COMPILER-BASED MITIGATIONS OF VULNERABILITIES IN SYSTEMS SOFTWARE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Scott A. Carr

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2017

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

Dr. Mathias Payer

Department of Computer Science

Dr. Aniket Kate

Department of Computer Science

Dr. Suresh Jagannathan

Department of Computer Science

Dr. Xiangyu Zhang

Department of Computer Science

Approved by:

Dr. William J. Gorman

Head of the Departmental Graduate Program

This dissertation is dedicated to the countless people who have taught me throughout my life, to my family, to my friends, and most of all to Nikki.

ACKNOWLEDGMENTS

I thank my collaborators, reviewers, editors, and proof readers for their invaluable efforts which greatly improved this dissertation. To my advisor, Mathias Payer, I am grateful for the guidance provided and for the continual feedback throughout the last three years. In our research group, we hold ourselves to a high standard, which I believe shows in the quality of our work. To my committee members, Suresh Jagannathan, Xiangyu Zhang, and Aniket Kate, I thank you for your insightful questions and suggestions that helped clarify and strengthen this document. I truly appreciate all the feedback and advice on my papers, presentations, and dissertation from my peers and friends, including Nathan Burow, Daniele Midi, Jeff Avery, Gregory Essertel, Bader AlBassam. Thank you to past and present members of my research group, Ahmed Hussain, Abe Clements, Craig West, Alessandro Di Federico, Jacek Rzeniewiwcz, Dominik Preikschat, Derrick McKee, Hui Peng, Terry Hsu, Yuseok Jeon, Priyam Biswas, Kryiakos Ispoglou, Naif Almakdhub, Prashast Srivastava, and Sushant Dinesh. Lastly but crucially, I give my deepest gratitude to my family, especially to my parents and to Nicole, my wife, whose confidence in me never waivers.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Carr, Scott A. PhD, Purdue University, May 2017. Compiler-based Mitigations of Vulnerabilities in Systems Software. Major Professor: Mathias Payer.

Systems software written in C/C++ is plagued by bugs, which attackers exploit to gain control of systems, leak sensitive data, or perform denial-of-service attacks. This plethora of vulnerabilities is caused by C/C++ not enforcing memory or type safety in language by design, instead they leave security checks to the programmer.

Previous research primarily focuses on preventing control-flow hijack attacks. In a control-flow hijack attack, the attacker manipulates a return address or function pointer to cause code of her choosing to be executed. Abadi et al. propose Control-Flow Integrity (CFI), to prevent such attacks, but as our CFI survey shows, CFI mechanisms have varying degrees of precision. Researchers exploit the imprecision in CFI implementations to evade their protection. One area of imprecision in CFI mechanisms is virtual functions in C++ programs. Attackers can re-target virtual function calls to other invalid functions as part of an exploit. Our work, VTrust, provides specialized protection for C++ virtual functions with low overhead.

As CFI mechanisms improve, and are widely deployed, attackers will follow the path of least resistance towards other attack vectors, e.g., non-control-data attacks. In a non-control-data attack the attacker manipulates ordinary variables (not return addresses, function pointers, etc.) to carry out the attack. Non-control-data attacks are not prevented by CFI, because the control-flow follows a valid path in the original program. The attack is carried out by modifying only non-control-data. To address this emerging problem, we have developed Data Confidentiality and Integrity (DCI) which allows the programmer to select which data types should be protected from corruption and information leakage by the attacker.

In this dissertation, we propose that by using static analysis and runtime checks, we can prevent attacks targeted at sensitive data with low overhead. We have evaluated our techniques, VTrust and DCI, on the SPEC CPU2006 benchmarks, the Firefox web browser, and the mbedTLS cryptographic library. Our results show our implementations have lower performance overhead than other state-of-the-art mechanisms. In our security evaluation, we have several case studies which show our defenses mitigate publicly disclosed vulnerabilities in widely deployed software. In future work, we plan to improve our static sensitivity analysis for DCI and investigate new methods for automatically identifying sensitive data.

1 INTRODUCTION

We develop compiler-based tools that mitigate vulnerabilities in systems software. Systems software is almost exclusively written in C/C++, and these unsafe languages allow subtle programmer errors to become devastating vulnerabilities. Attackers exploit these vulnerabilities to gain control of the system or steal information. Our techniques detect when attacks occur and abort the program to stop the attack.

This section presents our problem area in the context of its challenges and existing work. Included are our thesis statement, a summary of our contributions and an outline of this dissertation. Note that this section omits detailed definitions of several terms for brevity. See Chapter 2 for background information.

1.1 Securing Systems Software

The goal of our research is to prevent attackers from exploiting the endless stream of new vulnerabilities found in systems software. Vulnerabilities are errors that an attacker can exploit to hijack the system or leak sensitive data. Modern systems software is written in unsafe languages such as C/C++. When writing code in unsafe languages, the programmer is responsible for securing the code. For example, a common operation in C programs is copying a string into a buffer, but the language itself does not protect against the string overflowing the buffer. It is the programmer's responsibility to check if the buffer is large enough to hold the string.

Attackers are so successful at finding bugs in critical software that we conclude new tools are needed to facilitate writing secure software. Examples of high-profile vulnerabilities include Heartbleed [1], StageFright[1], ShellShock[2], Ghost[3] and Dirty

[1]`http://www.kb.cert.org/vuls/id/924951`
[2]`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271`
[3]`https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0235`

COW[4]. We divide all software security tools into three broad categories: i) safe programming languages, ii) formal verification, and iii) vulernability mitigation in C/C++. Our work falls into the third category. We provide a high-level intuition for the challenges and trade-offs between these categories in the next three subsections.

1.1.1 Safe Systems Programming Languages

Researchers and practitioners have developed many new safe programming languages which aim to fix the deficiencies of C/C++, (e.g., Rust [2]). Usually these languages provide the type and memory safety[5] that C/C++ do not. If we rewrote our systems software in these safe languages, whole categories of vulnerabilities would be eliminated. However, in the near term, manual rewriting is not a practical solution because production-quality systems software (operating systems, web browsers, cryptographic libraries, etc.) are millions of lines of code. Rewriting everything required for a complete system requires a massive human effort. Another complication is that safe languages often rely on virtual machines or libraries written in unsafe languages. For example, the Java HotSpot Virtual Machine[6] is written in C/C++. Notable efforts to rewrite large pieces of systems software in safe languages include the Servo[7] web browser engine written in Rust, and Singularity OS [3] written in a dialect of C#. Researchers have also proposed Cyclone and CCured [4, 5], which are dialects of C that were developed the eliminate memory safety errors.

The latest C++ standard, C++14 [6], establishes new language features aimed at making C/C++ safer. However, safe C++14 code is dramatically different from C++ code written under the old standards. Adopting safe C++14 practices requires rewriting code with a new standard library (i.e., the Guideline Support Library[8]). In effect, C++14 defines a new language, or at least a new dialect of C++, so the

[4]`https://dirtycow.ninja/`
[5]Type safety and memory safety are defined in Section 2.9 and Section 2.7 respectively.
[6]`http://openjdk.java.net/groups/hotspot/`
[7]`https://servo.org`
[8]`https://github.com/Microsoft/GSL`

same drawbacks of other new safe languages apply. Adopting it requires manual code rewriting and switching tools.

1.1.2 Formal Verification

Formal verification mathematically proves that software, as written in the source language, conforms to some specification. If the specification correctly encodes the security properties[9] we desire, and the software conforms to that specification, we conclude the software is secure[10]. Determining which security properties we desire, and writing a correct specification for complicated systems software are major challenges, thus we do not have a formal specification for our widely used systems software (OSes, servers, web browsers, etc.).

To date, researchers have not produced a verified OS or web browser that could widely replace the production-quality unverified alternatives, but continual progress has been made. A prominent work in this area is seL4 [7]. It is a microkernel consisting of around 9,000 lines of code (LoC), and is the first proven correct general purpose OS kernel. The state of the art in OS verification is CertiKOS [8] is the first kernel with formally proven multiprocess support. Both seL4 and CertiKOS are major achievements, but area not comparable to the Linux Kernel in terms of functionality. seL4 and CertiKOS are microkernels which are less than 10,000 LoC, while Linux is over fifteen million. Directly comparing the LoC count is partly misleading, because Linux is a monolithic kernel whereas seL4 and CertiKOS are microkernels. The relevant difference between monolithic and microkernels is monolithic kernels include device drivers as part of the kernel, but mircokernels do not. Even if we exclude the `drivers` directory for Linux, the LoC count is still over five million. The verification effort for the concurrency proof in CertiKOS was around two person-years, according to the authors, and seL4 required 20 person-years. The ratio of 10,000 LoC to 20

[9]The exact security properties we care about vary between different applications and formal verification techniques.

[10]For the *system* to be secure the hardware must also meet its specification.

person-years demonstrates the challenges in scaling up current formal verification tools to large code bases.

Formal verification and safe programming languages are complimentary as safe languages can be designed to facilitate proofs about programs written in the language.

1.1.3 Vulnerability Mitigation

The final common approach to combating vulnerabilities in systems software is to stop the vulnerabilities from being exploited by an attacker. Broadly, the idea behind this category of techniques is that we should develop automatic tools to find the vulnerabilities or detect when attacks occur. Techniques in this category work with the software as is, rather than requiring a rewrite in a new language or writing a formal specification. Our research falls into this category, and our goal is to compile a version of the program with exploit detection checks added. For example, attackers often exploit buffer overflows, so our compiler inserts new checks for out-of-bounds pointers into the program. These additional dynamic checks are called runtime monitors. The runtime monitors check for suspicious program behavior and abort the program if an attack is detected (before the attacker can hijack the system or leak sensitive data). The most notable work in this category is SoftBound + CETS [9, 10]. Importantly, these techniques work automatically, with little or no human effort required, however, the additional instructions needed for runtime monitoring impose performance overhead. We will describe our techniques which make use of compiler-inserted runtime monitors in detail throughout the remainder of this dissertation.

Static analysis is complimentary to runtime monitors. Most techniques that insert runtime monitors also include a static analysis component to determine statically that some subset of the code is safe (and thus does not need a runtime monitor).

Researchers have also proposed purely static bug finders that do not include a runtime component. In contrast to formal verification which aims to prove the entire program safe, static analyzers typically examine smaller pieces of the program (e.g., individual lines of code or functions). An example tool is μcheck [11], which is a framework for building static checking tools using a micro-grammar – a subset of the language specification. These checkers look for problematic patterns in the code, but do not show that the entire program is safe. The program may contain other bugs not covered by the checkers' set of patterns.

1.2 Challenges

The primary challenge that we face is that users of C/C++ demand high performance, but adding the security guarantees missing from C/C++ (i.e., memory safety and type safety) imposes overhead. Our security mechanisms insert runtime monitors into programs to detect incorrect behavior that may have been caused by an attacker. Example monitors include checking: if a write overflows a buffer (spatial memory safety), if a pointer points to deallocated memory (temporal memory safety), if a pointer points to an object of the incorrect type (type safety) or if a return address has been overwritten (control-flow integrity).

If we could statically determine that a program is safe (or that it is unsafe due to a list of identified bugs), then we would have secure systems software. However, in the general case, every security property we are interested in (e.g., memory safety, type safety, and control-flow integrity) cannot be statically determined for an arbitrary C/C++ program. For example, which function a pointer points to is statically undecidable in general, so we cannot statically determine the exact control-flow graph (CFG) of a C/C++ program that uses function pointers. Instead, static analyzers might over-approximate the true CFG, and consider the function pointer to point to any function of a compatible type. Attackers can exploit this over-approximation. While this example concerns a very specific problem, the pattern emerges when trying to determine any interesting security property about an arbitrary C/C++ program, without executing the program.

Scalability is another major challenge. Since the systems software we care about consists of thousands or millions of LoC, our tools must work automatically. The programmer effort to adopt our tools must be constant, not a function of the number of LoC in the program. Since systems software is dependent on C/C++ ecosystem, we cannot change the programming languages, operating system interface, or Application Binary Interface (ABI) [12] to ensure compatibility with other tools and libraries.

1.3 Attacker Model

We assume that the original program is benign, but may contain bugs. More precisely, the original application should not leak sensitive information or crash when given valid input not controlled by an attacker. However, we assume that the attacker can construct malicious input, which exploits bugs in the program, allowing her to write or read any memory that the program can legally read or write. The operating system is in the Trusted Computing Base (TCB), so we assume virtual memory page permissions are enforced. These permissions allow us to designate a range of memory addresses as being readable, writable, or executable (or some combination of those three). Unless stated otherwise, we assume code is not writable [13], so the attacker cannot perform a code injection attack.

We assume the attacker may make multiple attempts at compromising our system, meaning we cannot rely on randomization [14]. However, Address Space Layout Randomization (ASLR) [15], which randomly rearranges objects in memory for each execution, is complimentary to our techniques, offering defense in depth.

1.4 Thesis Statement

This dissertation presents our techniques which provide targeted defense for sensitive data in C/C++ applications. Our objective is to recompile existing C/C++ programs with additional security monitors to detect attacks before the attacker can leak confidential information or take control of the system.

The thesis statement is:

> *Using compiler-based static analysis and runtime monitors, we can strongly protect critical data in systems software that are left unprotected by existing low overhead defenses.*

Our key insight is that *not all data are equally critical.* Sensitive data like encryption keys, authentication tokens, and passwords can be used to impersonate the system or gain privileges, whereas most other data in a program is less useful to the attacker. Code-pointers (return addresses, function pointers, jump targets) are also targeted by attackers to change the control-flow of the program. We have designed our mechanisms according to this selective protection principle. Previous approaches which enforce a uniform policy for all data (critical and non-critical) lead to high overhead [16]. This has ruled out their widespread deployment. In contrast, with our techniques *we can strongly and completely protect* ***critical data only*** *at low overhead*, allowing us to secure systems with minimal performance impact.

Existing low overhead defenses provide either incomplete or probabilistic protection. In contrast, our tools catch all attacks target at sensitive data under our threat model, but may have false positives in pathological programs. For example, C programs rarely use pointer arithmetic to move a pointer between fields of a `struct`. Such programs are non-portable across architectures (and therefore technically violate the C standard [17]), but can execute successfully if the programmer is careful and knows the target architecture. This technique violates our memory safety policy as described in Section 2.7, causing a false alarm – false in the sense that this is the programmer's intended behavior, not an attack.

1.5 Contributions

The contributions of this dissertation are the design, implementation, and evaluation of targeted protection mechanisms for systems software that protect critical data at low overhead. Our survey of existing mechanisms shows that previously used metrics do not effectively distinguish the strength of previous works, and that new techniques are needed to defend emerging attack vectors. We have designed and implemented two such techniques providing targeted protection for sensitive data and vtables. We evaluate our mechanisms' performance on a variety of benchmarks, present detailed case studies, and discuss their security effectiveness. Specifically our contributions are:

1. Systematization and evaluation of prior work in Control-flow Integrity (CFI), assessing the performance of the mechanisms on the same hardware platform for the first time, qualitatively and quantitatively comparing the mechanisms' provided security, and identifying cross-cutting concerns and future directions for CFI research.
2. Design and implementation of VTrust, a CFI defense mechanism targeted at preventing vtable-based attacks in C++ programs.
   (a) Separate compilation support to protect applications and libraries, increasing our mechanism's practicality.
   (b) Reduced overhead compared to similar state-of-the-art mechanisms, always less than 8% across our benchmarks.
   (c) Security case studies showing VTrust mitigates vulnerabilities found in Firefox, software deployed to millions of users, and state-of-the-art vtable attacks.
3. Design of a new security policy Data Confidentiality and Integrity (DCI) which allows the programmer to select which data types are protected
   (a) Open-source prototype implementation of DCI (DataShield) based on the LLVM compiler infrastructure
   (b) Case study of applying DataShield to a production quality TLS/SSL library (mbedTLS), showing our technique is applicable to large code bases

(c) Drop-in replacement C/C++ shared standard libraries that are instrumented by DataShield, which allow the user to link programs instrumented by DataShield with their system libraries. Uninstrumented programs can use the instrumented libraries as well.

(d) Security evaluation showing DCI mitigates a publicly disclosed vulnerability in mbedTLS by detecting a proof-of-concept attack

1.6 Publications

This dissertation draws from our projects which have appeared in publications at several high-quality security conferences and journals. Our CFI survey paper will appear in ACM Computing Surveys 2018 [18]. VTrust was published in the Internet Society Network and Distributed System Security Symposium (NDSS) 2016 [19]. DCI [20] will appear in ACM Asia Conference on Computer and Communication Security (AsiaCCS) 2017, and we presented a poster on the same topic at IEEE Symposium on Security and Privacy 2015 [21].

1.7 Summary and Outline

In this dissertation, we present the design, implementation, and evaluation of our defense mechanisms which automatically protect critical data in C/C++ programs. Protecting critical data is complicated by C/C++'s lack of type and memory safety, making it difficult to efficiently prevent programmer errors from becoming security vulnerabilities. Many software security approaches have been developed including new safe languages, formal verification, runtime monitors, static bug finding, and auditing. We developed compiler-based approaches which combine static analysis with runtime monitors to provide strong always-on protection to C/C++ programs.

The remainder of this dissertation is laid out as follows. In Chapter 2, we provide the relevant background information necessary to understand the details of our defense mechanisms. Detailed discussion and results from our CFI survey appears in Chapter 3.

We describe the design and implementation of VTrust in detail in Chapter 4 and in Chapter 5 we do the same for DCI. In Chapter 6, we evaluate our mechanisms' security and performance. We discuss our future work in Chapter 7 and related work in Chapter 8. In Chapter 9, we conclude this dissertation.

2 SECURITY BACKGROUND

In this chapter, we present the necessary background information to understand the remainder of this dissertation. First, we describe the primary attack vectors adversaries use to hijack an application or corrupt/leak sensitive data. Next, we discuss defenses that are already deployed across modern desktops and servers. These defenses do not prevent all attacks, but we assume these baseline protections are in place on the systems we are protecting. For context, we summarize categories of academic research in defense techniques and provide important examples of each category. Next, we transition towards technical details that are relevant to systems software security research. In particular, we discuss the aspects of the C/C++ programming languages [6,17] which lead to insecure code or provide targets to attackers. The safety of a C/C++ program is left to the developer, with the language favoring flexibility and performance over correctness guarantees. Specifically we describe manual memory management, the difficulties in analyzing C/C++, Just in Time Compilation (JIT), and dynamic dispatch. Next, we define the most common security properties in security research. They are Control-Flow Integrity (CFI), memory safety (spatial and temporal), and type safety. With the discussion of each property, we include example tools that enforce the property, and a high-level intuition of how the mechanisms work.

2.1 Attack Vectors

Attackers have developed many attack types against C/C++ applications. The broadest categories of attacks are control-flow hijack attacks and non-control-data attacks. We also describe vtable attacks, which are a specialized control-flow hijack attack.

2.1.1 Control-Flow Hijack Attacks

The goal of a control-flow hijack attack is to divert the program execution's control-flow from the programmer's intended control-flow. The common examples of control-flow hijack attacks include stack smashing, code injection, and code reuse.

In a stack smashing attack, the attacker uses a buffer overflow on the stack to overwrite a return address stored on the stack with her chosen value. When the function returns, the execution will jump to the attacker-chosen instruction. Given that the attacker can direct execution to her chosen instruction, the next issue is determining which instruction the attacker wants to execute.

If the attacker can write to executable memory, she may perform a code injection attack. In a code injection attack, the attacker inserts code into the running program and jumps to the inserted code.

Modern systems employ code injection defenses (e.g., DEP). In this case the attack must reuse existing code, a so-called code reuse attack. For example, the attacker may redirect a call to a library function (e.g., `system`) in a code reuse attack called a return-to-libc attack. Or the attacker may stitch together sequences of instructions ending with return instructions in a return-oriented-programming (ROP) attack.

Other than return addresses, attackers often target function pointers. If the attacker can overwrite the value of a function pointer and that function pointer is invoked, she can dictate the control-flow of the program.

2.1.2 VTable Attacks

Several specialized attacks have been developed for vtables in C++. A vtable is an array of function pointers which are used to determine the target function of a virtual function call at runtime. See Section 2.5.4 for details. Broadly there are two categories of attacks, vtable injection and vtable corruption. The distinction is that vtable injection attacks create new vtables that did not exist in the original program, while vtable corruption reuses existing vtables. To change the vtable that will be used to dispatch a virtual function call, the attacker changes the object's vtable pointer to point to her chosen vtable. Alternatively to changing the vtable pointer the attacker maybe overwrite the vtable itself. However, vtable corruption is rare as modern compilers store vtables in read-only memory. Attackers have discovered that the program loader may copy vtables to writable memory when classes with virtual functions are defined in multiple modules [22]. The most notable vtable attack is Counterfeit Object-oriented Programming (COOP) [23], which performs arbitrary computation by chaining C++ virtual calls. A detailed discussion of vtable attacks can be found in Section 4.3.2.

2.2 Non-Control-Data Attacks

Non-control-data attacks are when the attacker corrupts or leaks information in the program without changing the program's intended control-flow. These non-control-data attacks can be as devastating as control-flow hijack attacks [24]. The HeartBleed Bug [1] in OpenSSL is an example of non-control-data attack that could be used to leak information. One way of exploiting the HeartBleed Bug is to send a malicious packet to the server where the length field of the message sent is greater than the actual message. This causes the server to echo back to original message plus whatever data happened to be next to the message packet in the server's memory. For a detailed discussion of non-control-data attacks please see Section 5.3.3

2.3 Widely Deployed Defenses

Severally widely deployed mechanism have been developed, but attackers have learned how to bypass them. Most modern desktop/server systems have stack canaries [25], Data Execution Prevention (DEP) [26], and Address Space Layout Randomization (ASLR) [15] enabled.

Stack canaries are values (randomly chosen at program startup) that are placed below each return address on the stack. We will call the randomly chosen value the expected value. It is stored in a protected location that the attacker cannot corrupt. Before each function return, the canary value on the stack is compared to the expected value. If the values do not match, this indicates an attacker has overwritten the stack. Stack canaries prevent stack smashing attacks where an attacker uses a continuous buffer overflow on the stack to overwrite the return address.

DEP uses page permissions to prevent code injection attacks. Page permissions are an operating system feature that controls whether a process can read, write, or execute a given chunk of memory (a page). DEP ensures that no page is both writable and executable at the same time so any new data the attacker may write into the program's memory will not be executable. Therefore, DEP prevents code injection attacks. Page permissions are used by other defense mechanisms to prevent an attacker from overwriting sensitive data. A commonly used technique is to allocate a protected page that is only made writable when the mechanism needs to update the data on the page, but is made read-only when the application is executing. This prevents the attacker from exploiting the application to overwrite the protected page's data.

ASLR is a probabilistic technique that dynamically changes the address of objects in memory on a per-execution basis. Under ASLR, the addresses of libraries, code, globals, the top of the stack, and heap will be placed at randomly determined addresses. This stops the attacker from knowing beforehand what the exact address of some object will be based on prior executions. ASLR does not achieve a purely random memory layout in practice. For example, if an attacker can leak the address of a C standard library (libc) function, she can calculate the address of any other libc function using the offset between the same two functions on her own system (assuming the attacked system and the attackers system use the same library version). Due to information leaks, ASLR does not completely defeat code reuse attacks, including return-to-libc and ROP, but makes them more difficult.

While these mechanisms have been widely deployed, they do not prevent all attacks. Instead, they make attacks more difficult. Accordingly, attackers and defenders are actively researching new techniques in an attempt to win the security arms race.

2.4 Defense Techniques Research

Many defense techniques have been proposed by software security researchers and these techniques broadly fall into the categories static analysis, runtime monitors, and logging.

2.4.1 Static Bug Finding

Static bug finding tools examine the program program source code without compiling and running it. In the context of security, these tools typically look for bugs that an attacker could exploit. As mentioned in Section 2.5, static analysis of C/C++ is difficult because of lack of type and memory safety and pointer aliasing. As such, many analysis tools are limited to look for patterns in the code that indicate there *may* be a bug. Stronger static analyses include symbolic execution, model checking, and formal verification. An interesting direction for future work is semi-static methods – methods that augment static analysis with some form of dynamic program execution. For example, Driller [27] combines symbolic execution with fuzzing (i.e., dynamically executing and testing a program on many automatically generated inputs).

The primary drawback of bug finding tools is that they require the programmer to manually fix the bug once it has been found. Correctly fixing the bug, without introducing new bugs can be difficult in complex software. However, automated repair, combining static bug finding with automatic program rewriting, is an emerging research topic [28].

2.4.2 Runtime Monitors

In contrast to static bug finding tools which never execute the program, runtime monitors are dynamic. They insert new instructions into the program to maintain and check metadata. When the program executes, failed checks are evidence of an attack. Examples of checks performed by runtime monitors include array bounds checking, checks if a pointer points to allocated memory, and type cast verification. All the mechanisms discussed in Section 2.8, in addition to our work, fall into this category. The main drawback of runtime monitors is that the additional instructions impose performance overhead. However, runtime monitors can check for dynamic program states which cannot be determined statically. Also, tools that insert runtime monitors automatically can be utilized with minimal programmer effort. Runtime monitors typically abort the program when the check fails (i.e., an attack or other malicious condition is detected).

2.4.3 Logging and Auditing

Logging is another form of runtime defense, but instead of having the criteria to determine if an attack has occurred being built into the program, as in a runtime monitor, security relevant data is logged and later analyzed (or audited) off-line. A canonical example of logging is taint or provenance tracking where data is marked with a tag indicating its origin [29]. For example in macOS files downloaded through a web browser are marked as being downloaded from the internet and when they are opened a warning is displayed to the user. Logging mechanisms do not prevent the attack when it occurs, but can detect a wider variety of attacks as a human analyst determines if an attack occurred. Another example of logging are Intrusion Detection Systems (IDS), which monitor network traffic [30] or collect performance metrics from a host [31] to scan for malicious behavior.

2.5 Security Properties of C/C++

C/C++ are systems programming languages, meaning that they are designed for programming software such as performance critical code, operating systems, web browsers, web servers, and cryptographic libraries. C/C++ give the programmer precise control over resources and memory used by the program. For example, C/C++ programmers may carefully optimize the memory layout of data structures for space efficiency, and precisely control when memory is allocated, deallocated, and initialized for performance efficiency.

However, the low-level control provided by C/C++ comes with drawbacks. The language specifications and compilers do not prevent errors like writing past the end of a buffer. Nothing prevents a programmer from dereferencing a pointer to deallocated memory, freeing the same memory twice, or reading uninitialized memory. Without a strong type system, it is up to the programmer to ensure type casts are correct. For example, a source of errors in C++ is casting from parent class to child class (downcasting). However, this is unsafe because a parent class may have many child classes and the code may cast to the incorrect child class.

2.5.1 Manual Memory Management

In C, manual memory management for the heap is performed as follows. The programmer requests a number of bytes of memory from the operating system using `malloc`, `calloc`, or `realloc` and releases a block of memory by passing a pointer to the beginning of the block to `free`. The program can request additional memory on the stack with `alloca`, which is automatically deallocated when the function returns.

In C++ the `new` keyword is roughly equivalent to `malloc`, and `delete` is roughly equivalent to `free`. C++'s constructors and destructors can automatically perform actions when an object is allocated or deallocated.

Not all memory objects are manually created and destroyed by the programmer. Instructions for allocating a stack frame, to hold a function's local variables, and deallocating the stack frame are created by the compiler. Stack frames are created at the beginning of each function call and destroyed before each function returns. Note that compilers can optimize tail-recursive calls to reuse stack frames. The program loader maps global variables into the program's memory space when the program starts or a new library is loaded. The loader also maps the program's binary code into memory.

2.5.2 Analysis Challenges

C/C++ programs are challenging to analyze due to separate compilation, pointer aliasing, and lack of type and memory safety.

