Computing Science, 2026

Control Flow Integrity in Practice: Retrospectives, Realities, and Automated Enforcement

Sabine Houy

UMEÅ UNIVERSITY

ISBN: 978-91-8070-888-3 (print)
ISBN: 978-91-8070-889-0 (pdf)
Electronic version available at: http://umu.diva-portal.org/
Printed by: Scandinavian Print Group, Skarpnäck, 2026

Abstract

Control Flow Integrity (CFI) is a well-established mitigation against control-flow hijacking attacks arising from memory corruption vulnerabilities. Over the past two decades, numerous CFI mechanisms have been proposed and integrated into modern compilers and software ecosystems. Despite this progress, CFI remains difficult to adopt in practice, and deployment decisions, compatibility constraints, and engineering overhead strongly influence its real-world security impact.

This dissertation investigates Control Flow Integrity from the perspective of practical adoption and deployability. Rather than treating CFI as a purely theoretical protection, it examines how CFI is selected, integrated, and maintained in real-world software systems, and why these steps often fall short of idealized designs. The dissertation is structured around four complementary studies that together trace the path from measurement to guidance, to deployment experience, and finally to automated enforcement.

The first study presents a large-scale empirical analysis of deployed binaries to assess the current state of LLVM-CFI adoption across major software platforms. It shows that while CFI deployment is increasing in some ecosystems, it remains uneven and limited, leaving substantial portions of the attack surface unprotected. The second study addresses the lack of practical guidance for developers by introducing a systematic taxonomy that maps LLVM-CFI variants to common classes of memory corruption vulnerabilities. This taxonomy provides actionable recommendations to support incremental, informed adoption of CFI in existing codebases.

The third study examines the practical challenges of deploying CFI in a complex, production-grade runtime. Through a detailed case study of integrating LLVM-CFI into a modern Java Virtual Machine, it demonstrates that compatibility issues, manual exclusions, and maintenance effort are central obstacles to effective enforcement, even when strong CFI mechanisms are available. These findings highlight the gap between CFI as designed and CFI as deployed.

Building on these insights, the dissertation introduces an automated framework for CFI policy generation and enforcement. By reducing manual effort and mitigating compatibility barriers, this approach enables more consistent and scalable CFI deployment across large and evolving software systems.

Overall, the dissertation shows that the effectiveness of Control Flow Integrity in practice is shaped less by the availability of CFI mechanisms than by the feasibility of adopting them. By combining empirical measurement, practical guidance, deployment experience, and automation, this work contributes toward a more realistic and actionable understanding of CFI and provides concrete support for improving its deployment in real-world software systems.

Preface

This thesis contains the following papers.

Paper I Sabine Houy and Alexandre Bartel, **Twenty years later: Evaluating the Adoption of Control Flow Integrity**, *ACM Transactions on Software Engineering and Methodology, Volume 34, Issue 4, Article 103, 2025.*

Paper II Sabine Houy, Bruno Kreyssig, Timothée Riom, Alexandre Bartel, and Patrick McDaniel, **SoK: A Practical Guideline and Taxonomy to LLVM's Control Flow Integrity**, *IEEE Secure Development Conference (SecDev), 2025.*

Paper III Sabine Houy and Alexandre Bartel, **Lessons Learned and Challenges of Deploying Control Flow Integrity in Complex Software: the Case of OpenJDK's Java Virtual Machine**, *IEEE Secure Development Conference (SecDev), 2024.*

Paper IV Sabine Houy, Bruno Kreyssig, and Alexandre Bartel, **CFIghter: Automated Control Flow Integrity Enablement and Evaluation for Legacy C/C++ Systems**, *Under review, 11th IEEE European Symposium on Security and Privacy, 2026.*

In addition to the papers included in this thesis, the following publications were published within the studies but not included in this thesis.

Paper V Timothée Riom, Sabine Houy, Bruno Kreyssig, Alexandre Bartel. **Evaluating the maintainability of Forward-Porting vulnerabilities in fuzzer benchmarks**. *41st International Conference on Software Maintenance and Evolution (ICSME), 2025.*

Paper VI Bruno Kreyssig, Sabine Houy, Timothée Riom, and Alexandre Bartel. **Sleeping Giants-Activating Dormant Java Deserialization Gadget Chains through Stealthy Code Changes**. *ACM SIGSAC Conference on Computer and Communications Security (CCS), 2025.*

Paper VII Bruno Kreyssig, Sabine Houy, Hantang Zhang, Timothée Riom, and Alexandre Bartel. **GadgetBuilder: An Overhaul of the Greatest Java Deserialization Exploitation Tool**. *30th Nordic Conference on Secure IT Systems (NordSec), 2025.*

The work in this thesis was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Acknowledgements

I would like to express my sincere and heartfelt gratitude to Professor Alexandre Bartel for his guidance, support, and encouragement throughout this work. Beyond his invaluable scientific mentorship, their understanding, integrity, and personal support during challenging periods were essential to my ability to continue and complete this dissertation.

I would like to thank my fellow PhD students and colleagues, who made the everyday life of a PhD both engaging and bearable. I am especially grateful to Bruno, Cornelia, Tim, Divya, and Leila for the many conversations, shared frustrations, mutual support, and friendships that developed along the way.

This research was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), which is gratefully acknowledged.

Finally, I would like to thank my family for their unconditional support, patience, and understanding throughout the years of my doctoral studies. Their encouragement and belief in me provided a constant source of motivation, enabling me to persevere through both the challenges and the rewarding moments of this journey.

