Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. "ConFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software." In *Proc. 28th USENIX Security Symposium*, August 2019.

ConFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software

Xiaoyang Xu
University of Texas at Dallas

Masoud Ghaffarinia*
University of Texas at Dallas

Wenhao Wang*
University of Texas at Dallas

Kevin W. Hamlen
University of Texas at Dallas

Zhiqiang Lin
Ohio State University

Abstract

ConFIRM (CONtrol-Flow Integrity Relevance Metrics) is a new evaluation methodology and microbenchmarking suite for assessing compatibility, applicability, and relevance of *control-flow integrity* (CFI) protections for preserving the intended semantics of software while protecting it from abuse. Although CFI has become a mainstay of protecting certain classes of software from code-reuse attacks, and continues to be improved by ongoing research, its ability to preserve intended program functionalities (*semantic transparency*) of diverse, mainstream software products has been under-studied in the literature. This is in part because although CFI solutions are evaluated in terms of performance and security, there remains no standard regimen for assessing compatibility. Researchers must often therefore resort to anecdotal assessments, consisting of tests on homogeneous software collections with limited variety (e.g., GNU Coreutils), or on CPU benchmarks (e.g., SPEC) whose limited code features are not representative of large, mainstream software products.

Reevaluation of CFI solutions using ConFIRM reveals that there remain significant unsolved challenges in securing many large classes of software products with CFI, including software for market-dominant OSes (e.g., Windows) and code employing certain ubiquitous coding idioms (e.g., event-driven callbacks and exceptions). An estimated 47% of CFI-relevant code features with high compatibility impact remain incompletely supported by existing CFI algorithms, or receive weakened controls that leave prevalent threats unaddressed (e.g., return-oriented programming attacks). Discussion of these open problems highlights issues that future research must address to bridge these important gaps between CFI theory and practice.

1 Introduction

Control-flow integrity (CFI) [1] (supported by vtable protection [29] and/or software fault isolation [73]), has emerged as one of the strongest known defenses against modern control-flow hijacking attacks, including return-oriented programming (ROP) [60] and other code-reuse attacks. These attacks trigger dataflow vulnerabilities (e.g., buffer overflows) to manipulate control data (e.g., return addresses) to hijack victim software. By restricting program execution to a set of legitimate control-flow targets at runtime, CFI can mitigate many of these threats.

Inspired by the initial CFI work [1], there has been prolific new research on CFI in recent years, mainly aimed at improving performance, enforcing richer policies, obtaining higher assurance of policy-compliance, and protecting against more subtle and sophisticated attacks. For example, between 2015–2018 over 25 new CFI algorithms appeared in the top four applied security conferences alone. These new frameworks are generally evaluated and compared in terms of performance and security. Performance overhead is commonly evaluated in terms of the CPU benchmark suites (e.g., SPEC), and security is often assessed using the RIPE test suite [80] or with manually crafted proof-of-concept attacks (e.g., COOP [62]). For example, a recent survey systematically compared various CFI mechanisms against these metrics for precision, security, and performance [13].

While this attention to performance and security has stimulated rapid gains in the ability of CFI solutions to efficiently enforce powerful, precise security policies, less attention has been devoted to systematically examining which general classes of software can receive CFI protection without suffering compatibility problems. Historically, CFI research has struggled to bridge the gap between theory and practice (cf., [84]) because code hardening transformations inevitably run at least some risk of corrupting desired, policy-permitted program functionalities. For example, introspective programs that read their own code bytes at runtime (e.g., many VMs, JIT compilers, hot-patchers, and dynamic linkers) can break after their code bytes have been modified or relocated by CFI.

Compatibility issues of this sort have dangerous security ramifications if they prevent protection of software needed in mission-critical contexts, or if the protections must be weak-

*These authors contributed equally to this work.

ened in order to achieve compatibility. For example, due in part to potential incompatibilities related to *return address introspection* (wherein some callees read return addresses as arguments) the three most widely deployed compiler-based CFI solutions (LLVM-CFI [69], GCC-VTV [69], and Microsoft Visual Studio MCFG [66]) all presently leave return addresses unprotected, potentially leaving code vulnerable to ROP attacks—the most prevalent form of code-reuse.

Understanding these compatibility limitations, including their impacts on real-world software performance and security, requires a new suite of CFI functional tests with substantially different characteristics than benchmarks typically used to assess compiler or hardware performance. In particular, CFI relevance and effectiveness is typically constrained by the nature and complexity of the target program's *control-flow paths* and *control data dependencies*. Such complexities are not well represented by SPEC benchmarks, which are designed to exercise CPU computational units using only simple control-flow graphs, or by utility suites (e.g., GNU Coreutils) that were all written in a fairly homogeneous programming style for a limited set of compilers, and that use a very limited set of standard libraries chosen for exceptionally high cross-compatibility.

To better understand the compatibility and applicability limitations of modern CFI solutions on diverse, modern software products, and to identify the coding idioms and features that constitute the greatest barriers to more widespread CFI adoption, we present ConFIRM (CONtrol-Flow Integrity Relevance Metrics), a new suite of CFI tests designed to exhibit code features most relevant to CFI evaluation.[1] Each test is designed to exhibit one or more control-flow features that CFI solutions must guard in order to enforce integrity, that are found in a large number of commodity software products, but that pose potential problems for CFI implementations.

It is infeasible to capture in a single test set the full diversity of modern software, which embodies myriad coding styles, build processes (e.g., languages, compilers, optimizers, obfuscators, etc.), and quality levels. We therefore submit ConFIRM as an extensible baseline for testing CFI compatibility, consisting of code features drawn from experiences building and evaluating CFI and randomization systems for several architectures, including Linux, Windows, Intel x86/x64, and ARM32 in academia and industry [7, 33, 45, 47, 75, 77–79].

Our work is envisioned as having the following qualitative impacts: (1) CFI designers (e.g., compiler developers) can use ConFIRM to detect compatibility flaws in their designs that are currently hard to anticipate prior to full scale productization. This can lower the currently steep barrier between prototype and distributable product. (2) Defenders (e.g., developers of secure software) can use ConFIRM to better evaluate code-reuse defenses, in order to avoid false senses of security. (3) The research community can use ConFIRM to identify and prioritize missing protections as important open problems worthy of future investigation.

We used ConFIRM to reevaluate 12 publicly available CFI implementations published in the open literature. The results show that about 47% of solution-test pairs exhibit incompatible or insecure operation for code features needed to support mainstream software products, and a *cross-thread stack-smashing* attack defeats all tested CFI defenses. Microbenchmarking additionally reveals some performance/compatibility trade-offs not revealed by purely CPU-based benchmarking.

In summary, our contributions include the following:

- We present ConFIRM, the first testing suite designed specifically to test compatibility characteristics relevant to control-flow security hardening evaluation.
- A set of 20 code features and coding idioms are identified, that are widely found in deployed, commodity software products, and that pose compatibility, performance, or security challenges for modern CFI solutions.
- Evaluation of 12 CFI implementations using ConFIRM reveals that existing CFI implementations are compatible with only about half of code features and coding idioms needed for broad compatibility, and that microbenchmarking using ConFIRM reveals performance trade-offs not exhibited by SPEC benchmarks.
- Discussion and analysis of these results highlights significant unsolved obstacles to realizing CFI protections for widely deployed, mainstream, commodity products.

Section 2 begins with a summary of technical CFI attack and defense details important for understanding the evaluation approach. Section 3 next presents ConFIRM's evaluation metrics in detail, including a rationale behind why each metric was chosen, and how it impacts potential defense solutions; and Section 4 describes implementation of the resulting tests. Section 5 reports our evaluation of CFI solutions using ConFIRM and discusses significant findings. Finally, Section 6 describes related work and Section 7 concludes.

2 Background

CFI defenses first emerged from an arms race against early code-injection attacks, which exploit memory corruptions to inject and execute malicious code. To thwart these malicious code-injections, hardware and OS developers introduced Data Execution Prevention (DEP), which blocks execution of injected code. Adversaries proceeded to bypass DEP with "return-to-libc" attacks, which redirect control to existing, abusable code fragments (often in the C standard libraries) without introducing attacker-supplied code. In response, defenders introduced Address Space Layout Randomization (ASLR), which randomizes code layout to frustrate its abuse. DEP and ASLR motivated adversaries to craft even

[1] https://github.com/SoftwareLanguagesSecurityLab/ConFIRM

more elaborate attacks, including ROP and Jump-Oriented Programming (JOP) [11], which locate, chain, and execute short instruction sequences (gadgets) of benign code to implement malicious payloads.

CFI emerged as a more comprehensive and principled defense against this malicious code-reuse. Most realizations consist of two main phases: (1) A program-specific *control-flow policy* is first formalized as a (possibly dynamic) control-flow graph (CFG) that whitelists the code's permissible control-flow transfers. (2) To constrain all control flows to the CFG, the program code is instrumented with guard code at all computed (e.g., indirect) control-flow transfer sites. The guard code decides at runtime whether each impending transfer satisfies the policy, and blocks it if not. The guards are designed to be uncircumventable by confronting attackers with a chicken-and-egg problem: To circumvent a guard, an attack must first hijack a control transfer; but since all control transfers are guarded, hijacking a control transfer requires first circumventing a guard.