Separate compilation means that libraries and applications are compiled separately. Often in C/C++ even submodules of the same application/library are compiled separately. Separate compilation prevents security tools from employing whole program analysis. On most systems there is an installed binary version of the C/C++ standard libraries shared among the system's applications. This presents a challenge to security mechanisms that insert runtime monitors as the system libraries will be an attack surface if they are unmodified.

It is common in C/C++ programs for many pointers to point to the same object. This is called pointer aliasing. Without knowing all aliases, an analyzer cannot accurately track the state of the pointed-to object, because the object may have been changed through an unknown alias. For objects that do not escape the current function, the analyzer can find all aliases to the local object, but for objects that escape, alias analysis is much more difficult.

Another analysis issue is that the lack of type and memory safety means that knowing what type of a object a pointer points to or if a pointer points to a valid object is impossible statically in the general case. For example, it is valid for a `void*` pointer to point to an object of any type. However, to use the pointer, it must be cast to another type. This cast from `void*` to another type is unchecked in C, so an analyzer cannot assume pointers point to an object of the correct type in general. It is also very common to create dangling pointers, pointers that point to invalid memory. This can be because the pointed-to object has been freed, or the pointer to an array element has been incremented past the last element of the array.

2.5.3 Just-In-Time Compilation

Web browsers are among the most widely used and often attacked C++ programs. Accordingly, web browsers are of particular interest to software security researchers, but browsers are even more difficult to analyze than typical C/C++ programs. Modern browsers are millions of lines of code (LoC) and include a JavaScript compiler that performs just-in-time compilation (JIT) of JavaScript code. The reason for implementing JIT is that the generated code is more efficient than interpreting the original JavaScript. Code generated by the JIT compiler opens another attack surface, as the attacker can tamper with the JIT's generated code. Since the JavaScript code is downloaded from websites the browser visits, the JavaScript code is untrusted and must be isolated from the browser's internal data structures. Designing defense mechanisms that can generically protect any JIT compiler is an unsolved problem. However, RockJIT [32] (which is specific to Google's V8 JavaScript engine) suggests this is a promising direction for future research.

2.5.4 Dynamic Dispatch

Indirect control-flow transfers occur frequently in C/C++ programs, and are relevant to security as after an indirect control-flow transfer the next instruction to be executed cannot be determined statically in general. To make our discussion concrete, we will use the names of x86 instructions, but other desktop/server architectures have analogous instructions. We classify control-flow transfers on the basis of direction. Backward transfers are returns from a function. Forward transfers are function calls. Forward transfers to an address that is not the beginning of a function are possible, but are not relevant to this dissertation. For example, `switch` statements can be implemented using jumps which are a type of forward control-flow transfer.

The most frequent kind of indirect control-flow transfer is the return instruction. A return instruction (`ret`) executes at the of end of each function, and `ret` transfers execution to next instruction after the instruction that called the returning function.

Function pointers allow C/C++ programmers to create variables that point to, and call, dynamically determined functions. Calling a function through a function pointer is typically compiled to a call instruction (`call`) with a computed address, whereas a normal function call is compiled to a `call` with a fixed address. While this dissertation and our research are architecture-agnostic (except when explicitly stated otherwise), instruction set architectures (ISA) that have no instructions analogous to `call`, and `ret` are out of scope, except when discussed in our CFI survey, Chapter 3.

C++ introduces a new type of function call not directly supported by C, namely virtual function calls. In C++, programmers can create a class which contains any number of members. Members can either be data or functions that manipulate the class's data. Classes may inherit the members of another class. The class that inherits is called the child class and the class being inherited from is called the parent. To allow a parent class's member function to be overridden, the programmer marks the function `virtual`. Classes with virtual member functions are call polymorphic classes. Every instantiated object of a polymorphic class has special pointer (a vtable pointer) which points to a table of virtual member functions (a vtable). When a virtual member function is called at runtime, the target function is determined by examining the vtable pointed to by the object's vtable pointer. This is called dynamic dispatch. In Chapter 4, Figure 4.2 shows a detailed dynamic dispatch example.

2.6 Control-Flow Integrity

Control-Flow Integrity is a very active research topic [33–46]. CFI mechanisms generally combine a static analysis to determine the program's Control-Flow Graph (CFG) with runtime monitors to ensure the program's execution does not deviate from this CFG. The aim of CFI is to prevent control-flow hijacking attacks.

The simplest CFI implementations conceptually group all the possible targets of an indirect call and assign the group of targets a label. In the binary, this label is inserted before each target function in the group. A check is inserted before the indirect call to determine if the dynamically calculated target has the matching label. Our survey in Chapter 3 discusses these mechanisms in detail.

2.7 Memory Safety

We described the mechanics of manual memory management for C/C++ in Section 2.5.1, but in this section we describe issues that arise from using these mechanics incorrectly and unsafely. First, we will give a high-level intuition for the different kinds of memory safety errors, then we will discuss the errors in depth throughout the remainder of this section.

A program is memory safe if every memory read/write in the program satisfies the following. Given pointer P, which was most recently assigned to point to object O:

1. Writes that dereference P only write O.
2. Reads that dereference P only read data that had been written to O – not uninitialized data.
3. O has not been deallocated.

Otherwise, the read/write through pointer P is a memory error.

There are two categories of memory errors, spatial and temporal. Spatial memory errors occur when a pointer is dereferenced outside the bounds of the intended object. For example, if a programmer requests an allocation of 100 bytes (i.e., `buf = malloc(100)`) and attempts to read the 101st byte (i.e., `buf[100]`), that is a spatial memory error. Spatial memory errors are often buffer overflows similar to the above example. Temporal errors occur when a pointer to a deallocated object is dereferenced. Use-after-free is the canonical example of a temporal memory error, but double free (attempting to free the same memory object twice), and reading uninitialized memory are also considered temporal memory errors. These errors typically occur when memory is deallocated in multiple code locations in a program, and the programmer makes an incorrect assumption about where an object will be deallocated.

```c
void fn() {
  char* buf = malloc(6);
  strcpy(buf, "hello");
  char* ptr = buf+2;
  ...
}
```

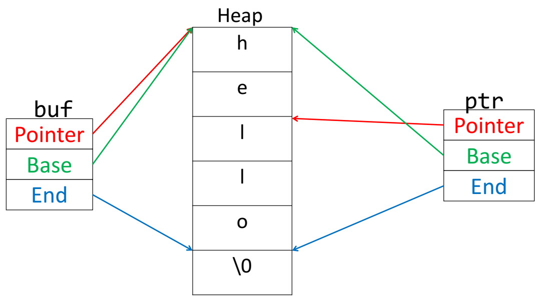


Figure 2.1. Example C code demonstrating pointers as capabilities to read or write a string buffer.

2.7.1 Spatial Memory Safety

A security-centric view of spatial memory safety is that pointers are capabilities to read or write some memory object [47]. Conceptually, every pointer is a 3-tuple, $\langle pointer, base, end\rangle$, where *pointer* is the current value of the pointer, base is the lowest address the pointer is allowed to point to, and end is the highest. A simple example is show in Figure 2.1. Here, both `buf` and `ptr` have the capability to read the entire buffer allocated by `malloc`, but the current value of the pointers are different.

An alternative definition of a spatial memory error is *any error that would not occur if objects were infinitely far apart in memory* [47]. C/C++ programs should not rely on the space between objects in memory because the spacing is under the control of the compiler and the memory allocator. In effect, any program that relies on the offset between two distinct objects is unsafe.

2.7.2 Temporal Memory Safety

Temporal memory safety errors are errors which occur because the programmer made an incorrect assumption – that memory has not been deallocated, or is initialized. In Figure 2.2, two examples of temporal memory errors are shown. In Figure 2.2a, there is an error on line 4, because the memory pointed to by `x` has been deallocated. In Figure 2.2b, the variable `y` is allocated on the stack, so when the function `f` returns,

```
1 char *x = malloc(10);
2 x[3] = 'a';
3 free(x);
4 x[0] = 'b'; // error
```

(a) A use-after-free error on the heap.

```
1 char* f() {
2   char y = 'f';
3   return &y;
4 }
5 void g() {
6   char z = *(f()); // error
7 }
```

(b) Return of stack local variable.

Figure 2.2. Temporal memory error examples. In (a) a pointer to deallocated heap memory is referenced. In (b) the stack frame containing the local variable will be deallocated when the function returns.

`y` will be automatically deallocated. The returned pointer will point to deallocated memory and dereferencing this pointer (line 6) is a temporal safety violation.

Temporal memory safety can be implemented with a capability, where each capability contains an allocation ID. The allocation ID is stored in both the capability associated with the pointer and the allocated object. Then the mechanism inserts a check before each pointer dereference to determine if the allocation ID of the capability and the allocation ID of the pointed-to object match. CETS [10] uses this scheme, but calls the allocation ID stored in the object the "lock" and the allocation ID stored in the capability the "key."

2.8 Memory Safety Enforcement Mechanisms

Researchers have proposed many mechanisms to detect memory errors in C/C++ applications. Most approaches combine static analysis with runtime checks, because statically determining if a C/C++ program is memory safe is impossible in general. In the example in Figure 2.3a, the function `input` can return any value. For instance, `input` might return a user entered value. We cannot determine if line 3 is safe statically, however, if we insert runtime checks, as in Figure 2.3b, we can be sure the program is now memory safe. At a high-level, it is the goal of many mechanisms to transform Figure 2.3a into Figure 2.3b automatically.

```
char* buf = malloc(10);
unsigned int i = input();
buf[i] = 'a'; // unsafe
...
```

(a) An unsafe program.

```
char* buf = malloc(10);
unsigned int i = input();
if (i < 9) {
  buf[i] = 'a'; // safe
} else {
  abort();
}
...
```

(b) The program from (a) rewritten to be memory safe.

Figure 2.3. A potentially unsafe program. If the value of `i` is within the bounds of `buf` the program will execute normally. If not, the program may crash or overwrite unintended data.

The preceding example was contrived because the size of the array (`buf`) was fixed, so it was obvious how to rewrite the code to make it statically safe. Usually, for heap allocations the mechanism cannot statically determine the size of an array so it records the array's size at runtime. Sizes are recorded for any memory object accessed through pointers (e.g., arrays, structs, class objects, etc.). The size can be encoded as a capability as described in Section 2.7.1. The data about objects (in any format) is referred to generically as metadata in the literature. Most mechanisms maintain a mapping from pointers to the associated metadata.

2.8.1 Memory Safety Using Pointer Checking

The main two approaches for mapping pointers to metadata are fat pointers and disjoint metadata. Fat pointers add data to the existing pointer encoding [48]. For example, on a 32-bit system a pointer is encoded in 4 bytes, but a fat pointer may be 8 or 16 bytes, to encode the metadata for that pointer. The main drawback of this approach is that it changes object layout and breaks compatibility with separately compiled binaries. For disjoint metadata the format of a pointer is unchanged and metadata is stored in a separate data structure. Often the address of a pointer maps to some entry in the metadata data structure. For a complete discussion on the metadata schemes see Nagarakatte et al. [16].

For enforcement mechanisms that ensure spatial safety based on pointer checking, the metadata for a given pointer is typically the base address and last address of the pointed-to object – which along with the pointer's value forms the capability tuple discussed in Section 2.7.1. This results in a memory overhead of two times the size of a pointer per pointer. For temporal safety mechanisms based on pointer checking the metadata is usually an allocation ID tracked for both the object itself and the pointer which points to the object. The metadata is typically created on object allocation, deleted on object deallocation, and propagated on pointer assignment.

2.8.2 Memory Safety Using Object Alignment

Alternatively to per-pointer metadata, mechanisms may use object padding or alignment to track the size of objects implicitly [49, 50]. For example, a mechanism could create various regions of fixed-sized objects and round-up allocations to the next size category.

Aligned regions can also be used for temporal safety. Objects of the same type have the same size, so if the mechanisms create an allocation region for each type it naturally aligns the objects [51] to known offsets. This way if memory is deallocated and reused an object of the same type will be reallocated to a given location. This mitigates use-after-free errors because the layout of the old object and the reallocated object are the same.

2.8.3 Protecting Metadata from Attackers

For mechanisms that are designed to be actively running in production, and thus potentially attacked, we have to ensure that the attacker does not tamper with the mechanism's metadata. For example, if the attacker could overwrite a pointer's capability, then she could cause a buffer overflow which would go undetected.

Mechanisms that enforce memory safety for every pointer protect the metadata automatically – the metadata should never be in bounds of any pointer defined in the original program, so can never be read or written by these pointers. However, mechanisms that enforce other security properties (e.g., partial memory safety) must protect their metadata through other means. They may maintain metadata in memory pages with read-only permissions, or isolate the metadata from the application with Software Fault Isolation (SFI) [52].

2.8.4 Important Memory Safety Mechanisms

The notable memory safety enforcement works include CCured [5], Cyclone [4], and SoftBound+CETS [9, 10]. Cyclone replaces manual memory management with region-based memory management, which statically defines the lifetime of objects in a given region. Both CCured and Cyclone classify pointers based on if their usage. So-called "safe" pointers are those that can be statically proven to be safe. Other pointers are unsafe. While CCured and Cyclone, define new C dialects, SoftBound works on unmodified C programs. The common theme among these mechanisms is that they identify pointers which are used in an unsafe way – typically this means the pointer is used in non-constant pointer arithmetic. The mechanisms track metadata about these unsafe pointers, and insert checks to determine if the pointer still points to a valid memory object when it is referenced.

2.9 Type Safety

A program that is type safe if whenever a pointer is dereferenced, the pointer and the pointed-to object are of compatible types. A type error occurs when a pointer of type T is dereferenced when the pointed to object is of an incompatible type U. Type safety in C is complicated by C programmers' habit of implementing ad-hoc inheritance by abusing `struct` layout rules, and C's admittance of `void*` pointers pointing to objects of any type. In C++, the meaning of "incompatible type" is also

```
1  class Parent {};
2  class ChildA : Parent {
3    int ID;
4  };
5  class ChildB : Parent {
6    float precision;
7  };
8
9  ChildA *ca = new ChildA();
10 Parent *p = new Parent();
11
12 Parent* p2 = static_cast<Parent*>(ca); // safe
13 ChildA* c2 = static_cast<ChildA*>(p); // unsafe
```

Figure 2.4. Examples of upcasting and downcasting. Line 12 is a downcast (from Child to Parent) which is always safe. Line 13 is an upcast (from Parent to Child) is is potentially unsafe.

governed by the type inheritance rules, as it is legal to dereference a pointer of a parent class type when the pointer points to an object of a child class type. Casting a pointer from child to parent type is called upcasting and is always safe. Casting a pointer from parent to child is called downcasting and can be unsafe if the parent class has multiple child classes and the true type of the child object is unknown.

An example of a type safety violation is shown in Figure 2.4. On Line 12 this cast is an upcast and is guaranteed to be safe. Since `ChildA` inherits `Parent` it is safe to cast `ChildA` pointers to `Parent`. On line 13 the cast is a downcast and is unsafe. `Parent` pointers may point to objects of any of the following types: `Parent`, `ChildA`, or `ChildB`. `Parent` pointers cannot be safely cast to `ChildA` pointers because the pointed to object might be of type `Parent` or `ChildB` which would cause the type of the pointer and the type of the pointed to object to mismatch (i.e., a type safety violation).

2.9.1 Type Safety Enforcement Mechanisms

The important works in this area are (in chronological order) UBSan [53], CaVer [54], and TypeSan [55]. Unfortunately, all these approaches are incomplete – they are not able to verify all casts in their evaluated programs. In particular, CaVer and TypeSan identify the type of an object at its allocation site, and their results show that some allocations are missed, leading to missing type information and unverified casts. However, the CaVer paper showed that these can be useful bug finding tools as the authors discovered two new vulnerabilities in Firefox.

2.10 Security Background Summary

Despite the best efforts of many researchers, new vulnerabilities are constantly being discovered in systems software. C/C++ present challenges to software security researchers and developers in that these languages do not provide type or memory safety guarantees. Widely used defenses (stack canaries, DEP, and ASLR) make executing these attacks more difficult but all deployed defenses can be bypassed. Common attack vectors control-flow hijacking using indirect transfers and non-control-data attacks. Software security researchers often combine static analysis and runtime monitors to try to detect and prevent these attacks, but the additional runtime monitor instructions lead to performance overhead. However, we argue that this approach provides strong defense with minimal required programmer effort, and our work aims to reduce these sources of overhead.

3 CONTROL-FLOW INTEGRITY

3.1 Abstract

Memory corruption errors in C/C++ programs remain the most common source of security vulnerabilities in today's systems. Control-flow hijacking attacks exploit memory corruption vulnerabilities to divert program execution away from the intended control flow. Researchers have spent more than a decade studying and refining defenses based on Control-Flow Integrity (CFI), and this technique is now integrated into several production compilers. However, so far no study has systematically compared the various proposed CFI mechanisms, nor is there any protocol on how to compare such mechanisms.

We compare a broad range of CFI mechanisms using a unified nomenclature based on (i) a qualitative discussion of the conceptual security guarantees, (ii) a quantitative security evaluation, and (iii) an empirical evaluation of their performance in the same test environment. For each mechanism, we evaluate (i) protected types of control-flow transfers, (ii) the precision of the protection for forward and backward edges. For open-source compiler-based implementations, we additionally evaluate (iii) the generated equivalence classes and target sets, and (iv) the runtime performance.

3.2 Introduction

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 [56]. The combination of three such defenses, (i) Address Space Layout Randomization (ASLR) [15], (ii) stack canaries [26], and (iii) Data Execution Prevention (DEP) [13] 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) [33] 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 an edge 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 branches, 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. Code is assumed to be static and immutable[1], either at compile time or in a binary. 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 [9] and temporal [10] 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 [58,59], Abadi et al.'s seminal paper [33] 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. 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) [41] 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

[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 [57], and the `ProcessDynamicCodePolicy` Windows API restrict how page permissions can be changed to prevent code injection and modification.

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 [60–62], 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 chapter 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.

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.

3.2.1 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. For those interested, a more comprehensive overview of static analysis techniques is available in Section 3.3.

3.2.2 Control-Flow Integrity Example

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 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

```
1    void foo(int a){
2        return;
3    }
4    void bar(int a){
5        return;
6    }
7    void baz(void){
8        int a = input();
9        void (*fptr)(int);
10       if(a){
11         fptr = foo;
12         fptr();
13       } else {
14         fptr = bar;
15         fptr();
16       }
17   }
```

Figure 3.1. Simplified example of over approximation in static analysis.

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 3.1. 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 (see Section 3.3).

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 3.1. 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.

3.2.3 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].

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.

[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.

- *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 [63–65].

- *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 `struct`s, 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 lookup. Such a dynamic dispatch using a `send`-method in Smalltalk, 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 dissertation 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 [66–68].

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 Link Time Optimization (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.

3.2.4 Classification of Static Analysis Precision

As we saw in Section 3.2.2, 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 dissertation, the following classification summarizes prior work to determine forward control-flow transfer analysis precision (see Section 3.3 for full details). 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

3.3 Prior Work on Static Analysis

Static analysis research has attracted significant interest from the research community. Following our classification of control-flows in Section 3.2.3, we are particularly interested in static analysis that identifies indirect calls/jump targets. Researchers refer to this kind of static analysis as *points-to analysis*. The wealth of information and results in points-to analysis goes well beyond the scope of this dissertation. We refer the interested reader to Smaragdakis and Balatsouras [69] and focus our attention on how points-to analysis affects CFI precision.

3.3.1 A Theoretical Perspective

Many compiler optimizations benefit from points-to analysis. As a result, points-to analysis must be sound at all times and therefore conservatively over-approximates results. The program analysis literature (e.g., [69–72]) expresses this conservative aspect as a *may*-analysis: A specific object "may" point to any members of a computed points-to set.

For the purposes of this dissertation, the following orthogonal dimensions in points-to analysis affect precision:

- *flow-sensitive* vs. *flow-insensitive*: this dimension states whether an analysis considers control-flow (sensitive) or not (insensitive).

- *context-sensitive* vs. *context-insensitive*: this dimension states whether an analysis considers various forms of context (sensitive) or not (insensitive). The literature further separates the following context information sub-categories: (i) call-site sensitive: the context includes a function's call-site (e.g., call-strings [73]), (ii) object sensitive: the context includes the specific receiver object present at a call-site [74], (iii) type sensitive: the context includes type information of functions or objects at a call-site [75].

Both dimensions, context and flow sensitivity, are orthogonal and a points-to analysis combining both yields higher precision.

Flow-Sensitivity. Figures 3.2a – 3.2c show the effect of flow sensitivity on points-to analysis. A flow-sensitive analysis considers the state of the program per line. We see, for instance, in Figure 3.2b how a flow-sensitive analysis computes the proper object type per allocation site. A flow-insensitive analysis, on the other hand, computes sets that are valid for the whole program. Or, simply put, it lumps all statements of the analyzed block (intra- or interprocedural) into one set and computes a single points-to set that satisfies all of these statements. From a CFI perspective, a flow-sensitive points-to analysis offers higher precision.

Context-Sensitivity. Figures 3.2d – 3.2f show the effects of context sensitivity on points-to analysis. In Figure 3.2d we see that the function `id` is called twice, with parameters of different dynamic types. Context-insensitive analysis (cf. Figure 3.2f), does not distinguish between the two different calling contexts and therefore computes an over-approximation by lumping all invocations into one points-to set (e.g., the result of calling `id` is a set with two members). A context-insensitive analysis, put differently, considers a function independent from its callers, and is therefore the forward control-flow transfer symmetric case of a backward control-flow transfers returning to many callers [70]. Context-sensitive analysis (cf. Figure 3.2e), on the other hand, uses additional context information to compute higher precision results.

```
Object o;
o= new A();
...

o= new B();
```

(a) Flow-sensitivity example.

```
1
2 o → A
3 ...
4
5 o → B
```

(b) Flow-sensitive result.

$o \rightarrow \{\texttt{A}, \texttt{B}\}$

(c) Flow-insensitive result.

```
// identity function
Object id(Object o) { return o; }

x= new A();
y= new B();
a= id(x);
b= id(y);
```

(d) Context-sensitivity example.

```
1
2
3 x → A
4 y → B
5 a → A;   id_1 → A
6 b → B;   id_2 → B
```

(e) Context-sensitive result.sult.

```
1
2
3 x → A
4 y → B
5 a → id;   id → A
6 b → id;   id → {A, B}
```

(f) Context-insensitive result.

Figure 3.2. Effects of flow/context sensitivity on precision.

The last two lines in Figure 3.2e illustrate the higher precision by inferring the proper dynamic types `A` and `B`. From a CFI perspective, a context-sensitive points-to analysis offers higher precision.

Object-Oriented Programming Languages. A C-like language requires call-string or type context-sensitivity to compute precise results for function pointers. Due to dynamic dispatch, however, a C++-like language should consider more context provided by object sensitivity [74, 76]. Alternatively, prior work describes several algorithms to "de-virtualize" call-sites. If a static analysis identifies that only one receiver is possible for a given call-site (i.e., if the points-to set is a singleton) a compiler can sidestep expensive dynamic dispatch via the vtable and generate a direct call to the referenced method. Class-hierarchy analysis (CHA) [77] and rapid-type analysis (RTA) [78] are prominent examples that use domain-specific information about the class hierarchy to optimize virtual method calls. RTA differs from CHA by pruning entries from the class hierarchy from objects that have not been instantiated. As a result, the RTA precision is higher than CHA precision [79]. Grove and Chambers [79] study the topic of call-graph construction and present a partial order of various approaches' precision (Figure 19, pg. 735). With regards to CFI, higher precision in the call-graph of virtual method invocations translates to either (i) more de-virtualized call-sites, which replace an indirect call by a direct call, or (ii) shrinking the points-to sets, which reduce an adversary's attack surface. Note that the former, de-virtualization of a call-site also has the added benefit of removing the call-site from a points-to set and transforming an indirect control-flow transfer to a direct control-flow transfer that need not be validated by the CFI enforcement component.

3.3.2 A Practical Perspective

Points-to analysis over-approximation reduces precision and therefore restricts the optimization potential of programs. The reduced precision also lowers precision for CFI, opening the door for attackers. If, for instance, the over-approximated set of computed targets contains many more "reachable" targets, then an attacker can use those control-flow transfers without violating the CFI policy. Consequently, prior results from studying the precision of static points-to analysis are of key importance to understanding CFI policies' security properties.

Mock et al. have studied dynamic points-to sets and compared them to statically determined points-to sets [80]. More precisely, the study used an instrumentation framework to compute dynamic points-to sets and compared them with three flow- and context-insensitive points-to algorithms. The authors report that static analyses identified 14% of all points-to sets as singletons, whereas dynamic points-to sets were singletons in 97% of all cases. In addition, the study reports that one out of two statically computed singleton points-to sets were optimal in the sense that the dynamic points-to sets were also singletons. The authors describe some caveats and state that flow and context sensitive points-to analyses were not practical in evaluation since

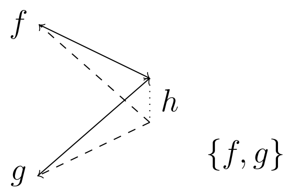


Figure 3.3. Backward control-flow precision. Solid lines correspond to function calls and dashed lines to returns from functions to their call sites. Call-sites are singletons whereas h's return can return to two callers.

they did not scale to practical programs. Subsequent work has, however, established the scalability of such points-to analyses [81–83], and a similar experiment evaluating the precision of computed results is warranted.

Concerning the analysis of de-virtualized method calls, prior work reports the following results. By way of manual inspection, Rountev et al. [84] report that 26% of call chains computed by RTA were actually infeasible. Lhotak and Hendren [76] studied the effect of context-sensitivity to improve precision on object-oriented programs. They find that context sensitivity has only a modest effect on call-graph precision, but also report substantial benefits of context sensitivity to resolve virtual calls. In particular, Lhotak and Hendren highlight the utility of object-sensitive analyses for this task. Tip and Palsberg [85] present advanced algorithms, XTA among others, and report that it improves precision over RTA, on average, by 88%.

3.3.3 Backward Control Flows

Figure 3.3 shows two functions, f and g, which call another function h. The return instruction in function h can, therefore, return to either function f or g, depending on which function actually called h at run-time. To select the proper caller, the compiler maintains and uses a stack of activation records, also known as stack frames. Each stack frame contains information about the CPU instruction pointer of the caller as well as bookkeeping information for local variables.

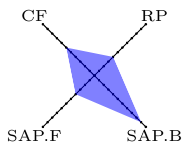
Since there is only one return instruction at the end of a function, even the most precise static analysis can only infer the set of callers for all calls. Computing this set, inevitably, leads to imprecision and all call-sites of a given function must therefore share the same label/ID such that the CFI check succeeds. Presently, the only known alternative to this loss of precision is to maintain a shadow stack and check whether the current return address equals the return address of the most recent call instruction.

3.3.4 Nomenclature and Taxonomy

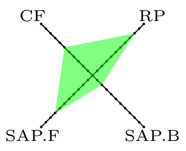
Prior work on CFI usually classifies mechanisms into fine-grained and coarse-grained. Over time, however, these terms have been used to describe different systems with varying granularity and have, therefore, become overloaded and imprecise. In addition, prior work only uses a rough separation into forward 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.

3.4 Security

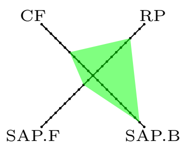
In this section we present a security analysis of existing CFI implementations. Drawing on the foundational knowledge in Section 3.2.1, we present a qualitative analysis of the theoretical security of different CFI mechanisms based on the policies that they implement. We then give a quantitative evaluation of a selection of CFI implementations. Finally, we survey previous security evaluations and known attacks against CFI.



