when compiled with CFIXX. CFIXX has 2.22% overhead without MPX protecting the metadata,

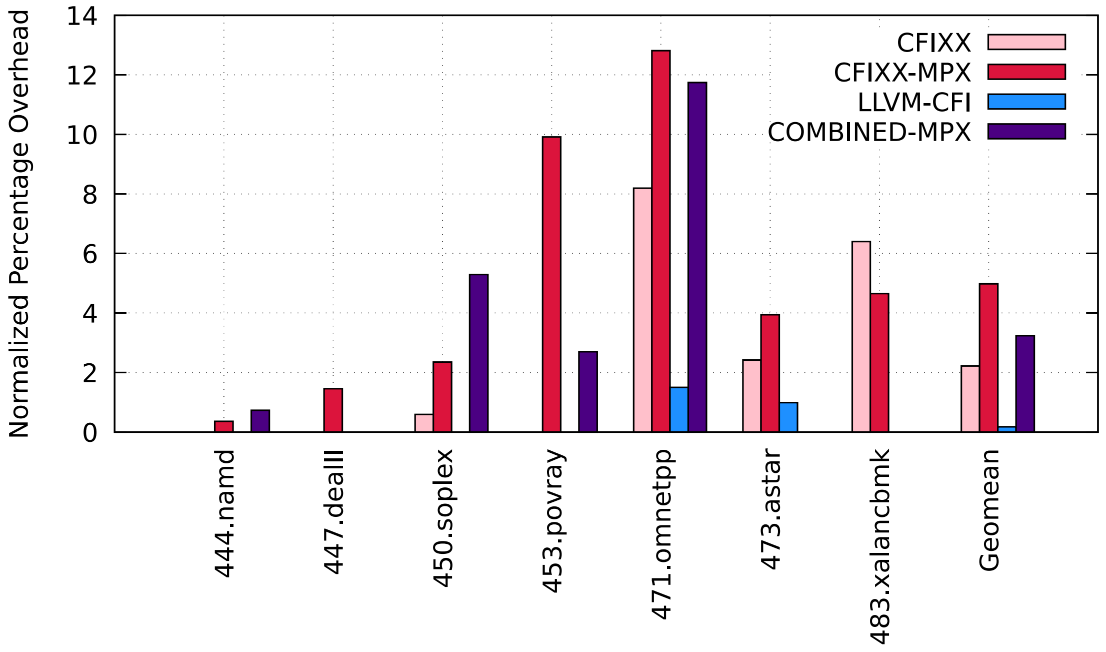


Figure 5.5.: SPEC CPU2006 performance measurements. Results are percentage overhead. LLVM-CFI requires LTO. As a result, the combined measurement also uses LTO.

and 4.98% with full MPX protection. These averages are geometric means over the full set of benchmarks, as is standard in the literature. LLVM includes a CFI implementation that protects virtual calls, using the same policy as VTrust and VTable Interleaving. As VTrust and VTable Interleaving are not yet open source, we evaluate against the LLVM CFI implementation. This protection is enabled with the `-fsantize=cfi-vcall` flag, and requires LTO. To compare CFIXX against, we reran the baseline with LTO as well to achieve an apples to apples comparison for measuring the performance impact of LLVM-CFI. Despite using LTO and a different baseline, its performance is reported in Figure 5.5 for comparison. LLVM-CFI fails to run the xalancbmk benchmark, but on the other benchmarks has 0.18% overhead, or an average of 1.25% on benchmarks with virtual calls. This performance is inline with the reported numbers from VTrust and VTable Interleaving, and reinforces the fact that CFIXX achieves almost equivalent performance with much stronger protections.

To show that OTI and CFI are orthogonal and can work together, we ran SPEC CPU2006 with both LLVM's CFI mechanism and CFIXX enabled. Only one benchmark, dealII, that runs with LLVM-CFI failed under combined protections — without *any* additional engineering effort. We anticipate minimal code changes will be required to make the two mechanisms fully compositional, and ensure that optimizations do not mix their checks. However, running all but one benchmark "out of the box" shows that the two are highly compatible, and that the remaining incompatibility is an engineering issue and not a fundamental design issue. The overhead with both protections enabled is in line with the overhead of only OTI — a benefit of adding the LTO optimizations required by LLVM's CFI mechanism.

