Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms

Yuan Li*
BNRist & INSC, Tsinghua University
Beijing, China
li-y18@mails.tsinghua.edu.cn

Mingzhe Wang*
Tsinghua University
Beijing, China
wmz18@mails.tsinghua.edu.cn

Chao Zhang✉
BNRist & INSC, Tsinghua University
Beijing, China
chaoz@tsinghua.edu.cn

Xingman Chen
Tsinghua University
Beijing, China
cxm16@mails.tsinghua.edu.cn

Songtao Yang
Tsinghua University
Beijing, China
yst18@mails.tsinghua.edu.cn

Ying Liu
BNRist & INSC, Tsinghua University
Beijing, China
liuying@cernet.edu.cn

Abstract

Control-flow integrity (CFI) is a promising technique to mitigate control-flow hijacking attacks. In the past decade, dozens of CFI mechanisms have been proposed by researchers. Despite the claims made by themselves, the security promises of these mechanisms have not been carefully evaluated, and thus are questionable.

In this paper, we present a solution to measure the gap between the practical security and the claimed theoretical security. First, we propose CScan to precisely measure *runtime feasible targets* of indirect control transfer (ICT) instructions protected by CFI, by enumerating all potential code addresses and testing whether ICTs are allowed to jump to them. Second, we propose CBench as a sanity check for verifying CFI solutions' *effectiveness against typical attacks*, by exploiting a comprehensive set of vulnerable programs protected by CFI and verifying the recognized feasible targets.

We evaluated 12 most recent open-source CFI mechanisms and discovered 10 flaws in most CFI mechanisms or implementations. For some CFIs, their security policies or protected ICT sets do not match what they claimed. Some CFIs even expand the attack surface (e.g. introducing unintended targets). To facilitate a deeper understanding of CFI, we summarize the flaws into 7 common pitfalls which cover the whole lifetime of CFI mechanisms, and reveal issues that affect CFI mechanisms in practical security.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

control flow integrity; evaluation; practical security

ACM Reference Format:
Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. 2020. Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372297.3417867

*Both authors contributed equally to this research.

1 Introduction

By exploiting memory vulnerabilities such as buffer overflow and use after free (UAF), an attacker can hijack the control flow of a victim program to execute malicious code. Control-flow integrity (CFI) is a type of defense against control flow hijacking. Since the first CFI solution [2] was proposed in 2005, dozens of CFI mechanisms have been developed. In general, CFI solutions instrument inline reference monitors for indirect control transfer (ICT) instructions, whose transfer targets could be compromised, to enforce that ICT instructions only jump to legitimate targets at runtime.

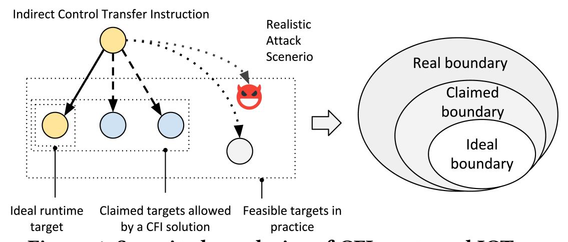
Indirect Control Transfer Instruction
Realistic Attack Scenerio
Real boundary
Claimed boundary
Ideal boundary
Ideal runtime target
Claimed targets allowed by a CFI solution
Feasible targets in practice

Figure 1: Security boundaries of CFI-protected ICTs.

As defense solutions, CFI mechanisms' security guarantees are their most important indicator. The levels of security guarantees provided by CFI can be illustrated by Figure 1. There may be multiple feasible targets for one ICT instruction. But when the program is actually running, there is only one target for each execution of the ICT instruction, which depends on the context information at the time. An ideal CFI mechanism should limit the feasible targets to a unique one for each execution, stopping all control flow hijacking attacks. In other words, the *ideal boundary* should contain only one target. At present, μCFI [27] can reach the ideal boundary in theory, but there are still some problems in its implementation. Usually, CFI mechanisms calculate their *claimed boundary* with some theoretical estimation or coarse measurement. Due to the deficiency of calculation, the feasible targets that CFI mechanisms actually allow (*real boundary*) could deviate from the claimed ones. Such differences reveal flaws in CFI solutions' design and implementation.

A flaw may allow feasible but *unintended targets*, which could be exploited by attackers in *realistic attack scenarios*. On one hand, the size of the real boundary reflects the attack surface; on the other hand, the existence of realistic attack scenarios indicate potential vulnerabilities. Specifically, to evaluate the practical security of CFI mechanisms, there are three questions still unclear:

• *RQ1: Where is the exact real boundary enforced by each CFI mechanism?* In the worst case, attackers could have very powerful yet realistic capabilities, e.g., reading from or writing arbitrary value to any accessible address at any time. When a CFI-protected ICT instruction is executed at runtime, which targets could attackers drive it to jump to? All these targets except only one are illegitimate.

• *RQ2: Does the real boundary match the boundary claimed by the CFI mechanism?* In theory, each CFI mechanism only allows each ICT instruction to jump to a limited number of targets, i.e., the claimed boundary. Targets in the real boundary but outside the claimed boundary are all unintended to the CFI mechanism.

• *RQ3: Are the feasible targets (especially the unintended targets) realistic?* There are many types of vulnerabilities and many ways to exploit vulnerabilities in practice. What attack scenarios will cause CFI mechanisms to fail and hijack ICT instructions to transfer to those feasible targets, even those targets unintended to the CFI? Failing to defeat such attacks shows that unintended targets are realistic.

In this paper, we propose a new solution to systematically evaluate the practical security of CFI mechanisms and answer the aforementioned questions, in order to further shed light on developing CFI solutions with strong security guarantees. The solution consists of two orthogonal components:

- A **tool CScan**, for measuring *runtime feasible targets*, i.e. the set of targets reachable from each CFI-protected ICT instruction at runtime (the real boundary as in Figure 1). CScan traverses all potential code addresses and filters addresses permitted by the deployed CFI mechanism. Each permitted address at an ICT instruction is a potential target that the control flow could be hijacked to at runtime.
- A **sanity check CBench**, for verifying *effectiveness against typical attacks*, i.e. control flow hijacking attacks that could bypass CFI (the realistic attack scenario as in Figure 1). Each failed test case proves that the target CFI mechanism is vulnerable and certain feasible targets (even unintended to CFI) are realistic.

Furthermore, we evaluated 12 most recent open-source CFI mechanisms with CBench and 9 with CScan (3 skipped due to compatibility issues). Results showed that most of them are flawed. We further analyzed the root causes of the flaws, and summarized 7 common pitfalls:

- At the design stage, ❶ *imprecise analysis methods* could be chosen, which limits the upper bound of a CFI mechanism can achieve; ❷ *improper runtime assumptions* could be made, which fails in subtle attack scenarios.
- At the implementation stage, security could be compromised in favor of compatibility, resulting ❸*unprotected corner code*. The compiler faces challenges from ❹*unexpected optimization* (which disables protection), and the runtime itself could have ❺*incorrect implementation* and ❻*mismatched specification*; it could also bring ❼*unintended targets*.

The pitfalls summarized above cover the whole lifetime of CFI mechanisms, and prevail in most popular CFI mechanisms. Discussion and analysis of these pitfalls reveal issues that affect CFI mechanisms in practical security. To help researchers avoid these pitfalls in future works, we also open-source our solution[1].

In summary, we make the following contributions:

(1) We present a lightweight and general tool CScan to precisely measure runtime feasible targets of ICT instructions, providing evidence of security risks faced by CFI mechanisms.
(2) We develop a sanity check for typical attack scenarios, to demonstrate the feasibility of identified security risks.
(3) We conduct a large scale analysis on representative CFI implementations, and provide a thorough study of their weaknesses.
(4) We further demonstrate 7 common pitfalls in CFI implementations which cover the whole lifetime of CFI mechanisms.

2 Background

Control-flow Integrity (CFI) [35] was first proposed in 2005 to mitigate control-flow hijacking attacks. The core idea is to limit the targets of each ICT instruction to a pre-computed legitimate set, ensuring the integrity of the control-flow. Typically, the legitimate set of transfer targets could be statically computed based on the control flow graph (CFG), and the security check can be statically instrumented into the target program and get executed at runtime. A large number of CFI mechanisms have been proposed to provide various supports in the last decade.

Binary Support. Some CFI solutions such as Opaque CFI [37], CCFIR [72], binCFI [75] and CFIMon [66] are able to deploy on binary programs without source code. They have a broader application, but also suffer from inaccuracy of legitimate transfer targets. They usually make special efforts to recover control flow graphs from binaries or simply mark all address-taken functions and call-site preceded instructions as legitimate transfer targets. In contrast, source-level CFI solutions [40] [30] could utilize the source code to build a more accurate control-flow graph and provide a finer granularity of protections. For certain scenarios such as protecting virtual function calls, source-level CFI solutions [28] [71] could utilize type analysis to further reduce transfer targets.

Modular Support. Many CFI mechanisms have poor modular support, and require global information to deploy CFI without compatibility issues. Modular support is also called support for DSO (dynamically shared objects). There are two specific forms of modular support issues: (1) integration of multiple modules which are hardened by CFI separately, and (2) integration of CFI-protected modules with non-protected modules. Binary solutions like CCFIR [72] solve these two issues by weakening the security and allowing more targets than necessary. MCFI [40] provides support for the first issue. It instruments each module individually, and generates new CFGs when modules are linked together, and updates the security checks accordingly. However, recent CFI solutions which provide stronger security guarantees, e.g., μCFI [27] and OS-CFI [31], cannot provide modular support yet, leaving it an urgent problem to solve.

Dynamically Generated Code Support. Inlined CFI instrumentation for dynamically generated code could be overwritten

[1]Our repo presents updated results as CFI mechanisms evolve.
https://github.com/vul337/cfi-eval

by attackers and become useless, since the code pages are writable. NaCl-JIT [5] implements a weak coarse-grained CFI mechanism by verifying the generated code before emitting. RockJIT [41], JITScope [69] and DCG [52] provide CFI support for dynamically generated code by separating the write permission from the dynamic code itself in several ways.