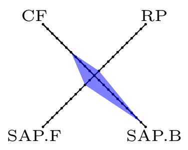
(a) Orig. CFI [33]



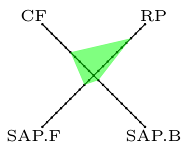
(b) Hypersafe [35]



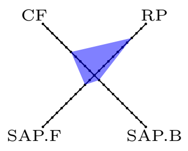
(c) CF-Locking [86]



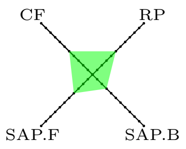
(d) MoCFI [87]



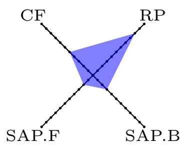
(e) MIP [42]



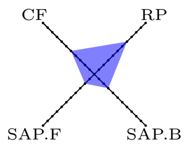
(f) kBouncer [88]



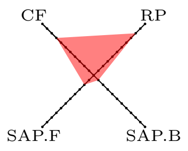
(g) CF-Restrictor [89]



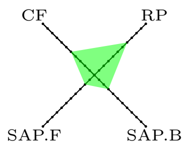
(h) CCFIR [39]



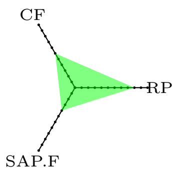
(i) bin-CFI [41]



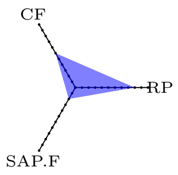
(j) ROPecker [90]



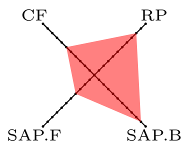
(k) KCoFI [44]



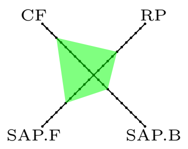
(l) SafeDispatch [91]



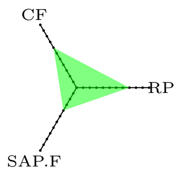
(m) T-VIP [92]



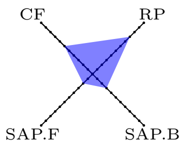
(n) HW-asst. CFI [93]



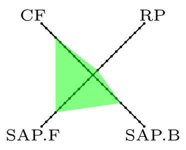
(o) RockJIT [32]



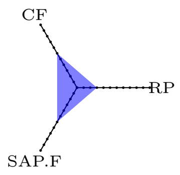
(p) VTV [46]



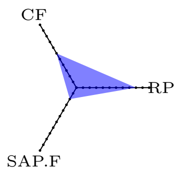
(q) O-CFI [94]



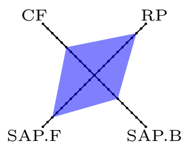
(r) C-CFI [95]



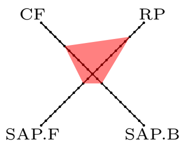
(s) vfGuard [96]



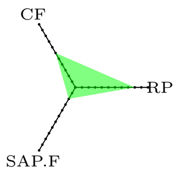
(t) VTint [97]



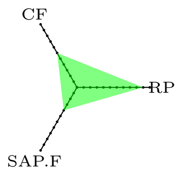
(u) PathArmor [98]



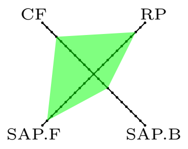
(v) CFIGuard [99]



(w) CFGuard [100]



(x) VTI [101]



(y) Kernel CFI [102]

Figure 3.4. 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.

3.4.1 Qualitative Security Guarantees

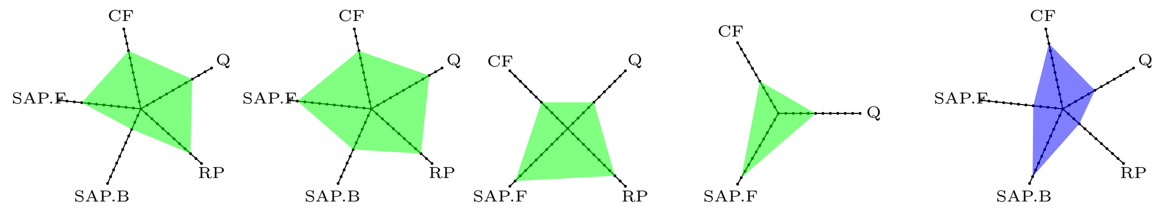
Our qualitative analysis of prior work and proposed CFI implementations relies on the classifications of the previous section (cf. Section 3.2.1) to provide a higher resolution view of precision and security. Figure 3.4 summarizes our findings among four dimensions based on the author's reported results and analysis techniques. Figure 3.5 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 3.4.2.

In Figure 3.4 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 3.2.3;
- 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 3.2.4; and
- SAP.B: static-analysis precision of backward control-flows, assigned based on our classification in Section 3.2.4.

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.



(a) MCFI [45] (b) πCFI [43] (c) IFCC [46] (d) LLVM-CFI-3.7 (2015) (e) Lockdown [103]

Figure 3.5. Quantitative comparison: control-flows (CF), quantitative security (Q), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B).

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 [33], MoCFI [87], HAFIX [93,104], and Lockdown [103]. 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 same label. Labeling, consequently, is a substantial source of imprecision (see Section 3.4.2 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 [80]. Borrowing a term from information-flow control [105], 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 [33], control-flow locking [37], CF-restrictor [89], CCFIR [39], MCFI [45], KCoFI [44], and RockJIT [32].

According to the criteria established in analyzing points-to precision, we find that at the time of this writing, πCFI [43] offers the highest precision due to leveraging dynamic points-to information. πCFI's predecessors, RockJIT [32] and MCFI [45], 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 [103] 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 [46] 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 [35], all other evaluated

CFI implementations with supporting academic publications offer lower precision of varying degrees, at most as precise as SAP.F.3.

3.4.2 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 [46]. 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 3.1. 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.

$$EC * \frac{1}{LC} = QuantitativeSecurity \tag{3.1}$$

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.

3.4.3 Implementations

We evaluate four compiler-based, open-source CFI mechanisms IFCC, LLVM-CFI, 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.

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 [106]. 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,

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

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).

3.4.4 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 3.5, see Table 3.1 for the full data set.

Table 3.1. Full quantitative security results for number of equivalence classes.

Benchmark	CFI Implementation															
	MCFI	πCFI	MCFI	πCFI	MCFI	πCFI	MCFI	πCFI	IFCC				LLVM-CFI		Lock-	Dynamic
	back edge				forward edge				single	arity	simpl.	full	3.7	3.9	down	
			no tail call				no tail call									
400.perlbench	978	310	1192	429	38	30	38	30	1	6	12	40	0	36	4	83
401.bzip2	484	82	489	86	14	10	14	10	1	2	2	2	0	2	3	12
403.gcc	2219	1260	3282	1836	98	90	98	90	0	0	0	0	0	94	3	197
429.mcf	475	96	475	96	12	8	12	8	0	0	0	0	0	0	3	0
445.gobmk	922	283	1075	230	21	17	21	17	0	0	0	0	0	11	4	0
456.hmmer	663	134	720	147	14	9	14	9	0	0	0	0	0	3	4	9
458.sjeng	540	119	557	125	13	9	13	9	1	1	1	1	0	1	3	1
462.libquantum	495	88	519	102	12	8	12	8	1	1	1	1	0	0	4	0
464.h264ref	773	285	847	327	21	15	21	15	0	0	0	0	0	9	4	59
471.omnetpp	1693	581	1784	624	357	321	357	321	0	0	0	0	114	35	0	224
473.astar	1096	226	1108	237	166	150	166	150	0	0	0	0	1	1	6	1
483.xalancbmk	6161	2381	7162	2869	1534	1200	1534	1200	0	0	0	0	2197	260	6	1402
433.milc	602	169	628	180	13	9	13	9	0	0	0	0	0	1	4	3
444.namd	1080	217	1087	224	166	150	166	150	1	1	1	5	4	4	6	12
447.dealII	2952	817	3468	896	293	258	293	258	0	0	0	0	43	15	0	95
450.soplex	1444	432	1569	479	321	291	321	291	1	7	0	186	41	9	6	157
453.povray	1748	650	1934	743	218	204	218	204	0	0	0	0	29	33	6	49
470.lbm	465	70	470	74	12	8	12	8	0	0	0	0	0	0	4	0
482.sphinx3	633	239	677	257	13	9	13	9	0	0	0	0	0	1	4	2

Figure 3.6 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 3.6 and Figure 3.7 to show the relative cost of protecting vtables in C++ relative to protecting all indirect call sites.

We quantify the set sizes for each of the four implementations in Figure 3.7. 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.

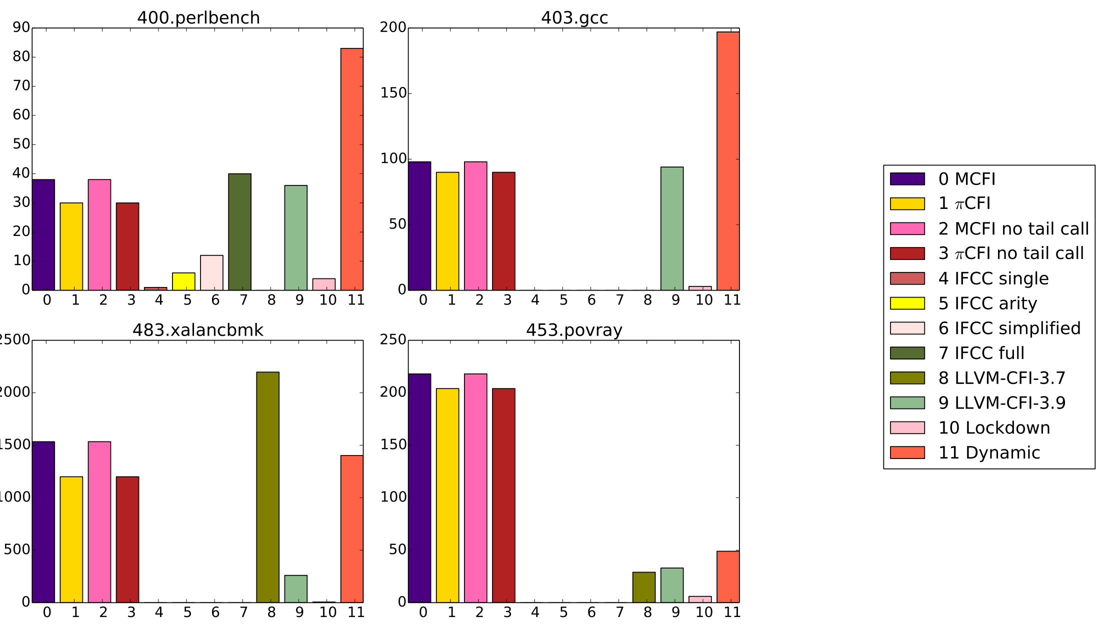


Figure 3.6. Total number of forward-edge equivalence classes when running SPEC CPU2006 (higher is better).

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

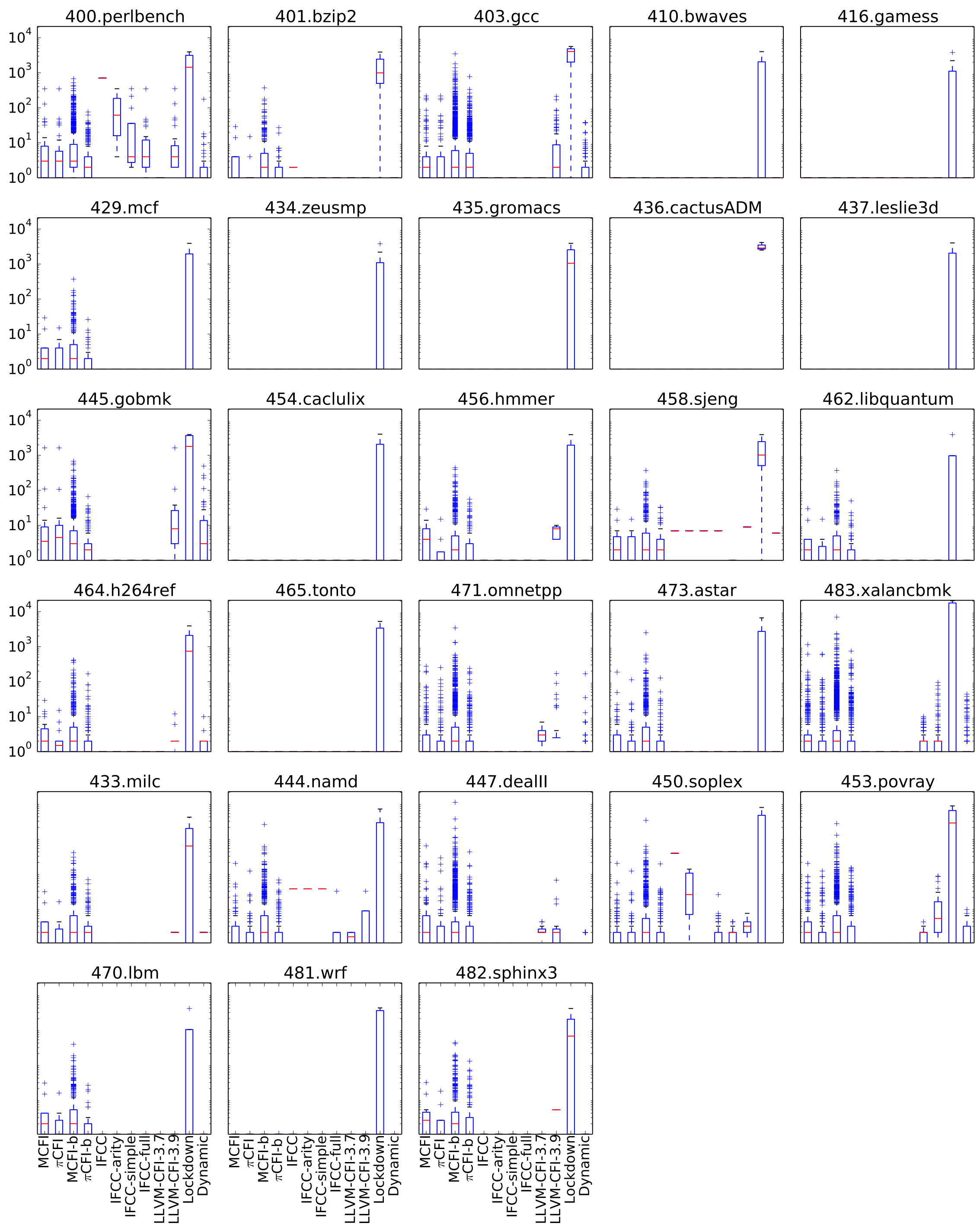


Figure 3.7. Whisker plot of equivalence class sizes for different mechanisms when running SPEC CPU2006. (Smaller is Better)

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 the whisker plots show, 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 3.7 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 3.7 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 3.7 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.

3.4.5 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 [41] 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 [45]. 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 [60–62]. 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) [23] 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 [107] 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 [62] 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 [108] 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 [109]. 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 [14] shows that ASLR might not be enough to hide this secret data from an adversary.

3.5 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. [56] 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 [110].

3.5.1 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 3.2.

Table 3.2. Measured and reported CFI performance overhead (%) on the SPEC CPU2006 benchmarks. The 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

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 [43]. 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 [101].

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 [56] of 5%. This dispenses with the concern that CFI enforcement is too costly in practice compared to alternative mitigations including those based on randomization [111]. 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 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

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

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 [97]. 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 [46] 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 be compared to the reported column for πCFI. We measured

[5]Non-CFI related hardware improvements, such as better branch prediction [112], also help to reduce performance overhead.

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 [112] 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 3.6. 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.

3.5.2 Reported CFI Performance

The right-hand side of Table 3.2 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) [113] 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 3.2 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 3.4i and Figure 3.4w). 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 3.3 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 [33, 94] whereas others evaluate performance using, e.g., web browsers [39, 91], or web servers [103, 114]. 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 3.3. 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 [115]	PCMark Vantage, NovaBench, 3DMark06, Peacekeeper, Sunspider, SuperPI 16M	0.5%
SafeDispatch [91]	Octane, Kraken, Sunspider, Balls, linelayout, HTML5	2.0%
CCFIR [39]	SPEC2kINT, SPEC2kFP, SPEC2k6INT	^{C} 2.1%
kBouncer [88]	wmplayer, Internet Explorer, Adobe Reader	^{C} 4.0%
OCFI [94]	SPEC2k	4.7%
CFIMon [114]	httpd, Exim, Wu-ftpd, Memcached	6.1%
Original CFI [33]	SPEC2k	16.0%

3.6 Summary

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 chapter, 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.

This survey of the CFI literature provides the context for our other two projects VTrust and Data Confidentiality and Integrity (DCI). Specifically, VTrust is a specialized CFI, for preventing attacks on vtables in C++. VTrust's motivation is to decrease the number of allowed targets at virtual function call sites. DCI is concerned with the attack surface not protected by CFI, namely non-control-data attacks.

4 VTRUST

4.1 Abstract

Virtual function calls are one of the most popular control-flow hijack attack targets. Compilers use a virtual function pointer table, called a *vtable*, to dynamically dispatch virtual function calls. These vtables are read-only, but pointers to them are not. VTable pointers reside in objects that are writable, allowing attackers to overwrite them. As a result, attackers can divert the control-flow of virtual function calls and launch *VTable hijacking* attacks. Researchers have proposed several solutions to protect virtual calls. However, they either incur high performance overhead or fail to defeat some VTable hijacking attacks.

In this chapter, we propose a lightweight defense solution, *VTrust*, to protect all virtual function calls from VTable hijacking attacks. It consists of two independent layers of defenses: *virtual function type enforcement* and *vtable pointer sanitization*. Combined with modern compilers' default configuration, i.e., placing vtables in read-only memory, VTrust can defeat all VTable hijacking attacks and supports modularity, allowing us to harden applications module by module. We have implemented a prototype on the LLVM compiler framework. Our experiments show that this solution only introduces a low performance overhead, and it defeats real world VTable hijacking attacks.

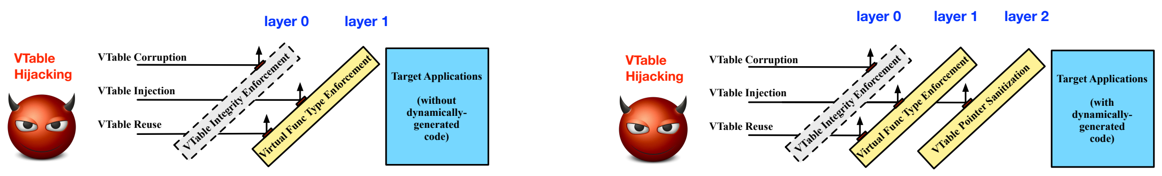
4.2 Introduction

Control-flow hijacking is the dominant attack vector to gain code execution on current systems. Attackers utilize memory safety vulnerabilities in a program and its libraries to tamper with existing data or prepare their own data structures in a target process' memory. When used by the program, the tampered data will redirect benign control-flow to attacker controlled locations. Attackers usually continue by reusing existing code sequences, e.g., Return Oriented Programming (ROP [66, 67, 116]) or Jump Oriented Programming (JOP [117]), to gain full code execution capabilities on the victim system, despite existing defenses like Data Execution Prevention (DEP [26]), Address Space Layout Randomization (ASLR [15]), or stack canaries [25].

Many defense mechanisms have been proposed to protect programs against control-flow hijacking attacks, including memory safety solutions [9, 10, 118] and Control-Flow Integrity (CFI) solutions [33–46]. See Chapter 3 for a detailed discussion and comparison of previous work in CFI. Memory safety solutions stop memory corruption and provide a strong security guarantee, but with a high performance overhead (often larger than 30%). Recent work has shown CFI is a practical approach. Researchers have created implementations of CFI that were incorporated into GCC and LLVM [46]. The most recent operating system Windows 10 has deployed a coarse-grained CFI by default [119]. CFI solutions typically either provide a coarse-grained protection, or incur a high performance overhead. A control-flow hijack attack usually targets return instructions, indirect jumps and indirect calls[1] to control the program counter. Out of these three, indirect calls, which are frequently used for virtual calls in programs written in C++, are receiving increasing attention from attackers. For example, over 80% of attacks against Chrome utilize use-after-free vulnerabilities and virtual function calls [120], whereas about 91.8% of indirect calls are virtual calls [46]. More than 50% of known attacks targeting Windows 7 exploit use-after-free vulnerabilities and virtual calls [121].

Modern compilers use a table (called vtable), consisting of virtual function pointers, to dynamically dispatch virtual calls. Attackers may tamper with these vtables, or pointers to them, and launch VTable hijacking attacks [19], including *VTable corruption* attacks that corrupt writable vtables, *VTable injection* and *VTable reuse* attacks that overwrite vtable pointers with references to fake or existing vtables (or even plain data).

[1]In rare cases, attackers may hijack the program via other instructions, e.g., iret. Such attacks are uncommon in the real world.


layer 0
layer 1
VTable Hijacking
VTable Corruption
VTable Injection
VTable Reuse
VTable Integrity Enforcement
Virtual Func Type Enforcement
Target Applications (without dynamically-generated code)
layer 0
layer 1
layer 2
VTable Hijacking
VTable Corruption
VTable Injection
VTable Reuse
VTable Integrity Enforcement
Virtual Func Type Enforcement
VTable Pointer Sanitization
Target Applications (with dynamically-generated code)

(a) Single layer of defense for applications without dynamically generated code

(b) Two layers of defense for applications with dynamically generated code

Figure 4.1. Illustration of VTrust's overall defense. The layer 0 defense (i.e., placing vtables in read-only sections to protect their integrity) is deployed by modern compilers by default, and thus provides an extra layer of defense for free. The layer 1 defense enforces virtual functions' type at runtime. It defeats all VTable reuse attacks, and also defeats VTable injection attacks if there are no writable code sections. The layer 2 defense enforces the validity of vtable pointers. It defeats VTable injection attacks even if there are writable code sections.

Modern compilers place vtables in read-only sections, defeating VTable corruption attacks by default. But VTable injection attacks are still one of the most popular attacks, and VTable reuse attacks are also practical and hard to defeat [23].

Researchers have proposed several defenses against VTable hijacking attacks. SafeDispatch [91] resolves the set of legitimate virtual functions (or vtable) for each virtual function call site at compile time, and validates the runtime virtual function pointer (or vtable) against this legitimate set. It requires an exact class hierarchy analysis, and involves a heavy runtime lookup operation. Moreover, it requires recompilation of all modules when a new module is added to the application or the inheritance hierarchy changes. Virtual Table Verification (VTV) [46] also validates the runtime vtable against a legitimate set. It supports incremental compilation by updating class hierarchy information at runtime, but also incurs high performance overhead, especially when the class inheritance graph is complex. VTint [19] uses binary rewriting to protect the integrity of vtables, and blocks corrupted or injected vtables from being used, but fails to protect against VTable reuse attacks. Our survey in Chapter 3 qualitatively compares these mechanisms as shown in Figure 3.4. The research paper COOP [23] shows that VTable reuse attacks are practical and even Turing-complete in real applications.

In this dissertation, we propose a lightweight solution VTrust to protect virtual calls from all VTable hijacking attacks. It first validates the validity of virtual function pointers, and then optionally validates the validity of vtables. As shown in Figure 4.1, it consists of two layers of defense: (1) *virtual function type enforcement* and an optional layer (2) *vtable pointer sanitization*. In the first layer, we instrument virtual calls with an additional check to match the runtime target function's type with the one expected in the source code. Each virtual function call site is enforced to invoke virtual functions with the same name and argument type list, and a compatible class relationship. Ideally, this layer is able to defeat all VTable hijacking attacks, if we can check virtual functions' type at runtime, e.g., by utilizing RTTI (RunTime Type Information). However, this would cause a very high performance overhead [54].

Our solution encodes the virtual functions' type information into hash signatures, and matches the signatures at runtime. It provides a fine-grained protection against VTable hijacking attacks. Essentially, it is a C++-aware fine-grained CFI policy. As far as we know, all existing signature-based CFI solutions do not utilize the name of virtual functions and its associated class information to protect virtual calls, causing a loss of precision. On the other hand, taking function name and class information into consideration is not a trivial task. We are the first to present such a C++-aware precise signature-based CFI implementation for virtual calls.

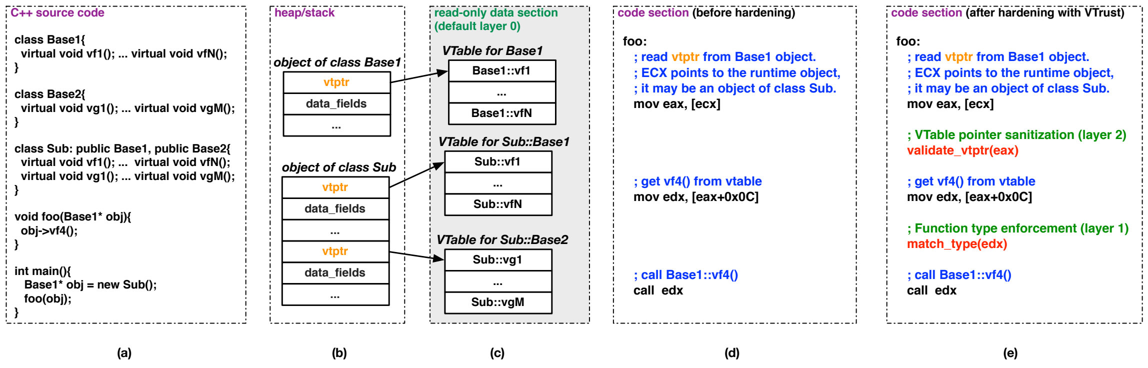
Unlike other CFI solutions [33], VTrust supports separate compilation. The signatures can be computed within each module, without any dependency on external modules, allowing us to harden applications module by module. It introduces a very low performance overhead, e.g., 0.31% for Firefox and 0.72% for SPEC CPU2006. It is able to defeat all VTable reuse attacks, including the COOP attack [23]. It is also able to defeat all VTable injection attacks, if target applications do not have writable code (e.g., dynamically generated code). Given that modern systems are protected by DEP, attackers cannot overwrite read-only code to forge virtual functions with correct signatures. Thus this layer of defense is practical and useful in the real world, because (1) most applications do not have writable code, and (2) forging signatures in writable code is hard due to defenses like ASLR and JIT spraying [122] mitigations. We strongly recommend deploying this defense in practice.

For applications with writable code, attackers may launch VTable injection attacks, as shown in Figure 4.1(b). Traditional signature-based CFI solutions fail to defeat this type of attack. We provide an extra optional layer of defense, to defeat VTable injection attacks that are launched by forging virtual functions with signatures in writable code memory. In this layer, we ensure that each vtable pointer points to a *valid* VTable at runtime by sanitizing the writable and untrusted vtable pointers. More specifically, we encode legitimate vtable pointers when initializing objects and decode them before virtual function calls. In this way, it blocks illegal (forged) virtual functions from being used, by blocking illegal vtables from being used. Even if attackers can forge virtual functions with correct signatures to bypass the first layer of defense, they cannot call them because vtables are all sanitized.

Since this layer of defense changes the representation of vtable pointers, it ensures we can protect all uses of vtable pointers, e.g., RTTI lookup or virtual base objects indexing, or corner cases like custom virtual calls written in assembly. Traditional solutions, e.g., SafeDispatch [91] and VTV [46], only cover regular virtual call instructions and are unable to identify these attack surfaces or protect them from being exploited.

We implemented a prototype on the LLVM compiler framework and tested the prototype on the SPEC CPU2006 benchmark [123], and the browser Firefox. The first layer and the second layer of defense introduce an overhead of about 0.31% and 1.80% respectively for Firefox, and an overhead of about 0.72% and 1.40% respectively for SPEC CPU2006. We also evaluated VTrust against several real world exploits targeting browsers, as well as exploits targeting some real world CTF (Capture The Flag) challenge programs. It showed VTrust is able to defeat them all.