Both CFI phases can be source-aware (implemented as a source-to-source transformation, or introduced during compilation), or source-free (implemented as a binary-to-binary transformation). Source-aware solutions typically benefit from source-level information to derive more precise policies, and can often perform more optimization to achieve better performance. Examples include WIT [5], NaCl [81], CFL [11], MIP [48], MCFI [49], RockJIT [50], Forward CFI [69], CCFI [42], πCFI [51], MCFG [66] CFIXX [14] and μCFI [35]. In contrast, source-free solutions are potentially applicable to a wider domain of software products (e.g., closed-source), and have a more flexible deployment model (e.g., consumer-side enforcement without developer assistance). These include XFI [26], Reins [78], STIR [77], CCFIR [84], bin-CFI [87], BinCC [74], Lockdown [54], TypeArmor [72], OCFI [45], OFI [75] and τCFI [47].

The advent of CFI is a significant step forward for defenders, but was not the end of the arms race. In particular, each CFI phase introduces potential loopholes for attackers to exploit. First, it is not always clear which policy should be enforced to fully protect the code. Production software often includes complex control-flow structures, such as those introduced by object-oriented programming (OOP) idioms, from which it is difficult (even undecidable) to derive a CFG that precisely captures the policy desired by human developers and users. Second, the instrumentation phase must take care not to introduce guard code whose decision procedures constitute unacceptably slow runtime computations [34]. This often results in an enforcement that imprecisely approximates the policy. Attackers have taken advantage of these loopholes with ever more sophisticated attacks, including Counterfeit Object Oriented Programming (COOP) [62], Control Jujutsu [28], and Control-Flow Bending [15].

These weaknesses and threats have inspired an array of new and improved CFI algorithms and supporting technologies in recent years. For example, to address loopholes associated with OOP, *vtable protections* prevent or detect virtual method table corruption at or before control-flow transfers that depend on method pointers. Source-aware vtable protections include GNU VTV [68], CPI [40], SAFEDISPATCH [37], Readactor++ [19], and VTrust [82]; whereas source-free instantiations include T-VIP [29], VTint [83], and VfGuard [58].

However, while the security and performance trade-offs of various CFI solutions have remained actively tracked and studied by defenders throughout the arms race, attackers are increasingly taking advantage of CFI compatibility limitations to exploit unprotected software, thereby avoiding CFI defenses entirely. For example, 88% of CFI defenses cited herein have only been realized for Linux software, but over 95% of desktops worldwide are non-Linux.[2] These include many mission-critical systems, including over 75% of control systems in the U.S. [39], and storage repositories for top secret military data [53]. None of the top 10 vulnerabilities exploited by cybercriminals in 2017 target Linux software [25].

While there is a hope that small-scale prototyping will result in principles and approaches that eventually scale to more architectures and larger software products, follow-on works that attempt to bridge this gap routinely face significant unforeseen roadblocks. We believe many of these obstacles remain unforeseen because of the difficulty of isolating and studying many of the problematic software features lurking within large, commodity products, which are not well represented in open-source codes commonly available for study by researchers during prototyping.

The goal of this research is therefore to describe and analyze a significant collection of code features that are routinely found in large software products, but that pose challenges to effective CFI enforcement; and to make available a suite of CFI test programs that exhibit each of these features on a small scale amenable to prototype development. The next section discusses this feature set in detail.

3 Compatibility Metrics

To measure compatibility of CFI mechanisms, we propose a set of metrics that each includes one or more code features from either C/C++ source code or compiled assembly code. We derived this feature set by attempting to apply many CFI solutions to large software products, then manually testing the functionalities of the resulting hardened software for correctness, and finally debugging each broken functionality step-wise at the assembly level to determine what caused the hardened code to fail. Since many failures manifest as subtle forms of register or memory corruption that only cause the program to crash or malfunction long after the failed operation completes, this debugging constitutes many hundreds

[2] http://gs.statcounter.com/os-market-share/desktop/worldwide

Table 1: CONFIRM compatibility metrics

Compatibility metric	Real-world software examples
Function Pointers	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Callbacks	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, TeXstudio, Visual Studio, Windows Defender, WinSCP
Dynamic Linking	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Delay-Loading	Adobe Reader, Calculator, Chrome, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, Visual Studio, WinSCP
Exporting/Importing Data	7-Zip, Apache, Calculator, Chrome, Dropbox, Firefox, MS Paint, MS PowerPoint, PowerShell, TeXstudio, UPX, Visual Studio
Virtual Functions	7-Zip, Adobe Reader, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, TeXstudio, Visual Studio, Windows Defender, WinSCP
CODE-COOP Attack	Programs built on GTK+ or Microsoft COM can pass objects to trusted modules as arguments.
Tail Calls	Mainstream compilers provide options for tail call optimization. e.g. `/O2` in MSVC, `-O2` in GCC, and `-O2` in LLVM.
Switch-Case Statements	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, PuTTY, TeXstudio, Visual Studio, WinSCP
Returns	Every benign program has returns.
Unmatched Call/Return Pairs	Adobe Reader, Apache, Chrome, Firefox, JVM, MS PowerPoint, Visual Studio
Exceptions	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, Visual Studio, Windows Defender, WinSCP
Calling Conventions	Every program adopts one or more calling convention.
Multithreading	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
TLS Callbacks	Adobe Reader, Chrome, Firefox, MS Paint, TeXstudio, UPX
Position-Independent Code	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Memory Protection	7-Zip, Adobe Reader, Apache, Chrome, Dropbox, Firefox, MS PowerPoint, PotPlayer, TeXstudio, Visual Studio, Windows Defender, WinSCP
JIT Compiler	Adobe Flash, Chrome, Dropbox, Firefox, JVM, MS PowerPoint, PotPlayer, PowerShell, Skype, Visual Studio, WinSCP
Self-Unpacking	Programs decompressed by self-extractors (e.g., UPX, NSIS).
Windows API Hooking	Microsoft Office family software, including MS Excel, MS PowerPoint, etc.

Table 2: Source code compiled to indirect call

Source code	Assembly code
1 void foo() { return; }	
2 void bar() { return; }	
3 void main() {	
4 void (*fptr)();	1 ...
5 int n = input();	2 call _input
6 if (n)	3 test eax, eax
7 fptr = foo;	4 mov edx, offset_foo
8 else	5 mov ecx, offset_bar
9 fptr = bar;	6 cmovnz ecx, edx
10 fptr();	7 call ecx
11 }	8 ...

of person-hours amassed over several years of development experience involving CFI-protected software.

Table 1 presents the resulting list of code features organized into one row for each root cause of failure. Column two additionally lists some widely available, commodity software products where each of these features can be observed in non-malicious software in the wild. This demonstrates that each feature is representative of real-world software functionalities that must be preserved by CFI implementations in order for their protections to be usable and relevant in contexts that deploy these and similar products.

3.1 Indirect Branches

We first discuss compatibility metrics related to the code feature of greatest relevance to most CFI works: indirect branches. Indirect branches are control-flow transfers whose destination addresses are computed at runtime—via pointer arithmetic and/or memory-reads. Such transfers tend to be of high interest to attackers, since computed destinations are more prone to manipulation. CFI defenses therefore guard indirect branches to ensure that they target permissible destinations at runtime. Indirect branches are commonly categorized into three classes: indirect calls, indirect jumps, and returns.

Table 2 shows a simple example of source code being compiled to an indirect call. The function called at source line 5 depends on user input. This prevents the compiler from generating a direct branch that targets a fixed memory address at compile time. Instead, the compiler generates a register-indirect call (assembly line 7) whose target is computed at runtime. While this is one common example of how indirect branches arise, in practice they are a result of many different programming idioms, discussed below.

Function Pointers. Calls through function pointers typically compile to indirect calls. For example, using gcc with the `-O2` option generates register-indirect calls for function pointers, and MSVC does so by default.

Callbacks. Event-driven programs frequently pass function pointers to external modules or the OS, which the receiving code later dereferences and calls in response to an event. These callback pointers are generally implemented by using function pointers in C, or as method references in C++. Callbacks can pose special problems for CFI, since the call site is not within the module that generated the pointer. If the call site is within a module that cannot easily be modified (e.g., the OS kernel), it must be protected in some other way, such as by sanitizing and securing the pointer before it is passed.

Dynamic Linking. Dynamically linked shared libraries reduce program size and improve locality. But dynamic linking has been a challenge for CFI compatibility because CFG edges that span modules may be unavailable statically.

In Windows, *dynamically linked libraries* (DLLs) can be loaded into memory at load time or runtime. In load-time dynamic linking, a function call from a module to an exported DLL function is usually compiled to a memory-indirect call targeting an address stored in the module's *import address table* (IAT). But if this function is called more than once, the compiler first moves the target address to a register, and then generates register-indirect calls to improve execution performance. In run-time dynamic linking, a module calls APIs, such as `LoadLibrary()`, to load the DLL at runtime. When loaded into memory, the module calls the `GetProcAddress()` API to retrieve the address of the exported function, and then calls the exported function using the function pointer returned by `GetProcAddress()`.