Hardware Utilization. Certain hardware features could help improve the performance or security of CFI solutions. For instance, PT-CFI [25], PITTYPAT [19], CFIMON [66] μCFI [27] and PathArmor [57] take advantage of Intel PT and LBR to obtain runtime information, to compute a smaller set of legitimate targets with an acceptable performance overhead. On the other hand, TSX-based CFI [38] and GRIFFIN [21] utilize hardware features such as Intel TSX [30] and MPX [31] to ensure that the instrumented code or metadata will not be interfered with or tampered with by attackers.

Security Guarantees. Early CFI schemes, e.g., Lockdown [46] and BinCFI [75], *statically* built an imprecise CFG and provided coarse-grained CFI. Source-level CFI solutions could utilize type information to provide finer CFG and stronger guarantees. Some of them have been integrated into standard compilers and kernels, such as Clang-CFI [56], RAP [55], and Control Flow Guard by Microsoft [54]. Recent *dynamic* CFI solutions (e.g., μCFI [27]) further utilize runtime information to improve the precision of CFGs, determining a unique target for an ICT in some cases.

3 Related Work

3.1 Security Evaluation Methods

To precisely evaluate the security of CFI mechanisms, proper evaluation methods should be used. After investigation, we found that existing security evaluation methods are not adequate, and hereby present two new methods. We analyzed 56 CFI solutions to figure out their security evaluation methods. As shown in Table A3, they can be divided into three categories: theoretical verification, experimental verification and quantitative analysis.

3.1.1 Theoretical Verification. Some CFI schemes, e.g., SafeDispatch [28] and PARTS [34], only provide theoretical analysis to illustrate their security and reliability. For example, SafeDispatch outlines its implementation and theoretically analyzes the feasibility and effectiveness against VTable hijacking attacks.

However, differences always lie in between realistic implementations and theoretical models. Therefore, theoretical verification cannot provide a strong support for precise security evaluation.

3.1.2 Experimental Verification. It refers to validating the effectiveness of CFI solutions with existing exploits, including:

- Advanced exploits that could bypass other defenses, including COOP [51], ROP [49] and SROP [7] attacks.
- Exploits against known vulnerabilities, including certain CVE vulnerabilities from the National Vulnerability Database (NVD).
- Exploits against test suites consisting of crafted vulnerable programs, e.g., the buffer overflow testbed RIPE [65].

If a CFI mechanism fails to pass the experimental verification, it is clear that the security guarantee of this CFI is weak. However, passing all the experimental verification cannot reveal how strong a security guarantee the CFI mechanism provides, since the verification benchmark is of limited range and cannot iterate all possibilities. For instance, the RIPE testbed only considers a limited number of buffer overflow vulnerabilities. There could be many other types of vulnerabilities and exploit skills in practice.

3.1.3 Quantitative Analysis. Quantitative analysis refers to evaluating the security strength of CFI mechanisms in a quantitative way, enabling direct comparison between different mechanisms. Existing CFI solutions widely use three common indicators to demonstrate their security strengths, i.e., number and length of code gadgets, Average Indirect target Reduction (AIR), and number of theoretically allowed targets.

Number of code gadgets. Modern control flow hijacking attacks, e.g., ROP and COOP, usually rely on existing code gadgets to bypass deployed defenses. Attacks are more likely to succeed if more code gadgets are available, since more choices are available for payload construction. After deploying a CFI mechanism, the number of available code gadgets would change.

Therefore, some CFI solutions, e.g., CCFIR [72], [22], CCFI [36], BinCFI [75] and Opaque CFI [37], evaluate the number of code gadgets available before and after deploying CFI. A CFI mechanism with a smaller number of gadgets in general (but not always) provides a stronger security guarantee.

However, the number of gadgets measured by different testing tools may vary. So, a unified evaluation tool is required to make fair comparison. On the other hand, this method only demonstrates the security guarantee to some extent, and cannot reflect the exact boundary enforced by each CFI mechanism.

Average indirect target reduction (AIR). After deploying a CFI mechanism, each ICT could only jump to a limited number of targets. This AIR method represents the percentage of ICT targets blocked by each CFI mechanism. It was first proposed in BinCFI [75], and has been widely used by many papers.

But researchers [23] [56] have pointed out that, this method could not reasonably reflect the effectiveness of CFI. A coarse-grained CFI mechanism could also have an AIR value higher than 99%, but provides a poor security guarantee, which means that AIR cannot clearly reflect the difference in safety strength between the coarse-grained CFI scheme and the fine-grained CFI scheme.

Number of targets theoretically allowed by CFI. Instead of AIR, some CFI solutions directly evaluate the number of targets of each ICT instruction allowed by the CFI policy in theory. This number represents the claimed boundary of security guarantees provided by the CFI. The smaller the number is, the stronger the security guarantee is. For static CFI mechanisms, e.g., TypeArmor [58], this number is determined by the statically computed CFG, varying along with the precision of CFG. For dynamical CFI mechanisms, e.g., OS-CFI [31] and CFI-LB [30], this number is reduced by refining the CFG with runtime context.

However, the claimed boundary is not accurate. In practice, CFI-protected ICT instructions still can jump to unintended targets. More details are discussed in section 7.

3.2 CFI Evaluations

Given the large amount of proposed CFI solutions, one would wonder which CFI is better. Three relevant works have been proposed to evaluate them as shown in Table A4.

Burow et.al. [9] conducted a survey on the accuracy, security and performance of CFI mechanisms. Regarding the security evaluation, they profiled the runtime transfer targets of ICT instructions for the SPEC CPU2006 benchmarks, then used the profiled targets as

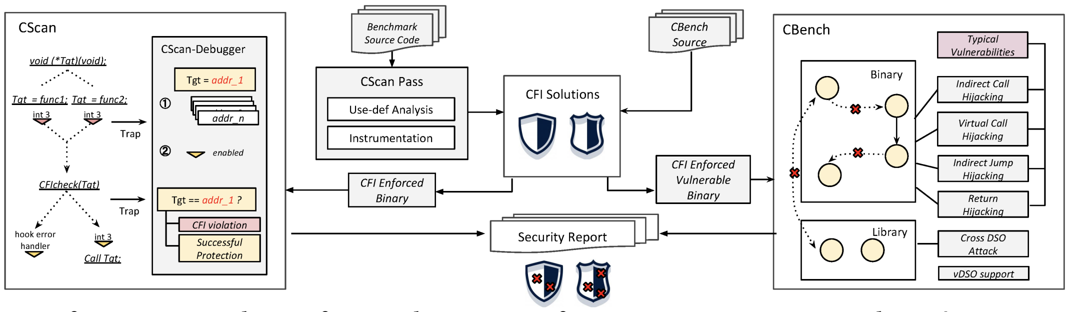

Figure 2: Overview of our security evaluation framework, consisting of two components, i.e., CScan and CBench. CScan instruments target programs to accurately collect the number of feasible targets for each ICT at runtime. CBench consists of a set of vulnerable programs and typical attacks against them, and demonstrates the effectiveness of target CFI mechanisms.

the lower bounds on the legitimate targets and compared them to the legitimate targets claimed by CFI mechanisms. However, the profiling was not complete due to the limitation of dynamic testing, and the comparison to lower bounds did not reflect the actual security guarantee of CFI mechanisms. Our evaluation shows that its claims on the strengths of some different CFI mechanisms are not accurate.

CONFIRM [68] presents a test suite to evaluate the compatibility and applicability of control-flow integrity solutions, without considering the security guarantees. For example, inside the *vtbl-call* and *tail-call* test cases of CONFIRM, no vulnerability-oriented tests exist and only the compatibility and performance are tested. Instead, CBench focuses on security testing.

LLVM-CFI [39] uses a unified static analysis framework to simulate security policies, thus the estimation cannot precisely reflect the metric on real-world programs. Moreover, its static analysis framework could not simulate modern CFI mechanisms that dynamically refine legitimate targets for ICT instructions.

Therefore, *existing security evaluations of CFI mechanisms are not adequate.* More specifically, they cannot answer the aforementioned two research questions, i.e., what the real boundaries enforced by CFI mechanisms are, and whether the risks between the claimed boundaries of CFI mechanisms and the real boundaries are realistic.

In this paper, we try to answer these questions. Table A4 shows the detailed comparison between our work and existing CFI evaluations. We evaluated two new security methods on ICT instructions to provide a more precise result on the security of CFI mechanisms.

4 Methodology

4.1 Our Methods

To precisely evaluate the security boundary and answer the three questions raised in Section 1, we propose CScan and CBench respectively to evaluate the practical security of the CFI mechanisms through two orthogonal aspects.

Runtime feasible targets. For each CFI-protected ICT instruction, it can jump to a specific set of targets at runtime. A powerful yet realistic attacker could hijack this ICT instruction to (and only to) jump to any target in this set. Therefore, the set of runtime feasible targets is the real security boundary of a CFI mechanism, which includes the claimed boundary and some unintended targets. The number of runtime feasible targets of a specific ICT instruction represents the runtime risks of being hijacked at that instruction (i.e., RQ1). Moreover, the number of unintended targets represents the flaws introduced by the target CFI implementation (i.e., RQ2). As illustrated in Figure 2, CScan obtains the accurate runtime feasible targets by debugging the target CFI-protected benchmark applications and testing the feasibility of potential targets at each ICT instruction. ICTs with fewer targets do not guarantee the infeasibility of exploits, but it can reflect the difficulty of exploits to a certain extent. Besides, compared with our method, AIR is too rough to even tell coarse-grained CFI from fine-grained CFI [9].