Glossary

CFG	Control Flow Graph
CFI	Control Flow Integrity
LLVM-CFI	LLVM's compiler-based implementation of CFI, integrated into Clang
LLVM IR	LLVM Intermediate Representation, human-readable counterpart to bytecode
vtable	*virtual table*: compiler-generated lookup table of function pointers used in object-oriented programming (like C++) to enable dynamic dispatch
vptr	*virtual pointer*: is a hidden pointer automatically added by a C++ compiler to objects of a class with virtual functions, pointing to that class's `vtable`

Contents

Chapter 1

Introduction

1.1 Research Motivation

Software systems written in memory-unsafe languages such as C and C++ continue to underpin modern computing infrastructure. Operating systems, browser engines, virtual machines, device drivers, and multimedia libraries rely on low-level code that offers performance and precise control over hardware resources. However, this programming model exposes systems to memory-corruption vulnerabilities, such as buffer overflows [6, 41, 40, 26], use-after-free errors [22], and type confusion [23], which consistently rank among the most severe and frequently exploited classes of software weaknesses. Decades after their initial discovery [45, 18], these vulnerabilities continue to dominate national vulnerability databases and security advisories [15], underscoring their persistent relevance.

Over the years, a wide range of mitigations have been introduced to reduce the exploitability of memory errors. Stack canaries [16], non-executable memory (NX) or Data Execution Prevention (DEP) [20], Address Space Layout Randomization (ASLR) [46], pointer authentication [19], hardened memory allocators, fortified standard libraries, and various compiler-level checks collectively form a layered set of protections. While these mitigations are effective against specific exploit strategies, each has known limitations. Canaries can be bypassed through information leaks or non-linear control flows; memory disclosures or side channels can weaken ASLR; pointer authentication can be subverted in the presence of replayable signatures; and hardened allocators do not prevent control-flow hijacking if an attacker gains influence over code pointers.

Control Flow Integrity (CFI) was proposed as a complementary mitigation to restrict unintended control-flow transfers and limit an attacker's ability to hijack program execution [1]. Although CFI predates modern code-reuse attacks, it has since become a central defense against techniques such as Return-Oriented Programming (ROP) [11, 57] and Jump-Oriented Program-

ming (JOP) [27, 10, 34]. Unlike earlier mitigations that focus on memory layout or metadata integrity, CFI restricts indirect control transfers so that they may only target a statically permitted set of destinations. By constraining the attacker's ability to chain unintended instructions into expressive payloads, CFI reduces the available attack surface even when other mitigations have been bypassed.

Although memory-unsafe languages such as C and C++ are the primary source of today's memory-corruption vulnerabilities, the relevance of CFI extends beyond these ecosystems. Modern platforms increasingly adopt memory-safe languages, most notably Rust, yet still rely on compiler-enforced control-flow validation as an additional hardening layer. LLVM already provides experimental CFI support for Rust [51, 50], and large-scale software ecosystems such as Android plan to expand their use of Rust [58] while continuing to employ CFI for components that remain performance-critical or require interoperability with legacy C and C++ code. Recent developments illustrate why this is necessary in practice: for the first time, a Rust component in the Linux kernel received a Common Vulnerabilities and Exposures designation (CVE-2025-68260), affecting the Android Binder subsystem [36]. The issue originates from an `unsafe` Rust block which introduced a race condition and potential memory corruption that could lead to a system crash or kernel panic. This incident underscores that the use of Rust alone does not eliminate the need for additional defenses against subtle control-flow and memory errors in large, mixed-language systems. Additionally, a complete transition to memory-safe languages is unlikely in the near future, and because CFI plays a complementary role even within memory-safe languages [51, 50], the technique is expected to remain relevant across evolving software stacks.

While several compiler toolchains provide their own CFI implementations, this dissertation focuses specifically on the variants integrated into LLVM/-Clang. LLVM-CFI has become the prevalent enforcement mechanism in modern software ecosystems, including Android [47, 55], major Linux distributions [14, 21, 63], and large-scale open-source projects [49, 53]. Moreover, LLVM-CFI offers finer-grained enforcement policies than alternatives such as GCC's *virtual table verification* (VTV) [66, 67, 60, 61], including multiple forward-edge variants that rely on precise type metadata and whole-program visibility [62, 64]. These properties enable more expressive and restrictive target sets, making LLVM-CFI particularly suitable for studying both the strengths and the compatibility challenges of modern CFI. GCC-based mechanisms are referenced where historically or conceptually relevant, but they are not part of the technical foundation of this work. Restricting the scope to LLVM-CFI ensures conceptual coherence and allows the dissertation to concentrate on the ecosystem where CFI is most actively developed and deployed.

Although CFI significantly restricts an attacker's ability to perform code-reuse attacks, it does not fully eliminate the risk of exploitation in systems written in memory-unsafe languages. CFI can be circumvented when target sets are overly permissive, when type information is incomplete, or when dynamic be-

havior contradicts static assumptions [10, 37]. Conversely, CFI is highly effective when combined with other mitigations: ASLR reduces knowledge of code locations, pointer authentication protects return addresses or `vtable` pointers, and hardened allocators prevent the corruption of specific metadata in the first place. This complementary role positions CFI as a crucial component in modern hardening pipelines, not as a stand-alone solution, but as part of an ensemble of mitigations whose combined effect significantly raises the bar for attackers.

Despite its theoretical robustness and its integration into modern compiler infrastructures such as LLVM/Clang [65, 62], CFI adoption in real-world systems remains surprisingly limited. In Paper I [29] we show that fewer than one percent of binaries in major Linux distributions include CFI instrumentation, and adoption in Android remains restricted to selected components [47, 5]. These observations raise a broader question that lies at the heart of this dissertation: **If CFI is mature, available, and conceptually well understood, why is it so rarely enabled in practice?**

A key part of the answer lies in the practical challenges of deploying CFI in complex, performance-critical software ecosystems. Type-based forward-edge CFI relies on assumptions about whole-program visibility, type consistency, symbol resolution, and predictable dispatch mechanisms [64, 67]. Real-world code routinely violates these assumptions through mixed-language components, dynamic loading, custom polymorphism models, legacy patterns, or engineering practices that predate modern compiler analyses [28]. When these incompatibilities manifest, enabling CFI can result in compilation failures, linkage issues, runtime `SIGILL` signals, or subtle functional regressions. Such failures discourage developers from adopting a mitigation that, in principle, is straightforward to enable.