Additionally, MSVC (since version 6.0) provides linker support for delay-loaded DLLs using the `/DELAYLOAD` linker option. These DLLs are not loaded into memory until one of their exported functions is invoked.

In Linux, a module calls functions exported by a shared library by calling a stub in its *procedure linkage table* (PLT). Each stub contains a memory-indirect jump whose target depends on the writable, lazy-bound *global offset table* (GOT). As in Windows, an application can also load a module at runtime using function `dlopen()`, and retrieve an exported symbol using function `dlsym()`.

Supporting dynamic and delay-load linkage is further complicated by the fact that shared libraries can also export data pointers within their export tables in both Linux and Windows. CFI solutions that modify export tables must usually treat code and data pointers differently, and must therefore somehow distinguish the two types to avoid data corruptions.

Virtual Functions. Polymorphism is a key feature of OOP languages, such as C++. Virtual functions are used to support runtime polymorphism, and are implemented by C++ compilers using a form of late binding embodied as *virtual tables* (vtables). The tables are populated by code pointers to virtual function bodies. When an object calls a virtual function, it indexes its vtable by a function-specific constant, and flows control to the memory address read from the table. At the assembly level, this manifests as a memory-indirect call. The ubiquity and complexity of this process has made vtable hijacking a favorite exploit strategy of attackers.

Some CFI and vtable protections address vtable hijacking threats by guarding call sites that read vtables, thereby detecting potential vtable corruption at time-of-use. Others seek to protect vtable integrity directly by guarding writes to them. However, both strategies are potentially susceptible to COOP [62] and CODE-COOP [75] attacks, which replace one vtable with another that is legal but is not the one the original code intended to call. The defense problem is further complicated by the fact that many large classes of software (e.g., GTK+ and Microsoft COM) rely upon dynamically generated vtables. CFI solutions that write-protect vtables or whose guards check against a static list of permitted vtables are incompatible with such software.

Tail Calls. Modern C/C++ compilers can optimize tail-calls by replacing them with jumps. Row 8 of Table 1 lists relevant compiler options. With these options, callees can return directly to ancestors of their callers in the call graph, rather than to their callers. These mismatched call/return pairs affect precision of some CFG recovery algorithms.

Switch-case Statements. Many C/C++ compilers optimize switch-case statements via a static dispatch table populated with pointers to case-blocks. When the switch is executed, it calculates a dispatch table index, fetches the indexed code pointer, and jumps to the correct case-block. This introduces memory-indirect jumps that refer to code pointers not contained in any vtable, and that do not point to function boundaries. CFI solutions that compare code pointers to a whitelist of function boundaries can therefore cause the switch-case code to malfunction. Solutions that permit unrestricted indirect jumps within each local function risk unsafety, since large functions can contain abusable gadgets.

Returns. Nearly every benign program has returns. Unlike indirect branches whose target addresses are stored in registers or non-writable data sections, return instructions read their destination addresses from the stack. Since stacks are typically writable, this makes return addresses prime targets for malicious corruption.

On Intel-based CISC architectures, return instructions have one of the shortest encodings (1 byte), complicating the efforts of source-free solutions to replace them in-line with secured equivalent instruction sequences. Additionally, many hardware architectures heavily optimize the behavior of returns (e.g., via speculative execution powered by shadow stacks for call/return matching). Source-aware CFI solutions that replace returns with some other instruction sequence can therefore face stiff performance penalties by losing these optimization advantages.

Unmatched call/return Pairs. Control-flow transfer mechanisms, including exceptions and setjmp/longjmp, can yield flows in which the relation between executed call instructions

and executed return instructions is not one-to-one. For example, exception-handling implementations often pop stack frames from multiple calls, followed by a single return to the parent of the popped call chain. Shadow stack defenses that are implemented based on traditional call/return matching may be incompatible with such mechanisms.

3.2 Other Metrics

While indirect branches tend to be the primary code feature of interest to CFI attacks and defenses, there are many other code features that can also pose control-flow security problems, or that can become inadvertently corrupted by CFI code transformation algorithms, and that therefore pose compatibility limitations. Some important examples are discussed below.

Multithreading. With the rise of multicore hardware, multithreading has become a centerpiece of software efficiency. Unfortunately, concurrent code execution poses some serious safety problems for many CFI algorithms.

For example, in order to take advantage of hardware call-return optimization (see §3.1), most CFI algorithms produce code containing guarded return instructions. The guards check the return address before executing the return. However, on parallelized architectures with flat memory spaces, this is unsafe because any thread can potentially write to any other (concurrently executing) thread's return address at any time. This introduces a TOCTOU vulnerability in which an attacker-manipulated thread corrupts a victim thread's return address after the victim thread's guard code has checked it but before the guarded return executes. We term this a cross-thread stack-smashing attack. Since nearly all modern architectures combine concurrency, flat memory spaces, and returns, this leaves almost all CFI solutions either inapplicable, unsafe, or unacceptably inefficient for a large percentage of modern production software.

Position-Independent Code. *Position-independent code* (PIC) is designed to be relocatable after it is statically generated, and is a standard practice in the creation of shared libraries. Unfortunately, the mechanisms that implement PIC often prove brittle to code transformations commonly employed for source-free CFI enforcement. For example, PIC often achieves its position independence by dynamically computing its own virtual memory address (e.g., by performing a call to itself and reading the pushed return address from the stack), and then performing pointer arithmetic to locate other code or data at fixed offsets relative to itself. This procedure assumes that the relative positions of PIC code and data are invariant even if the base address of the PIC block changes.

However, CFI transforms typically violate this assumption by introducing guard code that changes the sizes of code blocks, and therefore their relative positions. To solve this, PIC-compatible CFI solutions must detect the introspection and pointer arithmetic operations that implement PIC and adjust them to compute corrected pointer values. Since there are typically an unlimited number of ways to perform these computations at both the source and native code levels, CFI detection of these computations is inevitably heuristic, allowing some PIC instantiations to malfunction.

Exceptions. Exception raising and handling is a mainstay of modern software design, but introduces control-flow patterns that can be problematic for CFI policy inference and enforcement. Object-oriented languages, such as C++, boast first-class exception machinery, whereas standard C programs typically realize exceptional control-flows with gotos, longjumps, and signals. In Linux, compilers (e.g., gcc) implement C++ exception handling in a table-driven approach. The compiler statically generates read-only tables that hold exception-handling information. For instance, gcc produces a `gcc_except_table` comprised of *language-specific data areas* (LSDAs). Each LSDA contains various exception-related information, including pointers to exception handlers.

In Windows, *structured exception handling* (SEH) extends the standard C language with first-class support for both hardware and software exceptions. SEH uses stack-based exception nodes, wherein exception handlers form a linked list on the stack, and the list head is stored in the *thread information block* (TIB). Whenever an exception occurs, the OS fetches the list head and walks through the SEH list to find a suitable handler for the thrown exception. Without proper protection, these exception handlers on the stack can potentially be overwritten by an attacker. By triggering an exception, the attacker can then redirect the control-flow to arbitrary code. CFI protection against these SEH attacks is complicated by the fact that code outside the vulnerable module (e.g., in the OS and/or system libraries) uses pointer arithmetic to fetch, decode, and call these pointers during exception handling. Thus, suitable protections must typically span multiple modules, and perhaps the OS kernel.

From Windows XP onward, applications have additionally leveraged *vectored exception handling* (VEH). Unlike SEH, VEH is not stack-based; applications register a global handler chain for VEH exceptions with the OS, and these handlers are invoked by the OS by interrupting the application's current execution, no matter where the exception occurs within a frame.

There are at least two features of VEH that are potentially exploitable by attackers. First, to register a vectored exception handler, the application calls an API `AddVecoredExceptionHandler()` that accepts a callback function pointer parameter that points to the handler code. Securing this pointer requires some form of inter-module callback protection.

Second, the VEH handler-chain data structure is stored in the application's writable heap memory, making the handler chain data directly susceptible to data corruption attacks. Windows protects the handlers somewhat by obfuscating them using the `EncodePointer()` API. However, `EncodePointer()` does not implement a cryptographically secure function (since doing so would impose high overhead);

it typically returns the XOR of the input pointer with a process-specific secret. This secret is not protected against memory disclosure attacks; it is potentially derivable from disclosure of any encoded pointer with value known to the attacker (since XOR is invertible), and it is stored in the *process environment block* (PEB), which is readable by the process and therefore by an attacker armed with an information disclosure exploit. With this secret, the attacker can overwrite the heap with a properly obfuscated malicious pointer, and thereby take control of the application.

From a compatibility perspective, CFI protections that do not include first-class support for these various exception-handling mechanisms often conservatively block unusual control-flows associated with exceptions. This can break important application functionalities, making the protections unusable for large classes of software that use exceptions.