Effectiveness against typical attacks. The types of vulnerabilities and exploitation methods may vary. A CFI mechanism could occasionally miss certain corner cases and fail to provide adequate defenses. Therefore, its effectiveness against a comprehensive set of typical attacks also shows the strength of its security. Ideally, if a set of typical attacks is complete, i.e., including all potential types of attacks in the real world, the security boundary of a CFI mechanism can be represented as the effectiveness against this set. However, it is infeasible in practice to include all potential attacks. Therefore, the effectiveness evaluation is not complete. But still, given a comprehensive set of typical attacks, we could evaluate a CFI mechanism's effectiveness, and demonstrate corner cases where it fails to protect, and even prove whether the feasible targets exceeding the ideal boundary are realistic (i.e., RQ3). CBench provides a comprehensive set of typical attacks, and evaluates the effectiveness of target CFI mechanisms on these attack scenarios.

4.2 CScan: Feasible Targets Recognition

Assumptions. We assume the CFI-protected applications will be attacked by a powerful attacker, who has found proper vulnerabilities and is able to exploit them to read from or write arbitrary value to arbitrary accessible addresses at any time. This threat model is powerful yet realistic. On the other hand, we assume target CFI mechanisms satisfying two common requirements: 1) they use some specific signals or handler functions to process security violations. 2) they complete the security checks before the ICT instruction jumps. Most CFI solutions meet these requirements.

4.2.1 Overall Workflow

At the high level, CScan acts like a hypothetical attacker, by modifying ICT transfer targets at runtime and validating whether it could pass the deployed CFI security checks. Specifically, it recognizes all runtime feasible targets for ICT instructions in target CFI-protected applications as follows.

First, it locates ICT instructions in target applications, and traces back instructions that are responsible for setting their transfer targets with certain source values (e.g., from memory). A powerful attacker could overwrite these in-memory source values with vulnerabilities and then tamper the transfer targets. Therefore, CScan simulates an attacker by inserting debug breakpoints after these transfer target assignment instructions and modifying the runtime transfer targets externally.

Second, it locates security violation handler functions used by CFI mechanisms, and instruments them to restore the program execution context. Each time the handler is invoked, the hypothetical attacker fails to pass the CFI check. Then, the context restoring operation in the handler enables the attacker to test another transfer target fast and safely.

Third, it instruments the identified ICT instructions with code stubs, which could dump the addresses of the ICT instruction and its runtime transfer target. The recorded information reflects all ICT instructions' runtime feasible targets that could pass the deployed CFI checks.

4.2.2 Design Choices

To provide a precise evaluation of runtime feasible targets, `CScan` must meet the following requirements.

- *Locate ICT instructions precisely and exhaustively.* `CScan` recognizes ICT instructions with a source code analysis pass, and traces back instructions that define the transfer targets with state-of-the-art use-def analysis.
- *Enumerate transfer targets selectively.* The instrumented debugging stubs that act as a hypothetical attacker cannot enumerate all potential transfer targets due to the high overhead. Instead, only executable memory addresses will be enumerated. Moreover, each debugging stub will be invoked at most once.
- *Retrieve CFI check results accurately.* `CScan` hooks the error handler to get notified when the forged transfer targets fail to pass CFI checks, and instruments ICT instructions to dump forged transfer targets successfully pass CFI checks.

4.3 `CBench`: Typical Attacks Evaluation

Although `CScan` provides the real boundary of a CFI mechanism, it cannot determine whether the unintended targets are realistic, and what attack scenarios will cause the CFI to fail. Therefore, we further propose `CBench` to evaluate CFI mechanisms' effectiveness against typical attacks. In practice, there are many types of programs and vulnerabilities, and many ways to exploit them. So, to conduct a thorough evaluation, we constructed a comprehensive set of typical attacks in `CBench`. `CBench` can be used as a sanity check for CFI developers. Beyond that, it also reflects the general security level of CFI to a certain extent.

4.3.1 Typical ICT Hijacking

For x86 applications, there are three types of ICT instructions, i.e., indirect calls, indirect jumps and return instructions. Each could be exploited in different ways.

Indirect Call Hijacking Indirect calls can be hijacked if the function pointers being used are tampered with.

- *Function pointer overwrite.* A typical exploit is overwriting the function pointer to an arbitrary address.
- *Function pointer reuse.* Attackers could also overwrite pointers or offsets used in a memory dereference, loading an existing function pointer from memory and hijacking the control flow when this pointer is used.

Virtual Call (VTable) Hijacking Virtual call is a special type of indirect calls, which is usually used by dynamical dispatchers, e.g., those used by C++ polymorphism. Targets of virtual calls are retrieved from VTables, which are indexed by objects that are writable, and thus could be tampered with.

- *VTable injection.* Attackers could inject a fake VTable consisting of fake virtual function pointers into memory, and overwrite vulnerable objects, then hijack the control flow when virtual functions are being invoked.
- *VTable reuse.* Attackers could also overwrite vulnerable objects to point to existing VTables, and hijack virtual calls to existing virtual functions. COOP [51] is a typical VTable reuse attack.

Indirect Jump Hijacking Indirect jumps could be exploited in a same way as indirect calls. There are two common cases in practice.

- *Tail call.* A tail function call usually will be compiled into an indirect jump. Attackers could exploit them in the same way as indirect calls. But certain defenses may overlook this case.
- *Setjmp/longjmp.* This function pair is used by C programs to handle exceptions. *Setjmp* will save the current context into a buffer `jmp_buf`, and *longjmp* will restore the saved context by an indirect jump. Attackers could overwrite `jmp_buf` and hijack the control flow when `longjmp` is invoked.

Return Hijacking Return addresses are stored on the stack and could be overwritten by attackers to launch hijacking.

- *Return address overwrite.* Attackers could overwrite the return addresses on the stack to point to arbitrary code regions, including ROP gadgets. ROP [49] is a typical method to bypass data execution prevention (DEP).
- *Return address reuse.* Attackers could overwrite the return addresses to existing return addresses, to bypass certain security checks, e.g., PARTS [34].

4.3.2 Typical Vulnerabilities.

Different vulnerabilities enable different types of attacks. Some attacks can break certain CFI schemes. Therefore, `CBench` covers a wide range of vulnerabilities.

- *Buffer overflow and underflow.* Such vulnerabilities cause out-of-bound memory read or write. There are two specific types of buffer overflows, including stack-based and heap-based.
- *Integer overflow.* Due to the limitation of hardware resources, integers in programs are of limited ranges. Arithmetic operations would yield results exceeding the ranges and break programs' semantics.
- *Use after free (UAF).* This is a typical type of temporal vulnerabilities, in which a dangling pointer refers to a memory region allocated to a new object. Dereferencing the dangling pointer will corrupt the new object.
- *Race condition.* This is another type of temporal vulnerability, in which two threads could access a shared resource (e.g., memory) concurrently. Different access orders could yield different program behaviors.
- *Type confusion.* Polymorphic objects could be used as base class instances (i.e., up-cast) or derived class instances (i.e., down-cast) at runtime. Unsafe down casts will wrongly interpret a smaller base class object as a larger derived class object, causing memory corruption when such an object is dereferenced.

4.3.3 Compatibility Issues

When deploying CFI mechanisms to real world applications, there are several compatibility issues, which could also cause security concerns in some cases.

Inline assembly support Some applications may have some inline assembly code. CFI solutions that solely rely on source code analysis may overlook these assembly code, and leave potential attack surface. Attackers could exploit ICT instructions in inline assembly code to bypass CFI.

DSO support Modern applications consist of multiple modules, i.e., DSO (Dynamic Shared Objects). However, many CFI mechanisms only retrieve information from a single module, and cannot provide full support for ICT instructions that transfer to targets in external modules (DSO).

- *Cross-DSO Forward Control-flow Transfer.* Indirect call or jump instructions could jump to targets in external modules. Even worse, the transfer targets (e.g., function pointers) could be retrieved from external modules as well. CFI mechanisms may fail to determine the validity of these targets.
- *Cross-DSO Backward Control-flow Transfer.* Return instructions could also transfer to external modules as well, especially when a CFI-protected function is invoked as a callback in external modules.

vDSO support. For performance reasons, Linux kernel exports a small set of frequently invoked kernel routines to user space processes. As a result, the runtime address space of a target program has some external code and data. CFI mechanisms may overlook this part too and leave potential attack surfaces. CBench also covers this case in the benchmark.

5 Implementation

In this section, we'll present the implementation details of (1) CScan, i.e., the tool used for precisely evaluating the number of runtime feasible targets, and (2) CBench, i.e., the test suite used for verifying the effectiveness of CFI mechanisms against typical attacks.

5.1 CScan

We implemented CScan in Rust with 3500 LoC in total. The main workflow of our CScan is described as follows.

First of all, CScan semantically analyzes the benchmark applications' source code, and finds instructions that load ICT instructions' transfer targets from memory. If the memory is writable, CScan will insert a software interrupt instruction INT3 after the load instruction. Otherwise, the transfer targets are loaded from read-only memory (e.g., VTable), and CScan iteratively finds the source (e.g., objects) of such memory and instruments INT3 if the source is writable. If the transfer targets are loaded from a chain of read-only memory, then no instrumentation will be made.

At runtime, these software interrupts will be triggered, and CScan will disassemble the preceding memory load instruction with a customized disassembler based on Capstone [1]. It then overwrites the value of the register used in the memory load operation with crafted values, to simulate an attacker who can tamper with writable memory. CScan will iterate all executable code addresses (for direct transfer target loads) or all accessible addresses (for nested loads like virtual function pointer loads) and use them as crafted values one by one. Then, CScan monitors the program's further behavior.

If the tampered transfer targets cannot pass the security checks employed by the CFI mechanism, specific error handlers, e.g., UD2, INT3, exit functions, will be executed to prevent control-flow hijacking attacks. CScan listens to such signals or hooks such handlers, and restores the program context to the previously instrumented INT3 instruction, then begins next iteration of transfer target overwriting.

If the tampered transfer targets pass the security checks employed by the CFI mechanism, the ICT instruction will be hijacked to the tampered target. CScan sets a breakpoint before each ICT. At runtime, if such a breakpoint is hit, CScan will record the address of the ICT instruction as well as the tampered transfer target. The set of such addresses represent the exact runtime feasible targets of each ICT instruction.

Performance optimization. An ICT may be executed multiple times at runtime, but it is not necessary to test its feasible targets every time.