This discrepancy between CFI's conceptual strengths and its practical deployability provides a central motivation for this dissertation. The aim is to (1) understand why CFI remains challenging to deploy in real-world software, (2) which compatibility issues arise when it is used at scale, and (3) how its integration into existing systems can be improved through empirical analysis, systematic guidance, and dedicated tooling.

1.2 Research Objectives

The overarching objective of this dissertation is to investigate why Control Flow Integrity, despite its conceptual maturity and extensive compiler support, remains challenging to deploy in real-world software systems. The work seeks to bridge the gap between theoretical advances in CFI and the practical requirements of large, heterogeneous, and performance-critical codebases. To this end, the dissertation formulates a series of interrelated research objectives that collectively address LLVM-CFI from the perspectives of deployment, applicability, and tooling.

A first objective is to **understand the current state of LLVM-CFI adoption** across major software ecosystems. Although LLVM/Clang provides production-ready forward- and backward-edge CFI mechanisms, little is known about their use outside controlled experimental settings. A first objective of this dissertation, therefore, aims to establish a foundational picture of CFI deployment in practice by identifying adoption patterns and characterizing the technical conditions under which LLVM-CFI appears in production binaries. This objective is addressed in Paper I [29], which introduces a static detection approach to identify the presence of CFI enforcement and reports the first ecosystem-scale measurement of LLVM-CFI usage.

A second objective is to **clarify the practical security role of LLVM-CFI** within the broader landscape of memory-safety mitigations. CFI is frequently described as a mitigation against code-reuse attacks, yet the relationship between specific classes of memory-corruption vulnerabilities and individual LLVM-CFI variants is often left unspecified. The dissertation seeks to provide a systematic mapping between memory-corruption weaknesses and the CFI policies that target them, using real CVEs and exploit reproductions to assess when CFI is likely to provide meaningful protection. The goal is not to establish a universal metric of CFI effectiveness, but to delineate its applicability boundaries and offer actionable guidance to developers. Paper II [31] contributes directly to this objective by evaluating how different LLVM-CFI variants interact with real-world vulnerability classes and by articulating which attack patterns CFI can reasonably be expected to mitigate.

A third objective concerns the **compatibility challenges that arise when LLVM-CFI is applied** to large, mature, or highly optimized software systems. Modern codebases rely on complex build infrastructures, mixed-language components, custom dispatch mechanisms, evolving type hierarchies, and dynamic linking, which often violate the assumptions required by type-based CFI. The dissertation, therefore, studies CFI deployment in a representative real-world codebase, namely the OpenJDK Java Virtual Machine, to uncover structural, architectural, and semantic sources of incompatibility. A central aspect of this objective is to demonstrate the substantial manual effort needed to diagnose and resolve CFI-related failures in such a setting. Enabling CFI requires careful inspection of type relationships, symbol-visibility constraints, dispatch semantics, and runtime behavior, making the deployment process time-consuming and error-prone. By systematically analyzing these challenges, the work explains why developers frequently encounter unexpected failures when enabling CFI and why the required manual intervention constitutes a significant barrier to adoption. This investigation is carried out in Paper III [28], which provides a detailed case study of CFI deployment in OpenJDK.

A fourth objective is to explore how **CFI deployment can be supported and partially automated through dedicated tooling**. If incompatibilities and manual debugging constitute central obstacles to adoption, then progress on deployability requires approaches that help developers detect, diagnose, and repair CFI violations. The dissertation, therefore, investigates how link-time

analysis, runtime tracing, and guided policy relaxation can be combined into an automated pipeline to resolve common CFI-related build and runtime issues. The goal is to reduce the manual burden associated with adoption and to demonstrate that CFI can be integrated into complex codebases when supported by appropriate analysis and automation strategies. Paper IV [30] fulfills this objective by presenting an automated framework for repairing CFI-related incompatibilities and enabling CFI in software that would otherwise remain unverifiable. In doing so, this objective also completes the broader arc initiated in Objective I. Suppose CFI adoption remains limited, partly because the manual effort required for deployment is prohibitive. In that case, automating this process provides a direct means to address one of the root causes of the lack of real-world usage.

Taken together, these objectives form a coherent research agenda that examines CFI not only as a security mechanism but as a practical software-engineering challenge. By studying adoption, applicability, incompatibility, and automation, the dissertation provides a comprehensive foundation for understanding how LLVM-CFI can be made more accessible and more reliably deployable in modern software ecosystems.

1.3 Methodology

The methodology of this dissertation is guided by the overarching goal of understanding the deployability of Control Flow Integrity in real-world software systems and by the four research objectives outlined in the previous section. Because CFI is simultaneously a security mitigation, a compiler feature, and a software-engineering constraint, no single methodological paradigm is sufficient to study its behavior in practice. The dissertation, therefore, adopts a mixed-methods approach that combines large-scale empirical measurement, vulnerability- and exploit-driven analysis, in-depth system study, and the design and evaluation of automated tooling.

To investigate the state of CFI deployment, the dissertation employs a quantitative empirical methodology grounded in static program analysis. A custom detection tool, SeeCFI, was developed to identify LLVM-CFI instrumentation in compiled binaries without requiring source code or build metadata. This tool operates on compiler-emitted metadata, symbol information, and binary-level patterns associated with CFI enforcement, enabling a systematic scan of large software repositories. By applying this analysis to major Linux distributions and other software ecosystems, the dissertation establishes a reproducible empirical baseline for understanding the real-world adoption of LLVM-CFI. The resulting dataset provides objective insights into the frequency of CFI use, where it appears, and which technical conditions correlate with successful deployment.

To clarify the practical security role of CFI, the dissertation adopts a vulnerability-centric methodology that combines controlled-experiment design

with a grounded analysis of real-world CVEs. Rather than relying solely on synthetic examples or theoretical reasoning, the work examines publicly disclosed vulnerabilities and proof-of-concept exploits across a range of weakness classes. Each vulnerability is reconstructed, instrumented, and evaluated under different LLVM-CFI policies to observe how enforcement affects exploitability. This method provides concrete evidence of the conditions under which CFI prevents control-flow hijacking and delineates its protection boundaries. The approach emphasizes reproducibility, empirical validation, and an understanding of security properties as they manifest in practical attack scenarios.