Calling Conventions. CFI guard code typically instruments call and return sites in the target program. In order to preserve the original program's functionality, this guard code must therefore respect the various calling conventions that might be implemented by calls and returns. Unfortunately, many solutions to this problem make simplifying assumptions about the potential diversity of calling conventions in order to achieve acceptable performance. For example, a CFI solution whose guard code uses `EDX` as a scratch register might suddenly fail when applied to code whose calling convention passes arguments in `EDX`. Adapting the solution to save and restore `EDX` to support the new calling convention can lead to tens of additional instructions per call, including additional memory accesses, and therefore much higher overhead.

The C standard calling convention (`cdecl`) is caller-pop, pushes arguments right-to-left onto the stack, and returns primitive values in an architecture-specific register (`EAX` on Intel). Each architecture also specifies a set of caller-save and callee-save registers. Caller-popped calling conventions are important for implementing variadic functions, since callees can remain unaware of argument list lengths.

Callee-popped conventions include `stdcall`, which is the standard convention of the Win32 API, and `fastcall`, which passes the first two arguments via registers rather than the stack to improve execution speed. In OOP languages, every nonstatic member function has a hidden *this pointer* argument that points to the current object. The `thiscall` convention passes the *this pointer* in a register (`ECX` on Intel).

Calling conventions on 64-bit architectures implement several refinements of the 32-bit conventions. Linux and Windows pass up to 14 and 4 parameters, respectively, in registers rather than on the stack. To allow callees to optionally spill these parameters, the caller additionally reserves a *red zone* (Linux) or 32-byte *shadow space* (Windows) for callee temporary storage.

Highly optimized programs also occasionally adopt non-standard, undocumented calling conventions, or even blur function boundaries entirely (e.g., by performing various forms of function in-lining). For example, some C compilers support language extensions (e.g., MSVC's `naked` declaration) that yield binary functions with no prologue or epilogue code, and therefore no standard calling convention. Such code can have subtle dependencies on non-register processor elements, such as requiring that certain Intel status flags be preserved across calls. Many CFI solutions break such code by in-lining call site guards that violate these undocumented conventions.

TLS Callbacks. Multithreaded programs require efficient means to manipulate thread-local data without expensive locking. Using *thread local storage* (TLS), applications export one or more TLS callback functions that are invoked by the OS for thread initialization or termination. These functions form a null-terminated table whose base is stored in the PE header. For compiler-based CFI solutions, the TLS callback functions do not usually need extra protection, since both the PE header and the TLS callback table are in unwritable memory. But source-free solutions must ensure that TLS callbacks constitute policy-permitted control-flows at runtime.

Memory Protection. Modern OSes provide APIs for memory page allocation (e.g., `VirtualAlloc` and `mmap`) and permission changes (e.g., `VirtualProtect` and `mprotect`). However, memory pages changed from writable to executable, or to simultaneously writable and executable, can potentially be abused by attackers to bypass DEP defenses and execute attacker-injected code. Many software applications nevertheless rely upon these APIs for legitimate purposes (see Table 1), so conservatively disallowing access to them introduces many compatibility problems. Relevant CFI mechanisms must therefore carefully enforce memory access policies that permit virtual memory management but block code-injection attacks.

Runtime Code Generation. Most CFI algorithms achieve acceptable overheads by performing code generation strictly statically. The statically generated code includes fixed runtime guards that perform small, optimized computations to validate dynamic control-flows. However, this strategy breaks down when target programs generate new code dynamically and attempt to execute it, since the generated code might not include CFI guards. *Runtime code generation* (RCG) is therefore conservatively disallowed by most CFI solutions, with the expectation that RCG is only common in a few, specialized application domains, which can receive specialized protections.

Unfortunately, our analysis of commodity software products indicates that RCG is becoming more prevalent than is commonly recognized. In general, we encountered RCG compatibility limitations in at least three main forms across a variety of COTS products:

1. Although typically associated with web browsers, *just-in-time* (JIT) compilation has become increasingly relevant as an optimization strategy for many languages,

including Python, Java, the Microsoft .NET family of languages (e.g., C#), and Ruby. Software containing any component or module written in any JIT-compiled language frequently cannot be protected with CFI.

2. Mobile code is increasingly space-optimized for quick transport across networks. *Self-unpacking executables* are therefore a widespread source of RCG. At runtime, self-unpacking executables first decompress archived data sections to code, and then map the code into writable and executable memory. This entails a dynamic creation of fresh code bytes. Large, component-driven programs sometimes store rarely used components as self-unpacking code that decompresses into memory whenever needed, and is deallocated after use. For example, NSIS installers pack separate modules supporting different install configurations, and unpack them at runtime as-needed for reduced size. Antivirus defenses hence struggle to distinguish benign NSIS installers from malicious ones [21].

3. Component-driven software also often performs a variety of obscure *API hooking* initializations during component loading and clean-up, which are implemented using RCG. As an example, Microsoft Office software dynamically redirects all calls to certain system API functions within its address space to dynamically generated wrapper functions. This allows it to modify the behaviors of late-loaded components without having to recompile them all each time the main application is updated.

   To hook a function f within an imported system DLL (e.g., `ntdll.dll`), it first allocates a fresh memory page f' and sets it both writable and executable. It next copies the first five code bytes from f to f', and writes an instruction at $f'+5$ that jumps to $f+5$. Finally, it changes f to be writable and executable, and overwrites the first five code bytes of f with an instruction that jumps to f'. All subsequent calls to f are thereby redirected to f', where new functionality can later be added dynamically before f' jumps to the preserved portion of f.

   Such hooking introduces many dangers that are difficult for CFI protections to secure without breaking the application or its components. Memory pages that are simultaneously writable and executable are susceptible to code-injection attacks, as described previously. The RCG that implements the hooks includes unprotected jumps, which must be secured by CFI guard code. However, the guard code itself must be designed to be rewritable by more hooking, including placing instruction boundaries at addresses expected by the hooking code ($f+5$ in the above example). No known CFI algorithm can presently handle these complexities.

3.3 Compositional Defense Evaluation

Some CFI solutions compose CFI controls with other defense layers, such as randomization-based defenses (e.g., [8, 9, 18, 45, 52, 77]). Randomization defenses can be susceptible to other forms of attack, such as memory disclosure attacks (e.g., [27, 63–65]). ConFIRM does not test such attacks, since their implementations are usually specific to each defense and not easy to generalize.

Evaluation of composed defenses should therefore be conducted by composing other attacks with ConFIRM tests. For example, to test a CFI defense composed with stack canaries, one should first simulate attacks that attempt to steal the canary secret, and then modify any stack-smashing ConFIRM tests to use the stolen secret. Incompatibilities of the evaluated defense generally consist of the union of the incompatibilities of the composed defenses.

4 Implementation

To facilitate easier evaluation of the compatibility considerations outlined in Section 3 along with their impact on security and performance, we developed the ConFIRM suite of CFI tests. ConFIRM consists of 24 programs written in C++ totalling about 2,300 lines of code. Each test isolates one of the compatibility metrics of Section 3 (or in some cases a few closely related metrics) by emulating behaviors of COTS software products. Source-aware solutions can be evaluated by applying CFI code transforms to the source codes, whereas source-free solutions can be applied to native code after compilation with a compatible compiler (e.g., gcc, LLVM, or MSVC). Loop iteration counts are configurable, allowing some tests to be used as microbenchmarks. The tests are described as follows:

fptr. This tests whether function calls through function pointers are suitably guarded or can be hijacked. Overhead is measured by calling a function through a function pointer in an intensive loop.

callback. As discussed in Section 3, call sites of callback functions can be either guarded by a CFI mechanism directly, or located in immutable kernel modules that require some form of indirect control-flow protections. We therefore test whether a CFI mechanism can secure callback function calls in both cases. Overhead is measured by calling a function that takes a callback pointer parameter in an intensive loop.

load_time_dynlnk. Load-time dynamic linking tests determine whether function calls to symbols that are exported by a dynamically linked library are suitably protected. Overhead is measured by calling a function that is exported by a dynamically linked library in an intensive loop.

run_time_dynlnk. This tests whether a CFI mechanism supports runtime dynamic linking, whether it supports retrieving symbols from the dynamically linked library at runtime,

and whether it guards function calls to the retrieved symbol. Overhead is measured by loading a dynamically linked library at runtime, calling a function exported by the library, and unloading the library in an intensive loop.

delay_load *(Windows only)***.** CFI compatibility with delay-loaded DLLs is tested, including whether function calls to symbols that are exported by the delay-loaded DLLs are protected. Overhead is measured by calling a function that is exported by a delay-loaded DLL in an intensive loop.

data_symbl. Data and function symbol imports and exports are tested, to determine whether any controls preserve their accessibility and operation.

vtbl_call. Virtual function calls are exercised, whose call sites can be directly instrumented. Overhead is measured by calling virtual functions in an intensive loop.

code_coop. This tests whether a CFI mechanism is robust against CODE-COOP attacks. For the object-oriented interfaces required to launch a CODE-COOP attack, we choose Microsoft COM API functions in Windows, and gtkmm API calls that are part of the C++ interface for GTK+ in Linux.

tail_call. Tail call optimizations of indirect jumps are tested. Overhead is measured by tail-calling a function in a loop.