For context-insensitive CFI schemes such as Clang-CFI, Clang-CFI-DSO, TypeArmor , MCFI, and TSX-based CFI, transfer targets allowed by each ICT instruction will not change. To accelerate testing, CScan dynamically replaces the INT3 with NOP. As a result, CScan will only detect and test each ICT for once.

For context-sensitive CFI schemes, e.g., OS-CFI and CFI-LB, transfer targets allowed by each ICT instruction may change each time it is executed. CScan collects the context information of each ICT instruction, and dynamically filter out tested contexts. Note that, πCFI is a special context-sensitive CFI solution, which gradually extends the transfer target set of an ICT instruction each time the ICT is executed, and reduces the security guarantee gradually. To reflect the highest security strength, each πCFI-protected ICT is tested only once.

Corner cases For compatibility, we have made special adjustments to CScan for some CFI implementations.

TSX-based CFI allows jumping to any executable address beginning with a special TSX transaction instruction. CScan therefore traverses all executable address spaces to find the same machine code as the instruction, and uses them as crafted transfer targets.

MCFI/πCFI maintains two tables: one Bary Table storing IDs of ICT instructions and one Tary Table storing the IDs of transfer targets, and queries these two tables to find IDs of a specific ICT and the target being used, then checks if they match. Only targets recorded in the Tary Table could pass the CFI check. Therefore, CScan only iterates targets in this table to test target ICT instructions.

Lockdown caches checked target addresses to optimize runtime performance. If a target address is found in the cache, Lockdown directly jumps to that address without CFI checks. To accurately get the number of addresses passed through the check, we modified the default configuration of Lockdown to suppress the cache. Further, Lockdown does not store transfer targets in registers, but passes them to the CFI check functions as parameters via the stack. CScan therefore overwrites these parameters directly.

5.2 CBench

To demonstrate the effectiveness of each CFI scheme against typical attacks, we developed the CBench test suite consisting of 23 vulnerable C/C++ programs. These programs represent different attack scenarios, and are briefly listed in Table A5. We then protect these vulnerable programs with target CFI mechanisms, and evaluate the hardened programs' security with crafted attacks. There are some details worth mentioning when developing the test suite CBench.

Regarding the VTable reuse attacks, we not only test whether an attacker can reuse other types of VTables, but also test whether they can reuse VTables of base classes. Some fine-grained CFI schemes use runtime information to predict targets. In order to detect whether such CFI schemes (such as μCFI) can effectively process runtime information in complicated scenarios, we still use the VTable pointer of the same type of objects as the test target.

For other forward ICTs, we test whether they can jump to the following types of targets: functions with the same function type, functions with different return types, functions with different parameter types, functions with different (more or less) parameters, code gadgets and instructions located after call instructions.

For backward ICTs, we test whether they can return to: address on the same stack space, addresses within the same function as the intended target, return addresses of different functions and different stack spaces, addresses of function entries and code gadgets.

For setjmp/longjmp functions, some versions of their implementations (e.g., in MUSL libc) save a direct copy of contexts, e.g., *RSP, RBP, RIP* in 64 bit programs, while some other implementations (e.g., glibc) will encrypt the saved contexts. The former is more likely to be tampered with. CBench therefore includes a test case for this case.

6 Evaluation

With the proposed security evaluation metrics and corresponding evaluation tools, we further evaluated a set of representative CFI mechanisms, and demonstrated the evaluation results.

• *Answer to RQ1:* We firstly ran CScan to recognize all feasible runtime targets of ICT instructions, which are protected by target CFI mechanisms. From the evaluation results, we could identify the real boundaries of each CFI mechanism, and then compare different CFI mechanisms' security guarantees head-to-head.

• *Answer to RQ2:* We also evaluated the claimed boundary of each CFI mechanism, and presented the gap between claimed boundaries and real boundaries.

• *Answer to RQ3:* We ran CBench to evaluate target CFI mechanisms' effectiveness against typical attacks, and proved whether the unintended targets within the real boundary are realistic.

6.1 Selection of Target CFI Mechanisms

As aforementioned, we have analyzed 56 CFI solutions and compared their evaluation methods. But most of these CFI solutions either have not released the source code to the public, or only work on specific platforms. So, it is infeasible to deploy our evaluation framework to assess those CFI solutions in detail. To the best of our efforts, we collected 12 open source CFI implementations for detailed security analysis. These 12 implementations are representative solutions and cover the characteristics of most CFI mechanisms: support for applications with or without source code, context sensitive and insensitive policy, static and dynamic CFG, utilization of hardware features, and so on.

More specifically, we evaluated (1) two modes of CFI solutions shipped with the latest compiler Clang 9.0, (2) several classic type-based CFI mechanism, including BinCFI [75], MCFI [40], Lockdown [46], and πCFI [42], and (3) almost all open source CFI solutions presented at top-tier conferences in the past years, including OS-CFI [31], PARTS [34], μCFI [27] and CFI-LB [30]. Note that, all CFI solutions presented in the four top-tier security conferences since 2018, except Pittypat [19] that is not open source, are evaluated in our experiment. We believe these CFI mechanisms can well represent the progress of CFI research in past years.

Environment. We use the default configuration provided by the open-source project or the recommended configuration in their *README* files and the experimental environment declared in their paper, and use their test cases (if any) to ensure the prototype system works well. And the compilation environment of Clang 9.0 is Ubuntu 16.04.

Fairness vs Completeness. The compiler compatibility and feature support is different among CFIs, thus subtle changes in configuration lead to vastly different evaluation results. For fairness, in CScan we only evaluate the major contribution of CFI, i.e. protection of forward edges. To fill the missing parts, we further simulate realistic configurations in CBench for completeness. There are two cases. (1) Backward ICT: most CFI solutions [19, 27, 30, 31, 46] proposed in recent years focus on how to protect forward ICT instructions (e.g., indirect call and jump), and leave backward ICT instructions (e.g., return) to shadow stack [67]. Considering that the shadow stack mechanism is commonly used, we only evaluate the number of feasible targets of forward ICT instructions in CScan, but evaluate the classic ROP attack in CBench for completeness. (2) Tail call optimization: As claimed in the official code repository,MCFI/πCFI have to turn off compilers' tail call optimization to get accurate call stack information. Considering that tail calls could also affect other CFI implementations, we disable tail call optimization for all of them in CScan, but add it back in CBench due to its popularity in practical applications.

6.2 Number of Runtime Feasible Targets

6.2.1 Experiment Setup.

Out of 12 CFI mechanisms, CScan is only able to evaluate 9 of them. Three CFI mechanisms, i.e., μCFI, BinCFI and PARTS, adopt special techniques to deploy CFI checks, making it challenging to evaluate. We leave it as a future work to support these special CFI mechanisms. μCFI uses a second process for monitoring, which is asynchronous with the running program to dynamically analyze the running program to speculate the unique target. CScan will modify the run-time state of the running program, which will affect the dynamic analysis of the monitoring process. BinCFI employed an anti-debugging mechanism, prohibiting CScan to modify runtime values. PARTS protects the integrity of code pointers with cryptography signatures, and CScan cannot effectively iterate all potential pointers with valid signatures. For the similar reason, CScan currently cannot test CFI schemes with encrypted pointers (e.g. CCFI [36]).

Benchmark. As CONFIRM [68] shows, existing CFI implementations have enormous unresolved challenges in terms of compatibility. The effectiveness of these CFI implementations on most real-world programs cannot be evaluated due to incompatibility. To test real-world applications, we evaluate Nginx using the hello-nginx test module, which is commonly used in CFI evaluation, as the sample of the real-world programs. There are still compatibility issues. Lockdown cannot make Nginx compiled by Clang work properly (which we need), OS-CFI fails to compile Nginx.

To compare with existing evaluations [9], we also choose SPEC CPU2006 as the benchmarks, and evaluate their forward ICT instructions. However, there are no forward ICTs in two benchmarks,

Table 1: Number of runtime feasible targets for ICT instructions of SPEC CPU2006 benchmarks protected by CFI.

SPEC CPU Benchmark & Nginx	CFI Solution																	
	Clang-CFI-DSO		Clang-CFI		TSX-RTM		TSX-HLE		MCFI		πCFI		Lockdown (x86)		OS-CFI		CFI-LB	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median
perlbench	26.73	8	22.03	4	23445	23445	3673	3673	23.27	5	22.27	4	5533.2	5533	-	-	95741.81	5.5
bzip2	1	1	1	1	6829	6829	1901	1901	1	1	1	1	968	968	1	1	1	1
gcc	*9.40	*3	*8.91	*3	64558	64558	7386	7386	32.63	12	27.46	8	17565.51	17565.51	-	-	26.61	9
gobmk	600.84	8	600.84	8	18969	18969	4480	4480	605.51	32	602.18	17	10851	10851	-	-	596.75k	12
hmmer	10	10	10	10	10924	10924	2339	2339	10	10	1	1	1850	1850	1	1	10	10
sjeng	7	7	7	7	7805	7805	1945	1945	7	7	7	7	1122	1122	7	7	7	7
h264ref	2.06	2	2.06	2	10417	10417	2391	2391	2.06	2	1	1	1785	1785	†1.67	†2	3.56	4
omnetpp	*12.34	*3	*12.34	*3	-	-	-	-	16.33	1	10.90	1	8335	8335	-	-	2076.3k	13.5
astar	1	1	1	1	-	-	-	-	1	1	1	1	4237	4237	1	1	1	1
xalancbmk	*7.19	*3	*7.19	*3	-	-	-	-	44.57	2	17.33	2	46605	46605	-	-	†8775k	†11645k
milc	2	2	2	2	8149	8149	2036	2036	2	2	1	1	1206	1206	1.66	2	2	2
namd	40	40	40	40	-	-	-	-	40	40	16	16	4222	4222	1	1	40	40
dealII	*3.46	*2	3.6	2.5	-	-	-	-	38.15	3	15.13	2	29104	29104	-	-	†3096.6k	†5972.0k
soplex	*3.33	*3	3.45	3	-	-	-	-	2.14	1	1.66	1	6101	6101	1	1	†2979.6k	†5972.0k
povray	*13.81	*3	*13.81	*3	-	-	-	-	14.31	3	13.69	3	8903	8903	-	-	4635.7k	5910.5k
sphinx3	5	5	5	5	9432	9432	2170	2170	5	5	1	1	1505	1505	5	5	5	5
nginx	*28.93	*28.0	28.49	28	13489	13489	3091	3091	483.16	30.0	471.97	18.0	-	-	-	-	† 3098.64k	11

***: programs successfully executed via blacklisting some functions; †: programs aborted before exiting; -: compilation failed or cannot work normally.**

i.e., `429.mcf` and `462.libquantum`. So, we tested 16 SPEC CPU2006 C/C++ benchmarks. As it is infeasible to cover all ICT instructions in a benchmark program with dynamic testing, we use the same configuration and the same test inputs provided by SPEC2006 to test all CFI implementations.