To study compatibility challenges, the dissertation employs a qualitative, diagnostic methodology centered on a system-level analysis of a large, industry-grade codebase. The OpenJDK Java Virtual Machine is an exemplary case study owing to its size, complexity, and long development history. Deploying LLVM-CFI in this environment requires examining compiler behavior, build-system configuration, type hierarchies, symbol visibility, dispatch mechanisms, and runtime interactions. Incompatible patterns are identified through a combination of compiler diagnostics, build failures, runtime exceptions, differential testing, and manual inspection of both source and intermediate representations. This approach reveals structural and semantic reasons why real-world software violates CFI assumptions and explains the substantial manual effort needed to resolve such violations.

Finally, to explore how CFI deployment can be automated, the dissertation adopts a system-design-and-implementation-based methodology to construct, refine, and evaluate a practical tooling pipeline. The approach is implemented in the automated repair framework CFIghter, which integrates link-time analysis, runtime tracing, and guided policy relaxation into an iterative process that resolves common CFI-related build and runtime issues. The resulting system is evaluated on real software to assess its ability to reduce manual intervention, restore functionality under CFI, and enable enforcement in cases where deployment would otherwise fail. This methodological component demonstrates how insights from earlier objectives can be operationalized into concrete tools that improve the practical deployability of LLVM-CFI.

Together, these complementary methodological strategies allow the dissertation to approach CFI from multiple angles: as an empirical phenomenon observable at the ecosystem scale, as a security mechanism measurable through practical vulnerability testing, as a source of engineering constraints revealed through case study analysis, and as a deployability problem amenable to automation. This multi-perspective methodology ensures that the findings are both theoretically grounded and practically relevant, providing a rigorous foundation for the results and contributions presented in the following chapters.

1.4 Thesis Outline

This dissertation is structured to reflect the progression of its research objectives and to integrate the four included papers into a coherent narrative. The thesis begins with background chapters that introduce the conceptual and technical foundations necessary to understand the challenges of deploying Control Flow Integrity in real-world software, followed by the papers that constitute the primary research contributions. It concludes with an integrative reflection on the results.

Chapter 2 introduces essential background on memory-corruption vulnerabilities and modern exploitation techniques. It motivates the need for complementary mitigations such as Control Flow Integrity and provides the security context in which CFI operates. Chapter 3 presents an overview of LLVM-CFI, its enforcement principles, available variants, and the assumptions that underlie its correctness. This chapter establishes the technical foundation for understanding both the potential and the limitations of compiler-enforced control-flow protection. Chapter 4 discusses the practical obstacles that arise when enabling CFI in large, heterogeneous software systems. It highlights structural, semantic, and build-system-related factors that complicate deployment and explains why real-world adoption remains limited.

Chapter 5 contains the four peer-reviewed papers that constitute the core contributions of this dissertation. Together, they follow a logical progression:

- Paper I investigates LLVM-CFI adoption at ecosystem scale and establishes the empirical motivation for the dissertation.
- Paper II examines how LLVM-CFI interacts with real-world vulnerability classes and clarifies its practical applicability.
- Paper III studies the compatibility challenges that arise when deploying CFI in a large, industry-grade codebase, revealing structural barriers that hinder adoption.
- Paper IV develops an automated framework for repairing CFI-related violations and enabling enforcement in complex software projects.

Each paper is preceded by a short contextual introduction that links it to the broader research objectives. Chapter 6 summarizes the dissertation by synthesizing the insights gained from the four papers. It reflects on the implications for secure software development and compiler-based mitigations. Further, it outlines promising avenues for future work, including improved tooling and increased interoperability with emerging memory-safe languages.

The dissertation concludes with the four papers, presented in their final accepted or at least submitted versions (only Paper IV). Their substantive content is unchanged, although the formatting has been adapted for consistency and improved readability within the dissertation.

Chapter 2

Memory Corruption and Software Exploits

Memory corruption remains a longstanding and deeply entrenched threat in systems software [3, 59]. Its persistence reflects the continued reliance on programming models that provide low-level memory access without accompanying safety guarantees. Although the consequences of memory errors are well understood and substantial work has been invested in mitigation, memory corruption continues to underlie many high-impact vulnerabilities in widely deployed software systems [52, 15, 13, 35]. This chapter provides a concise but rigorous overview of why memory corruption occurs, how attackers exploit it, and why such vulnerabilities remain relevant in today's computer systems. The discussion serves as the foundation for Chapter 3, which introduces Control Flow Integrity (CFI) as a systematic approach to constrain the effects of memory corruption.

2.1 Why Memory Corruption Happens

The C and C++ programming languages remain central to performance-critical and low-level system components because they offer precise control over memory layout, pointer manipulation, and object representation. This expressiveness comes at the cost of memory safety. Violations of bounds, lifetimes, and type assumptions result in undefined behavior, which the language standards deliberately leave unspecified [32]. As a consequence, compilers may rely on the assumption that such violations do not occur when reasoning about program behavior and applying optimizations. These optimizations can obscure or exacerbate the effects of faulty memory usage, and programs may continue executing in a corrupted state after a memory error, creating conditions that can later be exploited.

One prominent category of memory misuse consists of spatial errors [54, 52,

41, 26, 40]. These occur when a program accesses memory outside the bounds of an allocated object. In Listing 2.1, line 3 copies `n` bytes from `src` into the fixed-size buffer `buf`. Because the code on line 3 does not verify that `n` fits within `buf`, the write may extend beyond its allocated space. Once the write overlaps adjacent stack locations, the program may silently corrupt saved registers or the return address, thereby redirecting execution upon function return.

```
1 void copy(char *src, size_t n) {
2     char buf[16];
3     memcpy(buf, src, n);            // No bounds check on n
4 }
```

Listing 2.1: Spatial out-of-bounds write caused by a missing bounds check.