In summary, our VTrust defense solution has the following key advantages:



```
C++ source code

class Base1{
  virtual void vf1(); ... virtual void vfN();
}

class Base2{
  virtual void vg1(); ... virtual void vgM();
}

class Sub: public Base1, public Base2{
  virtual void vf1(); ...  virtual void vfN();
  virtual void vg1(); ... virtual void vgM();
}

void foo(Base1* obj){
  obj->vf4();
}

int main(){
   Base1* obj = new Sub();
   foo(obj);
}
```

```
code section (before hardening)

foo:
   ; read vtptr from Base1 object.
   ; ECX points to the runtime object,
   ; it may be an object of class Sub.
   mov eax, [ecx]

   ; get vf4() from vtable
   mov edx, [eax+0x0C]

   ; call Base1::vf4()
   call  edx
```

```
code section (after hardening with VTrust)

foo:
   ; read vtptr from Base1 object.
   ; ECX points to the runtime object,
   ; it may be an object of class Sub.
   mov eax, [ecx]

   ; VTable pointer sanitization (layer 2)
   validate_vtptr(eax)

   ; get vf4() from vtable
   mov edx, [eax+0x0C]

   ; Function type enforcement (layer 1)
   match_type(edx)

   ; call Base1::vf4()
   call  edx
```

Figure 4.2. Illustration of a virtual function call `Base1::vf4()`, including the source code (a), the runtime memory layout (b and c) and the executable code (d), as well as the executable code after deploying VTrust's defense (e). The layer 2 defense is only necessary for applications with writable code sections. The layer 1 defense is sufficient for most applications.

- This solution is *effective*. It provides a fine-grained protection for virtual calls and defeats all different *VTable hijacking* attacks. Case-studies show that it defeats real world exploits.
- The defense is *efficient*. For applications without dynamic generated code, it introduces about 0.72% performance overhead. For other applications, it introduces an overhead of 2.2%.
- It has *modularity support*, allowing us to harden applications module by module. Whenever a new module is added, or the class inheritance tree grows, we do not need to recompile other modules.
- Its program analysis process is lightweight and fast. Unlike other solutions, VTrust does not require the whole class inheritance graph of the applications.

4.3 Threat Model

We assume a powerful yet realistic attacker model. Our model gives the attacker full control over all writable memory and allows arbitrary reads from any readable memory. While being conservative, this assumption is realistic as an attacker may use a vulnerability repeatedly, e.g., spawning threads to attack other threads.

4.3.1 Defense Mechanisms

We assume that protections against code injection and code corruption are in place (e.g., through DEP/NX bits for non-executable regions). All current operating systems make use of DEP. We also assume that attackers cannot remap memory regions (e.g., setting a vtable region writable, or setting a data region executable) to bypass DEP.

Our defense mechanism protects virtual calls only, we therefore assume auxiliary protections for return instructions, indirect jump instructions and any other indirect call instructions are deployed in the application. For example, we assume that the compiler uses fixed base addresses and bound checks when compiling jump tables (e.g., for switch statements). In other words, we assume indirect control transfers except virtual function calls are all well-protected. Attackers cannot hijack the control flow until they reach the virtual function calls. In addition, non-control-data attacks [24, 124] that may lead to control-flow hijacking are out of the scope of this work.

4.3.2 Attack Surface

As specified in the C++ ABI [12], all virtual functions are dispatched through vtables. As shown in Figure 4.2(d), a virtual function is dispatched in three steps: (i) read the vtable pointer from the object, (ii) dereference the vtable pointer (plus the target function's index) to get the target virtual function pointer, and (iii) invoke the target virtual function by indirectly calling the function pointer. The latter two steps may be encoded in one instruction, e.g., `call [eax+0x0C]`.

The vtable mechanism enables virtual function dispatch, access to the base class object, and runtime type information (RTTI). However, it also introduces an attack surface. Objects are usually allocated at runtime and stored in writable memory (e.g., on the heap or stack), so the vtable pointers are untrusted and may be overwritten by attackers. As a result, the target virtual functions read from vtables are untrusted and may be hijacked.

Any successful attack against virtual function calls must either (i) corrupt a vtable or (ii) a vtable pointer, to change the target virtual function pointer. *VTable corruption* attacks modify vtables directly, overwriting function pointers in vtables with attacker-controlled values. For this attack vector the adversary directly controls the target of the virtual function call. Modern compilers place vtables in read-only sections, defeating this kind of attacks by default. The alternate form of attack corrupts the vtable pointer, forcing the program to load the vtable from an alternate location. For this attack vector, the attackers indirectly control the target virtual function calls.

There are two flavors of the latter attack: *VTable injection* attacks and *VTable reuse* attacks, depending on where the overwritten vtable pointers point to. In VTable injection attacks, the adversary injects a surrogate virtual table that is populated with attacker-controlled virtual function pointers. In VTable reuse attacks, the attacker reuses existing vtables out of context (e.g., using a different class' vtable or using an offset to a vtable), or even reuses existing data as vtables.

In practice, VTable injection attacks are the most frequently used VTable hijacking attack vector. Attackers can craft arbitrary vtables containing invalid virtual function pointers pointing to code gadgets (e.g., ROP gadgets) or dynamically generated code sequences (e.g., JIT spraying). Combined with ROP, VTable injection is very easy to launch and reliable.

In VTable reuse attacks, attackers reuse existing vtables or data that looks like vtables (i.e., an array of function pointers). The attacker redirects a virtual call of a class A through attacker-chosen vtables to any function of any class B or any other existing code in memory as long as there is a pointer pointing to that code. This attack is a form of call-oriented programming (through virtual call gadgets).

VTable reuse attacks can bypass defenses like VTint [19]. As shown in one recent CTF (Capture The Flag) event [125], a defense similar to VTint is deployed on one challenge binary, and many teams have successfully bypassed this defense by launching VTable reuse attacks.

Researchers also proved VTable reuse attacks are realistic. The COOP (Counterfeit Object-oriented Programming) attack proposed by Schuster et.al. [23] introduces a specific type of VTable reuse attack. By stitching several virtual functions, attackers may execute arbitrary code. COOP shows that this attack is (i) practical in real applications (e.g., Firefox), and (ii) Turing complete for realistic conditions.

This attack surface is even larger in applications that have writable code (e.g., dynamic generated code). In these applications, even if there are no legitimate vtables or virtual functions in dynamic code memory, attackers may forge them in the dynamic code memory and launch VTable injection and VTable reuse attacks.

Moreover, vtable pointers are not only used in virtual calls. Attackers can overwrite vtable pointers to hijack the RTTI lookup or virtual base object indexing operations. Unlike existing defenses, VTrust protects these other uses from attacks too.

4.4 Design

In this section, we will describe the design of VTrust. We start with an overview of the defense solution, then explain the design of each defense layer.

4.4.1 Overview of VTrust

VTrust uses two layers of defenses to protect the integrity of virtual function calls in-depth, as shown in Figure 4.1 and Figure 4.2.

Virtual Function Type Enforcement. VTrust ensures that the actual runtime type of a virtual function call matches the static type declared in the source code. In this way, attackers cannot divert virtual calls to invalid functions or code. For a particular virtual call site, only functions with the correct type can be invoked, and thus VTable reuse attacks (including the COOP attack) are infeasible. It also stops all VTable injection attacks, if attackers cannot forge functions.

vtable pointer sanitization. VTrust sanitizes vtable pointers at runtime to enforce vtables' validity. So, even if attackers can tamper with the vtable pointers, they can only make them point to the beginning of existing vtables. It thus defeats VTable injection attacks. It also stops most VTable reuse attacks, since attackers can only reuse existing vtables, not part of them or plain data.

The combination of these two layers of defense can protect applications from all VTable hijacking attacks. For applications without dynamically generated code, code pages will not be writable. This stops the attacker from creating her own vtables and functions with forged signatures. In this configuration, VTrust's first layer of defense, virtual function type enforcement, is sufficient to protect against all VTable hijacking attacks.

4.4.2 Virtual Function Type Enforcement

As a first layer of defense, we enforce that the runtime target virtual function matches the type expected in the source code. According to the C++ specification, the derived class will override the parent class' virtual function if and only if it defines a function with the same name, parameter type list (but not the return type), constant and volatile qualifiers, and same reference qualifiers. Moreover, for each particular virtual function call site, the object has a statically declared type (i.e., class), and only virtual functions defined in this class or its sub-classes can be invoked.

VTrust provides a precise protection for virtual calls. It enforces that each virtual call site's static type to matches the invoked virtual function's dynamic type. For the types to match, all of the criteria from the C++ specification's virtual function override requirement must match in addition to the function's class information. Otherwise, the control-flow will be blocked.

More concretely, for a particular virtual function call site, all legitimate runtime target virtual functions meet the following requirements:

- The *function name* must be the same, except for virtual destructor functions and virtual calls that use class member function pointers. The destructor functions always have the same name as the class name (except the leading character ∼). The class member function pointer may be bound to virtual functions with different names.

- The *argument type list* must be identical, except for the hidden argument *this* pointer that references the runtime object on which the virtual function works;

- The *qualifiers* must be identical, including the constant, volatile, and reference qualifiers.

- The *class relationship* must be compatible. The runtime virtual function must belong to a derived class of the static class declared on the virtual function call site. For example, for a virtual call site that expects a virtual function from a specific static class `Foo`, only virtual functions belonging to classes derived from `Foo` can be invoked at runtime.

Omitting any requirements here would expose more attack surface. For example, if we only consider the class information, then attackers may launch attacks (e.g., COOP) to make vtable pointers pointing to the middle of legitimate vtables, to invoke any virtual function of any compatible class.

On the other hand, any virtual function that meets these requirements can be legally invoked at a particular virtual call sites, per the C++ specification. So we cannot further reduce the set of legitimate transfer targets, without breaking the program's functionality. In other words, this layer of defense provides the most fine-grained protection for virtual calls.

This layer also provides a strong protection against VTable hijacking attacks. It prevents the attacker from invoking any virtual function whose type does not match the call site type. Even if attackers can control vtable pointers and make them point to existing vtables or data (i.e., VTable reuse attacks) or even to attacker-controlled vtables (i.e., VTable injection attacks), they cannot invoke arbitrary virtual functions. Instead, they have to either (1) reuse existing virtual functions with correct type, or (2) find someway of forging the virtual function and the type. However, the first bypass is not exploitable, since it is legitimate control flow. The second bypass does not work, if target programs do not have writable code.

In the following, we will discuss some design choices, as well as the advantages and limitations of this solution.

Fast Runtime Matching. It would be slow to validate the type (including name, argument type list and so on) byte by byte at runtime, especially for validating whether a class is derived from another class. To facilitate the runtime type check, we encode the virtual function's type information into a word-sized signature. A simple comparison between the expected signature (a constant) and the runtime target function's signature (a memory value) is sufficient to validate the function's type.

More specifically, the signatures are statically computed (i) per virtual function based on the function's prototype and (ii) at the call sites based on the available call-site information. For each virtual function, we will place the signature together with its code. For each virtual function call site, we will instrument a security check, to match the target function's signature with the one expected on this call site. As the signature generation happens at compile time, VTrust can statically ensure that there are no collisions with other functions.

This signature-based type check provides a good runtime performance, better than existing solutions including SafeDispatch [91], VTV [46] and RockJIT [32]. In general, they have to check whether the runtime vtable or function is in a pre-computed set. As a result, they need to do a slow lookup operation for each virtual call site.

Complex Inheritance Support. When generating the signature for a virtual function, we need to decide its owner class. A virtual function definition may be shared between several classes inherited from a same ancestor. But we can only place one signature with the function. So we choose the **top-most primary ancestor** that defines this virtual function's interface as the owner class.

Assuming a virtual call site expects a virtual function `Foo::func()`, and the virtual function `func()` is first defined in class `Bar` among `Foo's` all ancestors, then this virtual call site will use `Bar` as the owner class to compute the expected signature. For the function `Foo::func()`, it also will use `Bar` as the owner class to compute its signature.

Modularity Support. To compute the signatures, we only need the ancestor information for the target class. As a result, we can generate signatures for virtual call sites and virtual functions when compiling one single module. We can simply analyze one module at a time, and extract the virtual function's type and compute signatures based on the type. This process is simple and fast. Moreover, it has natural modularity support, allowing us to compile target applications module by module. When the class inheritance changes or a new module is added, the default incremental compilation model is sufficient to update everything.

On the other hand, solutions like SafeDispatch and VTV all need the descendant classes information for each class to perform the checks. It thus requires a whole-program analysis, either by compile-time analysis or runtime merging, to get the knowledge of the complete class hierarchy. As a result, our solution has a better compile-time analysis speed and modularity support.

Dynamic Loading Support. Our defense supports dynamic loading. Whether or not a new library is loaded, each virtual function's signature and each virtual call site's expected signature will not change. In other words, even if the class inheritance graph may change, our solution remains effective without the need to update any runtime information.

Solutions like VTV have to load runtime information to update the class inheritance hierarchy, when loading a new module. However, this process can be done with initializer functions themselves, without modifying the dynamic loader.

Limitations. Similar to traditional signature-based CFI solutions, this layer of defense is also vulnerable if attackers can forge code with correct signatures, i.e., the target application has writable code. Our solution VTrust can defeat this kind of attack, by deploying an extra layer of defense: vtable pointer sanitization.

Moreover, the way we determine a virtual function's owner class also leaves some attack surface. For example, suppose (1) class `Grand` is `Parent`'s primary base class, (2) `Parent` is `Child`'s primary base class, and (3) the virtual function `func` is first defined in `Grand`, then for a virtual call site which requires `Child::func`, the virtual functions `Grand::func` and `Parent::func` are also allowed, since they share a same top-most primary base class `Grand`. In practice, this should leave only a small attack surface, as all these virtual functions (e.g., `Grand::func`, `Parent::func` and `Child::func`) by design should perform a similar action (i.e., generating similar output and side-effects for the same input) but only in different ways. VTV also has a similar attack surface [126].

Furthermore, similar to SafeDispatch and VTV, our solution also faces a compatibility issue when an external unhardened library is loaded into the process (e.g., by invoking the system call `dlopen`). For unhardened libraries, there are no signatures associated with its virtual functions, and the security checks we added to the current application will fail if the external virtual function is used. We also deploy a fail-safe error handler similar to the one used in VTV. If an unhardened library is used, we build a whitelist of target virtual functions. When the security check fails, the error handler will go through this whitelist, checking whether the target virtual function is in the list. If not, a security violation alert is thrown.

In summary, this layer of defense defeats all VTable reuse attacks, including the COOP attack. It can also defeat all VTable injection attacks if target applications have no writable code. Essentially, it is a C++-aware fine-grained CFI policy. Comparing with other solutions for C++ programs, it is much faster in compile-time analysis and runtime execution. It also has natural modularity support without added complexity, as well as a good dynamic loading support. So, we strongly recommend deploying this layer of defense in practice.

4.4.3 VTable Pointer Sanitization

As discussed in the previous section, for applications supporting dynamic code generation, attackers may bypass the first layer of defense by launching VTable injection attacks and utilizing dynamically generated code to forge virtual functions with correct types (i.e., signatures). As a result, we introduce a second layer of defense, to limit the source of virtual functions. More specifically, we sanitize the vtable pointers to enforce that they point to valid vtables, and thus only existing virtual functions can be invoked on a particular virtual call site.

One straightforward solution is to enforce the integrity of vtable pointers. But unlike vtables themselves, vtable pointers cannot be set to read-only, because these pointers are usually members of objects that are on the writable heap or stack. In order to enforce their integrity, we have to track data-flow at runtime, and prevent all memory write operations from overwriting vtable pointers. Obviously, this solution results in unacceptable performance overhead.

Instead, we enforce the validity rather than the integrity of vtable pointers. Our solution enforces the runtime vtable pointers to be `valid`, even if attackers have tampered with them. Existing defenses all fall into this category. VTint [19] checks whether the target vtable is read-only and T-VIP [127] works in a similar way. They both fail to defeat some attacks (e.g., COOP). SafeDispatch [91] and VTV [46] check whether the runtime vtable is in a pre-computed legitimate vtable set, introducing high performance overhead.

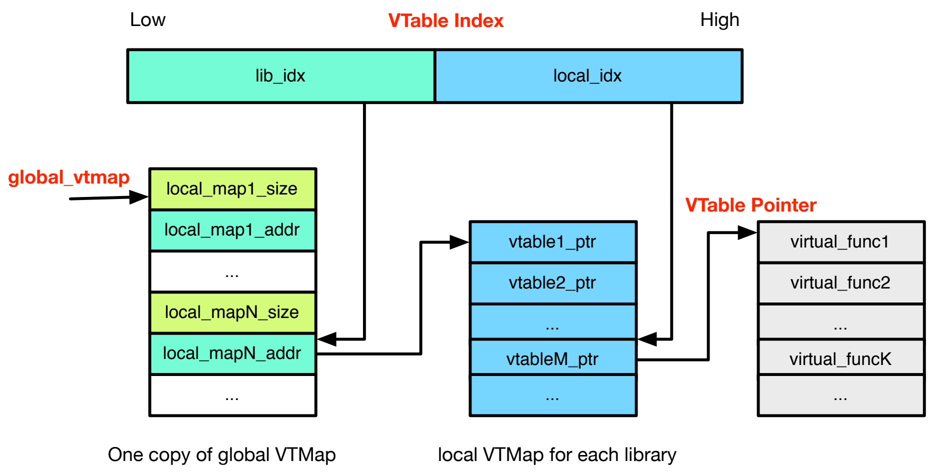


Figure 4.3. VTable pointer sanitization solution. Each vtable pointer will be encoded to a vtable index consistinf of a `lib_idx` and a `local_idx`. A global VTable pointer map (pointed by `global_vtmap`) will be used to decode vtable indexes.

We propose a novel solution to enforce vtable pointers' validity, by encoding vtable pointers into vtable indexes when initializing them and decoding vtable indexes when they are accessed (e.g., for virtual calls). In this way, even if attackers can control the vtable pointers in objects, these pointers will first be decoded before being used in virtual calls. As a result, only valid vtables can be used to perform the virtual calls.

A straightforward encoding and decoding solution works in this way: we maintain a whitelist of all legitimate vtable pointers, encode each vtable pointer to an index of this whitelist when assigning vtable pointer to objects, and decode the vtable pointers (i.e., indexes) that are read from the runtime objects to the original pointers by using the whitelist. This solution is simple and can defeat all VTable injection attacks. However, it does not support shared libraries. For example, when a vtable is defined in a shared library, its pointer can hardly be encoded to a unique index, because this library may be used in different applications, i.e., this vtable pointer may be recorded in several whitelists.

Instead, we use a separate whitelist (denoted as local VTMap) for each single library, as well as a global map (denoted as global VTMap) to perform the vtable pointer encoding and decoding. As shown in Figure 4.3, each vtable pointer will be encoded into a vtable index (i.e., an integer of the bit width same as the platform), consisting of two sub-indexes of the same bit width: (1) a `lib_idx` that represents the index (to the global VTMap) of the library that uses this vtable; and (2) a `local_idx` that represents the index (to the local VTMap) of the vtable inside the library. When decoding a vtable index, we use its lower half (i.e., `lib_idx`) to retrieve the local VTMap's address from the global VTMap, then use its upper half (i.e., `local_idx`) to retrieve the vtable pointer from the local VTMap.

When compiling a single library, we can build the local VTMap for it, and assign an index `local_idx` to each vtable that is **used** in current library. These indexes are all statically assigned, so that the library can be shared among different applications without modifications. Then we can statically compute the library's index `lib_idx`, either by (1) manually specifying or (2) automatically scanning existing libraries indexes and computing a different one.

The global VTMap of each application must be initialized at runtime. Whenever a library is loaded, its local VTMap's address will be registered in this global VTMap, at the specific entry numbered with `lib_idx`. We also store the size of the local VTMap into the global VTMap, for further bound checks when accessing the local VTMaps. As a result, this solution provides a good support for incremental compilation and dynamic loading. Similar to virtual function type enforcement, it may cause incompatibility issues when working with unhardened libraries that have vtables.

This layer essentially provides a whitelist protection. Although attackers may overwrite the vtable pointers, the pointers will be decoded (and therefore validity checked) before being used in virtual calls. As a result, this defense defeats VTable injection attacks. We emphasize that, this layer of defense also provides partial protection against VTable reuse attacks, even if the first layer of defense is absent. It enforces that only legitimate vtables can be used for virtual calls. So attackers can only reuse existing vtables, rather than any other data or the middle of existing vtables on which the COOP attack is based. However, in theory, attackers may still

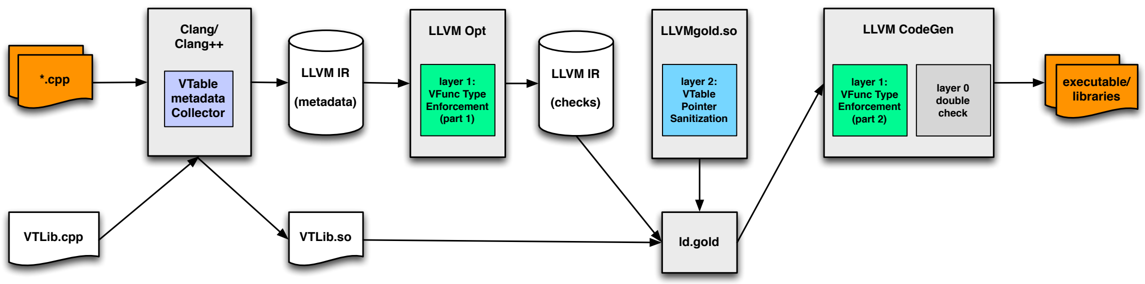


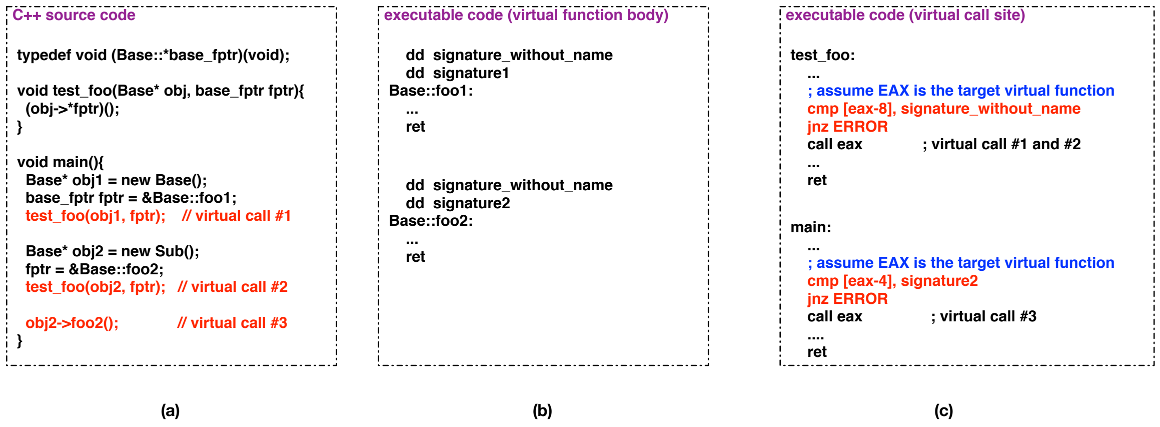
Figure 4.4. Illustration of VTrust's workflow. The first layer of defense is implemented as a compile-time optimization pass and a code generation step. The second layer of defense is implemented as a link-time optimization pass.

launch some VTable reuse attacks by only reusing existing vtables. Therefore, both defense layers are needed to prevent attacks.

Limitations. Our solution for selecting `lib_idx` can avoid conflicts when all code is compiled on the same machine. If code is compiled on different machines, it is not possible to statically determine a unique `lib_idx` by manual specification or by scanning for other libraries. This is an engineering challenge we plan to address in future work.

One possible solution is to have the library loader resolve the conflicts like it does for relocation. In fact, conflicts should not be that common in practice, as the number of possible values of `lib_idx` is much larger than the number of libraries a typical application links against. Another solution is to use the library name rather `lib_idx` to index the local VTMap, which eliminates the conflicts but makes the runtime decoding a little slower.

Alternative VTable Pointer Sanitization Solutions. We also tested a range check solution to validate VTable pointers. It records all legitimate VTable sections' address ranges, updates this information when libraries are loaded and unloaded, and validates whether the runtime VTable falls in any of these address ranges. However, this alternative is not precise as the VTable encoding and decoding solution we discussed earlier, and also has a higher performance overhead.



C++ source code

```
typedef void (Base::*base_fptr)(void);

void test_foo(Base* obj, base_fptr fptr){
  (obj->*fptr)();
}

void main(){
  Base* obj1 = new Base();
  base_fptr fptr = &Base::foo1;
  test_foo(obj1, fptr);    // virtual call #1

  Base* obj2 = new Sub();
  fptr = &Base::foo2;
  test_foo(obj2, fptr);   // virtual call #2

  obj2->foo2();            // virtual call #3
}
```

(a)

executable code (virtual function body)

```
    dd  signature_without_name
    dd  signature1
Base::foo1:
    ...
    ret


    dd  signature_without_name
    dd  signature2
Base::foo2:
    ...
    ret
```

(b)

executable code (virtual call site)

```
test_foo:
    ...
    ; assume EAX is the target virtual function
    cmp [eax-8], signature_without_name
    jnz ERROR
    call eax           ; virtual call #1 and #2
    ...
    ret


main:
    ...
    ; assume EAX is the target virtual function
    cmp [eax-4], signature2
    jnz ERROR
    call eax            ; virtual call #3
    ....
    ret
```

(c)

Figure 4.5. Illustration of the class member function pointer issue and solution. Here, virtual functions `Base::foo1` and `Base::foo2` have a same function type but different name. We instrument an extra signature that is computed without function name before the function body. For virtual call sites that use class member function pointers, we compare this special signature instead of the signature with function names.

4.5 Implementation

We have implemented a prototype of VTrust using the LLVM [128] compiler suite version 3.4 for x64.

Figure 4.4 shows the workflow of VTrust. In general, there are four steps in our implementation.

First, we collect vtable related information of the current compilation unit in the compiler frontend (i.e., Clang++), including all virtual functions, all vtable constants, all virtual call instructions, as well as all vtable assignment and read operations. It is worth noting that we do not need to collect the class inheritance information during the compilation, unlike SafeDispatch [91] and VTV [46].

This information is kept in the form of LLVM metadata and function attributes, and passed to the optimizer and linker. Some optimizations may remove or replace some LLVM instructions, e.g., the instruction combination optimization. When instructions are removed, some LLVM metadata may be discarded, which causes problems in our link-time analysis. In our prototype, we keep multiple copies of metadata in different instructions when compiling, as well as in the compilation module. During the link-time analysis, we will perform a cross verification to detect such metadata missing issues, and recover the metadata automatically based on other copies.

Second, we instrument type signature checks before virtual call instructions (i.e., the first layer of defense) when performing compile-time optimization.

Third, we utilize LLVM's link-time optimization support (based on the linker's gold plugin feature) to deploy our second layer of defense. More specifically, we encode and decode the vtable pointers before use.

Finally, we instrument type signatures before the body of each virtual function (for the first layer of defense) during code generation. We also verify that all vtables are placed in read-only sections (i.e., layer 0 defense).

We also provide a runtime library `VTLib.so` for some runtime APIs used by the security checks that we instrumented. All the hardened modules are then linked together by the gold linker, generating the final executable or libraries.

4.5.1 Virtual Function Type Enforcement

In the first layer of defense, VTrust will enforce that each runtime virtual function has a matching type (i.e., the name, argument type list, qualifiers, and class information) with the one expected on a virtual call site. To make the runtime check more efficient, we encode the type information into a word-sized hash value. This hash value is used as the signature for each virtual function, and is compared before the virtual function call site. If they do not match, a security violation is detected and the program is terminated.