To evaluate CFIXX's robustness and impact on real world code, we recompiled the Chromium browser on FreeBSD. FreeBSD was used because its ports system (i) supports clang, and (ii) abstracts out the details of applications' build systems. Consequently, FreeBSD allows for more rapid prototyping of clang-based compilers.

To benchmark Chromium, we ran the Kraken benchmark from Mozilla, the Octane benchmark from Google, and JetStream, which is an independent benchmark. As mentioned in Section 5.5, Chromium performs a `memcpy` of an allocated object. Running these benchmarks under CFIXX required a manual code change to maintain our metadata in the face of this non-standard code. We added three lines of code (a call to our metadata create function, and a forward declaration of that function) to maintain our metadata. FreeBSD does not yet support MPX, so our evaluation of Chromium is without MPX protection.

Browsers are benchmarked using suites of JavaScript tests. CFIXX has the following overheads: 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream. The slow down is unnoticeable to users. For this low cost, CFIXX provides strong control-flow hijacking guarantees for Chromium. By enforcing OTI, CFIXX guarantees the correctness of dynamic dispatch on a per object basis — something CFI policies cannot do. As shown here, it can easily be combined with CFI to mitigate object swapping attacks. Consequently, we believe that CFIXX should be deployed on all browsers.

5.7 Related Work

Defenses against control-flow hijacking attacks have been studied for many years. Instead of listing all proposed policies, we will focus on control-flow integrity defenses that specifically target virtual dispatch and C++. We refer to a recent survey on CFI defenses for a more complete overview [**?**].

VTrust [89] presented the current state-of-the-art policy for protecting dynamic dispatch. VTrust protects pointers to virtual tables through a two-layered system. During runtime, it enforces that the virtual function target matches the expected function, as determined statically from source. VTrust ensures that the target function name, argument type list, and class relationship match through the use of hash signatures. The second layer, which is optional and only needed for applications with writable code, encodes legitimate pointers to virtual tables during object creation, and allows only such pointers to be used. VTrust can stop COOP attacks if they cause virtual calls to violate its computed target set, but not all COOP attacks. VTable Interleaving [90] improves the performance of the VTrust policy by rearranging how vtables are organized in memory, efficiently packing the newly organized tables to reduce memory overhead, and reduces virtual call dispatch verification to a simple range check. Neither mechanism is open source, impeding comparative evaluation of performance and security policies. VIP [91] builds on these policies by adding static analysis to reduce the valid target set at virtual call sites.

VTrust, VTable Interleaving, and VIP have less overhead than CFIXX, but provide less security. While VIP leverages extensive analysis of pointers to shrink target sets as much as possible, the authors still report 1,173 call sites in Chrome with more than 64 targets. Determining the object types that can reach a virtual call site is a very difficult problem for static techniques, which are forced to be over-approximate or prevent legitimate execution paths. This over-approximation limits the security that they can provide.

PittyPat [93] performs an online analysis of execution traces in combination with a statically computed control flow graph to determine legitimate targets of indirect control flow transfers. Consequently, PittyPat is fully path sensitive, which makes its static analysis significantly more precise. PittyPat reports that 90%+ of call sites are reduced to one target. However, a subset of call sites with a large number of targets remains. OTI removes this remaining imprecision for C++.

πCFI [88] builds off of MCFI [87] by dynamically activating edges in the control flow graph on demand. This limits the set of targets available to attackers. However, the target set can still grow to be the full statically computed set. Consequently, while πCFI makes attacks more difficult, it is fundamentally no more secure than MCFI.

SafeDispatch [124] provides vtable protections against some control-flow hijacking attacks, through the use of class hierarchy analysis and run-time checks to confirm that the called method is a valid implementation of the called method. SafeDispatch incurs higher overhead (2.1% vs. 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream) and cannot protect against COOP. In addition, SafeDispatch does not support separate compilation. CFIXX supports separate compilation, making its use in large projects practical.