Temporal errors form a second major category [54, 52, 22]. A use-after-free occurs when memory is accessed after the program has deallocated it, resulting in a dangling pointer. In Listing 2.2, line 8 frees the object referenced by `w`. The subsequent write on line 10 performs an assignment through a stale pointer, and the call on line 11 invokes the function pointer embedded in memory that may no longer belong to the original object. If between `free(w)` and the use of `w`, an attacker reallocates the freed region with controlled data, the overwritten function pointer may dispatch execution to attacker-controlled code [42].

```
1  typedef struct {
2      void (*callback)(void);
3      int value;
4  } widget;
5
6  void process() {
7      widget *w = malloc(sizeof(widget));
8      free(w);                    // Lifetime ends here
9
10     w->value = 42;              // Undefined behavior
11     w->callback();              // Attacker may control this call
12 }
```

Listing 2.2: Use-after-free enabling corruption of an embedded function pointer.

A third category, type confusion [54, 52, 23], occurs when programs make incorrect assumptions about the type or structure of an object. This is especially relevant in C++ systems, where dynamic dispatch relies on predictable object layouts defined by the application binary interface (ABI) [33]. Objects with virtual methods contain a pointer to a virtual method table (`vtable`), which provides method dispatch entries. If an attacker overwrites this pointer, virtual calls may no longer resolve to their intended targets. Figure 2.1 illustrates how substituting the legitimate `vtable` with a forged one redirects the dispatch.

Across spatial, temporal, and type confusion errors, a shared characteristic emerges. Control-flow relevant state, such as return addresses, function pointers, and `vtable` pointers, resides alongside ordinary program data. Once

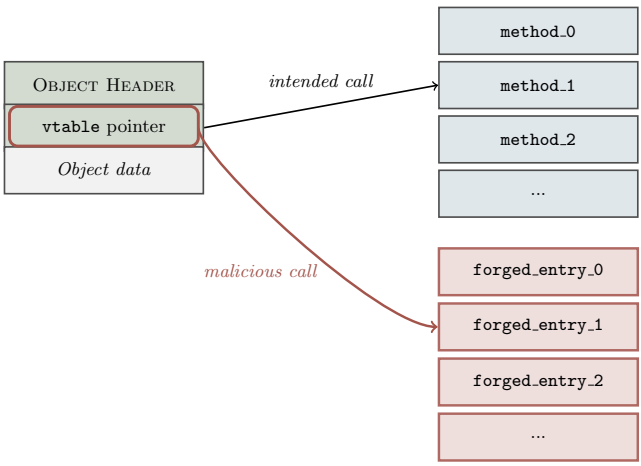

Figure 2.1: If the attacker successfully replaces the `vtable` pointer with a forged one but the index in the `vtable` stays as originally intended, then the execution (`call`) is redirected to the forged entry (`forged_entry_1`).

attackers can corrupt these values, the program cannot rely on its intended control flow.

2.2 How Memory Corruption Becomes Exploitation

Memory corruption by itself does not guarantee exploitation. Attackers typically transform an initial vulnerability into more expressive primitives. These primitives often include arbitrary memory reads, which reveal object layouts and code addresses, and arbitrary memory writes, which allow the attacker to modify sensitive program state. Even restricted versions of these capabilities can enable control-flow redirection.

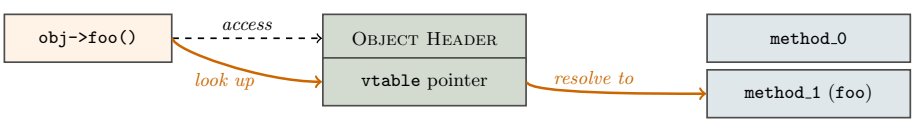

Figure 2.2: Virtual dispatch for `obj->foo()`. The call resolves through the `vtable` pointer embedded in the object.

Virtual dispatch in C++ illustrates how minor corruptions can undermine control flow. In Listing 2.1, a single unchecked write could overwrite an object's `vtable` pointer. A call such as `obj->foo()` compiles into a sequence that loads the `vtable` pointer and indexes into the dispatch table to obtain a target

address. Figure 2.2 shows how this process unfolds in practice. If an attacker overwrites the `vtable` pointer, the same call site may dispatch to arbitrary code. Figure 2.3 contrasts correct and corrupted dispatch paths. The two objects differ only in the value of their `vtable` pointers, yet their behavior diverges completely during dynamic dispatch.

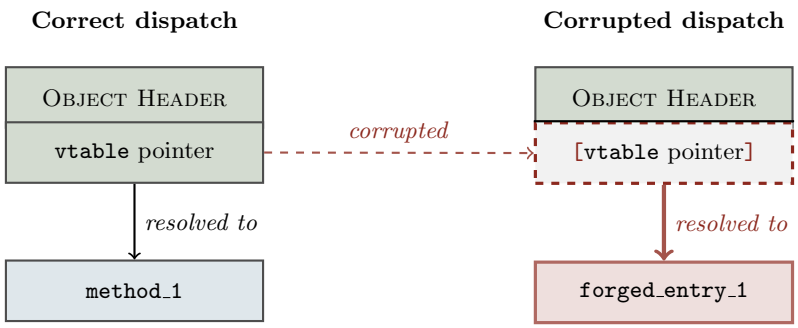

Figure 2.3: Correct and corrupted virtual dispatch. Overwriting the `vtable` pointer causes the call to dispatch to attacker-controlled code.

Exploitation often follows a sequence of steps. Figure 2.4 illustrates a representative use-after-free exploitation process. An object is allocated, then freed prematurely and replaced with a forged structure. A subsequent virtual call then dispatches to attacker-controlled code. Such patterns appear frequently in real-world exploits [9].

Traditional mitigations, such as a non-executable stack or stack canaries, reduce the likelihood of control-flow hijacking but do not eliminate it. In practice, all such mitigation techniques can be bypassed under specific conditions. Once execution is diverted to attacker-controlled code, these mechanisms no longer prevent exploitation. Return-oriented programming, introduced by Shacham [56], demonstrates that attackers can construct expressive behavior from short instruction sequences already present in memory. These techniques do not require code injection and instead rely on redirecting control flow.