switch. Indirect jumps associated with switch-case control-flow structures are tested, including their supporting data structures. Overhead is measured by executing a switch-case statement in an intensive loop.

ret. Validation of return addresses (e.g., dynamically via shadow stack implementation, or statically by labeling call sites and callees with equivalence classes) is tested. Overhead is measured by calling a function that does nothing but return in an intensive loop.

unmatched_pair. Unmatched call/return pairs resulting from exceptions and setjmp/longjmp are tested.

signal. This test uses signal-handling in C to implement error-handling and exceptional control-flows.

cppeh. C++ exception handling structures and control-flows are exercised.

seh *(Windows only)***.** SEH-style exception handling is tested for both hardware and software exceptions. This test also checks whether the CFI mechanism protects the exception handlers stored on the stack.

veh *(Windows only)***.** VEH-style exception handling is tested for both hardware and software exceptions. This test also checks whether the CFI mechanism protects callback function pointers passed to `AddVecoredExceptionHandler()`.

convention. Several different calling conventions are tested, including conventions widely used in C/C++ languages on 32-bit and 64-bit x86 processors.

multithreading. Safety of concurrent thread executions is tested. Specifically, one thread simulates a memory corruption exploit that attempts to smash another thread's stack and break out of the CFI-enforced sandbox.

tls_callback *(Windows source-free only)***.** This tests whether static TLS callback table corruption is detected and blocked by the protection mechanism.

pic. Semantic preservation of position-independent code is tested.

mem. This test performs memory management API calls for legitimate and malicious purposes, and tests whether security controls permit the former but block the latter.

jit. This test generates JIT code by first allocating writable memory pages, writing JIT code into those pages, making the pages executable, and then running the JIT code. To emulate behaviors of real-world JIT compilers, the JIT code performs different types of control-flow transfers, including calling back to the code of JIT compiler and calling functions located in other modules.

api_hook *(Windows only)***.** Dynamic API hooking is performed in the style described in Section 3.

unpacking *(source-free only)***.** Self-unpacking executable code is implemented using RCG.

5 Evaluation

5.1 Evaluation of CFI Solutions

To examine CONFIRM's effect on real CFI defenses, we used it to reevaluate 12 major CFI implementations for Linux and Windows that are either publicly available or were obtainable in a self-contained, operational form from their authors at the time of writing. Our purpose in performing this evaluation is not to judge which compatibility features solutions should be expected to support, but merely to accurately document which features are currently supported and to what degree, and to demonstrate that CONFIRM can be used to conduct such evaluations.

Table 3 reports the evaluation results. Columns 2–6 report results for Windows CFI approaches, and columns 7–14 report those for Linux CFI. All Windows experiments were performed on an Intel Xeon E5645 workstation with 24 GB of RAM running 64-bit Windows 10. Linux experiments were conducted on different versions of Ubuntu VM machines corresponding to the version tested by each CFI framework's original developers. All the VM machines had 16GB of RAM with 6 Intel Xeon CPU cores. The overheads for source-free approaches were evaluated using test binaries compiled with most recent version of gcc available for each test platform. All source-aware approaches were applied before or during compilation with the most recent version of LLVM for each test platform (since LLVM provides greatest compatibility between the tested source-aware solutions).

Table 3: Tested results for CFI solutions on CONFIRM

	LLVM (Windows)					LLVM (Linux)							
Test	**CFI**	**ShadowStack**	**MCFG**	**OFI**	**Reins**	**GCC-VTV**	**CFI**	**ShadowStack**	**MCFI**	**πCFI**	**πCFI (nto)**	**PathArmor**	**Lockdown**
fptr	6.35%	⚠	20.13%	4.35%	4.08%	⚠	6.97%	⚠	✗	−14.00%	−13.79%	⚠	174.92%
callback	⚠	⚠	⚠	128.39%	114.84%	⚠	⚠	⚠	✗	✗	✗	⚠	✗
load_time_dynlnk	2.74%	⚠	8.83%	3.36%	2.66%	⚠	1.33%	⚠	30.83%	31.52%	34.05%	74.54%	1.45%
run_time_dynlnk	⚠	⚠	17.63%	12.57%	11.48%	⚠	4.44%	⚠	✗	✗	✗	1,221.48%	✗
delay_load⊞	N/A	N/A	8.16%	3.61%	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
data_symbl	✓	⚠	✓	✓	✗	✓	✓	⚠	✓	✓	✓	✓	✓
vtbl_call	5.62%	⚠	27.71%	35.94%	31.17%	33.56%	5.94%	⚠	✗	−8.19%	−9.31%	⚠	227.82%
code_coop	⚠	⚠	⚠	✓	✗	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
tail_call	6.17%	⚠	9.51%	0.05%	0.05%	⚠	6.82%	⚠	✗	−17.69%	−17.37%	⚠	178.06%
switch	−5.80%	⚠	3.51%	22.82%	17.69%	⚠	−6.93%	⚠	−29.01%	−27.19%	−28.46%	⚠	85.85%
ret	⚠	18.04%	⚠	49.34%	48.49%	⚠	⚠	20.88%	70.72%	72.40%	71.52%	⚠	106.71%
unmatched_pair	⚠	⚠	⚠	✓	✓	⚠	⚠	⚠	✓	✓	✓	⚠	⚠
signal	✓	⚠	✓	✗	✗	✓	✓	⚠	✓	✓	✓	✗	✓
cppeh	✓	⚠	✓	✓	✗	✓	✓	⚠	✓	✓	✓	✗	✓
seh⊞	✓	⚠	✓	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
veh⊞	⚠	⚠	⚠	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
convention	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
multithreading	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
tls_callback⊞,$	N/A	N/A	N/A	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
pic	✓	✓	✓	⚠	⚠	✓	✓	✓	✓	✓	✓	✓	✓
mem	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✗	✓	✗
jit	⚠	⚠	⚠	✗	✗	⚠	⚠	⚠	✗	✗	✗	⚠	✗
unpacking$	N/A	N/A	N/A	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A	✗	✗
api_hook⊞	⚠	⚠	⚠	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

(nto) stands for *no tail-call optimization*

%: CFI defense passes compatibility and security test, and microbenchmark yields indicated performance overhead

✓: same as %, but this test provides no performance number

⚠: CFI defense passes compatibility but not security check

✗: test does not compile (compilation error), or crashes at runtime

N/A: test is not applicable to the CFI mechanism being tested

⊞: test is Windows-only

$: test is only for source-free defenses

Two forms of compatibility are assessed in the evaluation: A CFI solution is categorized as *permissively compatible* with a test if it produces an output program that does not crash and exhibits the original test program's non-malicious functionality. It is *effectively compatible* if it is permissively compatible and any malicious functionalities are blocked. Effective compatibility therefore indicates secure and transparent support for the code features exhibited by the test.

In Table 3, Columns 2–3 begin with an evaluation of LLVM CFI and LLVM ShadowCallStack on Windows. With both CFI and ShadowCallStack enabled, LLVM on Windows enforces policies that constrain impending control-flow transfers at every call site, except calls to functions that are exported by runtime-loaded DLLs. Additionally, LLVM on Windows does not secure callback pointers passed to external modules not compiled with LLVM, leaving it vulnerable CODE-COOP attacks. Although ShadowCallStack protects against return address overwrites, its shadow stack is incompatible with unmatched call/return pairs.

Column 4 of Table 3 reports evaluation of Microsoft's MCFG, which is integrated into the MSVC compiler. MCFG provides security checks for function pointer calls, vtable calls, tail calls, and switch-case statements. It also passes all tests related to dynamic linking, including load_time_dynlnk, run_time_dynlnk, delay_load, and data_symbl. As a part of MSVC, MCFG provides transparency for generating position-independent code and handling various calling conventions. With respect to exception handling, MCFG is permissively compatible with all relevant features, but does not protect vectored exception handlers. MCFG's most significant shortcoming is its weak protection of return addresses. In addition, it generates call site guard code at compile-time only. Therefore, code that links to immutable modules or modules compiled with a different protection scheme remains potentially insecure. This results in failures against callback corruption and CODE-COOP attacks.

Columns 5–6 of Table 3 report compatibility testing results for Reins and OFI, which are source-free solutions for Windows. Reins validates control-flow transfer targets for function pointer calls, vtable calls, tail calls, switch-case statements, and returns. It supports dynamic linking at load time and runtime, and is one of the only solutions we tested that secures callback functions whose call sites cannot be directly instrumented (with a high overhead of 114.84%). Like MCFG, Reins fails against CODE-COOP attacks. However, OFI extends Reins with additional protections that succeed against CODE-COOP. OFI also exhibits improved compatibility with delay-loaded DLLs, data exports, all three styles of exception handling, all tested calling conventions, and TLS callbacks. Both Reins and OFI nevertheless proved vulnerable against attacks that abuse position-independent code and memory management API functions.