For each virtual function, we collect its type information from the frontend, compute its signature, and embed this signature right before the function body when generating the native code for this function.

For each virtual function call site, we collect the expected type and compute the expected signature. VTrust will then instrument a security check before this virtual call to match the function's runtime signature against the expected one.

Computing the signature based on a type information is easy, but collecting the type information is not. The qualifiers and argument list type are deterministic. However, the virtual function name and the class information are not.

- First, we cannot get a meaningful function name for virtual destructor functions. The derived class' virtual destructor function will overload the parent class' virtual destructor function, i.e., they use the same slot in the vtables. But they have different function names. For example, assuming `Foo` is derived from `Bar`, then their virtual functions use a same slot in the vtable, but with different names: ∼`Foo` and ∼`Bar`.

- Second, we cannot use the virtual function's owner class' information (e.g., name) to compute the signature. For a virtual function call site, we only know a static class. At runtime, the target virtual function may belong to another class. These two classes are often different and their names do not match.

- Third, we cannot get a meaningful function name for virtual call site that uses class member function pointers. As shown in Figure 4.5, the virtual call in function `test_foo` does not have a meaningful name.

For the first two issues, similar to choosing owner class for virtual functions as discussed in Section 4.4.2, we choose the destructor name of the top-most primary ancestor class among all its ancestors that defined virtual destructors.

For each virtual function in a class, there is one slot in the per-class vtable. If a derived class overrides a virtual function, then the overrider takes the slot at the same offset in the vtable of the derived class. The text in black in Figure 4.6 shows the basic vtable building algorithm used in LLVM. In general, the primary base class initializes the vtable first, and the derived class then updates the vtable slots with overrider functions and extends it with new functions.

Based on this basic algorithm, for each virtual function, we can easily retrieve the function name and class name of the top-most primary class that defines this function.

```
addVTableMethods(TgtClass):
  BaseClass = PrimaryBase(TgtClass)
  # recursive invocation
  addVTableMethods(BaseClass)
  # all overrider virtual functions
  for overrider in TgtClass:
    updateVTableEntry(overrider)
    oldFunc = get_overriden_func(overrider)
    if isVirtualDestructor(overrider):
      ancestorFuncName = get_func_name(oldFunc)
      register_func_name(overrider, ancestorFuncName)
    ancestorClassName = get_class_name(oldFunc)
    register_class_name(overrider, ancestorClassName)
  # all new virtual functions
  for newFunc in TgtClass:
    appendVTableEntry(newFunc)
    if isVirtualDestructor(newFunc):
      register_func_name(newFunc, '~'+TgtClass)
    register_class_name(newFunc, TgtClass)
```

Figure 4.6. VTable Building and Type Collection Algorithm (Python-style pseudocode). Text in orange is the code we instrumented to collect type information for virtual functions.

As shown in Figure 4.6, for each overrider function, we use the overridden function's name and class information. It is worth noting that, we use the information from the top-most primary class that first defines the virtual function, not from the top-most primary class of the static class (declared in the virtual call site). These two may be different since an object may contain several vtables, due to multiple inheritance.

In this way, we can compute the signature of the type without the complete class inheritance tree. We can collect all the information when the compiler builds the vtable. It makes our compile-time analysis very fast. Moreover, if the programmer extends the class inheritance tree in the future, we do not need to recompile existing modules.

For the third issue, we instrument special security checks for virtual call sites that use class member functions. It reads the signature from a different offset to the function body, which is computed without the function name information. This signature still has the class information, as well as the function prototype information, and thus is strong enough.

4.5.2 VTable Pointer Sanitization

In the second layer of defense, VTrust will sanitize vtable pointers at runtime, to enforce they are *valid*. More specifically, we will encode vtable pointers when they are assigned to objects, and decode them when they are used.

VTable Pointer Encoding. When analyzing a module (i.e., a library or executable since we are working on link-time optimization), we discover all vtable pointer assignment operations, including (1) assigning vtable pointers to runtime created objects in constructor functions; (2) filling vtable construction tables (VTT, an auxiliary data structure for complex class inheritance) with vtable pointers; (3) assigning vtable pointers to static `typeinfo` objects that are used for RTTI; and (4) assigning vtable pointers to some constant static objects. It is worth noting that, for the latter three cases, there are no assignment instructions. Instead, the compiler will directly put the vtable pointer at the proper location.

For each of these vtable pointer assignments, we will replace the pointer with a (statically computed) constant index. As discussed in Section 4.4.3, this index consists of two parts: an index `local_idx` to the local library's VTMap, and the index `lib_idx` to the global VTMap. We use the order of each VTable pointer in the library as its `local_idx`, and use the build order of each library as its `lib_idx` (after scanning existing libraries and fixing conflicts). As a result, we can compute the constant index for each vtable pointer at link-time. After encoding, the vtable pointer of each runtime object will be a constant integer.

We create a global VTMap array, which will at runtime hold all loaded libraries' local VTMaps' addresses, in the support library `VTLib.so` for each application. For each library, we add an initialization function and a local VTMap array. The local

VTMap stores addresses of all vtables used in the current library. The initializer function is invoked automatically when the library is loaded, and registers the local VTMap to the global VTMap. More specifically, it updates the global VTMap's `lib_idx`-th entry to store the size of the local VTMap and its runtime address. This runtime update operation temporarily maps the global VTMap as writable when loading a library. Most applications only load libraries during initialization, therefore the risk of being attacked during this short time window is low.

VTable Pointer Decoding. After encoding the VTable pointers, we have to decode them at runtime, when the object's VTable pointer is read out and used for (1) accessing virtual function pointers for virtual calls; (2) accessing offsets of virtual base objects; or (3) accessing RTTI information.

To decode vtable pointers, we first parse the vtable index that is read from the runtime object into two parts: `local_idx` and `lib_idx`. Then we use the library index `lib_idx` to access the global VTMap, and get the library's local VTMap's address and size.

If the `lib_idx` is larger than the global VTMap's size, or the `local_idx` is larger than local VTMap's size, we raise a security violation exception and terminate the program. Otherwise, the `local_idx` is used to access the local VTMap, to get the value of the vtable pointer. Finally, this decoded pointer is used for virtual calls. In this way, even if attackers can overwrite the runtime vtable pointers, e.g., by exploiting use-after-free vulnerabilities, only vtables in the local VTMaps can be used in virtual calls after decoding.

This solution changes the representation of vtable pointers and will cause incompatibility issues if some libraries are not instrumented. But it covers all vtable uses and protects them from attacks. For example, if there are custom virtual calls written in assembly, as shown in the evaluation, traditional defenses may fail to protect them.

4.6 Summary

VTrust is a layered defense mechanism for preventing vtable-based attacks in C++. Virtual function type enforcement ensures that the type signature of the dynamic callee function matches the expected static signature at the call site. vtable pointer sanitization ensures that the vtable pointer of an object points to a valid vtable. Together these approaches prevent vtable attacks even when the application uses JIT.

VTrust is implemented as an LLVM pass and runtime library. The pass identifies every virtual function call and inserts the virtual function type enforcement check before each one (and optionally the vtable pointer sanitization check). The allowed virtual function targets are determined based on the class hierarchy and function signature. The pointer sanitization check is implemented by enumerating the valid vtables and checking if the vtable pointer points to one of them.

5 DATA CONFIDENTIALITY AND INTEGRITY

5.1 Abstract

Applications written in C/C++ are prone to memory corruption, which allows attackers to extract secrets or gain control of the system. With the rise of strong control-flow hijacking defenses, non-control data attacks have become the dominant threat. As vulnerabilities like HeartBleed have shown, such attacks are equally devastating.

Data Confidentiality and Integrity (DCI) is a low-overhead non-control-data protection mechanism for systems software. DCI augments the C/C++ programming languages with annotations, allowing the programmer to protect selected data types. The DCI compiler and runtime system prevent illegal reads (confidentiality) and writes (integrity) to instances of these types. The programmer selects types that contain security critical information such as passwords, cryptographic keys, or identification tokens. Protecting only this critical data greatly reduces performance overhead relative to complete memory safety.

Our prototype implementation of DCI, DataShield, shows the applicability and efficiency of our approach. For SPEC CPU2006, the performance overhead is at most 16.34%. For our case studies, we instrumented mbedTLS, astar, and libquantum to show that our annotation approach is practical. The overhead of our SSL/TLS server is 35.7% with critical data structures protected at all times. Our security evaluation shows DataShield mitigates a recently discovered vulnerability in mbedTLS.

5.2 Introduction

Preventing memory vulnerabilities in C/C++ code is a well-researched topic, but the widely adopted protection mechanisms focus on control-flow hijack attacks, neglecting non-control-data attacks. In a control-flow hijack attack, the attacker diverts the program's intended control-flow. However, in non-control-data attacks, the program execution follows a valid path through the program, but the data is controlled by the attacker.

Mature control-flow defenses – such as Stack Cookies [25], Address Space Layout Randomization (ASLR) [15], DEP [26], and Control-flow Integrity (CFI) [33] – are widely deployed. Next generation control-flow protection mechanisms such as CFI and Code-Pointer Integrity [118] are widely researched and are being deployed in production systems. For example, Microsoft's Control-Flow Guard [100] and LLVM-CFI [129] are available in production compilers. See Chapter 3 for a detailed survey of previous work in CFI. As these defenses improve, attackers will follow the path of least resistance and shift to non-control-data attacks, which are not prevented by any of the above mechanisms. As the high-profile HeartBleed bug showed [130], non-control-data attacks are as harmful as control-flow hijack attacks [24,107,131]. In fact, the Control-Flow Bending [62] non-control-data attack can achieve Turing-complete computation even with CFI protection in place.

Complete memory safety mechanisms stop both control-flow hijack attacks and non-control-data attacks, but have not been widely adopted. CCured [5], Cyclone [4], WIT [132] and SoftBound [9] are all different approaches that retrofit C/C++ with some form of memory safety, but (i) impose prohibitively high performance overhead and (ii) may run into compatibility issues with legacy code. In general, determining if a C/C++ program is memory safe is undecidable, so any complete protection mechanism must fall back, at least in part, to runtime checks. This requirement puts an intrinsic lower bound on the overhead of any complete memory protection mechanism. For this reason, we argue that program-wide precise memory protection imposes too much overhead for wide adoption.

We introduce Data Confidentiality and Integrity (DCI) to address these challenges. DCI limits the performance overhead by tagging data as either sensitive or non-sensitive, enforcing precise spatial and temporal safety checks only on sensitive data. For non-sensitive data, DCI only requires coarse bounds and imprecise temporal safety. Coarse checks can be performed with low overhead, avoiding expensive metadata lookups. The programmer specifies which data is fully protected and the DCI compiler and runtime enforces the security policy. In this way, DCI allows the programmer to control the overhead/protection trade-off. The DCI policy is:

(i) *Memory safety for sensitive data:* Pointers to sensitive objects can only be dereferenced if they point within the bounds of the associated (valid) memory object.

(ii) *Sandboxing for non-sensitive data:* Pointers to non-sensitive objects can be dereferenced if they point anywhere except a sensitive memory object.

(iii) *No data-flow between sensitive and non-sensitive data:* Explicit data-flow between sensitive memory objects and non-sensitive memory objects is forbidden. Sensitive and non-sensitive objects must reside in disjoint memory regions.

This policy provides spatial and temporal memory safety for sensitive data at runtime, and ensures both confidentiality and integrity of the sensitive data.

Our implementation of DCI, DataShield, consists of three key parts. First, DataShield provides a language the programmer uses to specify protection. The language is a small set of annotations that are added to existing C/C++ code. Second, compiler instrumentation identifies the sensitive variables and associated data-flows, and rewrites the program with additional security checks. Third, the runtime system creates and maintains the metadata.

Table 5.1. Performance Overhead of existing memory safety mechanisms (as reported by the authors).

Protection	Benchmarks	Avg. Overhead (%)
CCured	SPECINT95	81.75
CCured	Olden	44.33
Cyclone	N/A	38.25
SoftBound	SPEC2000, Olden	67

5.3 Motivation

In the following sections we: (i) discuss the performance overhead of memory safety, (ii) explain the relationship between memory safety, integrity, and confidentiality, and (iii) show that non-control-data attacks are not prevented by existing low-overhead defense mechanisms.

5.3.1 Memory Safety Overhead

The absence of memory errors is in general undecidable statically for C/C++. Any mechanism guaranteeing memory safety must (at least partially) fall back on dynamic checks and maintain metadata. While several mechanisms have been proposed with varying efficiency, they all require a runtime component that imposes some overhead [16]. Table 5.1 shows the average overhead of four existing complete safety mechanisms (as reported by the authors). The purpose of the table is not to compare the techniques with each other, but to show there is significant overhead for each mechanism. Straightforward comparison is impossible, since they use different benchmarks, computer architectures, software versions, and operating systems.

5.3.2 Memory Safety, Integrity, and Confidentiality

Data integrity and confidentiality are ensured by enforcing memory safety for both reads and writes through pointers. An integrity violation occurs when an attacker uses a pointer to write outside the intended memory object, while a confidentiality violation occurs when the attacker uses a pointer to read outside the intended object.

5.3.3 Non-Control-Data Attacks

To date, security researchers and attackers have primarily focused on control-flow hijack attacks. Such attacks were effective and simple to execute before protection mechanisms were in place. Stack Cookies [25], ASLR [15], and DEP [26] enforce code integrity and give probabilistic guarantees against code reuse attacks. The adoption of these defenses mitigated simple attacks, e.g., stack smashing or shell code injection. Attackers responded by moving to code reuse attacks, such as return-oriented programming [116, 117]. Following the same pattern, when CFI (or CPI) mechanisms [32, 43, 45, 46, 91, 98, 103, 118] become widely deployed, attackers will shift from control-flow hijack attacks to the next available form of attack, namely non-control-data attacks.

Non-control-data attacks [62, 107, 131] are harder to detect because only the data differs between benign and malicious executions. For example, to exploit the HeartBleed bug, the attacker sends a malicious request. For a benign request, the server echoes back the message. However, the attacker's malicious request causes a buffer overflow, because the value of the length field in her request is longer than the content [130]. The result is that the server sends back unintended additional data. The server's response includes the original message plus the data past the end of the message buffer. The exploit is devastating when the data after the buffer is sensitive, like a private encryption key.

While HeartBleed is an example of a confidentiality violation, non-control-data attacks can also result in integrity violations. For example, overwriting the user ID variable with the value of the administrator's user ID is one way to perform a privilege escalation attack. To prevent all non-control-data attacks we need both confidentiality and integrity of sensitive data. Neither of these attacks are caught by any of the mentioned defense mechanisms. Since non-control-data attacks can slip past these defenses and complete memory safety is too costly, we need new tools targeted at providing data protection with low overhead.

5.4 Threat Model

Our threat model assumes that the attacker can exploit the original benign program to perform arbitrary reads and writes through attacker-controllable memory errors. We assume the system is protected from code injection (or modification) attacks by DEP [26]. We assume that another protection mechanism is in place in the system to prevent control-flow hijacking. Side-channel attacks that leak information from the sensitive region are out of scope. The operating system, our compiler, and our instrumentation are in the trusted computing base (TCB).

5.5 Design

The key idea behind DCI is that we need protection against non-control-data attacks, but not all data in a program is equally important in terms of security. Relaxing the protection on the non-sensitive memory objects allows us to reduce the overhead of our protection mechanism relative to complete memory safety. Specifically, the non-sensitve checks are more efficient, and no metadata is tracked for non-sensitive objects. DCI ensures that despite the presence of memory vulnerabilities, the confidentiality and integrity of sensitive data is preserved.

Determining the subset of sensitive data is a pivotal decision. The current DCI implementation uses a programmer specification to describe which data is sensitive. While it would seem ideal to instead use a sophisticated static analysis to automatically infer sensitive data, the programmer intuitively has domain knowledge that is not available to a compiler or a static analyzer.

DCI separates the memory into two regions, a precisely protected region for sensitive data and a coarsely protected region for non-sensitive data. The mechanism enforces this separation and forbids any pointers from the non-sensitive region to the sensitive region. The DCI policy requires sensitive data to be precisely bounds checked in the same manner as complete memory safety. However, the policy relaxes the requirement for non-sensitive data. Pointers to non-sensitive data can be dereferenced if they point anywhere except a sensitive memory object. Such coarse bounds checking for non-sensitive data lowers overhead. Our policy requires coarse bounds checking (sandboxing) non-sensitive data as otherwise, an attacker could access sensitive data through a corrupted non-sensitive pointer. Thus, we must have a check on every pointer dereference, regardless of whether that pointer points to sensitive or non-sensitive data [14].

Finally, data-flow between variables in different regions is forbidden. This policy applies to primitive variables, pointers, and contents of aggregate types. This rule means that an object itself (`struct`, array, etc.) and all its sub-objects (members, elements, etc.) have the same sensitivity. This property is enforced at compile time by separating all variables into one of two disjoint sets – the sensitive set or the non-sensitive set.

Any attempt by the attacker to modify the sensitive data with a non-sensitive pointer, as well as a buffer overflow or use-after-free inside the sensitive region, causes our system to abort the program. An attacker may still modify sandboxed data through non-sensitive pointers, as the security policy does not protect such data.

5.5.1 Determining the Sensitive Types

DCI requires the programmer to specify which data is sensitive. A recent study showed that annotation burden is a primary factor affecting developers use of contracts [133], which are similar to our annotations. A survey of 21 open source projects that use contracts found that over 33% of program elements (classes or functions) were annotated [134]. For large code bases, annotating functions individually does not scale.

To mitigate the annotation burden problem, we design our protection specification to be type-based. When the programmer annotates a type, all instances of that type are sensitive. Annotating a type in effect annotates every function that uses that type as a parameter and every local or global variable of that type. This way, a small number of annotations give a specification for the entire program.

For example, to mark all instances of `struct circle` in the entire program as sensitive, the programmer would mark any instance with our provided annotation:

```
__attribute__((annotate("sensitive")))
struct circle c;
```

This has the effect of enabling sensitive protection to:

1. instances of `struct circle` and their contents;
2. instances of other types that contain `struct circle` as a member.

Implicit Sensitivity

A key design decision is how to handle interactions between non-sensitive and sensitive variables. The compiler can reject the program (and display an error message to the programmer) or it could automatically make the variable implicitly sensitive. To minimize annotation burden DataShield defaults to the second option.

Implicit sensitivity potentially leads to more variables being sensitive than the programmer expects, but greatly reduces the number of required annotations. To mitigate the problem of the programmer not knowing exactly what variables are sensitive, the compiler can report a list of sensitive variables and types to which protection was propagated. The programmer can then either iteratively modify the program or the sensitivity specification.

5.5.2 Sensitivity Rules

This section formalizes the sensitivity rules. To prevent data-flow between sensitive and non-sensitive variables, the first rule forbids direct interaction between variables of different sets.

1. x *[op]* y is only allowed when
   sensitivity(x) = sensitivity(y)

where *[op]* can be any binary operator. Unary operators do not change the sensitivity of their operand.

Constants take on the protection of their other operand:

2. sensitivity(x *[op]* c) $\leftarrow$ sensitivity(x)

where c is any constant.

These rules are sufficient for primitive values, but there are additional rules for pointers:

3. For pointers p and q, if q is *based on* p
   sensitivity(q) $\leftarrow$ sensitivity(p)

4. For pointer p and its pointee *p,
   sensitivity(p) $\leftarrow$ sensitivity(*p)

Here "based on" means that q is the result of pointer arithmetic on p. The additional pointer rules mean that the contents of a `struct` have the same sensitivity as a pointer to the `struct` and array elements have the same sensitivity as the array itself.

The final policy rules are:

5. Sensitive pointers can only be dereferenced within the bounds of a valid sensitive object.

6. Non-sensitive pointers cannot be dereferenced within the bounds of a sensitive object.

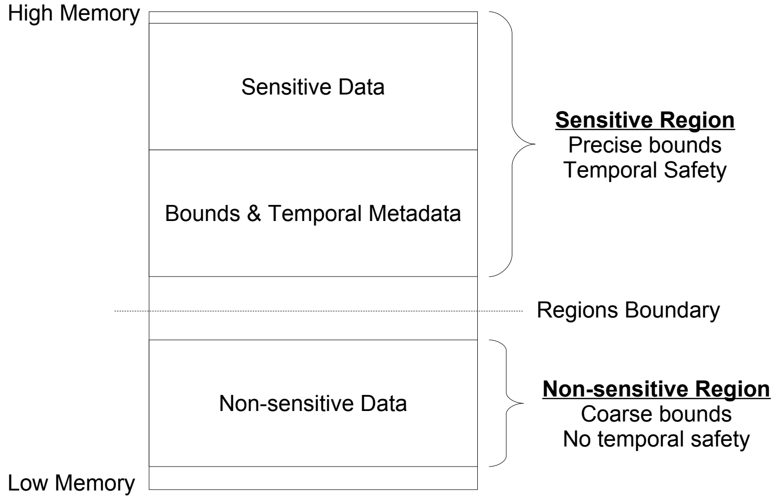


Figure 5.1. DataShield's runtime memory layout for sensitive and non-sensitive data. The sensitive region has a strict security policy that leads to instrumentation overhead, and the non-sensitive region has a relaxed security policy with minimal overhead.

Rule 5 prevents overflows between sensitive objects, and as side effect prevents overflows to non-sensitive variables as well (since they are out of bounds of the original sensitive object). It also prevents dereferencing pointers to sensitive objects when they point to unallocated memory (temporal errors). Rule 6 prevents overflows from non-sensitive objects to sensitive objects.

5.5.3 Enforcement

At runtime, DCI divides memory into two regions, associates metadata with each sensitive pointer in the program, and performs a check before each pointer dereference. A conceptual memory layout is shown in Figure 5.1.

The non-sensitive region simply contains all the non-sensitive data. The sensitive region holds both sensitive data and the metadata for the sensitive pointers.

Whenever a new sensitive memory object is allocated, the bounds of that object and a temporal check key are recorded as metadata. When a new non-sensitive object is allocated, no metadata is recorded because non-sensitive pointers have the implicit bounds of anywhere except a sensitive object and no temporal safety. Both sensitive and non-sensitive allocations use a custom allocator to ensure the new object resides in the appropriate region. Pointers in sensitive memory are restricted to point within the bounds of their intended memory object by checking their value against the associated metadata, while non-sensitive pointers may point anywhere in the non-sensitive region. The relaxed policy for non-sensitive pointers leads to more efficient checks and less metadata.

5.6 Implementation

DataShield implements DCI by extending the LLVM compiler. At a high-level, the compile-time portion of DataShield consists of collecting sensitive types, identifying sensitive variables and inserting new instructions to enforce the DCI policy. The new instructions create metadata, and perform sensitive or non-sensitive bounds checks. The runtime initializes the metadata data structure and implements the checks and region-based allocators. The compiler portion is implemented as an LLVM pass in 4,500 lines of C++ code. The DataShield runtime is 1,000 lines of C code.

5.6.1 Identifying Annotated Types

Most C/C++ compilers, including GCC and clang/LLVM, already have an annotation facility built in, requiring only minor modifications to support the type annotations that DataShield requires.

Our first pass scans all the code in a module, recording the annotated types as sensitive. When the programmer adds an annotation to her code, it appears in the LLVM IR as metadata. Identifying the set of sensitive types is as simple as parsing the metadata.

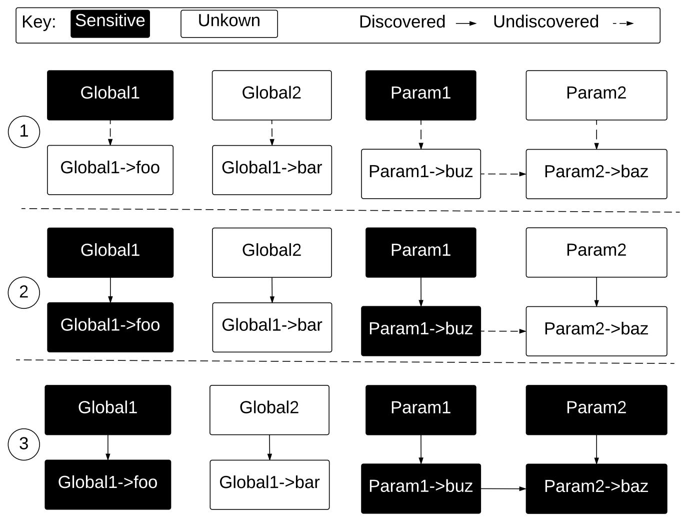


Figure 5.2. An example the sensitivity analysis. In iteration 1, only the sensitivity of globals and of function parameters are known. Then, the analysis applies abstract interpretation over the function body's instructions, discovering new relationships and adding variables into the sensitive set. The analysis concludes when a fixed point is reached in iteration 3. Arrows indicate that connected boxes must be in the same set according to the DCI policy rules.

5.6.2 Identifying Sensitive Variables

Once our implementation has identified the sensitive types, the compiler locates variables of those types. Declared variables of the sensitive types form the roots of the data-flow graph. The complier explores every execution path adding new variables to the sensitive set.

The data-flow analysis that finds all the explicitly and implicitly sensitive variables is inter-procedural and context-sensitive. It is a fixed-point analysis where we iteratively add more variables to the sensitive set. This analysis is depicted in Figure 5.2.

At the start of the analysis, only global variables and function arguments of the sensitive types are in the sensitive set. Variables that have data-flow with other variables in the sensitive set are unioned into the sensitive set. In our formalization, lowercase letters denote variables and uppercase letters denote types. We use the notation $x \in \mathit{Sens}$ to denote the variable x is in the sensitive set, and the notation $\mathit{sensType}(T) = \mathit{true}$ to denote that any of the following are true:

- T is annotated as sensitive;
- pointers to T are annotated as sensitive;
- T is a member of another type U that is annotated as sensitive type;
- T has a member of another type U that is annotated as sensitive type.

Note that the definition is recursive. For instance assume (i) a program has types T, U, and V, and (ii) V is a member of U and U is a member of T. Then if $\mathit{sensType}()$ is true for any of the types, then it is true for all three.

Pointers to primitive types (e.g., `void*, int*, char*, or float*`) are handled specially. We assume that the programmer does not intend to make *all* instances of, e.g., `char*` sensitive. If the programmer annotates a type which has a `char*` member, only `char*` based on the (sensitive) parent type are *explicitly* sensitive. Instances of `char*` that are not members of sensitive types are considered non-sensitive initially, but can become *implicitly* sensitive when our analysis discovers data-flow with other sensitive variables. This approach reduces the amount of primitive types (and broadly data at runtime) that need to be sensitive.

The analysis proceeds by abstract interpretation over the function's LLVM IR instructions, applying the transfer function in Figure 5.3. For brevity, we show only a subset of interesting instructions. The abstract domain for a given variable is whether it belongs to the sensitive set or not. Once a variable belongs to the sensitive set it can never leave the set. The rest of the instructions simply union the sensitivity of their operands. The one exception is LLVM's `CallInst`, as calling a function with mixed sensitivity arguments is legal under our policy. For instance, it could be that the mixed sensitivity arguments do not interact with each other inside the function body. Alternatively, if the mixed sensitivity arguments do interact, the non-sensitive arguments will be promoted to implicitly sensitive when the callee function is analyzed.

$$\textbf{T x = LoadInst(T* a)}$$
$$\text{if } \mathit{sensType}(T) \lor a \in \mathit{Sens} \lor x \in \mathit{Sens}$$
$$\text{then } \mathit{Sens} \cup \{x, a\}$$
$$\textbf{StoreInst(T x, T* y)}$$
$$\text{if } \mathit{sensType}(T) \lor x \in \mathit{Sens} \lor y \in \mathit{Sens}$$
$$\text{then } \mathit{Sens} \cup \{x, y\}$$
$$\textbf{T x = BitcastInst(U a)}$$
$$\text{if } \mathit{sensType}(T) \lor \mathit{sensType}(U) \lor x \in \mathit{Sens} \lor a \in \mathit{Sens}$$
$$\text{then } \mathit{Sens} \cup \{x, a\}$$

Figure 5.3. Abstract interpretation transfer function for finding implicitly sensitive variables. Other instructions simply propagate sensitivity.