Evaluation. We evaluated the chosen benchmarks with `CScan`, and counted the number of feasible runtime targets for each ICT instruction, i.e., the real boundary. Then, we use the information gathered at the compile time to determine whether each real feasible target of each ICT instruction satisfies the theoretical model of these research to draw the baseline, i.e., the claim boundary. Finally, we compared the claimed boundary against the real boundary, and filtered out unintended targets for ICT instructions.

Metric. To provide an intuitive comparison of actual boundaries, we calculate the mean and median number of feasible targets allowed by CFI for each ICT instruction to demonstrate the security. There are some special cases though. πCFI expands the dynamic CFG gradually and allows more targets to jump along with the program execution, therefore decreases the security guarantees gradually. We only count the number of feasible targets for each ICT when it is executed for the first time, showing the upper bound of πCFI's security guarantee. CFI-LB and OS-CFI provide context-sensitive protections. Each ICT instruction may have different targets in different contexts. Therefore, we first calculate the average number of feasible targets in different contexts for each ICT instruction, and then calculate the average of all ICT instructions.

6.2.2 Results Table 1 shows the mean and median number of feasible targets for ICTs in different CFI mechanisms.

Compatibility Issues. During testing, we found that CFI implementations except MCFI, πCFI and Lockdown, all have compatibility issues:

• *Compilation failure.* Lockdown, TSX-based CFI, and OS-CFI fail to compile some benchmarks or cannot make these work normally.

• *False positives with workarounds.* Some CFI mechanisms may wrongly report normal control transfers as illegitimate ones. Clang-CFI and Clang-CFI-DSO provide a `blacklist` scheme to skip protecting certain functions.

• *False positives without workarounds.* OS-CFI and CFI-LB have similar false positive issues, but provide no workarounds. So, during testing the target program may exit abnormally.

Feasible Targets (answers to RQ1). Table 1 shows the number of feasible runtime targets for ICT instructions protected by different CFI mechanisms. Overall, all CFI mechanisms cannot provide ideal protection for ICT instructions, i.e., each ICT is only allowed to jump to one target at runtime. On average, the number of feasible targets allowed by the context-sensitive solution OS-CFI is the smallest, while the one allowed by its base version CFI-LB is the largest.

• CFI-LB allows the largest number of targets, since it fails to protect a large number of ICT instructions and allows them to jump to arbitrary executable addresses.

• TSX-RTM allows the second largest number of targets, i.e., all function entries and call-preceded instructions.

• TSX-HLE allows a smaller number of targets, since it uses the `xacquire`/`xrelease` instructions to protect the integrity of return addresses used by backward ICT instructions.

• Lockdown further limits the number of feasible targets, especially when the targets are across DSOs.

• Clang-CFI utilizes precise type information to provide a fine-grained CFI, and allows a smaller set of targets. Clang-CFI-DSO enables support for DSOs and has similar results.

• OS-CFI utilizes the origin and call-site sensitivity to divide the targets of each ICT into smaller sets, and allows the smallest number of targets (according to the test result). But it has severe compatibility issues.

MCFI [40] is a CFI mechanism with modular support. It uses the structural equivalence of types to obtain the equivalence class of each ICT. As shown in Table 1, it provides less precise results than Clang-CFI/Clang-CFI-DSO, since the latter applies a more accurate type matching. For example, Clang-CFI enforces virtual function calls only jumping to virtual functions defined in compatible classes, while MCFI allows any functions with matching structural types. *The comparison result between MCFI and Clang-CFI is different from the conclusion of the previous survey work [9]*, which sorts CFI mechanisms by the product of the number of equivalence classes and the inverse of the size of the largest class. We suppose the the difference in results can be explained by method of data collection. Burrow et al[9] use the built-in reporting mechanism of MCFI and πCFI, while for the others they approximate the metrics with lower bounds by extending the instrumentation pass. In addition, the results from

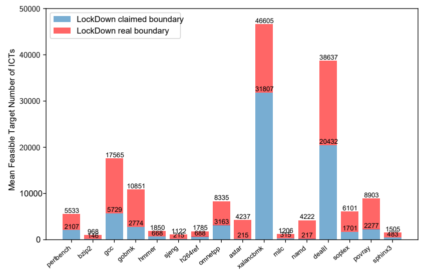

Figure 3: Claimed and real boundary enforced by Lockdown.

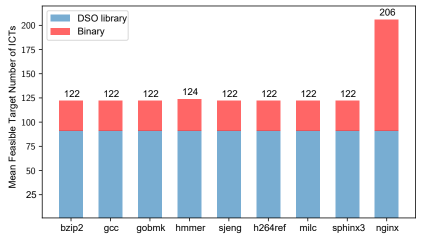

Figure 4: Unintended targets (in DSO libraries or current binary) exceeding the claimed boundary of TSX-based CFI .

the built-in reporting mechanism of MCFI and πCFI include all the used runtime libraries, getting high numbers of equivalence classes.

πCFI is an update to MCFI. It gradually builds the CFG at runtime, and thus allows a smaller equivalence class for each ICT than MCFI. Since the CFG will grow at runtime, but we only test each ICT when it is executed for the first time, so the evaluation result of πCFI shown in Table 1 represents the upper limit of its security strength. However, it is still less precise than Clang-CFI / Clang-CFI-DSO for some benchmarks (such as xalancbmk).

Comparison with Claimed Boundary (answers to RQ2). To answer the question of whether the real boundary of each CFI matches its claim, we evaluated the claimed boundary of each CFI. Results showed that, some CFI mechanisms, e.g., Lockdown, TSX-based CFI and MCFI/πCFI, will have unintended targets for certain ICT instructions. Figure 3 shows the difference between the claimed boundary (blue bar) and the real boundary (blue+red bar) enforced by Lockdown is significant. Figure 4 shows the average number of unintended targets is over 100 for each ICT protected by TSX-based CFI, while 90 of these targets reside in external libraries. The root causes will be analyzed in Section 7.

6.3 Effectiveness Against Typical Attacks

We have tested all the 12 chosen CFI implementations with CBench, to evaluate their effectiveness against typical attacks and **answer the question RQ3**. As shown in Table 2, all 12 CFI mechanisms fail to defeat some attacks.

6.3.1 Protecting Regular Code. First, we present the effectiveness of CFI mechanisms on regular ICT instructions in one program module.

Clang-CFI can defeat most attacks in the CBench test suite. However, it can be bypassed if attackers hijack an ICT instruction to jump to a function entry with the same type.

TSX-based CFI solutions have the worst performance. It cannot protect C++ applications, and allows ICT instructions to transfer to all function entries and *call-preceded instructions*. Even worse, it allows ICT instructions transfer to some special gadgets that can perform arbitrary address jump.

BinCFI has similar bad results. It allows forward ICT to transfer to function entry and *call-preceded instructions*, and backward ICT to return to *call-preceded instructions*. It is also vulnerable to Type Confusion attacks and VTable reuse attacks. Note that, indirect call and jump instructions should only transfer to function entries, but are allowed to jump to *call-preceded instructions* by binCFI. More specifically, if there is a call to the special exit function, the contents after this call instruction could be anything, even a function entry. BinCFI allows each ICT to jump to that specific location.

Lockdown allows forward ICT to transfer to function entries, and thus can be bypassed by type confusion and VTable reuse attacks as well. It uses shadow stack to protect backward ICT instructions, and can defeat ROP attacks.

πCFI provides a stronger defense than MCFI, by generating CFG at runtime and making decisions accordingly. However, they can be bypassed by VTable reuse attacks that exploit virtual functions with structurally equivalent function types.

CFI-LB enforces indirect call instructions to transfer to a subset of function entries with matching types according to the runtime context. But it cannot protect indirect jump or return instructions, and fails to defeat VTable reuse attacks.

μCFI and OS-CFI provide the most robust protection. In some cases, they could even identify the unique target. However, OS-CFI fails to protect tail calls, and μCFI fails to defeat code pointer reuse and VTable reuse attacks. More details will be analyzed in Section 7.

PARTS also enforces ICT instructions to jump to functions of the same type. However, it currently does not support C++ programs, and can be bypassed by Type Confusion attacks.

6.3.2 Protecting Corner Code. As discussed in Section 4, we also evaluated three corner cases, i.e., inline assembly code, Cross-DSO transfer, and vDSO code, in which it could cause compatibility issues and make CFI mechanisms ineffective. The result is also included in Table 2.

Protecting ICT Instructions In Inline Assembly. None of the source-level CFI mechanisms provide support for inline assembly code. Binary-level CFI mechanisms, e.g., binCFI and Lockdown, are able to provide necessary protections for them.

Protecting Control Transfers To DSO Targets. Clang-CFI, CFI-LB, OS-CFI, and μCFI do not support modular compilation, and thus cannot support Cross-DSO transfers. Forward ICT instructions cannot transfer to DSO targets, and backward ICTs cannot return to DSO targets. Lockdown, as a binary-level runtime protection scheme, takes into account the Cross-DSO issue. It allows forward ICT instructions to jump to functions imported from DSOs. However, it cannot protect backward ICT instructions, since the parent function could be used as callbacks by any function in DSO. Clang-CFI-DSO, TSX-based CFI, MCFI, and πCFI support modular compilation, and thus provide a security guarantee for Cross-DSO control transfers similar to intra-module transfers.