The GNU C-compiler does not yet have built-in CFI support, but includes *virtual table verification* (VTV). VTV is first introduced in gcc 4.9.0. It checks at virtual call sites whether the vtable pointer is valid based on the object type. This blocks many important OOP vtable corruption attacks, although type-aware COOP attacks can still succeed by calling a different virtual function of the same type (e.g., supertype). As shown in column 7 of Table 3, VTV does not protect other types of control-flow transfers, including function pointers, callbacks, dynamic linking for both load-time and run-time, tail calls, switch-case jumps, return addresses, error handling control-flows, or JIT code. However, it is permissively compatible with all the applicable tests, and can compile any feature functionality we considered.

As reported in Columns 8–9, LLVM on Linux shows similar evaluation results as LLVM on Windows. It has better effective compatibility by providing proper security checks for calls to functions that are exported by runtime loaded DLLs. LLVM on Linux overheads range from -6.93% (for switch control structures) to 20.88% (for protecting returns).

MCFI and πCFI are source-aware control-flow techniques. We tested them on x64 Ubuntu 14.04.5 with LLVM 3.5. The results are shown in columns 10–12 of Table 3. ΠCFI comes with an option to turn off tail call optimization, which increases the precision at the price of a small overhead increase. We therefore tested both configurations, observing no compatibility differences between πCFI with and without tail call optimizations. Incompatibilities were observed in both MCFI and πCFI related to callbacks and runtime dynamic linking. MCFI additionally suffered incompatibilities with the function pointer and virtual table call tests. For callbacks, both solutions incorrectly terminate the process reporting a CFI violation. In terms of effective compatibility, MCFI and πCFI both securely support dynamic linking, switch jumps, return addresses, and unmatched call/return pairs, but are susceptible to CODE-COOP attacks. In our performance analysis, we did not measure any considerable overheads for πCFI's tail call option (only 0.3%). This option decreases the performance for dynamic linking but increases the performance of vtable calls, switch-case, and return tests. Overall, πCFI scores more compatible and more secure relative to MCFI, but with slightly higher performance overhead.

PathArmor offers improved power and precision over the other tested solutions in the form of contextual CFI policy support. Contextual CFI protects dangerous system API calls by tracking and consulting the control-flow history that precedes each call. Efficient context-checking is implemented as an OS kernel module that consults the last branch record (LBR) CPU registers (which are only readable at ring 0) to check the last 16 branches before the impending protected branch. As reported in column 13, our evaluation demonstrated high permissive compatibility, only observing crashes on tests for C++ exception handling and signal handlers. However, our tests were able to violate CFI policies using function pointers, callbacks, virtual table pointers, tail-calls, switch-cases, return addresses, and unmatched call/return pairs, resulting

Table 4: Overall compatibility of CFI solutions

Tests	LLVM (Windows)*	MCFG	OFI	Reins	GCC-VTV	LLVM (Linux)*	MCFI	πCFI	πCFI (nto)	PathArmor	Lockdown
applicable	21	22	24	24	18	18	18	18	18	19	19
permissively compatible	21	22	20	12	18	18	11	14	14	16	14
effectively compatible	12	13	17	9	6	12	9	12	12	6	11
Permissive compatibility	100.00%	100.00%	83.33%	50.00%	100.00%	100.00%	61.11%	77.78%	77.78%	84.21%	73.68%
Effective compatibility	57.14%	59.09%	70.83%	37.50%	33.33%	66.67%	50.00%	66.67%	66.67%	31.58%	57.89%

*Compatibility of LLVM is measured with both CFI and ShadowCallStack enabled.

in a lower effective compatibility score. Its careful guarding of system calls also comes with high overhead for those calls (1221.48%). This affects feasibility of dynamic loading, whose associated system calls all receive a high performance penalty per call. Similarly, load-time dynamic linking exhibits a relatively high 74.54% overhead.

Lockdown enforces a dynamic control-flow integrity policy for binaries with the help of symbol tables of shared libraries and executables. Although Lockdown is a binary approach, it requires symbol tables not available for stripped binaries without sources, so we evaluated it using test programs specially compiled with symbol information added. Its loader leverages the additional symbol information to more precisely sandbox interactions between interoperating binary modules. Lockdown is permissively compatible with most tests except callbacks and runtime dynamic linking, for which it crashes. In terms of security, it robustly secures function pointers, virtual calls, switch tables, and return addresses. These security advantages incur somewhat higher performance overheads of 85.85–227.82% (but with only 1.45% load-time dynamic loading overhead). Like most of the other tested solutions, Lockdown remains vulnerable to CODE-COOP and multithreading attacks. Additionally, Lockdown implements a shadow stack to protect return addresses, and thus is incompatible with unmatched call/return pairs.

5.2 Evaluation Trends

ConFIRM evaluation of these CFI solutions reveals some notable gaps in the current state-of-the-art. For example, all tested solutions fail to protect software from our cross-thread stack-smashing attack, in which one thread corrupts another thread's return address. We hypothesize that no CFI solution yet evaluated in the literature can block this attack except by eliminating all return instructions from hardened programs, which probably incurs prohibitive overheads. By repeatedly exploiting a data corruption vulnerability in a loop, our test program can reliably break all tested CFI defenses within seconds using this approach.

Since concurrency, flat memory spaces, returns, and writable stacks are all ubiquitous in almost all mainstream architectures, such attacks should be considered a significant open problem. Intel Control-flow Enforcement Technology (CET) [36] has been proposed as a potential hardware-based solution to this; but since it is not yet available for testing, it is unclear whether its hardware shadow stack will be compatible with software idioms that exhibit unmatched call-return pairs.

Memory management abuse is another major root of CFI incompatibilities and insecurities uncovered by our experiments. Real-world programs need access to the system memory management API in order to function properly, making CFI approaches that prohibit it impractical. However, memory API arguments are high value targets for attackers, since they potentially unlock a plethora of follow-on attack stages, including code injections. CFI solutions that fail to guard these APIs are therefore insecure. Of the tested solutions, only PathArmor manages to strike an acceptable balance between these two extremes, but only at the cost of high overheads.

A third outstanding open challenge concerns RCG in the form of JIT-compiled code, dynamic code unpacking, and runtime API hooking. RockJIT [50] is the only language-based CFI algorithm proposed in the literature that yet supports any form of RCG, and its approach entails compiler-specific modifications to source code, making it difficult to apply on large scales to the many diverse forms of RCG that appear in the wild. New, more general approaches are needed to lend CFI support to the increasing array of software products built atop JIT-compiled languages or linked using RCG-based mechanisms—including many of the top applications targeted by cybercriminals (e.g., Microsoft Office).

Table 4 measures the overall compatibility of all the tested CFI solutions. Permissive and effective compatibility are measured as the ratio of applicable tests to permissively and effectively compatible ones, respectively. All CFI techniques embedded in compilers (*viz.* LLVM on Linux and Windows, MCFG, and GCC-VTV), are 100% permissively compatible, avoiding all crashes. LLVM on Linux, LLVM on Windows, and MCFG secure at least 57% of applicable tests, while GCC-VTV only secures 33%.

OFI scores high overall compatibility, achieving 83% permissive compatibility and 71% effective compatibility on 24 applicable tests. Reins has the lowest permissive compatibility score of only 50%. PathArmor and Lockdown are permissively compatible with 84% and 74% of 19 applicable tests. However PathArmor can only secure 32% of the tests, giving it the lowest effective compatibility score.

Table 5: Correlation between SPEC CPU and ConFIRM performance

SPEC CPU Benchmark	CFI Solution									Benchmark Correlation
	MCFG	Reins	GCC-VTV	LLVM-CFI	MCFI	πCFI	πCFI (nto)	PathArmor	Lockdown	
perlbench				2.4	5.0	5.0	5.3	15.0	150.0	**0.09**
bzip2	−0.3	9.2		−0.7	1.0	1.0	0.8	0.0	8.0	**−0.12**
gcc					4.5	4.5	10.5	9.0	50.0	**0.02**
mcf	0.5	9.1		3.6	4.5	4.5	1.8	1.0	2.0	**−0.39**
gobmk	−0.2			0.2	7.0	7.5	11.8	0.0	43.0	**−0.09**
hmmer	0.7			0.1	0.0	0.0	−0.1	1.0	3.0	**0.33**
sjeng	3.4			1.6	5.0	5.0	11.9	0.0	80.0	**−0.03**
h264ref	5.4			5.3	6.0	6.0	8.3	1.0	43.0	**−0.09**
libquantum				−6.9	0.0	−0.3	−1.0	3.0	5.0	**0.51**
omnetpp	3.8		5.8		5.0	5.0	18.8			**−0.52**
astar	0.1		3.6	0.9	3.5	4.0	2.9		17.0	**0.92**
xalancbmk	5.5		24.0	7.2	7.0	7.0	17.6		118.0	**0.94**
milc	2.0			0.2	2.0	2.0	1.4	4.0	8.0	**0.40**
namd	0.1		−0.1	0.1	−0.5	−0.5	−0.5	3.0		**0.98**
dealII	−0.1		0.7	7.9	4.5	4.5	4.4			**−0.36**
soplex	2.3		0.5	−0.3	−4.0	−4.0	0.9	12.0		**0.89**
povray	10.8		−0.6	8.9	10.0	10.5	17.4		90.0	**0.88**
lbm	4.2			−0.2	1.0	1.0	−0.5	0.0	2.0	**−0.22**
sphinx3	−0.1			−0.8	1.5	1.5	2.4	3.0	8.0	**0.31**
ConFIRM median	9.51	4.59	33.56	5.19	30.83	−11.10	−11.60	648.01	140.82	**0.36**

5.3 Performance Evaluation Correlation

Prior performance evaluations of CFI solutions primarily rely upon SPEC CPU benchmarks as a standard of comparison. This is based on a widely held expectation that CFI overheads on SPEC benchmarks are indicative of their overheads on real-world, security-sensitive software to which they might be applied in practical deployments. However no prior work has attempted to quantify a correlation between SPEC benchmark scores and overheads observed for the addition of CFI controls to large, production software products. If, for example, CFI introduces high overheads for code features not well represented in SPEC benchmarks (e.g., because they are not performance bottlenecks for CFI-free software and were therefore not prioritized by SPEC), but that become real-world bottlenecks once their overheads are inflated by CFI controls, then SPEC benchmarks might not be good predictors of real-world CFI overheads. Recent work has argued that prior CFI research has unjustifiably drawn conclusions about real-world software overheads from microbenchmarking results [70], making this an important open question.