2.3 Why This Still Matters in Practice

Memory corruption remains an influential class of vulnerabilities. Despite advances in programming languages, static analysis, and runtime mitigations, many long-lived systems rely on C and C++ for performance and control. Browsers, kernels, multimedia libraries, and network stacks continue to expose substantial attack surfaces built on memory-unsafe foundations.

Our empirical study (Paper I) [29] shows that attackers continue to exploit control-flow manipulation techniques even when modern defenses such as ASLR, stack canaries, hardened allocators, and non-executable memory are en-

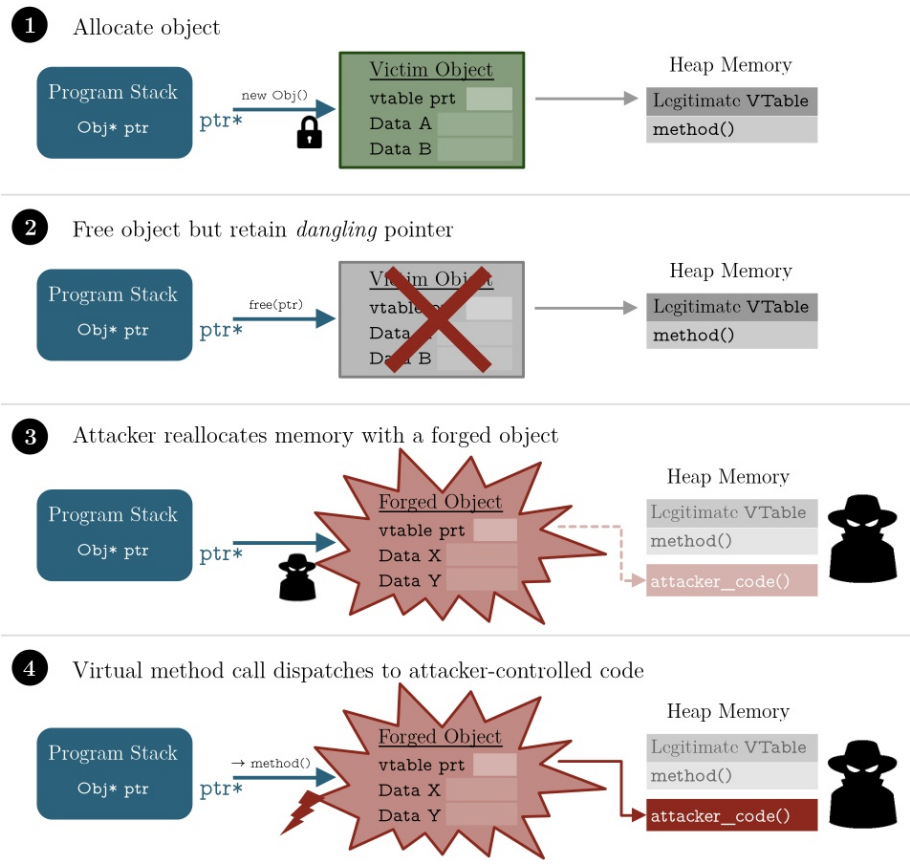

Figure 2.4: Representative exploitation timeline for a use-after-free vulnerability. An attacker replaces the freed object with a forged version.

abled. These mitigations raise the cost of exploitation but do not eliminate the underlying ability to corrupt control-flow relevant state. Prior systematization work highlights the longevity of memory corruption as a security problem and the incremental nature of many defensive advances [59]. Related work identifies similar trends across different application domains (Paper II) [31]. The persistence of memory-corruption vulnerabilities reflects the difficulty of migrating existing systems to memory-safe languages and the ongoing evolution of exploitation techniques.

These observations motivate a shift in perspective. Rather than attempting to eradicate all memory errors, which remains difficult for large and evolving codebases, one can impose structural restrictions on control flow. Control Flow Integrity embodies this idea. It seeks to limit indirect control-flow transfers to legitimate targets inferred from the program's intended behavior [2]. The next

chapter introduces this model and discusses how it constrains the impact of memory corruption in practice.

Chapter 3

Technical Foundations of Control Flow Integrity

Memory corruption vulnerabilities become security-critical when they allow attackers to influence control-flow decisions in low-level software. Such software extensively uses indirect control-flow mechanisms, such as function pointers, virtual dispatch, and jump tables, to improve modularity and performance. While Chapter 2 introduced memory corruption and its consequences at a conceptual level, the present chapter develops the technical foundations needed to understand how control flow is represented, manipulated, and constrained in compiled C and C++ systems. Control Flow Integrity (CFI) builds on a model of all legitimate control-flow transfers, but real toolchains and runtime environments leave significant room for ambiguity and imprecision. It was first introduced by Abadi et al. in 2005 [2] and revised in 2009 [1]. Five years later, the first compiler-chain (GNU/gcc & LLVM/Clang) implementations were proposed by Tice et al. [65]. This chapter, therefore, examines how compilers encode control flow, how these representations can be abused in the presence of memory corruption, and how contemporary CFI mechanisms, in particular LLVM's implementation, attempt to restrict attackers' influence. This foundation supports the practical perspective in Chapter 4 and prepares the ground for the enforcement-focused analyses presented in Chapter 5.

3.1 Control Flow as a Modeling Problem

CFI is fundamentally a model-based mitigation. It requires the compiler or analysis tool to derive the set of all legitimate targets for each indirect control-transfer instruction. While this seems straightforward in structured languages, C and C++ complicate matters substantially. Separate compilation, undefined behavior, pointer aliasing, function pointers, virtual dispatch, and aggressive optimizations all obscure the final set of dynamic control transfers that may

occur in the compiled program. Computing an exact control-flow graph (CFG) is infeasible for non-trivial systems code, and even compiler-internal CFGs represent only approximations tailored to specific optimization passes [59].