Protecting vDSO code. LockDown allows forward ICT instructions transfer to vDSO functions, which weakens the security guarantee. For example, attackers can hijack forward ICT to jump to `__kernel_rt_sigreturn` and launch an SROP attack [7]. Other CFI mechanisms do not support forward ICT transfer to vDSO. However, none of these solutions is able to provide protections for backward

Table 2: Effectiveness of CFI mechanisms against typical attack scenarios. The left-most column represents the specific attack scenario, and the right-most column represents the detail type of attack. The meaning of each symbol is shown in the label below the table.

Classification	TSX-based CFI		binCFI	LockDown	LLVM 9.0.0		MCFI	πCFI	CFI-LB	OS-CFI	μCFI	PARTS	CBench Testsuite	
	RTM	HLE			Clang-CFI	Clang-CFI-DSO								
Indirect Call	⊕ ⊗	⊕	⊕ ⊗ ⊙	⊕ ⊙	Δ	Δ	★	①	Δ	✓	✓	✓	Code Pointer	Overwrite
	⊕ ⊗	⊕	⊕ ⊗ ⊙	⊕ ⊙	Δ	Δ	★	★	Δ	✓	×	Δ		Reuse
Virtual Call	—	—	⊕ ⊗ ⊙	⊕ ⊙	✓	✓	✓	✓	∅	✓	∗	—	VTable Injection	
	—	—	×	×	✓	✓	②	②	÷	÷	✓	—	VTable Reuse	COOP
	—	—	×	×	✓	✓	②	②	÷	÷	③	—		UAF
Indirect Jump	⊕ ⊗	⊕	⊕ ⊗ ⊙	⊕ ⊙	Δ	Δ	★	①	×	×	✓	✓	Tail Call	Overwrite
	⊕ ⊗	⊕	⊕ ⊗ ⊙	⊕ ⊙	Δ	Δ	★	★	×	×	×	Δ		Reuse
	×	×	N/A	N/A	N/A	N/A	✓	✓	N/A	N/A	N/A	N/A	setjmp/longjmp	
Return Address	⊕ ⊗	⊕	⊗ ④	✓	—	—	✓	✓	—	—	✓	✓	Return Address Overwrite	
Type Confusion	×	×	×	×	✓	✓	✓	✓	∅	×	×	×	Function Type Confusion	
	—	—	×	×	✓	✓	✓	✓	×	×	×	—	Object Type Confusion	
Assembly Support	×	×	⊕ ⊗ ⊙	⊕ ⊙	×	×	×	×	×	×	×	×	Indirect Call/Jump	
Cross DSO Support	⋒	⋒	∗	⋄	—	Δ	⋒	⋒	—	—	—	—	Code Pointer Overwrite/Reuse	
	—	—	∗	⋄	—	✓	⋒	⋒	—	—	—	—	Object Injection/Reuse	
	⋒	⋒	∗	Δ	—	Δ	⋒	⋒	—	—	—	—	Callback	Code Pointer Overwrite/Reuse
	—	—	∗	⋒	—	✓	⋒	⋒	—	—	—	—		Object Injection/Reuse
	⋒	⋒	∗	✓	—	—	⋒	⋒	—	—	—	—	Return Address Overwrite	
vDSO Support	—	—	—	×	—	—	—	—	—	—	—	—	Code Pointer Overwrite	

✓: Success Protection, Δ: Targets of the same type, ★: Targets of the same structural type,÷: all virtual function of the same type ⊕: Function entries, ⊗: Callsites, ⊙: Immediate value used in assignment instruction, ⋄: Imported symbols only, ⋒: Same as Non-dso protection ×: Failed to protect, —: Unsupported, N/A: Target unavailable, ∅: Compilation failed. ∗: Crashed at runtime

①: Part of all function entries of the same structural type are feasible targets. ②: Successfully hijacked the control flow to all virtual function entries of the same type, but available syscalls are limited. ③: If the member variable is initialized in the base class, then in the VTable reuse attack that exploits the UAF vulnerability, the hijacked ICT only allows jumping to the base class instead of the same class. ④: Function entry instructions after `call exit()` are feasible targets.

ICT instructions in vDSO, since these instructions are introduced by the system and cannot be instrumented by any CFI mechanism.

7 Pitfalls

We further summarize 7 common pitfalls by studying the root causes of both the design flaws and implementation bugs. Noting that both types of issues cause security threats in practical applications, we use the development lifetime to organize them. We hope the organization can help to bridge the gap between practical security and claimed security.

7.1 Imprecise Analysis

CFI mechanisms rely on information provided by static or dynamic analysis to trim down the set of allowed targets. As the compiler gradually lowers the source code to machine code in multiple passes, information gradually get lost. By design, binary-level CFI mechanisms have less information to utilize; however, if implemented inappropriately, source-level CFI mechanisms also fail to maximize rich semantics.

LockDown: In the theoretical model, LockDown allows forward ICT instructions to jump to imported functions to support Cross-DSO transfers. Furthermore, ICT instructions may call external callback functions. LockDown is designed to resolve callbacks to reduce false positives. Specifically, it first identifies a set of candidate callback functions, and skips security checks if the target function is in this set or performs the CFI check otherwise. However, it wrongly marks many functions as candidate callbacks, and thus introduces unintended targets.

In addition, we found that the real boundary of each ICT instruction of LockDown contains not only all function entries but also all executable addresses that have been taken (i.e., pointer constants), introducing unintended targets as well.

MCFI MCFI performs type checking following the structural equivalence rule, which will introduce unintended targets. For example, MCFI considers the function `gtp_aa_confirm_safety` in `gobmk` and the static `child` function in `musl libc` as the same equivalent class. However, the parameter type of `gtp_aa_confirm_safety` is `char *`, and the parameter type of `child` is `void *`.

πCFI: πCFI is built based on MCFI, and has similar problems. As the instrumentation works on IR instead of source code, the type information of indirect control transfers gets lost. This IR-level design simplifies implementation, but reduces the precision.

μCFI: Since μCFI needs another monitoring process and does not meet the requirements of our testing tools, we did not test it with `CScan`. But we have evaluated it with our test suite `CBench`. As shown in Table 2, μCFI has very high precision in detecting pointer overwrite attacks, and can identify the unique target for many ICT instructions. However, it fails to defeat the code pointer reuse and VTable reuse attacks.

In the code pointer reuse attack scenario, we demonstrated an out-of-bounds attack. Assume there is a structure consisting of two arrays of function pointers (of different types), at runtime a function pointer is retrieved from the first array and invoked indirectly. If the pointer retrieval operation is tampered by the adversary, an out-of-bound memory access is yielded and a wrong pointer will be retrieved. There are two cases. First, the access is still in the structure, and another function pointer in the second array will be invoked, and μCFI fails to detect it. Second, the access is outside the structure, then any function pointer could be invoked. However, instead of reporting succeed and failed, μCFI reports a third state (`empty`) without reporting an attack. The root cause is imprecise analysis data collection: μCFI's analysis relies on runtime information (`constraining data`), but it cannot differentiate legitimate vs malicious ones.

In the VTable reuse attack scenario, if an object of a derived class is freed and later one of its virtual functions is invoked again, the adversary could override the freed object's VTable pointer and execute the virtual function defined in the base class. μCFI also fails to detect this attack due to the inaccuracy in the point-to analysis.

```
push    rax
pop     rax
pop     rcx     #Target address
mov     ecx,ecx
mov     rdi,QWORD PTR gs:0x14240   #BaryID
mov     rsi,QWORD PTR gs:[rcx]     #TaryID
cmp     rsi,rdi     #Compare
jne     0x400559 <check1>
jmp     rcx     #Transfer to target
```

Listing 1: An example gadget in MCFI/πCFI.

7.2 Improper Runtime Assumption

Enforcing security policies is challenging for CFI mechanisms. The runtime provides rich but legal methods for altering the control flow (e.g. longjmp/signal calls), even executing callbacks (e.g. constructor/destructor attributes). These scenarios are trivial, yet improper assumptions in these scenarios enable potential attacks.
MCFI/πCFI: MCFI/πCFI regards the `_init` function in the `.init` section and the `_fini` function in the `.fini` section as legal targets of indirect calls, thus introduceing unintended transfer targets.

These two functions are generated by the compiler, and are invoked by the `_start` function rather than any user-defined function. However, MCFI/πCFI adds them to the legal set and allows other ICT instructions to jump to them.

As shown in Listing 1, these two functions have a specific type of "void(void)", and perform initialization/finalization operations, then return with a specific gadget. Therefore, attackers can hijack ICT instructions protected by MCFI/πCFI to `_init` and `_fini`, and use them to skip the execution of some other code.

TSX-based CFI. TSX-based CFI uses `musl libc` instead of `glibc` as the C library, introducing security risks as well. Glibc can protect the contents of sensitive registers (such as *SP* and *PC*) in the buffer *jmp_buf* by mangling them. Musl, on the other hand, does not provide protections for *jmp_buf*. And thus the buffer *jmp_buf* can be easily tampered for a *JOP* attack. If the program uses the `setjmp/longjmp` function in musl libc and there is a buffer overflow vulnerability in the program, the attacker can utilize `setjmp/longjmp` to launch a JOP attack. Even worse, TSX-based CFI does not perform checks on the indirect jump instruction of the `longjmp` function, so an attacker can bypass it easily.

7.3 Unprotected Corner Code

Compatibility issues must be considered when implementing CFI mechanisms, and they also lead to security issues. As aforementioned, some ICT instructions may reside in corner code, e.g., inline assembly code or DSO libraries, or vDSO code. This threat is actually very severe. As shown in Table 2, ICT instructions in these cases are in general not protected by most CFI solutions.