The `BitCastInst` instruction propagates sensitivity in both directions and never removes sensitivity. If a non-sensitive pointer is cast to sensitive, or a sensitive pointer is cast to non-sensitive, both the original and cast pointers are considered sensitive.

When a callee function with sensitive arguments is discovered by the analysis, we clone a new version of that function which will be reanalyzed and rewritten with the appropriate sensitivity. At the original call site, the call to the original function is replaced with a call to the newly cloned function. The per-call site cloning is crucial because the same function may be called in different contexts with different argument sensitivities. For example, let there be a function with the signature: "`void foo(void* p);`". It is valid to call `foo` with the parameter `p` as any pointer type, and more relevantly to us, with a sensitive or non-sensitive pointer.

When the analysis concludes, every variable is either sensitive or non-sensitive. The analysis yields a conservative over-approximation of the sensitive set which, by design, will never lead to a security violation. Putting new variables into the sensitive set only enables precise bounds checking for more variables (at potentially increased performance overhead).

5.7 Runtime

At runtime, DataShield separates the sensitive and non-sensitive memory objects by creating two separate memory regions. The non-sensitive region resides in the lower memory addresses up to a fixed address, which is the highest possible non-sensitive address. DataShield uses $2^{32} - 1$ as the end of the non-sensitive region, but this is a configurable parameter. The sensitive memory region resides in the remaining memory addresses above the non-sensitive region.

While the boundary between the regions is fixed, data within the regions need not be stored at any fixed address. This means that our approach remains compatible with randomization techniques (e.g., ASLR).

Sensitive and non-sensitive heap- and stack-allocated variables are moved to the corresponding region. For the current implementation, the non-sensitive region contains a dedicated heap and stack, but there is no sensitive stack. All sensitive stack allocations are rewritten as sensitive heap allocations, so sensitive pointers are only stored in the sensitive heap or in registers. There is nothing inherent in the DCI policy that requires this implementation choice.

In addition to the two memory regions, DataShield needs a data structure for storing bounds and temporal metadata for sensitive memory objects. We store this metadata disjoint from the actual sensitive data to preserve the memory layout. This follows the approach of SoftBound [9], and allows system calls that take sensitive variables to work without modification. A detailed diagram of the memory layout and pointer-to-bounds metadata mapping is shown in Figure 5.4. Note that the bounds for the non-sensitive objects in Figure 5.4 are conceptual and so are the absolute addresses shown. Non-sensitive object bounds are not actually stored in the metadata table, they are hard coded in the coarse bounds check instructions. For a thorough discussion of the merits of disjoint metadata please see Nagarakette et al. [16]. To bounds check each sensitive pointer we need to store the base and last addresses, meaning we must save 16 bytes for each sensitive pointer. Note from the figure that

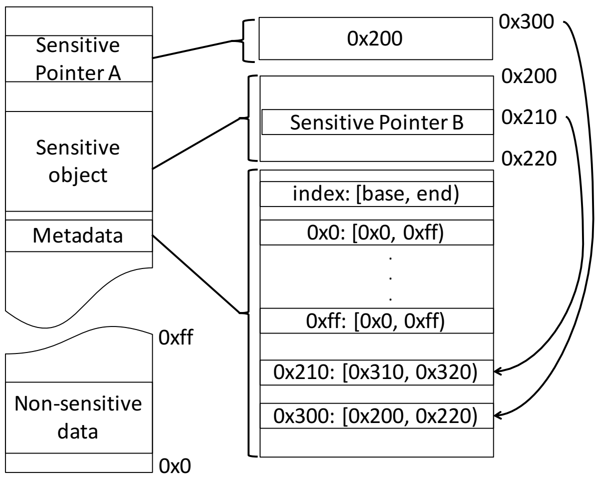


Figure 5.4. Detailed memory layout, showing the mapping between bounds and sensitive pointers using the pointer's address.

bounds are created and checked for sub-objects if the type of the sub-object is a sensitive pointer. Sensitive Pointer A and Sensitive Pointer B have their own bounds in this example.

Though unavailable in the current prototype, temporal metadata can be stored in the same metadata table. A discussion on temporal safety is in Section 5.7.6.

The runtime must protect the integrity of the metadata table. If the attacker could modify it, she could cause memory errors in the sensitive region. For the current prototype, the metadata table is stored inside the sensitive region. Keeping metadata in the sensitive region allows the coarse bounds check we apply to non-sensitive pointers to protect both the sensitive data and the metadata table.

5.7.1 Sensitive Globals and Constants

Normally, all constants and global data are loaded together by the program loader. However, to enforce the same security policy for global data as for heap and stack data, we use a linker script to map the sensitive and non-sensitive globals into their respective regions. After the sensitivity analysis finishes, sensitive globals and constants are marked with custom section names which are recognized by our linker script. Any non-sensitive globals are mapped to an address below the sensitive/non-sensitive region boundary. Sensitive globals and constants are instead mapped to an address above the boundary, and metadata is created for them in effectively the same manner as any other sensitive object.

5.7.2 Instruction Rewriting

The instruction rewriting step occurs after the sensitive variable analysis where the sensitivity of every variable is known. Before every pointer dereference, the compiler inserts the appropriate bounds check depending on the sensitivity of the pointer.

Allocations are replaced with calls to our region-based allocators (based on dl-malloc[1]) that ensure the allocated memory is in the correct sensitive or non-sensitive region.

5.7.3 Rewriting for Non-Sensitive Variables

We have implemented three types of coarse-grained bounds checks. DataShield inserts one of the three following coarse bounds check types, depending on the target processor and configuration, before every non-sensitive pointer dereference. All implementations enforce strong isolation. Considering recent advances in breaking information hiding [14, 135], our prototype avoids information hiding.

Software Mask. The software mask check has the widest compatibility. It only requires the target processor to have an `and` instruction. To mask a non-sensitive pointer, an `and` instruction with a pre-determined value is inserted before the pointer dereference. The mask clears the higher bits of the pointer, preventing the resulting value from pointing into the sensitive region before it is dereferenced.

Intel MPX Bounds Check. Intel MPX adds hardware support for bounds checking, including 4 bounds registers (`bnd0`–`bnd3`) and 7 new instructions. At

[1]http://g.oswego.edu/dl/html/malloc.html

program startup, our runtime initializes the `bnd0` register with the bounds of the non-sensitive region. The compiler inserts a `bndcu` instruction prior to every non-sensitive pointer dereference. The `bndcu` instruction checks the given pointer value against the upper bound stored in the given bounds register. In our case, it checks the pointer against the bounds of the non-sensitive region. By utilizing the 4 bounds registers, DataShield can support up to 4 non-sensitive regions (e.g., to sandbox different untrusted components) and the non-sensitive regions can reside anywhere in memory.

Address Override Prefix. An address override prefix before an instruction tells the processor to treat address operands as 32-bit values. An instruction with the address override prefix cannot access the sensitive region (since the sensitive region is above 2^{32} in this configuration). This bounds check is supported by any x86-64 processor. On x86-32 processors, previous work used segmentation registers [118], but segments are no longer enforced in x86-64.

5.7.4 Rewriting for Sensitive Variables

Bounds information is created when sensitive memory objects are allocated. The base and last addresses of the allocated object are recorded in the metadata table. Bounds metadata is propagated to other pointers on assignment. For example, extending our `struct circle` example above:

```
struct circle *c1, *c2;

// creates bounds information:
// base = address returned by malloc
// last = base + sizeof(struct circle)*10-1
c1 = malloc(sizeof(struct circle)*10);

// c2 gets assigned the bounds information
// (base and last) from c1
c2 = c1;
```

Prior to every occurrence of a sensitive pointer dereference, the compiler inserts a precise bounds check. The precise bounds check consists of a metadata table look up based on the address of the pointer, and a comparison of the sensitive pointer value with the upper and lower bound retrieved from the table. The coarse bounds enforcement for non-sensitive pointers is much faster than the precise bounds check for sensitive pointers because it consists of at most a single instruction (compared to several instructions and a memory access for the precise bounds check).

5.7.5 Standard Library Instrumentation

For complete protection, we must instrument both the application itself and the libraries the application uses. DataShield provides instrumented versions of musl[2] for the C standard library, and libc++[3] for the C++ standard library.

For compatibility, DataShield supports shared libraries, as they are used more commonly in practice than statically linked libraries. The issue with shared libraries is that they are compiled separately and ahead of time, without knowledge of the applications the library will be linked against. Since we cannot know all settings the

[2]https://www.musl-libc.org
[3]http://www.libcxx.llvm.org

library will be used in, we also cannot know if data-flow in an application will cause a particular library variable to be sensitive. We have two options to address this problem.

Option 1: Two Versions of Each Library. We compile two shared versions of each standard library, one that treats all data as sensitive and another that treats all data as non-sensitive. During compilation of the application, each call to the library is directed to the appropriate version, depending on the sensitivity of the arguments. Note that merging shared state between the two compiled library instances becomes challenging.

Option 2: Drop-in Replacement. We compile a drop-in replacement for the default system standard library, i.e., a single shared library that works with programs compiled with and without DataShield's instrumentation. To achieve this, we relocate all library objects to the non-sensitive region and do not insert any checks in the library. Internal checks in the library would fail when linked against applications not compiled with DataShield.

In our evaluation, we use Option 2. Benchmark programs typically make few standard library calls, so checks in libraries should not have a measurable effect on overhead.

Option 2 makes all library data non-sensitive, so application code that deals with non-sensitive data can directly use the library. However, with Option 2, the application cannot pass sensitive data to, or read sensitive data from, the library. Instead, we created wrappers for the standard library functions that propagate metadata (e.g., `memcpy`), and return pointers to library allocated memory objects (e.g., `getenv`.) The functions that return pointers to library allocated objects return pointers to safe region copies. Copies are made during program startup, so there is no opportunity for the attacker to corrupt the sensitive copy. Option 2 offers the added security of bounds checks in the application itself, while still allowing our libraries to be drop-in replacements.

5.7.6 Limitations

Our DataShield prototype does not support bounds or temporal checks of variadic arguments (a limitation shared with related work). This is an engineering issue, because for non-variadic functions we use the function signature to match the function arguments between the caller and callee. However, the variadic function may retrieve the variadic arguments in arbitrarily complicated ways. A straight-forward solution to this problem requires adding an argument to the variadic function prototypes and dynamically reading this new argument when *va_arg* is called to get the variadic arguments. This new argument would specify the number of arguments and the bounds and temporal metadata for each argument at the specific call site. A similar approach was proposed (but not implemented) in SoftBound [9]. In contrast, it is completely safe to pass non-sensitive pointers to variadic functions. We consider all non-sensitive pointers to have the same metadata, so we side-step the problem of matching up arguments to metadata across the caller/callee boundary.

Temporal metadata checking and tracking is not enforced in our current prototype. We could extend our prototype with temporal safety in the same manner that CETS [10] added temporal safety to SoftBound [9]. Following this plan, adding full temporal safety is an engineering effort.

The prototype does provide some temporal protection in that even if a pointer points to deallocated memory, it is impossible for a new object of the wrong sensitivity to be allocated in the pointed-to location. Concretely, given some sensitive pointer P, if $free(P)$ is called, the memory pointed to by P will be available to be reallocated. The most harmful type of temporal error occurs when a new object of a different type is allocated to where P points. However, DataShield mitigates this by guaranteeing that only sensitive objects will be allocated to where P points. The attack surface is limited by requiring a temporal error to exist in the portion of the program that uses sensitive data. Or in the case where there is only one sensitive type, DataShield provides region-based temporal safety analogously to DieHard(er) and Cling [50, 51, 136].

5.8 Summary

The DCI policy fully protects sensitive data while relaxing the requirements for non-sensitive data. This relaxation allows us to optimize the safety checks compared to complete memory safety. The prototype implementation of DCI, DataShield, is implemented using the LLVM compiler framework. The compiler pass identifies sensitive variables and inserts new instructions to perform checks and maintain metadata. The new instructions call DataShield's runtime library and DataShield can compile instrumented versions of the C/C++ standard libraries.

6 EVALUATION

In this chapter, we evaluate our implementations for efficiency and security. Our primary benchmark for measuring performance is the SPEC CPU2006 suite, which is standard in language-based security research. Additionally, select other pieces of software that are widely used (e.g., browsers) or security critical (e.g., cryptographic libraries). Evaluating security is challenging and we take the approach of examining case studies of known vulnerabilities or attacks to assess if our mechanisms mitigate them. First, we evaluate VTrust, second we evaluate the implementation of DCI (DataShield) and finally we summarize our results.

6.1 Evaluation Plan

To evaluate the runtime overhead induced by our instrumentation we compile our benchmark programs with and without our instrumentation, run them, and compare the time it takes to execute the program in each configuration. For VTrust, our benchmarks will be the SPEC CPU2006 programs that are written in C++ and the Firefox web browser. For DataShield, our benchmarks are the SPEC CPU2006 programs written in C/C++ and the cryptographic library mbedTLS.

Whenever possible we want to measure the components that contribute to the totally overhead of our instrumentation. For each mechanism, we present multiple configurations that provide different levels of protection and may be more appropriate depending on the level of overhead desired, hardware platform, or application.

To evaluate security guarantees, we search for CVEs that have been found in our benchmark programs and proof-of-concept exploits for these vulnerabilities. We then perform the exploit with our instrumentation enabled and verify the attack is detected.

6.2 VTrust

We evaluate our prototype implementation on the SPEC CPU2006 benchmarks and the open source web browser Firefox, to test our prototype's runtime performance and security. Additionally, we measure the attack surface of our benchmark programs by collecting virtual function call statistics.

6.2.1 Virtual Call Statistics

First, we measure the attack surface of VTable hijacking attacks by gathering virtual call related metrics for these benchmarks, Our results show that a large attack surface exists in real applications. Consequently, defenses like our solution VTrust should be deployed as soon as possible.

Statistics for SPEC CPU2006. Table 6.1 shows the total number of runtime indirect control flow transfers for SPEC CPU2006 benchmarks that include C++ code, and the static count of vtables and virtual calls. At runtime, indirect calls comprise between 1% and 33% of all indirect control-flow transfers. So, it is important to deploy defenses like VTrust to harden these applications.

We also found that, the proportion of return instructions is high for all benchmarks, calling for an efficient stack protection mechanism (which is orthogonal to our work).

Table 6.1. Virtual call related statistics for SPEC CPU2006 benchmarks written in C++. The unit `M` stands for millions, and `B` stands for billions.

	Runtime Profiling				Static Count	
	#inst	iCall	iJump	Return	vtbl.	vcall
444.namd	39M	1.17%	37.74%	61.08%	3	2
447.dealII	43B	1.01%	27.33%	71.66%	115	200
450.soplex	144M	3.78%	41.03%	55.19%	51	495
453.povray	29B	24.30%	2.43%	73.27%	48	112
471.omnetpp	22B	11.19%	18.63%	70.18%	127	1431
473.astar	15B	32.95%	0.07%	66.98%	2	0
483.xalanc.	36B	24.42%	7.81%	67.78%	29	4284

Moreover, we found that many indirect control flow transfers at runtime only have a single target. In our runtime profiling, an average of 26% indirect calls, 85% of indirect jumps, and 44% of return instructions only have one runtime target. Such a high number of control transfers with a single target indicates that a localized caching strategy, e.g., devirtualization and inline caching [137], might be an interesting future opportunity for optimization. This optimization not only improves the runtime performance, but also reduces some attack surfaces. VTV [46] showed the devirtualization can greatly improve the runtime performance.

Statistics on Firefox. Table 6.2 shows parts of the vtable-related information in the browser Firefox. We collect these information during compile-time optimization, and thus only libraries or executables are evaluated. The data of all object files are not included here. There are 39 libraries and executables in Firefox, but only the 12 libraries with the most vtables are listed in this table. The rest are omitted for brevity.

Columns 2-4 shows the vtable related information. The second column shows the count of vtables in each library (or executable). The library libxul.so has 15,801 vtables, indicating that there are thousands of classes in Firefox.

The third column shows the count of vtable pointer assignments. vtable pointers are usually assigned to runtime created objects in the constructor functions. In each constructor function, it may assign multiple vtable pointers to the objects due to multiple inheritance. The library libxul.so has 26,212 vtable pointer assignment operations, indicating that Firefox has thousands of constructors. For each vtable pointer assignment, the optional second layer of VTrust statically encodes the pointer to a constant index before it is assigned to the object.

The fourth column shows the count of vtable pointer read operations. Before accessing vtables (e.g., for virtual calls or RTTI), the vtable pointers will be read from the objects first. VTrust will decode these pointers at runtime. The library libxul.so has 72,874 vtable pointer read operations.

Table 6.2. VTable-related statistics of Firefox, including the count of (1) vtables, (2) vtable pointer assignments, (3) vtable pointers read operations, and (4) call instructions, as well as the ratio of indirect calls to call instructions, and the ratio of virtual calls to indirect calls.

library/	VTable			Call Instruction		
executable	#	assign	read	call	iCall	vCall
makeconv	173	470	1831	62625	6.66%	42.85%
genrb	173	473	1831	68429	6.35%	41.10%
icuinfo	173	470	1844	66600	6.35%	42.40%
genccode	173	470	1831	61037	6.81%	42.95%
gencmn	173	470	1831	61051	6.81%	42.95%
icupkg	175	476	1845	63197	6.89%	41.36%
pkgdata	175	476	1845	64363	6.75%	41.43%
gentest	174	471	1846	66640	6.35%	42.42%
gennorm2	179	478	1837	61831	6.78%	42.73%
gendict	174	472	1831	60896	6.83%	42.94%
js	1420	1991	3626	262502	23.26%	5.87%
libxul.so	15801	26212	72874	1720021	15.21%	27.48%

Most of these vtable pointer read operations are used for virtual calls. As shown in this table, there are 71,892 virtual calls (=1,720,021*15.21%*27.48%), close to the count of vtable pointer read operations. The remaining three columns in the table show the count of call instructions, the ratio of indirect calls to call instructions, and the ratio of virtual calls to indirect calls. It shows that about 40% of indirect calls are virtual calls, which is very high.

From this table, we can see that there is a large attack surface in Firefox. Attackers can find a lot of useful vtables and virtual call sites to launch VTable hijacking attacks. As the paper [23] discussed, it is practical to launch COOP attack in Firefox.

6.2.2 Performance Overhead

We have evaluated the performance overhead of VTrust on the SPEC CPU2006 benchmarks and the Firefox web browser.

Runtime Performance on SPEC CPU2006. To evaluate the performance overhead of VTrust we applied it to the SPEC CPU2006 benchmarks that are written

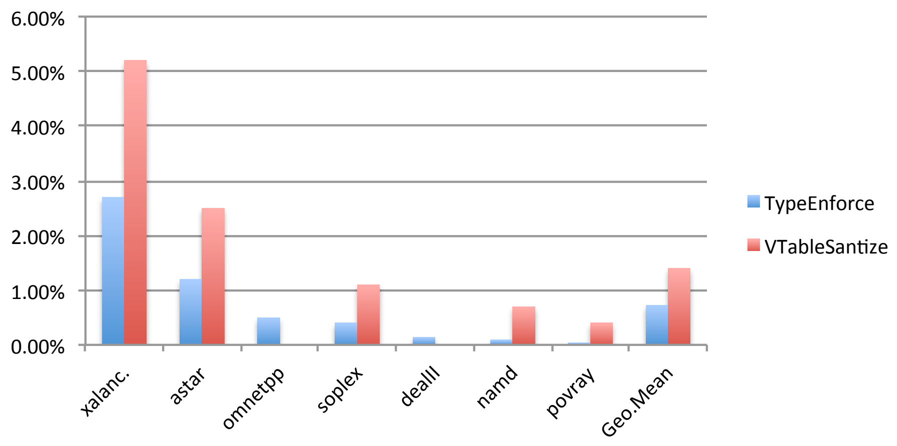


Figure 6.1. Performance overhead of VTrust on SPEC CPU2006, when enabling only the first layer of defense (virtual function type enforcement), or only the second layer (vtable pointer sanitization).

in C++. There are seven C++ benchmark programs in the SPEC CPU2006 suite. We ran the benchmarks under Ubuntu 14.04 LTS on a computer with an Intel Core i7-3770 with eight cores @ 3.40 GHz Processor and 16 GB RAM.

Our current prototype implementation of vtable pointer sanitization only works with the C++11 compatible library libcxx provided by LLVM, which has some compatibility issues with the 471.omnetpp and 447.dealII benchmarks. All other benchmarks and configurations work well. The overhead for virtual function type enforcement is very low, on average 0.72%. The worst case's overhead is about 2.7%. The average overhead for vtable pointer sanitization is 1.4%, and the worst is 5.2%. The average overhead of these two layers together is 2.2%, and the worst case is 8.0%.

Performance on Firefox. We also measure the performance overhead on a Firefox (version 34.0). We use six popular browser benchmarks, including Microsoft's LiteBrite [138], Google's Octane [139], Mozilla's Kraken [140], Apple's Sunspider [141], RightWare's BrowserMark [142] and PeaceKeeper [143], to test browsers' performance on JavaScript execution, HTML rendering and HTML5 support.

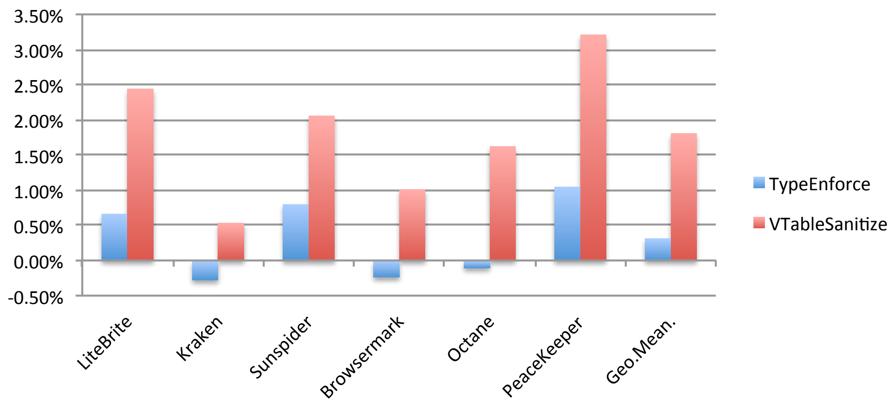


Figure 6.2. Performance overhead of VTrust on Firefox, when enabling only the first layer of defense (virtual function type enforcement), or only the second layer (vtable pointer sanitization).

We tested the browser using Ubuntu 14.04 on a computer with an Intel Core i7 with 12 cores @ 3.7 GHz Processor and 16GB RAM. We tested the performance overhead of VTrust's first and second layer separately.

As shown in Figure 6.2, the first layer of defense (i.e., virtual function type enforcement) introduces negligible performance overhead. On average, the performance overhead is about 0.31%. In the worst case, its performance overhead is about 1.05%. The overhead is so small that in some cases it is even negative. We attribute these fluctuations to caching effects, code layout, or system noise.

The second layer of defense (i.e., vtable pointer sanitization) introduces higher performance overheads. It has a performance overhead of about 1.81% on average. In the worst case, it introduces a performance overhead of 3.21%.

For applications with dynamically generated code (e.g., Firefox), we enforce both layers of VTrust. The performance overhead is close to the accumulation of the overheads of the first layer and the second layer. For example, if we enable both of these two defenses, it introduces an average performance overhead of 2.2%, close to the sum of the standalone virtual function type enforcement's overhead 0.31% and the overhead of vtable pointer sanitization 1.81%.

Practical Experience with Firefox. Firefox uses some tricks to support multiple platforms, and interactions between JavaScript and C++ code. It implements several special virtual functions `nsXPTCStubBase::StubNN`, where `NN` is a number ranging from 0 to 249, by using inline assembly code with mangled names (e.g., `__ZN14nsXPTCStubBase6StubNNEv`). The linker automatically resolves these virtual functions by name at runtime. So, the frontend compiler (e.g., Clang/Clang++) does not know the existence of these functions. As a result, VTrust fails to include signatures before these functions, causing false alarms when they are used at virtual call sites that have been instrumented with the first layer of defense. As a workaround, we modify the security violation handler to check whether the target is in the set of special virtual functions (for this application), when a security violation is detected.

Moreover, Firefox also has a special virtual call site, which is simulated in a special function `NS_InvokeByIndex`. This function will get an object and a method index as arguments, then do a simulated virtual function call: (1) it reads the vtable pointer from the argument object; (2) it reads the function pointer from the vtable using the method index; and (3) it calls the target method. This in fact is a virtual function call site, but the compiler frontend is not aware of it. As a result, VTrust fails to instruments checks, including the second layer of defense, for this call site. At runtime, encoded vtable pointers will be used here, and will then cause compatibility issues. We can identify this kind of corner cases, and instrument them with VTrust. It is worth noting that, all other existing defenses, including VTV and SafeDispatch, fail to identify this kind of corner cases, leaving them still vulnerable to VTable hijacking attacks.

6.2.3 Performance Comparison

Among all existing solutions that provide a strong protection against VTable hijacking attacks, VTV [46] has the lowest performance overhead. Other solutions, e.g., SafeDispatch, vfGuard and RockJIT, introduce a much higher performance overhead, which will be discussed in Table 8.1.

For the worst case SPEC benchmarks `astar` and `xalanc`, VTV introduces a performance overhead of 2.4% and 19.2% respectively, while VTrust introduces a comparable performance overhead of 3.7% and 7.9% respectively when enabling both two layers of defenses. An important point is that while the VTV paper reports a “lower bound” of 4.7% overhead on `xalanc`, this is not a valid comparison to VTrust. VTV’s “lower bound” configuration uses profile guided optimization (PGO), de-virtualization, and replaces the bodies of the VTV library functions with stubs. We do not consider PGO practical for complex software or a fair technique for benchmarks with a small number of input datasets, like the SPEC CPU2006 benchmarks. In fact, the VTV authors admit Chrome cannot be built with with PGO and de-virtualization. For the browser benchmark `sunspider` and `octane`, VTV introduces an overhead of 1.6% and 2.6% respectively when deployed on Chrome, while VTrust introduces an overhead of 2.8% and 1.5% respectively. So, VTrust introduces a similar performance overhead as VTV.

VTV validates the runtime vtable against a legitimate set that is updated when loading libraries are loaded. This validation needs to (1) dynamically update the legitimate set for each virtual call when a library is loaded, (2) resolve the *split-set problem* when a vtable set is created, and (3) perform a slow set lookup operation to validate the runtime vtable. Since the library loading usually finished before benchmark testing, and thus its overhead is not easy to evaluate. Moreover, when the legitimate set’s size is large, which is the common case for classes with many derived classes, it will take a longer time to do the runtime lookup.

Our solution VTrust has a negligible overhead of library loading. It only validates the signatures of target functions, and decodes the vtable pointers before they are used. It costs a constant time for each virtual call, and is faster than VTV in general. More important, for applications without dynamic code, VTrust only validates the signatures of target functions, which is much faster. So, in general, VTrust is faster than existing solutions.

Further, we can compare VTrust's overhead with all of the mechanisms from our CFI survey as shown in Table 3.2. On average, VTrust has lower overhead than the other mechanisms. However, the most useful comparison is VTrust against other virtual function call specific CFI mechanisms. We would expect mechanisms that protect all indirect calls to have higher overhead, in general, than mechanisms that only protect virtual calls.

6.2.4 Memory Overhead

We evaluated the memory overhead on Firefox in two scenarios: after a cold start and after running a sample benchmark.