VTint [132] protects against code injection attacks by preventing the use of virtual tables that are writable. In legitimate programs, objects' virtual tables are never changed, and thus any virtual table that is writable should never be used. VTint checks during runtime that any used virtual table is read-only. Recent attack classes, like COOP and vtable reuse, effectively bypass VTint.

vfGuard [131] extracts virtual tables and call sites from COTS binaries to determine a fine-grained CFI. Once all virtual tables and call sites are found, vfGuard then generates a list of targets for each virtual function and enforces this CFI policy through the use of a binary rewriter [160]. T-VIP [125] is another example of binary rewriting that protects against code-reuse attacks in a similar way to VTint. Again, COOP is able to overcome these defenses. Additionally, vfGuard imposes high overhead, making its widespread use unlikely. LLVM provides a virtual call CFI mechanism [63]. However, as shown in Section 5.6.1, CFIXX blocks more exploits with similar runtime overhead (reported 1% vs. 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream on Chromium). Additionally, LLVM CFI requires Link Time Optimization [161], which greatly increases the resources needed for compilation. CFIXX does not require Link Time Optimization.

An alternate approach to protecting application control flow is Code Pointer Integrity (CPI) and Code Pointer Separation (CPS) [65]. CPI attempts to ensure that pointers to code (return addresses, indirect call sites, etc.) are legitimate through the use of instrumentation and/or runtime checks. CPS is a more relaxed version of CPI, with lower overhead at the expense of some safety guarantees provided by CPI. CPI experienced over 40% performance overhead on some C++ SPEC CPU2006 tests, and over 80% on pybench. CPS had lower overhead, 2% on SPEC CPU2006. The key differences between CPS and CFIXX are that: (i) CFIXX fully protects virtual table pointers, whereas CPS prevents direct overwrites, but not those done through additional indirection, and (ii) CPS cannot prevent COOP attacks because it cannot detect synthetic objects. Consequently, CFIXX provides greater security for C++ at the same overhead.

Type confusion sanitizers [162, 163] also focus on protecting object types. They detect static cast violations dynamically, and do not protect dynamic dispatch. Sanitizers are orthogonal to defenses as they focus on detecting bugs, not protecting against adversarial attacks.

Full memory safety has been the subject of many years of research [13, 28, 30, 36]. Mechanisms that protect unsafe languages either incur large ($> 100\%$) overhead, or only provide probabilistic protections. If practical, full memory safety would eliminate a class of vulnerabilities used by attackers to corrupt program state, such as object types. For these reasons, whole address space memory protection remains an ongoing area of research.

5.8 Conclusion

We have presented Object Type Integrity (OTI). Our new defense policy guarantees that polymorphic objects have the correct type associated with them at run time. By doing so, we protect dynamic dispatch on a per object basis. OTI advances the state of the art in protecting dynamic dispatch by limiting objects to one target per call site, instead of a set of targets like CFI. OTI prevents the second stage of code-reuse attacks — corrupting control flow data — whereas CFI prevents the final stage — control-flow hijacking. Consequently, the two are orthogonal and compatible, and can be deployed together.

Our OTI implementation, CFIXX[4], has minimal overhead on CPU bound applications such as SPEC CPU2006: 4.98%. On key applications like Chromium, CFIXX has negligible overhead on JavaScript benchmarks: 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream. CFIXX protects all code in user space (including the libc and any shared libraries), and comprehensively hardens dynamic dispatch. Consequently, CFIXX is compatible with current large C++ applications such as Chromium, readily deployable, and advances the state of the art in protecting dynamic dispatch.

[4]The prototype implementation is open source at `https://github.com/HexHive/CFIXX`.

6 SUMMARY

We show compilers can be used to close the semantic gap between C/C++ and machine languages by enforcing full or partial memory safety. Our system CUP introduces a novel metadata scheme, and has half the overhead of the state of the art mechanism for memory safety. We also demonstrate how partial memory safety, where only certain regions of memory are protected, can be leveraged for “prevent the exploit” defenses. Our shadow stack work introduces a novel shadow stack mechanism Shadesmar that we recommend for deployment. Shadesmar has 3.65% performance overhead and fully support all C/C++ programming paradigms, and in particular stack unwinding. Our shadow stack work also investigates how hardware primitives can best be leveraged for partial memory safety. Building on our survey of existing CFI techniques, we introduce a new policy called Object Type Integrity that is fully precise for virtual calls in C++, thereby substantially improving the protection of forward-edge control-flow transfers. Our proof of concept OTI mechanisms, CFIX, has 2% performance overhead on Chromium, demonstrating OTI’s power protect widely used systems software that is commonly exposed to attacks. Combined, these defenses offer hope that control-flow hijacking attacks can be mitigated, substantially increasing the security of software.