To better understand the relationship between CFI-specific operation overheads and SPEC benchmark scores, we therefore computed the correlation between median performance of CFI solutions on ConFIRM benchmarks with their performances reported on SPEC benchmarks (as reported in the prior literature). Although ConFIRM benchmarks are not real-world software, they can serve as microbenchmarks of features particularly relevant to CFI. High correlations therefore indicate to what degree SPEC benchmarks exercise code features whose performance are affected by CFI controls.

Table 5 reports the results, in which correlations between each SPEC CPU benchmark and ConFIRM median values are computed as Pearson correlation coefficients:

$$\rho_{x,y} = \frac{(\sum_{i=1}^{n} x_i \times y_i) - (n \times \bar{x} \times \bar{y})}{(n-1) \times \sigma_x \times \sigma_y} \quad (1)$$

where x_i and y_i are the CPU SPEC overhead and ConFIRM median overhead scores for solution i, $\bar{x}$ and $\bar{y}$ are the means, and σ_x and σ_y are the sample standard deviations of x and y, respectively. High linear correlations are indicated by $|\rho|$ values near to 1, and direct and inverse relationships are indicated by positive and negative ρ, respectively.

The results show that although a few SPEC benchmarks have strong correlations (namd, xalancbmk, astar, soplex, and povray being the highest), in general SPEC CPU benchmarks exhibit a poor correlation of only 0.36 on average with tests that exercise CFI-relevant code features. Almost half the SPEC benchmarks even have negative correlations. This indicates that SPEC benchmarks consist largely of code features unrelated to CFI overheads. While this does not resolve the question of whether SPEC overheads are predictive of real-world overheads for CFI, it reinforces the need for additional research connecting CFI overheads on SPEC benchmarks to those on large, production software.

6 Related Work

6.1 Prior CFI Evaluations

We surveyed 54 CFI algorithms and implementations published between 2005–2019 to prepare ConFIRM, over half

of which were published within 2015–2019. Of these, 66% evaluate performance overheads by applying SPEC CPU benchmarking programs. Examples of such performance evaluations include those of PittSFIeld [43], NaCl [81], CPI [40], REINS [78], bin-CFI [87], control flow locking [10], MIP [48], CCFIR [84], ROPecker [16], T-VIP [29], GCC-VTV [69], MCFI [49], VTint [83], Lockdown [54], O-CFI [45], CCFI [42], PathArmor [71], BinCC [74], πCFI [51], VTI [12], VTrust [82], VTPin [61], TypeArmor [72], PITTYPAT [24], RAGuard [85], GRIFFIN [30], OFI [75], PT-CFI [33], HCIC [86], μCFI [35], CFIXX [14], and τCFI [47].

The remaining 34% of CFI technologies that are not evaluated on SPEC benchmarks primarily concern specialized application scenarios, including JIT compiler hardening [50], hypervisor security [41,76], iOS mobile code security [22,55], embedded systems security [2–4], and operating system kernel security [20,31,38]. These therefore adopt analogous test suites and tools specific to those domains [17,23,56,57,67].

Several of the more recently published works additionally evaluate their solutions on one or more large, real-world applications, including browsers, web servers, FTP servers, and email servers. For example, VTable protections primarily choose browsers as their enforcement targets, and therefore leverage browser benchmarks to evaluate performance. The main browser benchmarks used for this purpose are Microsoft's Lite-Brite [44] Google's Octane [32], Mozilla's Kraken [46], Apple's Sunspider [6], and RightWare's BrowserMark [59].

Since compatibility problems frequently raise difficult challenges for evaluations of larger software products, these larger-scale evaluations tend to have smaller sample sizes. Overall, 88% of surveyed works report evaluations on 3 or fewer large, independent applications, with TypeArmor [72] having the most comprehensive evaluation we studied, consisting of three FTP servers, two web servers, an SSH server, an email server, two SQL servers, a JavaScript runtime, and a general-purpose distributed memory caching system.

To demonstrate security, prior CFI mechanisms are typically tested against proof-of-concept attacks or CVE exploits. The most widely tested attack class in recent years is COOP. Examples of security evaluations against COOP attacks include those reported for μCFI [35], τCFI [47], CFIXX [14], OFI [75], PITTYPAT [24], VTrust [82], PathArmor [71], and πCFI [51].

The RIPE test suite [80] is also widely used by many researchers to measure CFI security and precision. RIPE consists of 850 buffer overflow attack forms. It aims to provide a standard way to quantify the security coverage of general defense mechanisms. In contrast, CONFIRM focuses on a larger variety of code features that are needed by many applications to implement non-malicious functionalities, but that pose particular problems for CFI defenses. These include a combination of benign behaviors and attacks.

6.2 CFI Surveys

There has been one prior survey of CFI performance, precision, and security, published in 2016 [13]. It surveys 30 previously published CFI frameworks, with qualitative and quantitative comparisons of their technical approaches and overheads as reported in each original publication. Five of the approaches are additionally reevaluated on SPEC CPU benchmarks.

In contrast, CONFIRM establishes a foundation for evaluating *compatibility* and *relevance* of various CFI algorithms to modern software products, and highlights important security and performance impacts that arise from incompatibility limitations facing the state-of-the-art solutions.

7 Conclusion

CONFIRM is the first evaluation methodology and microbenchmarking suite that is designed to measure applicability, compatibility, and performance characteristics relevant to control-flow security hardening evaluation. The CONFIRM suite provides 24 tests of various CFI-relevant code features and coding idioms, which are widely found in deployed COTS software products.

Twelve publicly available CFI mechanisms are reevaluated using CONFIRM. The evaluation results reveal that state-of-the-art CFI solutions are compatible with only about 53% of the CFI-relevant code features and coding idioms needed to protect large, production software systems that are frequently targeted by cybercriminals. Compatibility and security limitations related to multithreading, custom memory management, and various forms of runtime code generation are identified as presenting some of the greatest barriers to adoption.

In addition, using CONFIRM for microbenchmarking reveals performance characteristics not captured by metrics widely used to evaluate CFI overheads. In particular, SPEC CPU benchmarks designed to assess CPU computational overhead exhibit an only 0.36 correlation with benchmarks that exercise code features relevant to CFI. This suggests a need for more CFI-specific benchmarking to identify important sources of performance bottlenecks, and their ramifications for CFI security and practicality.

Acknowledgments

The authors thank Tyler Bletsch, Dimitar Bounov, Mihai Budiu, Yueqiang Cheng, Xuhua Ding, Hong Hu, Jay Ligatti, Ben Niu, Mathias Payer, Michalis Polychronakis, R. Sekar, Zhi Wang, and Qingchuan Zhao for their provision of CFI solution implementations and installation assistance for evaluations. The research reported herein was supported in part by ONR award N00014-17-2995, DARPA award FA8750-19-C-0006, NSF awards #1513704 and #1834215, and an NSF IUCRC award from Lockheed Martin.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. 12th ACM Conf. Computer and Communications Security (CCS)*, pages 340–353, 2005.

[2] A. Abbasi, T. Holz, E. Zambon, and S. Etalle. ECFI: Asynchronous control flow integrity for programmable logic controllers. In *Proc. 33rd Annual Computer Security Applications Conf. (ACSAC)*, pages 437–448, 2017.

[3] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-FLAT: Control-flow attestation for embedded systems software. In *Proc. 23rd ACM Conf. Computer and Communications Security (CCS)*, pages 743–754, 2016.

[4] S. Adepu, F. Brasser, L. Garcia, M. Rodler, L. Davi, A.-R. Sadeghi, and S. Zonouz. Control behavior integrity for distributed cyber-physical systems. *CoRR*, abs/1812.08310, 2018.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. 29th IEEE Sym. Security and Privacy (S&P)*, pages 263–277, 2008.

[6] Apple. Sunspider 1.0 JavaScript benchmark suite. https://webkit.org/perf/sunspider/sunspider.html, 2013.