The CFG evolves as optimizations reshape dependencies, inline functions, eliminate virtual calls, or introduce synthetic dispatch mechanisms. For CFI to remain effective, its analysis must stay consistent with such transformations. Mismatches between analysis and transformation are a recurring source of fragility and directly influence what can be enforced in practice. Before examining enforcement in detail, it is helpful to understand how C and C++ express indirect control flow and why these expressions challenge precise modeling.

3.2 Program Representation and Indirect Control Flow

3.2.1 Compiler Representations of Control Flow

A CFG models the possible execution paths within a function, and a call graph relates functions to potential callees. Compilers construct these representations in their intermediate form, such as LLVM IR, but indirect calls, function pointers, virtual dispatch, and opaque constructs complicate target recovery. For CFI, these indirect transfers are the critical enforcement points. Only if the compiler can ascribe a reasonably precise set of legitimate targets to an indirect transfer can it emit meaningful runtime checks. Low-level code, characterized by unrestricted pointer manipulation and weak static typing, often forces compilers to make conservative over-approximations, thereby expanding the set of edges that a CFI policy must admit. This tension between precision and compatibility with existing code reappears throughout the thesis.

3.2.2 Function Pointers

Function pointers exemplify the difficulty of recovering precise target sets even in simple cases. Listing 3.1 illustrates a common callback pattern.

Listing 3.1: Callback invocation through a function pointer.

```
1 void (*callback)(int);
2
3 void register_callback(void (*cb)(int)) {
4     callback = cb;
5 }
6
7 void run(void) {
8     callback(42);
9 }
```

The invocation on line 8 is an indirect call whose target depends on the runtime contents of `callback`. The compiler cannot readily infer which functions may flow into this pointer across large codebases. Complex control paths and aliasing relationships may separate assignments on line 4 and the call on line 8. Since CFI aims to remain sound at the level of static analysis, all statically plausible targets must be permitted, even if only a subset occur at runtime [59]. Several of the included papers show that callback-heavy frameworks lead to large function-pointer target sets in practice [29, 31, 59, 7, 65, 38]. A similar effect occurs when function pointers reside in heap-allocated objects. Listing 3.2 illustrates a representative use-after-free pattern.

Listing 3.2: Use-after-free on a structure containing a function pointer.

```
1  typedef struct {
2      void (*callback)(void);
3      int value;
4  } widget;
5
6  void process(void) {
7      widget *w = malloc(sizeof(widget));
8      /* ... initialise w ... */
9      free(w);                    /* Lifetime ends here */
10
11     w->value = 42;              /* Undefined behavior */
12     w->callback();              /* Attacker may control this call
           */
13 }
```

After the free on line 9, the writes on line 11 and the call on line 12 operate through a dangling pointer. If an attacker reallocates the freed region with controlled data, the overwritten function pointer is invoked as in Listing 3.1, but with a malicious target. CFI can reject such attempts only if the injected pointer falls outside the statically inferred target set.

3.2.3 Virtual Dispatch in C++

Polymorphism introduces another common source of indirect control flow. Under the standard C++ ABI, each polymorphic object contains a `vptr` referencing a table of virtual method pointers. Listing 3.3 provides a minimal example.

Listing 3.3: Virtual dispatch in C++.

```
1  struct Base {
2      virtual void f();
3  };
4
5  struct Derived : Base {
6      void f() override;
7  };
8
9  void g(Base *b) {
10     b->f();
11 }
```

The call `b->f()` resolves by reading the object's `vptr` and selecting the appropriate entry in its dispatch table. A conceptual object layout is presented in Figure 3.1.

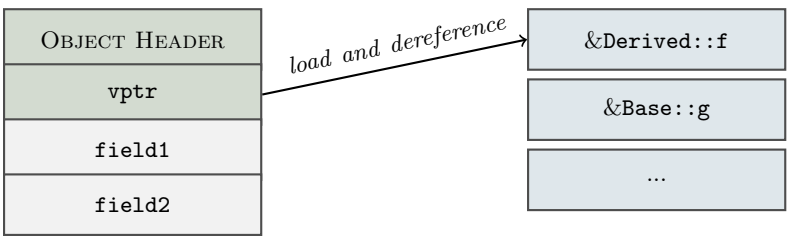

Figure 3.1: Conceptual layout of an object with its `vptr` and associated `vtable`.

Virtual dispatch is attractive for CFI because the `vtable` provides structured metadata. LLVM's `cfi-vcall` instruments these calls by verifying that the `vtable` referenced by the object's `vptr` carries a type identifier compatible with the static type at the call site. If an attacker corrupts the `vptr`, a type mismatch can trigger an abort (*Illegal instruction* signal) under the enforced CFI policy.

3.2.4 Inline Assembly and Opaque Constructs

Inline assembly blocks are opaque to compiler analysis. Since LLVM cannot interpret their semantics, it must conservatively assume that inline assembly may redirect control flow in unpredictable ways. Several of the included papers observe that this opacity can prevent LLVM from inserting CFI checks or may cause mismatches between static assumptions and generated code. Although inline assembly is necessary in systems code, it often complicates precise modeling and contributes to deployment fragility.

3.3 Threat Model and Attack Surface

Memory corruption enables attackers to influence control flow by modifying code pointers, object metadata, or stack-resident control data. Forward-edge hijacking refers to redirecting the target of an indirect call or jump, for example by corrupting a function pointer or a `vptr`. Backward-edge attacks manipulate return addresses, but they are typically mitigated by shadow stacks or hardware mechanisms such as Intel CET.

This chapter focuses on the forward edge. The measurement study in one of the included papers confirms that corruptions of function pointers and `vptrs` remain prevalent across large, mature software ecosystems. Because static analyses over-approximate target sets, attackers may be able to select powerful

yet type-compatible targets under certain conditions. CFI attempts to limit this freedom by enforcing per-site or per-type target restrictions.

3.4 Constructing a Practical Control-flow Model

CFI enforcement requires a model of all allowed targets for indirect transfers. Ideally, this model is both sound, admitting all targets that may occur at runtime, and precise, excluding those that cannot legitimately occur. In practice, soundness is limited by separate compilation and dynamic linking, and precision is constrained by the scalability of pointer analysis and the complexity of class hierarchies. Paper II develops a taxonomy that describes these trade-offs.