Bibliography

[1] A. One, “Smashing the stack for fun and profit,” *Phrack magazine*, 1996.

[2] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *CCS ’07*, 2007.

[3] 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 *SP ’15*, 2015.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *SP ’13*, 2013.

[5] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” in *TISSEC ’12*, 2012.

[6] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: A new class of code-reuse attack,” in *CCS ’11*, 2011.

[7] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity,” in *SEC’15*, 2015.

[8] 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 *SEC ’98*, 1998.

[9] M. Corporation, “A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003,” https://support.microsoft.com/en-us/kb/875352, 2013.

[10] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address obfuscation: An efficient approach to combat a broad range of memory error exploits.” in *SEC ’03*, 2003.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *CCS ’05*, 2005.

[12] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Softbound: Highly compatible and complete spatial memory safety for C,” in *PLDI ’09*, 2009.

[13] ——, “Cets: Compiler enforced temporal safety for C,” in *ISMM ’10*, 2010.

[14] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, “Memory errors: The past, the present, and the future,” in *RAID’12*, 2012.

[15] Mitre, https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0235.

[16] F. J. Serna and K. Stadmeyer, https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html.

[17] https://googleprojectzero.blogspot.com/2017/12/apacolypse-now-exploiting-windows-10-in_18.html.

[18] https://googleprojectzero.blogspot.com/2016/09/return-to-libstagefright-exploiting.html.

[19] https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

[20] https://googleprojectzero.blogspot.com/2017/02/attacking-windows-nvidia-driver.html.

[21] https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html.

[22] I. Beer, https://googleprojectzero.blogspot.com/2017/04/exception-oriented-exploitation-on-ios.html.

[23] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *SP ’16*, 2016.

[24] N. Burow, D. McKee, S. A. Carr, and M. Payer, “Cup: Comprehensive user-space protection for c/c++,” in *AsiaCCS ’18*, 2018.

[25] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, “Control-flow integrity: Precision, security, and performance,” *CSUR*, 2017.

[26] N. Burow, D. McKee, S. A. Carr, and M. Payer, “Cfixx: Object type integrity for c++ virtual dispatch,” in *NDSS’18*, 2018.

[27] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” in *ATC’12*, 2012.

[28] E. D. Berger and B. G. Zorn, “Diehard: Probabilistic memory safety for unsafe languages,” *SIGPLAN Not.*, 2006.

[29] G. Novark and E. D. Berger, “Dieharder: Securing the heap,” in *WOOT’11*, 2011.

[30] Duck, Yap, and Cavallaro, “Stack bounds protection with low fat pointers,” in *NDSS ’17*, 2017.

[31] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, “Preventing use-after-free with dangling pointers nullification.” in *NDSS ’15*, 2015.

[32] C. Evans, https://scarybeastsecurity.blogspot.com/2017/05/further-hardening-glibc-malloc-against.html?m=1, May 2017.

[33] TrendMicro, http://blog.trendmicro.com/pwn2own-day-1-recap/.

[34] M. Hicks, “What is memory safety?” 2014. [Online]. Available: http://www.pl-enthusiast.net/2014/07/21/memory-safety/

[35] V. V. D. Veen, https://github.com/vvdveen/memory-errors/.

[36] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors,” in *SEC ’09*, 2009.

[37] C. Evans, https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html.

[38] A. Sotirov, “Heap feng shui in javascript,” *Black Hat Europe*, 2007.

[39] “musl libc,” http://www.musl-libc.org/, 2016.

[40] “National Institute of Standards and Technology Juliet C/C++ test suite,” https://samate.nist.gov/SARD/testsuite.php, 2013.

[41] man7.org, http://man7.org/linux/man-pages/man3/alloca.3.html.

[42] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy software,” 2005.

[43] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, “Secure virtual architecture: A safe execution environment for commodity operating systems,” in *OSR ’07*, 2007.

[44] D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for c with very low overhead,” in *ICSE ’06*, 2006.