[7] E. Bauman, Z. Lin, and K. W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proc. 25th Network and Distributed Systems Security Sym. (NDSS)*, 2018.

[8] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proc. 27th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 158–168, 2006.

[9] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Sym.*, 2003.

[10] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proc. 27th Annual Computer Security Applications Conf. (ACSAC)*, pages 353–362, 2011.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attacks. In *Proc. 6th ACM Sym. Information, Computer and Communications Security (AsiaCCS)*, pages 30–40, 2011.

[12] D. Bounov, R. G. Kici, and S. Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Proc. 23rd Network and Distributed System Security Sym. (NDSS)*, 2016.

[13] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1):16:1–16:33, 2017.

[14] N. Burow, D. McKee, S. A. Carr, and M. Payer. CFIXX: Object type integrity for C++. In *Proc. 25th Network and Distributed System Security Symposium (NDSS)*, 2018.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the effectiveness of control-flow integrity. In *Proc. 24th USENIX Conf. Security (USENIX)*, pages 161–176, 2015.

[16] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and H. R. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proc. 21st Network and Distributed System Security Sym. (NDSS)*, 2014.

[17] R. Coker. Disk performance benchmark tool – Bonnie. https://www.coker.com.au/bonnie++, 2016.

[18] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conf.*, pages 63–77, 1998.

[19] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proc. 22nd ACM Conf. Computer and Communications and Security (CCS)*, pages 243–255, 2015.

[20] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proc. 35th IEEE Sym. Security and Privacy (S&P)*, pages 292–307, 2014.

[21] C. Crofford and D. McKee. Ransomeware families use NSIS installers to avoid detection, analysis. *McAfee Labs*, March 2017.

[22] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proc. 19th Network and Distributed System Security Sym. (NDSS)*, 2012.

[23] A. C. de Melo. Performance counters on Linux. In *Linux Plumbers Conf.*, 2009.

[24] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. Efficient protection of path-sensitive control security. In *Proc. 26th USENIX Security Sym.*, pages 131–148, 2017.

[25] S. Donnelly. Soft target: The top 10 vulnerabilities used by cybercriminals. Technical Report CTA-2018-0327, Recorded Future, 2018.

[26] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. 7th USENIX Sym. Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[27] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglu-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. 36th IEEE Sym. Security & Privacy (S&P)*, pages 781–796, 2015.

[28] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 901–913, 2015.

[29] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proc. 30th Annual Computer Security Applications Conf. (ACSAC)*, pages 396–405, 2014.

[30] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding control flows using Intel processor trace. In *Proc. 22nd ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 585–598, 2017.

[31] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proc. 1st IEEE European Sym. Security and Privacy (EuroS&P)*, pages 179–194, 2016.

[32] Google. Octane JavaScript benchmark suite. https://developers.google.com/octane, 2013.

[33] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace. In *Proc. 7th ACM Conf. Data and Application Security and Privacy (CODASPY)*, pages 173–184, 2017.

[34] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 28(1):175–205, 2006.

[35] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee. Enforcing unique code target property for control-flow integrity. In *Proc. 25th ACM Conf. Computer and Communications Security (CCS)*, pages 1470–1486, 2018.

[36] Intel. Control-flow enforcement technology preview, revision 2.0. Technical Report 334525-002, Intel Corporation, June 2017.

[37] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proc. 21st Network and Distributed System Security Sym. (NDSS)*, 2014.

[38] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight kernel protection against return-to-user attacks. In *Proc. 21st USENIX Security Sym.*, pages 459–474, 2012.

[39] F. Konkel. The Pentagon's bug bounty program should be expanded to bases, DOD official says. *Defense One*, 2017.

[40] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[41] D. Kwon, J. Seo, S. Baek, G. Kim, S. Ahn, and Y. Paek. VM-CFI: Control-flow integrity for virtual machine kernel using Intel PT. In *Proc. 18th Int. Conf. Computational Science and Its Applications (ICCSA)*, pages 127–137, 2018.

[42] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 941–951, 2015.

[43] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. 15st USENIX Security Sym.*, 2006.

[44] Microsoft. Lite-Brite Benchmark. https://testdrive-archive.azurewebsites.net/Performance/LiteBrite, 2013.

[45] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *Proc. 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[46] Mozilla. Kraken 1.1 JavaScript benchmark suite. http://krakenbenchmark.mozilla.org, 2013.

[47] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. τCFI: Type-assisted control flow integrity for x86-64 binaries. In *Proc. 21st Int. Sym. Research in Attacks, Intrusions, and Defenses (RAID)*, pages 423–444, 2018.

[48] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proc. 21st ACM Conf. Computer and Communications Security (CCS)*, pages 199–210, 2013.

[49] B. Niu and G. Tan. Modular control-flow integrity. In *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 577–587, 2014.

[50] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proc. 23rd ACM Conf. Computer and Communications Security (CCS)*, pages 1317–1328, 2014.

[51] B. Niu and G. Tan. Per-input control-flow integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 914–926, 2015.

[52] G. Novark and E. D. Berger. DieHarder: Securing the heap. In *Proc. 17th ACM Conf. Computing and Communications Security (CCS)*, 2010.

[53] Office of Inspector General. Evaluation of DHS' information security program for FY 2017. Technical Report OIG-18-56, Department of Homeland Security (DHS), 2018.

[54] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Proc. 12th Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 144–164, 2015.

[55] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *Proc. 29th Annual Computer Security Applications Conf. (ACSAC)*, pages 309–318, 2013.

[56] Postmark. Email delivery for web apps. https://postmarkapp.com, 2013.

[57] R. Pozo and B. Miller. SciMark 2. http://math.nist.gov/scimark2, 2016.

[58] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proc. 22nd Network and Distributed System Security Sym. (NDSS)*, 2015.

[59] RightWare. Basemark web 3.0. https://web.basemark.com, 2019.

[60] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Information and System Security (TISSEC)*, 15(1), 2012.

[61] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. VTPin: Practical vtable hijacking protection for binaries. In *Proc. 32nd Annual Computer Security Applications Conf. (ACSAC)*, pages 448–459, 2016.

[62] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming. In *Proc. 36th IEEE Sym. Security and Privacy (S&P)*, pages 745–762, 2015.

[63] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conf. Computer and Communications Security (CCS)*, pages 298–307, 2004.

[64] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. 34th IEEE Sym. Security & Privacy (S&P)*, pages 574–588, 2013.

[65] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proc. 2nd European Work. System Security (EUROSEC)*, pages 1–8, 2009.

[66] J. Tang. Exploring Control Flow Guard in Windows 10. Technical report, Trend Micro Threat Solution Team, 2015.

[67] The Wine Committee. Wine. http://www.winehq.org.

[68] C. Tice. Improving function pointer security for virtual method dispatches. In *GNU Cauldron Work.*, 2012.

[69] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proc. 23rd USENIX Security Sym.*, pages 941–955, 2014.

[70] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida. SoK: Benchmarking flaws in systems security. In *Proc. 4th IEEE Eurpean Sym. Security and Privacy (EuroS&P)*, 2019.

[71] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 927–940, 2015.

[72] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proc. 37th IEEE Sym. Security and Privacy (S&P)*, pages 934–953, 2016.

[73] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Sym. Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[74] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proc. 31st Annual Computer Security Applications Conf. (ACSAC)*, pages 331–340, 2015.

[75] W. Wang, X. Xu, and K. W. Hamlen. Object flow integrity. In *Proc. 24th ACM Conf. Computer and Communications Security (CCS)*, pages 1909–1924, 2017.

[76] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proc. 31st IEEE Sym. Security and Privacy (S&P)*, pages 380–395, 2010.

[77] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. 19th ACM Conf. Computer and Communications Security (CCS)*, pages 157–168, 2012.

[78] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proc. 28th Annual Computer Security Applications Conf. (ACSAC)*, pages 299–308, 2012.

[79] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu. Shingled graph disassembly: Finding the undecidable path. In *Proc. 18th Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD)*, pages 273–285, 2014.

[80] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Proc. 27th Annual Computer Security Applications Conf. (ACSAC)*, pages 41–50, 2011.

[81] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. 30th IEEE Sym. Security and Privacy (S&P)*, pages 79–93, 2009.

[82] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining trust on virtual calls. In *Proc. 23rd Network and Distributed System Security Sym. (NDSS)*, 2016.

[83] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Protecting virtual function tables' integrity. In *Proc. 22nd Network and Distributed System Security Sym. (NDSS)*, 2015.

[84] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zo. Practical control flow integrity and randomization for binary executables. In *Proc. 34th IEEE Sym. Security and Privacy (S&P)*, pages 559–573, 2013.

[85] J. Zhang, R. Hou, J. Fan, K. Liu, L. Zhang, and S. A. McKee. RAGuard: A hardware based mechanism for backward-edge control-flow integrity. In *Proc. ACM Int. Conf. Computing Frontiers (CF)*, pages 27–34, 2017.

[86] J. Zhang, B. Qi, Z. Qin, and G. Qu. HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things J.*, 6(1):458–471, 2019.

[87] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proc. 22nd USENIX Conf. Security (USENIX)*, pages 337–352, 2013.