After a cold start, the original Firefox uses about 130MB memory (resident set size, RSS). The hardened version Firefox uses about 133MB memory. The absolute memory overhead is about 4MB, and the relative memory overhead is about 3.1%.

Most of the memory overheads are from (1) the instrumented security checks including the type enforcement checks and vtable pointer decoding instructions, (2) the instrumented local VTMaps and the global VTMap that are used for vtable pointer encoding and decoding, (3) the instrumented signatures before each virtual function's body, and (4) the runtime supporting library VTLib.so, introduced by VTrust.

For example, there are 71892 virtual calls in the library libxul.so, and each virtual call costs about 40 bytes for the security checks. As a result, the security checks in this library takes about 2.9MB. Moreover, there are 15801 vtables in this library, taking about 128KB memory.

After running Firefox for a while, e.g., after testing the Kraken benchmark, the original Firefox uses about 299MB memory. The hardened Firefox uses about 303MB memory. The absolute memory overhead is still 4MB, close to the memory overhead in the cold start scenery. The relative memory overhead drops to 1.3%. Our solution

Table 6.3. Public VTable hijacking exploits against Firefox.

CVE-ID	Exploit Type	Vul App	Protected
CVE-2013-1690	VTable injection	FF 21	YES
CVE-2013-0753	VTable injection	FF 17	YES
CVE-2011-0065	VTable injection	FF 3	YES

VTrust does not use runtime allocated memory, so its absolute memory overhead stays constant.

6.2.5 Case Study: Real World VTable Injection Attacks

To evaluate the effectiveness of VTrust, we choose three public real world vtable hijacking exploits. These exploits are publicly available and all target the popular browser Firefox by exploiting use-after-free vulnerabilities. They all inject fake vtables and hijack the control flow. This is the most common VTable hijacking attack seen in practice. Table 6.3 shows details for these exploits, including the CVE-ID, target Firefox version, and the type of the exploits.

Experiments are carried out in a virtual machine running Ubuntu 14.04. For each exploit, we download the vulnerable Firefox's source code and compile it with VTrust. After hardening Firefox, we drive the browsers to access malicious URLs containing exploits. Results show that all the exploits we collected are blocked. Therefore, VTrust successfully protects applications from VTable injection attacks.

6.2.6 Case Study: Real World VTable Reuse Attacks

Since it is much easier to launch VTable injection attacks than VTable reuse attacks and no defenses against these attacks have been deployed, there are few VTable reuse attacks in real world. We found only one such case, besides the COOP attack published recently. In a recent Capture The Flag event [125], there is one challenge program (i.e., `zhongguancun`) that deploys a similar defense as VTint [97].

It checks if the runtime vtable is writable. If yes, it terminates the program. The only way to hijack its control flow is through VTable reuse attacks.

More specifically, this challenge program allocates a large buffer on the heap, and several objects close to this buffer. When passing a negative number to the program, it will overflow the buffer on the heap. By exploiting this vulnerability, attackers are able to overwrite the adjacent objects that have vtable pointers. They then overwrite this vtable pointer with a pointer to read-only memory, to bypass the deployed defense.

In fact, attackers can overwrite this vtable pointer to reference an offset in an existing vtable (i.e., a COOP attack). In this way, attackers can invoke a virtual function out of context. On the other hand, the program contains a virtual function that writes arbitrary content to the memory pointed by the function argument, allowing attackers to implement write-what-where primitives. Finally, attackers can overwrite control data, e.g., function pointers in the Global Offset Table, to hijack the control flow.

Several teams have solved this challenge, showing that VTable reuse attacks are feasible. The research paper COOP [23] also shows that VTable reuse attacks are practical in larger applications, e.g., Internet Explorer and Firefox.

To evaluate our solution's effectiveness against VTable reuse attacks, we collected several public exploits for this CTF challenge. Then we get the source code of the challenge from the author, and recompile the challenge using our tool VTrust. Finally we modify all these exploits to fit our new environment, and test them against the hardened challenge program. The result shows that all these exploits are blocked when the overwritten vtable pointer is used for virtual calls.

6.3 Data Confidentiality and Integrity

To evaluate the efficiency of our implementation prototype, we consider the major contributors to overhead. The first major source of overhead is coarsely bounds checking the non-sensitive object set. The second source is enforcing precise bounds on the sensitive set. There are other sources of overhead, such as initializing and allocating internal data structures, but these happen only once at program start up and are negligible for long lived programs.

A key feature of DCI is that the programmer decides which objects are in the sensitive set. This decision should have an effect on the measured overhead, so our evaluation must account for this decision. This presents a challenge because we cannot evaluate all possible ways to divide the program data into two non-interacting sets. Instead, we perform three experiments, that taken together give an overall picture of DataShield's overhead.

First, we evaluate microbenchmarks designed to vary the split between sensitive and non-sensitive data to quantify the ratio's effect on total measured overhead. We compare DataShield's overhead on these microbenchmarks to SoftBound + CETS, a complete memory safety mechanism, to show the reduced overhead of relaxed protection for non-sensitive data.

Second, we present three case studies where we assumed the role of the programmer. We examined the case study source codes and decided what data should be sensitive. We do not argue that our division of the program data into sensitive/non-sensitive is correct, optimal, or best in an objective sense. In our case studies, we annotated the important data types in the programs, leading to most of the data being sensitive. For our case study programs, we chose libquantum and astar from SPEC CPU2006, and mbedTLS, a TLS/SSL library.

Third, we evaluate the overhead's lower bound, i.e. the sensitive set is empty. This evaluation is an approximation of the case where only a small amount of data in a program is sensitive and it is accessed very infrequently. We evaluate all SPEC CPU2006 C/C++ programs in this configuration.

For all our evaluations, our platform was Ubuntu 14.04 LTS with an Intel Core i7-6600 3.4 GHz processor and 16 GB of RAM. The baseline compiler was clang 3.9 and all programs were compiled with Link Time Optimization (LTO) and O3 optimizations.

During our evaluation we discovered that our region-based allocator introduced performance speed-ups of up to 20% due to a massive reduction in page faults. We adjusted for this difference by replacing the default allocator with our region-based allocator when measuring baseline performance. Note that we did not modify the allocator used by SoftBound + CETS.

6.3.1 Microbenchmarks

To quantify the relationship between proportion of sensitive data and overhead, we created two microbenchmarks. In the benchmarks, we create a sensitive and a non-sensitive array, and the size of arrays are varied to control the sensitive to non-sensitive data ratio. We used the software masking implementation of coarse bounds checking for comparison against SoftBound + CETS, because the publicly available implementation uses software bounds checking.

In the first microbenchmark, `insertion-sort`, we sort arrays using insertion sort. This exaggerates the effect of the difference in array sizes because insertion sort's complexity is quadratic. For example, if the non-sensitive array has size N and the sensitive array has size $2N$, then we will execute four times as many sensitive pointer dereferences as non-sensitive.

The second microbenchmark is `find-max`, a simple implementation of a linear scan of an array of objects to find the element with the largest value for a particular integer field. We control the ratio of sensitive to non-sensitive objects by varying the sizes of the two arrays. For example, if the sensitive array has twice as many elements as the non-sensitive array, we know that there should be roughly twice as many sensitive pointer dereferences as non-sensitive – because the `find-max` algorithm is linear in the size of the array.

The results of this experiment are shown in Figure 6.3. The overhead of SoftBound + CETS is mostly constant across our experiments, as we would expect. However, as the amount of sensitive data increases, the overhead of DataShield increases towards the

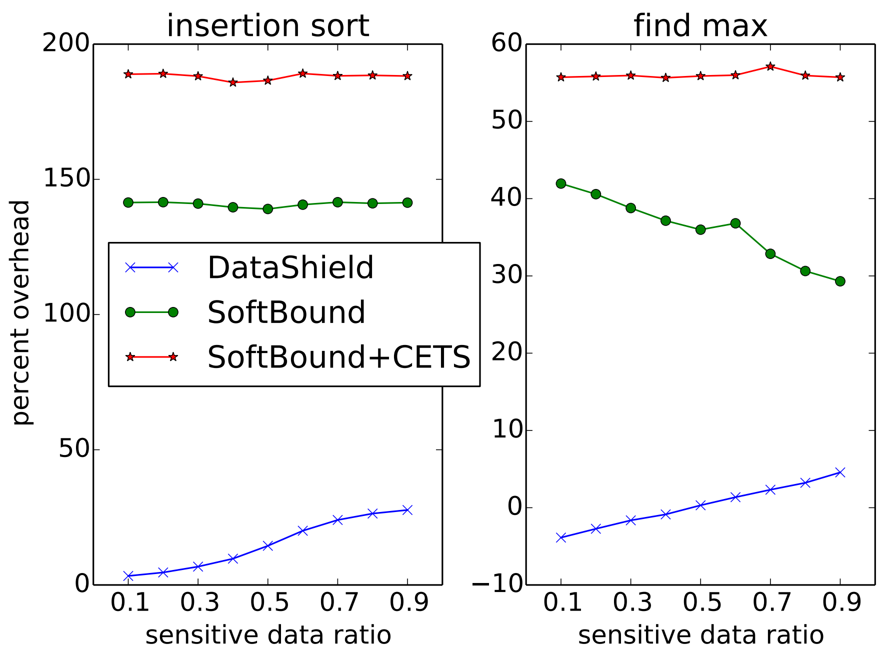


Figure 6.3. Performance overhead measured on two microbenchmarks when varying the proportion of sensitive to non-sensitive data. More sensitive data leads to higher overhead for DataShield but not for SoftBound + CETS.

SoftBound + CETS overhead. In Figure 6.3, SoftBound + CETS includes both spatial and temporal protection, but SoftBound is spatial protection only. All configurations of DataShield, even protecting up to 90% of the data, are faster than SoftBound. Beyond the overhead savings of enforcing memory safety on only a subset of the data, we attribute the additional performance improvements compared to SoftBound to, in part, local optimizations that reduce the number of times bounds are loaded and inlined versions of the checks. We also observed that the region-base allocators can have a large effect on heap locality.

From these experiments, we conclude that non-sensitive data does in fact incur lower overhead using our prototype versus sensitive data. Therefore, the total program overhead is a function of the amount of sensitive data in the program.

6.3.2 Case Study: libquantum

For our first case study, we evaluated libquantum from the SPEC CPU2006 benchmark suite with a subset of the program's data protected. We decided to protect the `quantum_reg_struct` type as it is one of the main types used by libquantum. To protect this type, we simply added our annotation to the header file that defines the type, i.e., "qureg.h." With precise bounds checking enabled for `quantum_reg_struct` and its sub-objects, we measured an overhead of 27.21% on the `ref` SPEC benchmark inputs. Unfortunately, we cannot compare our overhead to SoftBound as the current SoftBound version does not compile libquantum.

The purpose of the case study is not only to measure the performance overhead, but also to evaluate the difficulty of annotation. For this case study, adding just one annotation for `quantum_reg_struct` protected nearly every pointer in the program sensitive because that data type is used so commonly. We created a dynamic profiler to measure how many dereferenced pointers were sensitive versus non-sensitive. We measured only two non-sensitive pointer dereferences in this configuration. Note that benchmarks are geared towards a single purpose with all data heavily connected. This behavior is therefore expected.

6.3.3 Case Study: mbed TLS

For our second case study, we applied DataShield to
mbedTLS, a SSL/TLS library implemented in about $30,000$ lines of C code[1]. There are two main purposes for this case study:

- To show that the type-based annotation approach is scalable to large programs;
- To measure the overhead a system would incur when using a protected SSL/TLS library in practice.

[1]The name of the library was changed from PolarSSL to mbedTLS when it was acquired by ARM in 2015.

We annotated the type `ssl_context`, which is the most important type used by the library users. Most functions that are visible to clients take a `ssl_context` as a parameter. The context has fields of many different types: primitives, pointers, arrays, and function pointers. We recompiled the mbedTLS library with the context type annotated and built the included programs `ssl_client2` and `ssl_server2` against our protected library.

With only the type `ssl_context` annotated, we successfully protect all cryptography related memory objects in the client and server. In an example run of the sever, 52 non-sensitive pointers were dereferenced compared to over 1.6 million sensitive pointer dereferences. Note that in a production web server, it would have many more non-cryptographic functionalities, so in the other areas of the code there would be more non-sensitive pointer dereferences – which incur lower overhead.

Despite having a high percentage of sensitive objects, we measured the fairly low overhead of 35.7% when exchanging one million messages between `ssl_client2` and `ssl_server2`. This is partly due to not incurring instrumentation overhead when performing and waiting for IO, as the client and server communicate with each other over a socket. In practice the SSL/TLS client and server would be running on different machines connected across some network, so the time waiting for IO might be even greater.

In conducting this case study we found the type annotations to be straightforward to use, but we encountered a difficulty with function pointers with sensitive arguments. When the pointed-to function takes an explicitly sensitive type as a parameter there is no problem, and the analysis rewrites the pointed-to function correctly. However, if the caller function invokes a callee function through a pointer with an *implicitly* sensitive argument, the analysis can fail to match the callee and caller correctly and consider the argument as non-sensitive inside the callee. This situation always leads to a false positive policy violation in the callee, which luckily cannot lead to a security vulnerability but aborts the program. To address this problem, we added an annotation that marks the sensitivity of function pointer call sites and address-taken functions. We annotated 50 address-taken functions total for both the `ssl_client2` and `ssl_server2`. Most function pointer invocations do not need annotations, because the analysis usually determines sensitivity correctly if the caller uses the pointer at all – versus allocating a new object, not accessing it, and passing a pointer to the object to the callee.

6.3.4 Case Study: astar

For our third case study, we use astar, which is a SPEC CPU2006 benchmark. It is a path finding library implemented in 4,285 lines of C++. In this case study, we evaluated the effect of relaxing one of the policy rules on the number of bounds checks and performance. Specifically, we removed rule 1 from Section 5.5.2 for primitive types only. This relaxation allows sensitive primitive values (`int`, `float`, etc.) to leak information when they are added or subtracted with non-sensitive primitives, but leaves full protection in place for pointers. We refer to this related policy as "separation mode." We annotated the type `statinfot` and used the "rivers.cfg" input configuration.

With the full DCI policy enforced the measured overhead was 96%. In separation mode, the overhead was reduced to 9.12% and the number of sensitive bounds checks was reduced by from over 100 billion to 160 thousand. Our results show that the full policy is quite strict and results in a large portion of the program data being sensitive. However, if we relax the policy, as in separation mode, we can further control the security versus overhead trade-off.

6.3.5 SPEC CPU2006 Evaluation

To further evaluate the overhead of DataShield, we recompiled each of the SPEC CPU2006 C/C++ benchmarks with our instrumentation. The SPEC benchmarks are not ideal candidates for benchmarking security mechanisms like DataShield. Unlike browsers, web servers, and cryptographic libraries, the SPEC benchmarks are simple

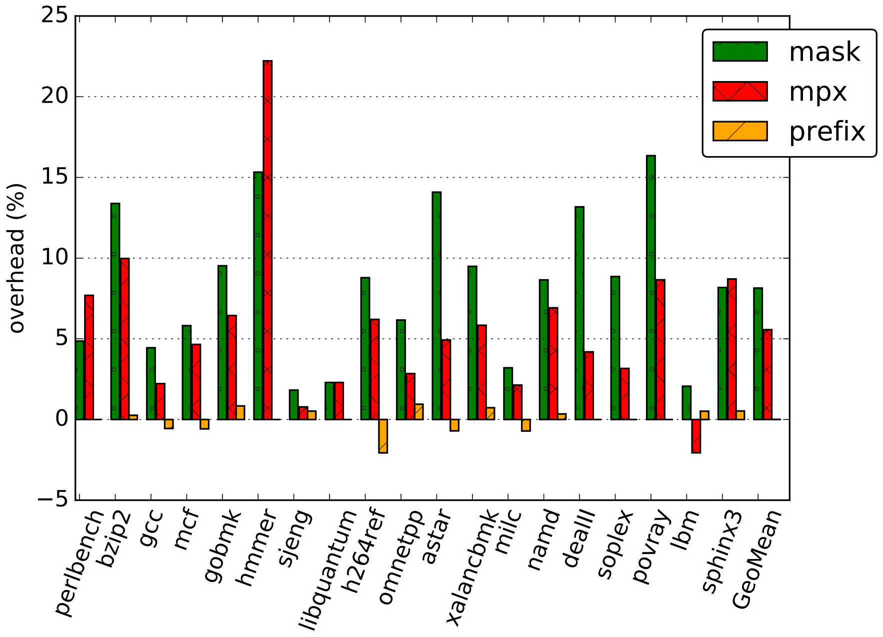


Figure 6.4. Performance overhead on SPEC CPU2006 for three non-sensitive protection options: masking, Intel MPX, and address override prefix.

programs with few types and none of the benchmarks deal with sensitive data. We include them since they are the de-facto standard for performance measurement.

We did not annotate these benchmarks for this experiment. Even though this experiment is run with an empty sensitive set, the bounds of the non-sensitive region are still enforced. This experiment is effectively measuring the overhead of the parts of a program that do not interact with sensitive data, independent of what sensitive data may exist in the program.

With SPEC CPU2006, we evaluated the three coarse-bounds check options, software mask, Intel MPX, and address override prefix. Moreover, to isolate the components of DataShield's non-sensitive overhead, we measured the overhead of software masking in integrity-only and confidentiality-only modes.

Comparison of Coarse Bounds Check Implementations. Depending on the target processor, the programmer may choose among three coarse bounds check implementations, namely software mask, Intel MPX, and address override prefix. Figure 6.4 shows the overhead of the three options on the SPEC CPU2006 C/C++ benchmarks, using the median of ten runs of each individual benchmark.

For the software mask coarse bounds check, the geometric mean across the benchmarks was 8.14%, and the difference between individual benchmarks is quite large (1.82% to 16.34%). The width of this range is due to some benchmarks having many pointer operations while others having much fewer. For MPX bounds checks and address override prefix the geometric means are 5.56% and 0.0013% respectively.

As expected, the address override prefix implementation had the lowest overhead – too small to measure reliably. The main reason for this is that the address override prefix bounds check does not introduce any additional instructions to the program, it just prefixes existing instructions. The main drawbacks are that this feature is unique to the x86-64 instruction set, and that the prefix applies to a fixed region $(0 - 2^{32})$.

To summarize, using a prefix offers best performance but constrains the location, maximum size of the region, and the ISA. Intel MPX has lower overhead than masking and can give fine-grained control over the location and the size of the region. Masking has the widest compatibility but is the slowest option.

Integrity and Confidentiality Overhead. We have evaluated the execution time overhead of DataShield in three different configurations: (i) integrity-only, (ii) confidentiality-only, and (iii) both confidentiality and integrity. These different configurations protect the confidentiality, integrity, or both of the sensitive region.

In integrity-only mode, *only stores* to pointed to memory locations are protected. In confidentiality-only mode, *only loads* from pointed to memory locations are protected. In the third mode, *all loads and stores* are protected.

Integrity-only is clearly useful on its own. Many mechanisms enforce only integrity including CFI, CPI, and WIT [33, 118, 132]. Conversely, confidentiality without integrity is brittle because the attacker can simply overwrite the metadata. We present the overhead of confidentiality-only mode to show the different components of the

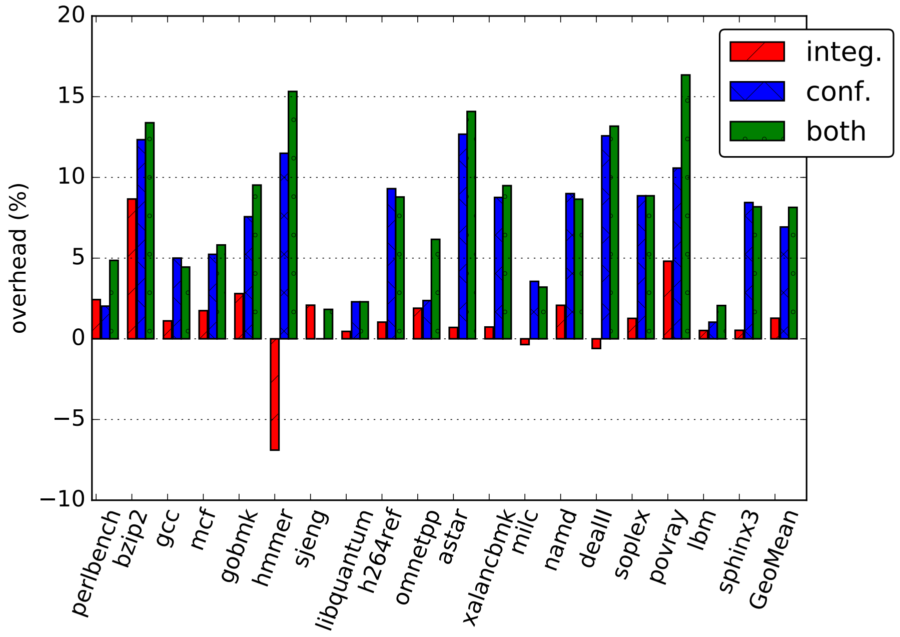


Figure 6.5. Performance overhead on SPEC CPU2006 isolated by protection type. Integrity-only protects writes, confidentiality-only protects reads, and “both” protects reads and writes.

overhead and for comparison to integrity-only mechanisms. Of course, the enforcement of both integrity and confidentiality is the strongest protection and incurs the highest overhead.

Figure 6.5 measures the overheads for the different modes on different runs, so integrity-only and confidentiality-only options do not sum up exactly to the combined integrity and confidentiality option due to measurement variation. One interesting aspect of this result is that confidentiality is more costly than integrity, as there are more memory reads than writes in the SPEC CPU2006 benchmarks.

6.3.6 Security Evaluation

To evaluate the security of our approach, we looked for Common Vulnerabilities and Exposures (CVEs) in our case study programs. Guido Vranken discovered a remote heap corruption vulnerability for mbedTLS in October 2015 [144] (CVE-2015-5291). The root cause of the vulnerability is a buffer overflow. Specifically, a malicious SSL/TLS server can create a session ticket that overflows the client's buffer when the session ticket is reused by the client, corrupting the client's heap. We recompiled mbedTLS 2.1.1 (an older version, before the vulnerability was patched) both with and without protection, and ran the malicious server against our clients. As expected, without DataShield protection the client's heap was corrupted, but with protection the attack caused a bounds violation and termination of the program. From this evaluation, we conclude that DCI is a useful defense mechanism for mitigating vulnerabilities in production software.

Qualitatively, DataShield provides deterministic protection for every sensitive variable in the sensitive set because there is a check on every pointer deference.

6.4 Summary

In this chapter we have evaluated the performance and security of our mechanisms. We showed that the performance overhead of VTrust is very low (less than 6%) and the performance overhead of DataShield is reduced relative to complete memory safety. Both mechanisms are able to detect exploits of recent CVEs found in production software.

7 DISCUSSION

In this section, we discuss our results, potential new directions for research, and unsolved challenges related to our work.

7.1 Benchmarking Control-Flow Integrity Mechanisms

As Table 3.2 shows, authors working in the area of CFI 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, i.e., the benchmarks which make many indirect calls. 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 [56].

7.2 Cross-cutting Concerns for Control-Flow Integrity

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.

7.2.1 Enforcement Mechanisms

The CFI precursor Program Shepherding [58] 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 [33]. 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 [46, 129] and Microsoft [100] 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 [93, 104, 145, 146] 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 [110].

There have also been efforts to repurpose existing hardware mechanisms to implement CFI [88, 90, 98, 99]. kBouncer [88] was the 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 [90] 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 [60–62]. The CFIGuard mechanism [99] uses the LBR feature in conjunction with hardware performance counters to heuristically detect ROP attacks. CFIMon [114] used the branch trace store, which records control-flow transfers to a buffer in memory, rather than the LBR for CFI enforcement. C-CFI [95] 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. O-CFI [94] leverages 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. Intel Control-flow Enforcement Technology (CET) [147] 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.

7.2.2 Open Problems

As seen in Section 3.4.1 most existing CFI implementations use ad hoc, imprecise analysis techniques when constructing their CFG. This unnecessarily weakens these mechanisms, as seen in Section 3.4.2. All future work in CFI should use flow-sensitive and context-sensitive analysis for forward edges, SAP.F.5 from Section 3.2.4. 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.

7.2.3 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 [44] 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. [102] 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. [32] 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 [137] 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` [148–150].

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 [151], 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 [89] propose a mechanism for Objective-C send-like dispatch, the generalisation to Smalltalk-style dispatch remains unsolved.

7.3 Data Confidentiality and Integrity Future Work

Automatically identifying sensitive data is an open problem we plan to investigate in future work. Additionally, we plan to enhance our implementation, DataShield, by improving the precision of the sensitivity analysis.

7.3.1 Automatically Identifying Sensitive Data

The main unsolved problem for targeted non-control-data attack mechanisms is identifying sensitive data in a generic way. Other approaches have targeted a specific piece of software and the authors use their domain knowledge to determine what is sensitive. An example of this is Kenali [152] which protects security checks inside the Linux Kernel. The approach is semi-automated in that all variables which have data-flow with the security check's return codes are found automatically. However, the authors manually determined that the return codes are the roots of the sensitive data-flows and this approach is not immediately generalizable to software other than the Linux Kernel. However, we believe generalizing Kenali's approach to track data-flow with variables upon which control-flow depends is an promising direction.

The challenge for automatically identifying sensitive data is that there is no information in C/C++ code that can be reliably found automatically that determines if a given variable is security critical.

7.3.2 Sensitivity Analysis

In future work, we plan to formalize and improve our sensitivity analysis. As discussed in Section 5.6.2 we over-approximate the sensitive set. This leads to higher overhead because the security checks on sensitive pointers are more expensive. Therefore, a more precise analysis would result in lower overhead.

Despite not presenting a complete formal proof, we do have some evidence for correctness. If a program runs successfully with instrumentation (which is the case in all our experiments) then we know that every check succeeded. Therefore, the static determination of sensitivity matched the true sensitivity of the pointer at runtime every time a pointer was dereferenced during program execution. This argument, however, does not provide conclusions about the correctness of non-exercised code paths.

We also plan to investigate using Intel MPX to enforce the bounds for both sensitive and non-sensitive pointers at runtime. We would need to modify our metadata data structure to allow Intel MPX's bounds look up instructions to access it.

7.4 Summary

In this section, we have discussed our results, limitations of our defenses, and future research directions. Based on our results, we show that the CFI community would benefit from greater benchmarking standardization. For our own work on DCI, we foresee the method for automatically identifying the smallest set of sensitive data as the next research problem.

8 RELATED WORK

The related worked for this dissertation broadly falls into three categories, (i) memory safety, (ii) control-flow integrity, and (iii) non-control-data attacks. Memory safety and control-flow integrity are well researched areas with many attacks and defenses. In fact, CFI attackers and defenders are in an arms race. Defenders propose a mechanism, attackers bypass it, and then the defenders propose a new defense against that attack, and the cycle repeats.

8.1 Memory Safety

Control-flow hijacking attacks (including VTable hijacking attacks) utilize memory safety bugs to modify the program state by tampering with code pointers (e.g., return addresses and function pointers), causing the control-flow to divert when the broken code pointers are used.

Effective defense mechanisms against these attacks can be classified on *when* they stop an attack [56]: (i) at the memory safety level by, e.g., checking bounds of memory access [9,153–155] or enforcing temporal safety on memory [10], (ii) when code-pointers are written (i.e., protecting a subset of data) [118], or (iii) when corrupted data is used in a computation like for Control-Flow Integrity (CFI) [33–46].

Memory safety-based defenses result in fairly high overhead as many memory read and write operations must be protected with additional guards. Some tools reduce the overhead by restricting protection to write operations only [132]. Code-Pointer Integrity [118] restricts the protection in another dimension by protecting a subset of all pointers: code pointers and any data structure that references code pointers.