[45] F. C. Eigler, “Mudflap: Pointer use checking for c/c+,” in *GCC Developers Summit*, 2003.

[46] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “Everything You Want to Know About Pointer-Based Checking,” in *SNAPL ’15*, 2015.

[47] D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for c with very low overhead,” in *ICSE’06*, 2006.

[48] “IntelÂő software development emulator,” https://software.intel.com/en-us/articles/intel-software-development-emulator, 2015.

[49] C. W. Otterstad, “A brief evaluation of intel® mpx,” in *SysCon ’15*, 2015.

[50] “Address sanitizer intel memory protection extensions,” https://github.com/google/sanitizers/wiki/AddressSanitizerIntelMemoryProtectionExtensions, 2016.

[51] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “Watchdog: Hardware for safe and secure manual memory management and full memory safety,” in *ISCA ’12*, 2012.

[52] S. Nagarakatte, M. M. Martin, and S. Zdancewic, “Watchdoglite: Hardware-accelerated compiler-based pointer checking,” in *CGO ’14*, 2014.

[53] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, “Hardbound: Architectural support for spatial safety of the c programming language,” in *ACM SIGARCH Computer Architecture News*, 2008.

[54] W. Chuang, S. Narayanasamy, and B. Calder, “Accelerating meta data checks for software correctness and security,” *Journal of Instruction-Level Parallelism*, 2007.

[55] J. Caballero, G. Grieco, M. Marron, and A. Nappa, “Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities,” in *ISSTA ’12*, 2012.

[56] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *ACM Sigplan notices*, 2007.

[57] F. Qin, S. Lu, and Y. Zhou, “Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs,” in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, 2005.

[58] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Memtracker: Efficient and programmable support for memory access monitoring and debugging,” in *HPCA’07*, 2007.

[59] S. H. Yong and S. Horwitz, “Protecting c programs from attacks via invalid pointer dereferences,” in *FSE ’03*, 2003.

[60] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Memtracker: Efficient and programmable support for memory access monitoring and debugging,” in *HPCA ’07*, 2007.

[61] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: A safe dialect of c,” in *ATC ’02*, 2002.

[62] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *CCS ’05*, 2005.

[63] “Control flow integrity,” http://clang.llvm.org/docs/ControlFlowIntegrity.html, 2016.

[64] “Control flow guard (windows),” https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[65] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *OSDI ’14*, 2014.

[66] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *AsiaCCS ’15*, 2015.

[67] T.-c. Chiueh and F.-H. Hsu, “Rad: A compile-time solution to buffer overflow attacks,” in *ICDCS’01*, 2001.

[68] L. Davi, A.-R. Sadeghi, and M. Winandy, “Ropdefender: A detection tool to defend against return-oriented programming attacks,” in *AsiaCCS ’11*, 2011.

[69] P. Team, “Pax address space layout randomization (aslr),” 2003.

[70] E. Goktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, “Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure,” in *EuroSP’18*, 2018.

[71] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *SP ’13*, 2013.

[72] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, “Aslr on the line: Practical cache attacks on the mmu,” *NDSS ’17*, 2017.

[73] E. Göktaş, R. Gawlik, and B. Kollenda, “Undermining information hiding (and what to do about it),” in *SEC ’16*, 2016.

[74] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, “Enabling client-side crash-resistance to overcome diversification and information hiding.” in *NDSS ’16*, 2016.

[75] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, “Poking holes in information hiding.” in *SEC ’16*, 2016.

[76] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, “Missing the point (er): On the effectiveness of code pointer integrity,” in *SP ’15*, 2015.

[77] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, “Adapting software fault isolation to contemporary cpu architectures.” in *SEC ’10*, 2010.

[78] S. McCamant and G. Morrisett, “Evaluating sfi for a cisc architecture.” in *SEC ’06*, 2006.

[79] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” in *SP ’09*, 2009.

[80] Y. Jeon, P. Biswas, S. A. Carr, B. Lee, and M. Payer, “Hextype: Efficient detection of type confusion errors for c++,” in *CCS ’17*, 2017.

[81] https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[82] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel, “Erim: Secure and efficient in-process isolation with memory protection keys,” *arXiv preprint arXiv:1801.06822*, 2018.