7.4 Unexpected Optimization

Modern compilers optimize code aggressively. Security checks in CFI mechanisms may leverage low-level constructs to detect anomaly; if considered as an undefined behavior, the compiler is free to optimize even remove the operation. The same applies to operations free of side-effects: some CFI mechanisms require collecting data before enforcing the policy. If the data collection operation is considered free of side effects, the compiler may also remove it.
OS-CFI: OS-CFI is an origin-sensitive CFI solution which uses SUPA [53] for static analysis of CFG. We have turned off the tail optimization option when evaluating with our tool `CScan`, so we specifically tested the tail optimization in our test suite `CBench`.

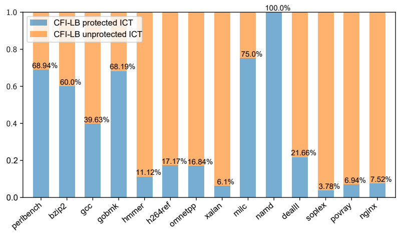

Figure 5: Protected vs. unprotected ICTs in SPEC benchmarks hardened by CFI-LB.

The results show that neither CFI-LB nor OS-CFI protects tail calls that are optimized for indirect calls. The root cause is that, if compiler optimization is turned on, *the OS-CFI checker function will be removed due to optimization.* Besides, SUPA fails to recognize any tail call during the static CFG analysis, causing security risks as well.
CFI-LB: As for CFI-LB, the checker function is also deleted due to optimization. In addition, when CFI-LB uses Intel Pin to generate dynamic CFG, it searches for a specific function entry before analysis. This function is an empty function and is removed if the optimization is turned on, thus cannot be found during the searching phase. So CFI-LB fails to generate the dynamic CFG. Therefore, neither the static CFG builder can analyze those ICTs at compile time nor the dynamic CFG builder can add them to the final CFG while running the program. As a result, the runtime CFI checks will be misled.

7.5 Incorrect Implementation

Implementing CFI mechanisms involves complex analysis techniques and linker or runtime adjustments. The low-level enforcement of a policy and high-level decision-making process of a policy are error-prone.

CFI-LB. As shown in Table 1, CFI-LB has the largest variance between the median and the mean value of feasible targets. We analyzed the code and found that the root cause is that, some ICT instructions were unprotected and could be hijacked to jump to all executable addresses in the target benchmark protected by CFI-LB.

Moreover, one of CFI-LB's checker functions has a logic bug, which silently ignores the case that the runtime target does not match the CFG, causing security risks too.

We counted the number of incorrect checker functions in the program to obtain the proportion of unprotected ICTs in all ICTs, and demonstrated the results in Figure 5.

7.6 Mismatched Specification

Hardware may provide extra security-related features such as NX bit, Intel Control-flow Enforcement Technology (CET), ARM Pointer Authentication (PA). The effective use of these features relies on both the user and the platform. Errors in either party lead to mismatched specification, and security issues result.

For example, ARM PA is designed to enforce the integrity of pointers, where the hardware can detect external entities with cryptographic signatures of pointers. The specification requires three values to generate the signature: a secret key, a modifier, and the pointer itself. However, the implementation of PA in QEMU mismatches with the specification, because the pointer itself does not affect the generated signature. Therefore, a local attacker could abuse this flaw to bypass ARM PA protection for all programs running on QEMU. We analyzed this bug and reported the details to

```
xend
add     BYTE PTR [rdi],cl
test    DWORD PTR [rcx],ebp
add     al,BYTE PTR [rax]
add     BYTE PTR [rcx+rcx*4-0x28],cl
mov     r10,rax
add     r10,0x3
jmp     r10
```

Listing 2: An example gadget in TSX-based CFI

QEMU developers, and a CVE is assigned. The mismatch of specification in QEMU implies weakened protections in CFI mechanisms relying on this feature. For example, the protection of PARTS running in QEMU can be easily defeated by a buffer-overflow; on the other hand, the very same CFI mechanism running in ARM Fixed Virtual Platforms (FVP) does provide protections as expected.

7.7 Unintended Targets Introduced by CFI

Most CFI schemes enforce their policies by checking logic around ICTs or inside their runtime libraries. The code instrumented by CFI can be potential (thus unintended) transfer targets to the adversary.

TSX-based CFI. In the RTM mode of TSX-based CFI, an ICT is allowed to transfer to any code snippets starting with an `xend` instruction, ending the transaction normally.

Any byte sequence starting with the machine code of the `xend` instruction is a feasible target for each ICT instruction. The common `libc` library has over 100 matching bytes, and thus over 100 unintended targets. Ironically, most of the unintended targets reside in the checker function instrumented by the TSX-based CFI itself.

TSX-based CFI provides many checker functions: (1) a unique checker function is provided for the return instruction, (2) a unique checker function is provided for each pair of (`register, ICT-type`), where `ICT-type` is either call or jump and `register` is the operand used in the ICT, and (3) more checker functions are provided for ICT instructions with memory operand rather than register operand.

Taking the checker function for `jmp rdx` as an example, if the attacker hijacks one ICT to transfer to the location of `xend` in this checker function, she/he can obtain the gadget shown in Listing 2. If she/he is also able to control the value of `rax, rdi, rcx`, then she/he could jump to arbitrary code with the last unintended and unprotected instruction `jmp r10`.

7.8 Summary

The pitfalls listed above imply that flaws exist in the whole lifetime of CFI mechanisms.

At the design stage, imprecise analysis methods could be chosen, which limits the upper bound of a CFI mechanism can achieve; improper runtime assumption could be made, which fails in subtle attack scenarios. For example, Lockdown's design cannot handle callbacks; uCFI's attack model lacks considering malicious data related to control-flow ; TSXCFI's inspection mechanism introduces unavoidable extra gadgets.

At the implementation stage, security could be compromised in favor of compatibility, and corner code remains unprotected. The compiler faces challenges from unexpected optimization (which disables protection). The runtime itself could be buggy in both implementation and specification, and could also bring unintended targets. These implementation bugs are introduced unintentionally, yet they still affect practical security and can be valuable lessons for developers. For example, MCFI/πCFI introduce unintended targets because of missing type information of ICTs; The OS-CFI and CFI-LB check functions will be removed after optimization; The logic bug of CFI-LB's check function causes the check to fail.

To avoid these factors that lead to security risks, we propose to perform more comprehensive and fine-grained evaluations on practical security. Our solution covers both the design and implementation stage, helps to identify and resolve issues of designing in advance, and makes up for the gap between design and implementation.

8 Discussion

Deployment of CFI Implementations. Some CFI implementations have higher requirements on the deployment environment and need to be the same as their experimental environment in order to operate normally. Some of the CFI implementations are not user-friendly and require a good understanding of the entire running process and sample scripts before they can be used. This consumes most of the time in our evaluation process. We would thank the majority of CFI program authors for their support, and hope that the usability of CFI mechanisms will be increased.

Security Issue and Compatibility. Many CFI solutions have bad compatibility for regular programs. Some CFI implementations even break the original semantic of programs, for example, MCFI/πCFI ignore init functions in protected binaries. Moreover, protected programs are often terminated with CFI violation during our evaluation due to the existence of false positive cases. CFI solutions including clang-cfi and clang-cfi-dso provide an interface for programmers to label specific functions which failed to be supported with a blacklist. And the ICTs in blacklisted functions will not be protected. Scalable leads to good compatibility though, but brings heavy tasks, inevitable cases and security issues. How to resolve this conflict is a question of practical significance.

Human Efforts in Testing CFI Implementations. CBench does not require any human effort to be adapted to new CFI solutions. CScan requires very few human efforts, thanks to the fact that the structure of most CFI solutions are similar. For corner cases, see Section 5.1 for details in customization.

9 Conclusion

This paper proposes a solution to evaluate practical security of CFI mechanisms. The solution consists of `CScan`, for measuring the real security boundary of CFI mechanisms, and `CBench`, for verifying effectiveness against typical attacks. With the proposed solution, we evaluated popular open-source CFI mechanisms. Among all the 12 evaluated mechanisms, 10 mechanisms were found with flaw(s). They either fail to protect all ICT instructions as claimed, or become parts of the attack surface, or become unintended targets of ICT instructions, or fail to correctly enforce the claimed CFI policy.

We further summarize the flaws into 7 common pitfalls in developing CFI mechanisms. The pitfalls prevail in most CFI mechanisms we studied. As a solution to these issues, we open source `CScan` and `CBench`, in order to help evaluating, understanding and promoting CFI mechanisms.

Acknowledgement

This work was supported in part by National Natural Science Foundation of China under Grant 61772307, 61772308, 61972224 and U1736209, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] [n.d.]. Capstone, the ultimzte disassembly framework. http://www.capstone-engine.org/.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 4.

[3] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. 2017. Ecfi: Asynchronous control flow integrity for programmable logic controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 437–448.

[4] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 743–754.

[5] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L Schuff, David Sehr, Cliff L Biffle, and Bennet Yee. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 355–366.

[6] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 353–362.

[7] Erik Bosman and Herbert Bos. 2014. Framing signals-a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 243–258.

[8] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving.. In *NDSS*.

[9] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.

[10] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2018. Cfixx: Object type integrity for c++ virtual dispatch. In *Prof. of ISOC Network & Distributed System Security Symposium (NDSS). https://hexhive. epfl. ch/publications/-files/18NDSS. pdf*.

[11] Ping Chen, Yi Fang, Bing Mao, and Li Xie. 2011. JITDefender: A defense against JIT spraying attacks. In *IFIP International Information Security Conference*. Springer, 142–153.

[12] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. (2014).

[13] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 38–49.

[14] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 292–307.

[15] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones.. In *NDSS*, Vol. 26. 27–40.

[16] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 74.

[17] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[18] P de Clercq. 2017. Hardware supported Software and Control Flow Integrity. (2017).

[19] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 131–148.

[20] Robert Gawlik and Thorsten Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 396–405.

[21] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 585–598.

[22] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 179–194.

[23] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 575–589.

[24] Jens Grossklags and Claudia Eckert. 2018. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, Vol. 11050. Springer, 423.

[25] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 173–184.