There are many proposed techniques that aim to add memory safety to C or a C dialect. Approaches that augment the C language include CCured [5] and Cyclone [4].

Both approaches are compiler-based and combine static analysis with runtime checks. DataShield is inspired by CCured and Cyclone in that it tries to make the porting process as easy as possible. There is a massive amount of legacy C code for which porting to a new language is too costly.

SoftBound [9] provides complete spatial memory safety but works on unmodified C code. CETS [10] is an extension to SoftBound that provides temporal safety. The main drawback of SoftBound + CETS, and complete memory safety in general, is overhead. Code-Pointer Integrity (CPI) [118] is a specialization of memory safety that only protects code pointers. This ensures control-flow integrity while reducing overhead relative to complete memory protection. DCI extends CPI's partial protection to other types of data. Key differences between CPI and DCI are:

1. CPI protects code pointers only (e.g., function pointers, return addresses, or indirect jumps) while DCI protects any type of data, not just pointer data;
2. DCI allows the programmer to specify what is protected whereas CPI exclusively focuses on code pointers;
3. DCI protects the content of objects along with pointer values whereas CPI protects pointer values only;
4. DCI enforces both integrity and confidentiality where CPI only enforces integrity.

Several approaches attempt to reduce the memory overhead of complete memory safety [16]. Hardware support [156–158] has been shown to reduce overhead. ASAP [159] is a tool that allows the programmer to specify the amount of overhead she is willing to accept then only inserts checks up to that budget. DCI also aims to reduce the performance overhead but never relaxes the policy on sensitive data. SAFECode [160] used static analysis to eliminate checks and its allocation pools are similar to DCI if we consider the sensitivity to be part of a variable's type. METAlloc reduces the cost of metadata look up [161]. PAricheck [162] reduces the cost of pointer arithmetic checks by labeling memory objects and checking if the result points to an object with the same label. Similar approaches that are memory allocator based include Cling, DieHard(er), and Baggy Bounds Checking [49–51, 136].

8.2 Control-Flow Integrity

CFI protects against control-flow hijacking attacks by adding guards before indirect control-flow transfers, which restricts each indirect control-flow transfer to the set of valid targets as determined by a static analysis (usually a type-based points-to analysis). CFI can stop attacks such as return-to-libc [163] and ROP [66]. HyperSafe [35] enforces a fine-grained CFI policy for virtual machine managers. Recent approaches [39, 41] directly rewrite binaries and provide a coarse-grained CFI protection.

However, CFI faced several adoption hurdles: the fine-grained CFI solutions usually do not support separate compilation, few of them provide precision protection for C++ programs, and many of them induce fairly high overhead. Relaxed implementations can be circumvented [60–62]. Our solution VTrust provides a fine-grained CFI for only virtual calls. It does not rely on a whole-program analysis and provides modularity support.

In Chapter 3, we systematically compare the existing work in CFI in depth.

8.2.1 VTable Hijacking Defense

Researchers also proposed some specific virtual call protection solutions. Table 8.1 shows a brief comparison between these solutions and our solution VTrust, including the effectiveness of each defense, the support of incremental building (i.e., modularity support), dynamic loading of external libraries (i.e., mixed code) and dynamic generated code (i.e., writable code), as well as the performance of compile-time class hierarchy analysis and runtime overhead.

Table 8.1. Comparison between defenses against VTable hijacking attacks, including whether they can (1) defeat VTable hijacking, and support (2) incremental building (i.e., modularity), (3) external libraries, and (4) writable code (i.e., dynamic generated code). This table also shows the comparison of (5) speed of class hierarchy analysis, (6) source code dependency, and (7) performance overhead. The abbreviation `SD` stands for SafeDispatch. In the `dynamic loading` column, `Y/N` means the defense supports loading hardened or analyzed libraries, but not unhardened ones.

Defense	Able to Defend?		Incremental	External	Writable	Class Hierarchy	Source Code	Performance
Solution	VTable injection	VTable reuse	Building	Libraries	Code	Analysis Speed	Dependency	Overhead
VTint [97]	y	partial	N/A	y	y	N/A	N	2%
T-VIP [127]	y	N	N/A	y	y	N/A	N	2.2%
vfGuard [96]	y	partial	N/A	Y/N	y	N/A	N	18.3%
original CFI [33]	partial	partial	N/A	Y/N	N	N/A	N	16%
VTGuard [164]	N	y	y	Y/N	N	N/A	y	$< 0.5\%$
SD-vtable [91]	y	y	N	Y/N	y	slow	y	30%
SD-method [91]	y	y	N	Y/N	y	slow	y	7%
RockJIT [32]	y	y	y	Y/N	y	slow	y	10.8%
VTV [46]	y	y	y	Y/N	y	fair	y	about 3%
VTrust	y	y	y	Y/N	y	fast	y	0.72% or 2.2%

The VTint [97], T-VIP [127] solutions are binary-rewriting based defense mechanisms. VTint places vtables in a special read-only section and adds instrumentation before virtual calls to check if the runtime target vtable is in this read-only section. It can defeat all VTable injection attacks, but only a few VTable reuse attacks (i.e., reusing existing data rather than vtables). Attackers may reuse existing vtables to launch attacks [23] to bypass it. T-VIP works in a similar way, but does not provide any protection against VTable reuse attacks. They both introduce a low performance overhead, and are able to protect applications with writable code.

VfGuard [96] is another binary level defense. It filters virtual functions at runtime based on some features, e.g., the index of the function inside a vtable. It uses dynamic instrumentation tool PIN [165] to validate these filters, and thus has a high performance overhead. Since the filters used by vfGuard are permissive, it only provides a partial protection against VTable reuse attacks. Moreover, all these three binary solutions rely on some heuristics to identify vtable related operations in programs, and may also cause false negatives in some cases, i.e., some virtual calls are not protected.

The original CFI [33] also provides some protection against VTable hijacking. However, it cannot defeat all VTable hijacking attacks, because it does not utilize the type information of virtual functions. Moreover, it does not support writable code and incurs a higher performance overhead.

VTGuard [164] is a lightweight source code level defense, similar to stack canaries, that instruments secret cookies at the end of legitimate vtables. Its performance overhead is extremely low. However, it is vulnerable to information leakage attacks. Attackers may leak the secret cookies and inject fake vtables with correct cookies. So, it cannot defeat VTable injection attacks, nor protect applications with writable code.

SafeDispatch [91], RockJIT [32] and VTV [46] work on programs' source code too. SafeDispatch resolves the set of legitimate vtables (or virtual functions) for each virtual function call by performing a class hierarchy analysis (CHA) at compile-time, and validates the runtime vtable (or virtual function) against this set. It requires a heavy compile-time class hierarchy analysis, which prevents the incremental compilation. It uses a set lookup operation that is slow to perform the security check, introducing a high performance overhead.

RockJIT is based on a fine-grained signature-based CFI solution MCFI [45] that only protects C code, and extends it to Just-in-Time compiled code and virtual calls. RockJIT also performs a CHA analysis like SafeDispatch, and introduces a very high performance overhead as well. Unlike SafeDispatch, it supports separate compilation by emitting the class hierarchy information into each module and combining them at link time. However, it also has to rebuild the whole program, when the class hierarchy changes. VTrust only uses signature matching (i.e., type enforcement) to protect virtual calls, without the requirement of class hierarchy information, which provides a better compatibility and performance.

Tice et al. [46] propose VTV and Indirect Function-Call Checks (IFCC) to protect indirect call and jump instructions. VTV validates if the target vtable is in a legitimate set. But it only analyzes parts of the class hierarchy information when compiling, and utilizes the runtime initializer functions to update the overall class hierarchy. In this way, it supports incremental building with a faster class hierarchy analysis.

However, it also needs a slow runtime set lookup operation to perform the security check. The performance overhead not only depends on the count of virtual calls, but also the size of the legitimate vtable set. In applications with complex class hierarchy, the performance overhead would be higher. Moreover, it needs to perform an extra check each time a new vtable set is created, to overcome the *split-set problem* [46], causing a high overhead when loading a new library.

VTV's overhead ranges from 2.4% to 19.2% for SPEC applications, and from 1.6% to 8.4% for the Chrome browser when testing different benchmarks. Our solution VTrust introduces an average overhead of 0.72% and 0.31% for SPEC and Firefox respectively, when only enabling the first layer. Even with the extra second layer of defense, the average overhead of VTrust is about 2.2%, comparable with VTV. VTV also introduces a profile guided optimization to perform de-virtualization, i.e., translating virtual calls to direct calls. This optimization helps improve the overall performance a lot. It can also be adopted by other defenses, such as VTrust.

Our solution VTrust uses the signature matching to enforce virtual functions' type and provide a most fine-grained CFI protection, and an optional extra layer of defense to validate vtable pointers' validity in case target applications have writable code allowing attackers to forge functions with correct signatures. It does not need any global class hierarchy information, and thus it has a faster static analysis and natural modularity support. Its performance overhead is also better than most of existing solutions. Moreover, the overhead of each security check instrumented by VTrust is constant, irrelevant to the class hierarchy. It is also able to identify corner cases, and provides a complete protection against VTable hijacking attacks.

All six of these source code level defenses support dynamically loaded libraries. When a hardened library is dynamically loaded into the process, the runtime class hierarchy is updated, so the newly loaded virtual functions are allowed to be called. However, if an unhardened library is loaded into the process, all these solutions may cause false positives because the class hierarchy is missing the classes from the loaded library. Usually, a special fallback failure function that tracks a whitelist can be embedded in the security check to catch such false positives.

8.3 Non-Control-Data Defenses

Yarra [166] is similar in concept to the DCI policy but Yarra focuses on programming language theory while our work targets a practical implementation. Yarra has two modes, whole program and targeted. Whole program mode is complete memory safety with metadata for each memory address. The runtime of gzip from SPEC INT2000 in whole program mode is 6x the baseline. In targeted mode, Yarra uses page protection to lock its protected data whenever unprotected functions are executing. This approach was inspired by Samurai [167] and has great compatibility because it can guarantee the integrity of the protected data even when running completely unknown and untrusted code. The drawback is that the overhead of updating the page permission is far higher than our implementation. Yarra's execution time of gzip in targeted mode is 2x the baseline.

Kenali [152] enforces the integrity of kernel security checks with a form of data-flow integrity. It is similar to DCI in that it attempts to infer the sensitive data from a set of sensitive data root variables. For Kenali, root variables are the error codes returned by kernel security checks but in DCI the roots can be any data type. The protection enforcement is stronger in DataShield than in Kenali. Kenali relies on information hiding to protect its stack and overflows between sensitive objects are not prevented by Kenali. We believe DCI offers a more flexible approach in that the programmer can control which data is sensitive and it works on a variety of programs whereas Kenali targets only the Linux kernel.

Shreds [168] is a new compartmentalization mechanism for protecting sensitive data. Unlike Shreds which treats all memory inside the shred as sensitive, DCI supports code that mixes sensitive and non-sensitive data. Shreds provides no protection against overflows between sensitive objects. If there is a memory error anywhere within the shred, the attacker can corrupt any memory inside the shred.

The inspiration for DataShield comes from the abundance of work on Control-Flow Integrity [33, 43, 45, 91, 98, 103, 118]. CFI mechanisms are becoming robust and practical, but they do not address non-control-data attacks. Chen et al. [24] argued that non-control-data attacks pose a significant and realistic threat. Recent work has called into question the security of CFI. Control-flow Bending [62], Control-flow Jujutsu [107], Counterfeit Object Oriented Programming [23], and Out of Control [60] showed there are multiple attack vectors to bypass CFI.

8.4 Isolation Mechanisms

DataShield's protection scheme is somewhat similar to the implementation of Monitor Integrity Protection (MIP) [42], except DataShield also enforces confidentiality.

The monitors in MIP are analogous to the sensitive data in DCI. Other similar isolation mechanisms include PittSFIeld [52], TRuE [169], and Native Client [170]. Other mechanisms combine isolation with a policy which forbid or allow certain system calls [58, 171].

8.5 Summary

Many memory safety mechanisms have been proposed by researchers, but have not seen wide spread adoption, primarily due to prohibitive overhead. Control-flow integrity has recently been integrated into production compilers, but is still a very active area of research. Non-control-data attacks may be the next attack vector after CFI, and non-control-data research is relatively nascent compared to CFI research.

9 CONCLUSION

Systems software, primarily written in C/C++, comprises the majority software modern computers run. Attackers exploit errors in C/C++ programs to: i) gain control of systems and ii) leak or corrupt sensitive data. Exploitable errors are prevalent in C/C++ programs partly due to the design of the programming languages. Memory and type safety of the program are not required by the language or enforced by the compiler, but are left solely to the programmer.

This dissertation presents our research on state-of-the-art defenses that protect critical data in systems software at low overhead. Our systems' designs are informed by the strengths and weaknesses in existing defenses and the latest, most sophisticated attacks. As our Control-Flow Integrity (CFI) survey shows, measuring the strength of existing protections is difficult. We need strong new tools to proactively defend emerging attack against sensitive data. VTrust protects vtables in C++, while Data Confidentiality and Integrity protects arbitrary programmer selected data.

We have evaluated the performance and security of our mechanisms. Both VTrust and DCI impose low overhead while mitigating vulnerabilities in production software. These evaluations support our thesis statement in Section 1.4, that we can provide strong protect to sensitive data in systems software at low overhead

Our contributions are summarized as:

1. Chapter 3 is an exhaustive survey and systematization of the CFI literature with the first direct comparison of these mechanisms on common platforms and security evaluation using quantitative and qualitative metric.

2. In Chapter 4 we present VTrust, a new mechanism for protecting virtual function calls in C++ with lower overhead than state-of-the-art mechanisms and modularity support.

3. In Chapter 5, we present the design of DCI, and its implementation DataShield. DCI is a novel technique for protecting programmer-selected sensitive data.

Future work for VTrust and our CFI survey are to develop more precise analyses, more precise than type-based analysis, and to develop a metric or method that accurately measures the security provided by a CFI mechanism. For DCI, our work could be expanded by generalizing the method for choosing the subset of data to protect automatically. This would make our technique applicable to more types of software and easier to adopt.

Our techniques provide strong protection, as evidenced by detecting CVEs found in production software, at low over head – 35.7% in our case study of applying DCI to mbedTLS, and under 6% for VTrust across both SPEC CPU2006 and Firefox.

REFERENCES

REFERENCES

[1] Neel Mehta and Codenomicon. The Heartbleed Bug. `http://heartbleed.com/`.

[2] The Rust Project Developers. The Rust Programming Language. https://doc.rust-lang.org/stable/book/.

[3] Galen Hunt and Jim Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 2007.

[4] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference (ATC)*, 2002.

[5] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *Transactions on Programming Languages and Systems (TOPLAS)*, 2005.

[6] ISO/IEC. ISO International Standard ISO/IEC 14882:2014(E). `https://isocpp.org/std/the-standard`, 2014.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[8] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[9] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[10] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management (ISMM)*, 2010.

[11] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[12] Itanium C++ ABI. `http://mentorembedded.github.io/cxx-abi/abi.html`.

[13] Microsoft. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/, 2006.

[14] Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[15] PaX-Team. PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[16] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[17] ISO/IEC. ISO International Standard ISO/IEC 9899:2011. http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2FISO%2FIEC+9899-2012, 2012.

[18] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michale Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Protection, security and performance. *ACM Computing Surveys (CSUR)*, 2018.

[19] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. VTrust: Regaining trust on virtual calls. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[20] Scott A. Carr and Mathias Payer. DataShield: Configurable data confidentiality and integrity. In *Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[21] Scott A. Carr and Mathias Payer. Poster: Data confidentiality and integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[22] Xinyang Ge, Mathias Payer, and Trent Jaeger. An evil copy: How the loader betrays you. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[23] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[24] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[25] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[26] Arjan van de Ven and Ingo Molnar. Exec Shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[27] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[28] Scott A. Carr, Francesco Logozzo, and Mathias Payer. Automatic contract insertion with CCBot. *Transactions on Software Engineering (TSE)*, 2016.

[29] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[30] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 1999.

[31] Mathias Payer. Hexpads: A platform to detect stealth attacks. In *International Symposium on Engineering Secure Software and Systems*, 2016.

[32] Ben Niu and Gang Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Conference on Computer and Communications Security (CCS)*, 2014.

[33] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Conference on Computer and Communications Security (CCS)*, 2005.

[34] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[36] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. Code pointer masking: Hardening applications against code injection attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[37] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[38] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, 2013.

[39] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[40] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Stephen McCamant, and Laszlo Szekeres. Protecting function pointers in binary. In *Asia Conference on Computer and Communications Security (ASIACCS)*, 2013.

[41] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

[42] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *Conference on Computer and Communications Security (CCS)*, 2013.

[43] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Conference on Computer and Communications Security (CCS)*, 2015.

[44] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[45] Ben Niu and Gang Tan. Modular control-flow integrity. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[46] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[47] Michael Hicks. What is memory safety? `http://www.pl-enthusiast.net/2014/07/21/memory-safety/`, 2014.

[48] Matthew S. Simpson and Rajeev K. Barua. Memsafe: ensuring the spatial and temporal memory safety of cat runtime. *Software: Practice and Experience*, 2013.

[49] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.

[50] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[51] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, 2010.

[52] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security Symposium*, 2006.

[53] The Chromium Projects. Undefined behavior sanitizer for Chromium. `http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer`, November 2014.

[54] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium*, 2015.

[55] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *Conference on Computer and Communications Security (CCS)*, 2016.

[56] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[57] Bill McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[58] Vladimir Kiriansky, Derek Bruening, Saman P Amarasinghe, et al. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

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

[60] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[61] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[62] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[63] James R Bell. Threaded code. *Communications of the ACM*, June 1973.

[64] P. M. Kogge. An architectural trail to threaded-code systems. *Computer*, March 1982.

[65] Eddy H Debaere and Jan M van Campenhout. *Interpretation and instruction path coprocessing*. Computer Systems. MIT Press, 1990.

[66] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Conference on Computer and Communications Security (CCS)*, 2007.

[67] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Conference on Computer and Communications Security (CCS)*, 2010.

[68] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 2012.

[69] Yannis Smaragdakis and George Balatsouras. Pointer Analysis. *Foundations and Trends in Programming Languages*, 2015.

[70] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. 1999.

[71] Michael Hind and Anthony Pioli. Which pointer analysis should I use? *ACM SIGSOFT Software Engineering Notes*, September 2000.

[72] Michael Hind. Pointer analysis. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[73] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis*, 1981.

[74] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *ACM SIGSOFT Software Engineering Notes*, 2002.

[75] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well. *ACM SIGPLAN Notices*, January 2011.

[76] O Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? *Compiler Construction*, 2006.

[77] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[78] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices*, October 1996.

[79] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, November 2001.

[80] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[81] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *Symposium on Foundations of Software Engineering*, 2006.

[82] Ben Hardekopf and Calvin Lin. The ant and the grasshopper. In *Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[83] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Symposium on Code Generation and Optimization (CGO)*, 2011.

[84] Atanas Rountev, Scott Kagan, and Michael Gibas. Evaluating the imprecision of static analysis. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2004.

[85] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, October 2000.

[86] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[87] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[88] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.

[89] Jannik Pewny and Thorsten Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[90] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert Huijie Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[91] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[92] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[93] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Design Automation Conference (DAC)*, 2014.

[94] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin Hamlen, and Michael Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[95] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In *Conference on Computer and Communications Security (CCS)*, 2015.

[96] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[97] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[98] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Conference on Computer and Communications Security (CCS)*, 2015.

[99] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-assisted fine-grained code-reuse attack detection. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[100] Microsoft. Visual studio 2015 — compiler options — enable control flow guard, 2015. `https://msdn.microsoft.com/en-us/library/dn919635.aspx`.

[101] Dimitar Bounov, Rami Kici, and Sorin Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[102] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[103] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-grained control-flow integrity through binary hardening. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2015.

[104] Orlando Arias, Lucas Davi, Matthias Hanreich, Yier Jin, Patrick Koeberl, Debayan Paul, Ahmad-Reza Sadeghi, and Dean Sullivan. HAFIX: Hardware-assisted flow integrity extension. In *Design Automation Conference (DAC)*, 2015.

[105] Andrei Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[106] Ben Niu and Gang Tan. MCFI readme. `https://github.com/mcfi/MCFI/blob/master/README.md`, 2015.

[107] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Conference on Computer and Communications Security (CCS)*, 2015.

[108] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Conference on Computer and Communications Security (CCS)*, 2015.

[109] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *International Conference on Formal Methods and Software Engineering (ICFEM)*, 2005.

[110] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Asia Conference on Computer and Communications Security (ASIACCS)*, 2015.

[111] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[112] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters: Don't trust folklore. In *Symposium on Code Generation and Optimization (CGO)*, 2015.

[113] Intel Inc. Intel 64 and IA-32 architectures. software developer's manual, 2013.

[114] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Conference on Dependable Systems and Networks (DSN)*, 2012.

[115] Ivan Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks.
`http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf`, 2012.

[116] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58):`http://phrack.com/issues.html?issue=67&id=8`, 2007.

[117] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Asia Conference on Computer and Communications Security (ASIACCS)*, 2011.

[118] Volodymyr Kuzentsov, Laszlo Szekeres, Mathias Payer, George Candea, Dawn Song, and R. Sekar. Code pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[119] Microsoft. Visual Studio 2015 Preview: Work-in-progress security feature. `http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx`.

[120] Caroline Tice. Improving function pointer security for virtual method dispatches. In *GNU Tools Cauldron Workshop*, 2012.

[121] Microsoft. Software vulnerability exploitation trends: Exploring the impact of software mitigations on patterns of vulnerability exploitation (2013). `http://download.microsoft.com/download/F/D/F/FDFBE532-91F2-4216-9916-2620967CEAF4/Software%20Vulnerability%20Exploitation%20Trends.pdf`, 2013.

[122] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser JIT engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[123] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.

[124] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *USENIX Security Symposium*, 2015.

[125] BlueLotus Team. BCTF challenge: Bypass vtable read-only checks. `https://github.com/ctfs/write-ups-2015/tree/master/bctf-2015/exploit/zhongguancun`, 2015.

[126] Istvan Haller, Enes Gkta, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. ShrinkWrap: VTable protection without loose ends. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[127] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[128] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Symposium on Code Generation and Optimization (CGO)*, 2004.

[129] Peter Collingbourne. LLVM — control flow integrity, 2015. `http://clang.llvm.org/docs/ControlFlowIntegrity.html`.

[130] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Internet Measurement Conference*, 2014.

[131] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[132] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[133] Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. Case studies and tools for contract specifications. In *International Conference on Software Engineering (ICSE)*, 2014.

[134] H.-Christian Estler, CarloA. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in Practice. In *Formal Methods*, 2014.

[135] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *USENIX Security Symposium*, 2016.

[136] Gene Novark and Emery D. Berger. Dieharder: Securing the heap. In *Conference on Computer and Communications Security (CCS)*, 2010.

[137] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming (ECOOP)*, 1991.

[138] Microsoft IE. LiteBrite: HTML, CSS and JavaScript Performance Benchmark. `http://ie.microsoft.com/testdrive/Performance/LiteBrite/`, 2014.

[139] Google. Octane JavaScript benchmark suite. `https://developers.google.com/octane/`, 2014.

[140] Mozilla. Kraken 1.1 JavaScript benchmark suite. `http://krakenbenchmark.mozilla.org/`, 2014.

[141] Apple. Sunspider 1.0.2 JavaScript benchmark suite. `https://www.webkit.org/perf/sunspider/sunspider.html`, 2014.

[142] RightWare. Browsermark 2.1 benchmark. `http://browsermark.rightware.com/`, 2014.

[143] FutureMark. Peacekeeper: HTML5 browser speed test. `http://peacekeeper.futuremark.com/`, 2014.

[144] Guido Vranken. CVE-2015-5291: remote heap corruption in ARM mbed TLS / PolarSSL, October 2015.

[145] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *Design Automation Conference (DAC)*, 2016.

[146] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-enforced control-flow integrity. In *Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[147] Baiju 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/`.

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

[149] Francisco 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.

[150] David Weston and Matt Miller. Windows 10 mitigation improvements. Black-Hat'16 `https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf`, 2016.

[151] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 2003.

[152] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[153] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *International Conference on Software Engineering (ICSE)*, 2006.

[154] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software: Practice and Experience*, January 1997.

[155] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference (ATC)*, 2005.

[156] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[157] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[158] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Symposium on Code Generation and Optimization (CGO)*, 2014.

[159] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[160] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[161] Istvan Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. METAlloc: Efficient and comprehensive metadata management for software security hardening. In *European Workshop on System Security*, 2016.

[162] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An efficient pointer arithmetic checker for C programs. In *Asia Conference on Computer and Communications Security (ASIACCS)*, 2010.

[163] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Conference on Computer and Communications Security (CCS)*, 2004.

[164] Matthew R Miller, Kenneth D Johnson, and Timothy William Burrell. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities, 2014. US Patent 8,683,583.

[165] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[166] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Ben Zorn. Modular protections against non-control data attacks. In *IEEE Computer Security Foundations Symposium (CSF)*, 2011.

[167] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai: Protecting critical data in unsafe languages. In *European Conference on Computer Systems (EUROSYS)*, 2008.

[168] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[169] Mathias Payer, Tobias Hartmann, and Thomas R Gross. Safe loading: A foundation for secure execution of untrusted programs. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[170] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (S&P)*, 2009.

[171] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *Conference on Virtual Execution Environments (VEE)*, 2011.

VITA

VITA

Scott A. Carr

EDUCATION

- Doctor of Philosophy, May 2017
  Computer Science, Purdue University
- Master of Science in Engineering, December 2011
  Computer Engineering, University of Michigan-Dearborn
- Bachelor of Science, May 2006
  Computer Engineering, Lawrence Technological University
  Summa Cum Laude

TEACHING EXPERIENCE

- Teaching Assistant, Purdue University, CS510 Software Engineering, Spring 2015.

PROFESSIONAL EXPERIENCE

Research Intern **Summer 2016**
Mozilla Corporation, San Francisco, CA.
Research Intern **Summer 2015**
Microsoft Research, Redmond, WA.
Research Intern **Spring 2014**
Microsoft Research, Redmond, WA.
Computer Scientist **July 2011**
Michigan Aerospace Corporation, Ann Arbor, MI.
Engineer in Research Senior **March 2008**
University of Michigan, Ann Arbor, MI.

PUBLICATIONS

1. Burow, N., **Carr, S. A.**, Brunthaler, S. Payer M., Nash, J., Larsen, P., Franz, M. *Control-flow Integrity: Protection, Security, and Performance*, ACM Computing Surveys 2018. To appear.

2. **Carr, S. A.**, Payer M, *DataShield: Configurable Data Confidentiality and Integrity*, Asia Conference on Communications and Computer Security 2017.

3. **Carr, S. A.**, Payer, M., Logozzo, F., *Automatic Contract Insertion with CCBot*, IEEE Transactions on Software Engineering 2016.

4. Zhang, C., **Carr, S. A.**, Li, T., Ding, Y., Song, C., Payer, M., Song, D., *VTrust Regaining Trust on Your Virtual Calls*, Network and Distributed Systems Security Symposium 2016.

5. **Carr, S. A.**, Payer M. *Poster: Data Confidentiality and Integrity.* IEEE Symposium on Security and Privacy (S&P) 2015.

6. **Carr, S. A.** Pittman, N., *Extensions to gNOSIS to Support Static Analysis of System Verilog HDL Code*, MSR-TR-2015-68, 2015.

7. Kim, S., Adams, D. E., Sohn, H., Rodriguez Rivera, G., Vitek, J., **Carr, S. A.**, and Grama, A., *Validation of Vibro-Acoustic Modulation of wind turbine blades for structural health monitoring using operational vibration as a pumping signal*, 2013, Proceedings of the 9th International Workshop on Structural Health Monitoring, Palo Alto, CA.