[83] https://www.keycdn.com/support/the-growth-of-web-page-size/.

[84] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *CCS ’10*, 2010.

[85] 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 *CCS ’15*, 2015.

[86] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, “The dynamics of innocent flesh on the bone: Code reuse ten years later,” in *CCS ’17*, 2017.

[87] B. Niu and G. Tan, “Modular control-flow integrity,” in *PLDI ’14*, 2014.

[88] ——, “Per-input control-flow integrity,” in *CCS ’15*, 2015.

[89] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, “Vtrust: Regaining trust on virtual calls,” in *NDSS ’16*, 2016.

[90] D. Bounov, R. Kici, and S. Lerner, “Protecting c++ dynamic dispatch through vtable interleaving,” in *NDSS ’16*, 2016.

[91] X. Fan, Y. Sui, X. Liao, and J. Xue, “Boosting the precision of virtual call integrity protection with partial pointer analysis for c++,” in *ISSTA ’17*, 2017.

[92] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “Practical context-sensitive cfi,” in *CCS ’15*, 2015.

[93] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, “Efficient protection of path-sensitive control security,” in *CCS ’17*, 2017.

[94] M. Prasad and T.-c. Chiueh, “A binary rewriting defense against stack based buffer overflow attacks.” in *ATC ’03*, 2003.

[95] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “Xfi: Software guards for system address spaces,” in *OSDI’06*, 2006.

[96] X. Ge, W. Cui, and T. Jaeger, “Griffin: Guarding control flows using intel processor trace,” in *ASPLOS ’17*, 2017.

[97] R. Qiao, M. Zhang, and R. Sekar, “A principled approach for rop defense,” in *ACSAC ’15*, 2015.

[98] A. Quach, M. Cole, and A. Prakash, “Supplementing modern software defenses with stack-pointer sanity,” in *ACSAC ’17*, 2017.

[99] A. Prakash and H. Yin, “Defeating rop through denial of stack pivot,” in *ACSAC ’15*, 2015.

[100] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-free: Defeating return-oriented programming through gadget-less binaries,” in *ACSAC ’10*, 2010.

[101] A. van de Ven and I. Molnar, “Exec shield,” https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[102] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity: Principles, implementations, and applications,” in *CCS ’05*, 2005.

[103] B. McCarty, *SELinux: NSA’s Open Source Security Enhanced Linux*. O’Reilly Media, Inc., 2004.

[104] V. Kiriansky, D. Bruening, and S. Amarasinghe, “Secure execution via program shepherding,” in *USENIX Security Symposium*, 2002.

[105] V. Kiriansky, "Secure execution environment via program shepherding," Master's thesis, Massachusetts Institute of Technology, 2013.

[106] PaX-Team, "Pax future," https://pax.grsecurity.net/docs/pax-future.txt, 2003.

[107] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *USENIX Security Symposium*, 2013.

[108] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *SP*, 2014.

[109] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *SEC '14*, 2014.

[110] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *SEC '14*, 2014.

[111] J. R. Bell, "Threaded code," *Communications of the ACM*, vol. 16, no. 6, pp. 370–372, jun 1973.

[112] Kogge, "An Architectural Trail to Threaded-Code Systems," *Computer*, 1982.

[113] E. H. Debaere and J. M. van Campenhout, *Interpretation and instruction path coprocessing*, ser. Computer systems. MIT Press, 1990.

[114] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2009.

[115] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE S&P'10*, 2010.

[116] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Annual Computer Security Applications Conference (ACSAC)*, New York, New York, USA, 2011.

[117] 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 *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[118] B. Niu and G. Tan, "Monitor integrity protection with space efficiency and separate compilation," in *CCS '13*, 2013.

[119] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *USENIX Security Symposium*, 2013.

[120] J. Pewny and T. Holz, "Control-flow restrictor: Compiler-based CFI for iOS," in *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[121] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity & randomization for binary executables," in *SP*, 2013.

[122] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *NDSS '14*, 2014.

[123] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *SP '14*, 2014.

[124] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: Securing c++ virtual calls from memory corruption attacks," 2014.

[125] R. Gawlik and T. Holz, "Towards automated integrity protection of c++ virtual function tables in binary programs," in *ACSAC '14*, 2014.