[26] Owen S Hofmann, Alan M Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring operating system kernel integrity with OSck. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 279–290.

[27] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1470–1486.

[28] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks.. In *NDSS*.

[29] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. 2012. kGuard: lightweight kernel protection against return-to-user attacks. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 459–474.

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

[31] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive control flow integrity. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 195–211.

[32] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-pointer integrity. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 147–163.

[33] Donghyun Kwon, Jiwon Seo, Sehyun Baek, Giyeol Kim, Sunwoo Ahn, and Yunheung Paek. 2018. VM-CFI: Control-Flow Integrity for Virtual Machine Kernel Using Intel PT. In *International Conference on Computational Science and Its Applications*. Springer, 127–137.

[34] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. {PAC} it up: Towards Pointer Integrity using {ARM} Pointer Authentication. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 177–194.

[35] MihaiBudiu MartnAbadi and Jay Ligatti ÚlfarErlingsson. 2005. Control flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, Virginia*. 340–353.

[36] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 941–951.

[37] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity.. In *NDSS*, Vol. 26. 27–30.

[38] Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, and Davide Balzarotti. 2016. Taming transactions: Towards hardware-assisted control flow integrity using transactional memory. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 24–48.

[39] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2019. Analyzing Control Flow Integrity with LLVM-CFI. *arXiv preprint arXiv:1910.01485* (2019).

[40] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 577–587.

[41] Ben Niu and Gang Tan. 2014. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1317–1328.

[42] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 914–926.

[43] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. 2017. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 259–284.

[44] Vasilis Pappas. 2012. kBouncer: Efficient and transparent ROP mitigation. *Apr* 1 (2012), 1–2.

[45] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent {ROP} Exploit Mitigation Using Indirect Branch Tracing. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 447–462.

[46] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 144–164.

[47] Jannik Pewny and Thorsten Holz. 2013. Control-flow restrictor: Compiler-based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 309–318.

[48] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries.. In *NDSS*.

[49] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012), 2.

[50] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: practical VTable hijacking protection for binaries. In

Proceedings of the 32nd Annual Conference on Computer Security Applications. ACM, 448–459.

[51] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 745–762.

[52] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation.. In *NDSS*.

[53] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. ACM, 460–473.

[54] Jack Tang and Trend Micro Threat Solution Team. 2015. Exploring control flow guard in windows 10. *Available at ht tp://blog. trendmicro. c om/trendlabssecurity-intelligence/exploring-control-flow-guard-in-windows* 10 (2015).

[55] PaX Team. 2015. Rap: Rip rop. In *Hackers 2 Hackers Conference (H2HC)*.

[56] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity. In *in GCC & LLVM. In 23rd USENIX Security Symposium (USENIX Security 14)(Aug. 2014), USENIX Association*. Citeseer.

[57] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 927–940.

[58] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.

[59] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. 2019. Control-Flow Integrity for Real-Time Embedded Systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[60] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. 2015. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 331–340.

[61] Wenhao Wang, Xiaoyang Xu, and Kevin W Hamlen. 2017. Object flow integrity. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1909–1924.

[62] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 380–395.

[63] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. 2018. Sponge-based control-flow protection for iot devices. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 214–226.

[64] Mario Werner, Erich Wenger, and Stefan Mangard. 2015. Protecting the control flow of embedded processors against fault attacks. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 161–176.

[65] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime Intrusion Prevention Evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM.

[66] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 1–12.

[67] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K Iyer. 2002. Architecture support for defending against buffer overflow attacks. *Coordinated Science Laboratory Report no. UILU-ENG-02-2205, CRHC-02-05* (2002).

[68] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. 2019. {CONFIRM}: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1805–1821.

[69] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. 2015. JITScope: Protecting web users from control-flow hijacking attacks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 567–575.

[70] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity.. In *NDSS*.

[71] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls.. In *NDSS*.

[72] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[73] Jun Zhang, Rui Hou, Junfeng Fan, Ke Liu, Lixin Zhang, and Sally A McKee. 2017. RAGuard: A hardware based mechanism for backward-edge control-flow integrity. In *Proceedings of the Computing Frontiers Conference*. ACM, 27–34.

[74] Jiliang Zhang, Binhang Qi, Zheng Qin, and Gang Qu. 2018. HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things Journal* 6, 1 (2018), 458–471.

[75] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for {COTS} Binaries. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 337–352.

A Appendix

Table A3: Existing security evaluation methods used in previous CFI studies.

Category	method	Paper
Theoretical Verification	Theoretical Methods	SafeDispatch [28], VM-CFI [33], HyperSafe [62], RAGuard [73], TSX-based CFI [38] CFI CaRE [43], PARTS [34], SOFIA [18], [64], [6], [8], Control-flow restrictor [47], C-FLAT [4]
Experimental Verification	Advanced Exploits, Known CVE Vulnerabilities	πCFI [42], CFIMon [66], CCFIR [72], CFIXX [10], [20], μCFI [27], τCFI [24], Lockdown [46], OSck [26], PT-CFI [25], kGUARD [29], vfGUARD [48], VTPIN [50], Binary code continent [60], OFI [61], JITDefender [11], PITTYPAT [19], kBouncer [44], ROPecker [12], TypeArmor [58], VTrust [71], VTint [70], [45], CFIBR [17], HCFI [13], HAFIX [16], OS-CFI [31], CFI-LB [30], CFI CaRE [43], [20], [59] ECFI [3], MoCFI [15], SCFP [63]
	RIPE test suite	GRIFFIN [21], CPI [32], BinCFI [75], CFI-LB [30]
Quantitative Analysis	Number of code gadgets	CCFIR [72], [22], CCFI [36], BinCFI [75], Opaque CFI [37], MCFI [40], RockJIT [41], PathArmor [57], KCoFI [14]
	Average Indirect-target Reduction (AIR)	MCFI [40], LockDown [46], BinCFI [75], [22], τCFI [24], vfGUARD [48], GCC-VTV & Clang-CFI [56], Binary code continent [60], KCoFI [14], HCIC [74]
	Number of theoretically allowed targets	PathArmor [57], πCFI [42], TypeArmor [58], PITTYPAT [19], OS-CFI[31], CFI-LB[30], μCFI [27]

Table A4: Comparison between CFI evaluations.

Evaluation	Motivation	Type of Methods	Evaluation Targets	Evaluation methods	Scalability
CFI survey [9]	Accuracy, Security, and Performance	Measurement	Accuracy of CFG (Security), Runtime Overhead (Performance)	EC (Equivalence Classes) / LC (Largest Class)	Scalable
ConFIRM [68]	Compatibility	Measurement	Coding Features and Idioms	Compatible or Not	Scalable
LLVM-CFI [39]	Security	Estimation	Accuracy of CFG	Theoretical Result	Static CFI only
Our Work	Security	Measurement	Targets of ICT	Count of Feasible Targets and Effectiveness Against Typical Attacks	Scalable

Table A5: CBench test suite details.

Classification			Exploitation Primitives	Targets of Control-flow Hijacking	Vulnerability Category
Indirect Call	Code Pointer Overwrite		Overwrite specific code ptr	Function entry of the same/different type , Non-function entry	Stack-based Overflow
	Code Pointer Reuse		Reuse specific code ptr	Function entry of the same/different type	Out-of-bound Access
Virtual Call	VTable Injection		Overwrite specific vptr	(virtual) Function entry of the same/different type, Non-function entry	Heap-based Overflow
	VTable Reuse		Reuse specific vptr	Same virtual function of base class/subclass, the original virtual function, Virtual function of the same/different type	Use-after-free
			COOP	Same virtual function of base class/subclass, the original virtual function, Virtual function of the same/different type	Heap-based Overflow
Indirect Jump	Tail Call Overwrite		Overwrite specific code ptr	Function entry of the same/different type, Non-function entry	Integer overflow caused Stack-based Overflow
	Tail Call Reuse		Reuse specific code ptr	Function entry of the same/different type	Race condition caused Out-of-bound Access
	setjmp/longjmp		Overwrite specific code ptr	Other call site, Function entry, Non-function entry	Heap-based Overflow
Return Address	Return Address Overwrite		Overwrite specific code ptr	Different call site for the same function with the same/different sp, Different call site for the different function with the same sp, Other call site, Function entry, code gadgets	Stack-based Overflow
Type Confusion	Function Type Confusion			Different type	Type Confusion
Type Confusion	Object Type Confusion			Different type	Type Confusion
Assembly Support	Indirect Call		Overwrite specific code ptr	Function entry of the same/different type, Non-function entry	Stack-based Overflow
	Indirect Jump		Overwrite specific code ptr	Function entry (of same/different type), Non-function entry	Stack-based Overflow
Cross DSO Support	Code Pointer Overwrite		Overwrite specific code ptr	Function entry of the same/different type, Non-function entry	Arbitrary-address-write
	Code Pointer Reuse		Reuse specific code ptr	Function entry of the same/different type	Out-of-bound Access
	Object Injection		Overwrite specific vptr	Function entry of the same/different type, Non-function entry	Heap-based Overflow
	Object Reuse		Reuse specific vptr	Same virtual function of base class/subclass, the original virtual functions, Virtual function of the same/different type	Use-after-free
	Callback	Code Pointer Overwrite	Overwrite specific code ptr	Function entry of the same/different type, Non-function entry	Arbitrary-address-write
		Code Pointer Reuse	Reuse specific code ptr	Function entry of the same/different type	Out-of-bound Access
		Object Injection	Overwrite specific vptr	Function entry of the same/different type, Non-function entry	Heap-based Overflow
		Object Reuse	COOP	Same virtual function of base class/subclass, the original virtual functions, Virtual function of the same/different type	Heap-based Overflow
	Return Address Overwrite		Overwrite specific code ptr	Different call site for the same function with the same/different sp, Different call site for the different function with the same sp, Other call site, Function entry, code gadgets	Stack-based Overflow
vDSO Support	Code Pointer Overwrite		Overwrite specific code ptr	other vDSO functions	Global-buffer-based Overflow