[126] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Annual Design Automation Conference (DAC)*, 2014.

[127] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[128] 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 *USENIX Security Symposium*, 2014.

[129] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[130] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, “CCFI: cryptographically enforced control flow integrity,” in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[131] A. Prakash, X. Hu, and H. Yin, “vfguard: Strict protection for virtual function calls in cots c++ binaries.” in *NDSS ’15*, 2015.

[132] C. Zhang, C. Song, K. Chen, Z. Chen, and D. Song, “Vtint: Defending virtual function tables’ integrity,” in *NDSS’15*, 2015.

[133] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “PathArmor: Practical ROP protection using context-sensitive CFI,” in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[134] P. Yuan, Q. Zeng, and X. Ding, “Hardware-assisted fine-grained code-reuse attack detection,” in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[135] X. Ge, N. Talele, M. Payer, and T. Jaeger, “Fine-Grained Control-Flow Integrity for Kernel Software,” in *IEEE European Symp. on Security and Privacy*, 2016.

[136] M. Payer, A. Barresi, and T. R. Gross, “Lockdown: Dynamic control-flow integrity,” *arXiv preprint arXiv:1407.0549*, 2014.

[137] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan, “HAFIX: Hardware-assisted flow integrity extension,” in *Annual Design Automation Conference (DAC)*, 2015.

[138] M. Mock, M. Das, C. Chambers, and S. J. Eggers, “Dynamic points-to sets,” in *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, 2001.

[139] A. Sabelfeld and A. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, jan 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1159651

[140] B. Niu and G. Tan, “Mcfi readme,” https://github.com/mcfi/MCFI/blob/master/README.md, 2015.

[141] 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.

[142] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity,” in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015.

[143] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi, “Losing control: On the effectiveness of control-flow integrity under stack attacks,” in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[144] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “A theory of secure control flow,” in *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, ser. ICFEM’05, 2005.

[145] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, “Missing the point: On the effectiveness of code pointer integrity,” in *SP*, 2015.

[146] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “SoK: Automated software diversity,” in *SP*, 2014.

[147] E. Rohou, B. N. Swamy, and A. Seznec, “Branch prediction and the performance of interpreters: Don’t trust folklore,” in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[148] Intel Inc., “Intel 64 and IA-32 architectures. software developer’s manual,” 2013.

[149] Y. Xia, Y. Liu, H. Chen, and B. Zang, “Cfimon: Detecting violation of control flow integrity using performance counters,” in *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2012.

[150] I. Fratric, “ROPGuard: Runtime prevention of return-oriented programming attacks,” http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf, 2012.

[151] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, “Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity,” in *Annual Design Automation Conference (DAC)*, 2016.

[152] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced Control-Flow Integrity," in *CODASPY*, 2016.

[153] B. Patel, "Intel releases new technology specifications to protect against rop attacks," 2016, http://blogs.intel.com/evangelists/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks/.

[154] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[155] Miscosoft, "Setprocessvalidcalltargets function," https://msdn.microsoft.com/en-us/enus/library/windows/desktop/dn934202(v=vs.85).aspx, 2015.

[156] F. Falcon, "Exploiting adobe flash player in the era of control flow guard," BlackHat EU'15 https://www.blackhat.com/docs/eu-15/materials/eu-15-Falcon-Exploiting-Adobe-Flash-Player-In-The-Era-Of-Control-Flow-Guard.pdf, 2015.

[157] D. Weston and M. Miller, "Windows 10 mitigation improvements," BlackHat'16 https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf, 2016.

[158] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[159] N. N. V. Database, https://nvd.nist.gov/.

[160] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI '05*, 2005.

[161] "Llvm link time optimization," http://llvm.org/docs/LinkTimeOptimization.html, 2017.

[162] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *SEC '15*, 2015.

[163] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *CCS '16*, 2016.

VITA

VITA

Nathan Burow graduated from Yale in 2011 with a BA in economics. He received a MS in computer science from Purdue in 2015. Nathan has worked for the Missile Defence Agency, Sandia National Laboratory, Northrop Grumman, and Draper Laboratory. Nathan was named to the all state football team in high school, and was a National Merit Finalist. He is the eldest spawn of Craig and Rebecca Burow.