Table 3.1 summarizes key obstacles to CFG precision.

Table 3.1: Sources of imprecision in CFG construction that affect CFI policies.

Source	**Effect on control-flow modeling**
Pointer aliasing	Expands function-pointer target sets because precise alias information is unavailable.
Separate compilation	Prevents whole-program visibility of object hierarchies and call targets.
Dynamic linking	Allows unforeseen overrides and target definitions at load time.
Optimizations	Transform or eliminate call edges, which invalidates naive assumptions about control flow.
Inline assembly	Forces conservative assumptions about opaque indirect control transfers.

Figure 3.2 depicts how imprecision interacts with attacker influence.

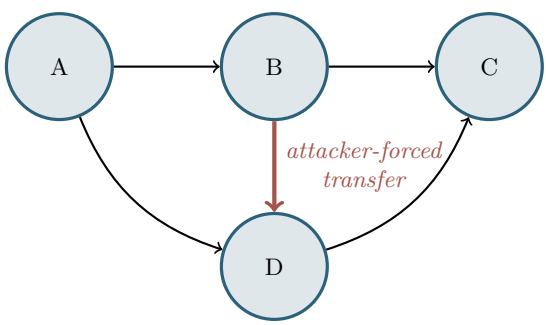

Figure 3.2: Imprecision in the static CFG may permit attacker-steered paths.

Even if both `C` and `D` are legitimate targets at compile time, an attacker may force execution along unintended sequences. This motivates finer-grained CFI

policies, although the effectiveness of these policies is limited by the modeling constraints described earlier.

3.5 LLVM's Implementation of Control Flow Integrity

LLVM implements a family of CFI schemes through sanitizers in Clang. These schemes instrument indirect transfers with dynamic checks that validate the runtime target against a statically derived type identifier. LLVM exposes seven variants, and each variant targets a different class of control-flow or type-safety violation [62].

LLVM associates each function type and each polymorphic class type with a stable identifier that is derived from its mangled name. Clang attaches this metadata to functions, call sites, casts, and `vtables`. A dedicated CFI pass instruments indirect calls by comparing the dynamic target against the expected type. If the comparison fails, the inserted trap aborts execution.

Figure 3.3 illustrates this process for `cfi-icall`.

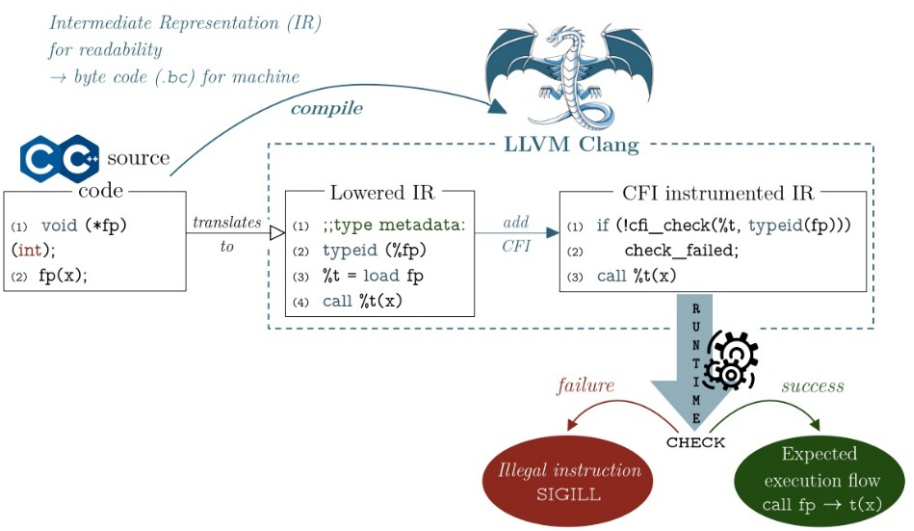

Figure 3.3: Simplified illustration of `cfi-icall`. The compiler assigns each function type a stable identifier, attaches type metadata to indirect call sites, and inserts a runtime check to validate the dynamic target. A mismatch aborts execution.

The seven CFI variants can be summarized as follows.

All variants rely on consistent visibility of type information. Whole-program compilation or link-time optimization is often required to ensure that LLVM

Table 3.2: Overview of LLVM-CFI's variants and their primary targets.

Variant	Primary construct	Informal goal
`cfi-vcall`	Virtual calls on polymorphic objects	Prevent calls through corrupted `vptrs` or forged objects.
`cfi-nvcall`	Non-virtual member function calls	Enforce type discipline on calls that depend on receiver type.
`cfi-icall`	Function-pointer calls	Restrict targets of function-pointer calls to compatible signatures.
`cfi-mfcall`	Member-function-pointer calls	Validate the structure and target type of member function pointers.
`cfi-derived-cast`	Base-to-derived casts	Detect unsafe downcasts that produce invalid object views.
`cfi-unrelated-cast`	Casts between unrelated types	Prevent misuse of generic pointers as arbitrary class types.
`cfi-cast-strict`	Strict base-to-derived casts	Close remaining gaps caused by layout-compatible hierarchies.

sees all relevant definitions. Symbol visibility annotations also matter, especially when dynamic linking introduces new subclasses or override relationships that are not visible at compile time. These constraints reappear in Chapter 4, where their impact on real-world deployments is examined.

3.6 Summary

This chapter surveyed the mechanisms underlying indirect control flow in C and C++, the challenges these mechanisms pose for static modeling, and the practical enforcement strategies implemented in LLVM-CFI. Function pointers, virtual dispatch, and opaque constructs contribute to a rich but difficult-to-model control-flow structure. CFI builds a model on top of this structure, but the model must contend with incomplete visibility, dynamic linking, and the intrinsic limitations of scalable analysis.

LLVM-CFI mechanisms illustrate how an abstract security policy becomes a concrete toolchain transformation. The seven variants offered by Clang target different language constructs and type-safety violations, and they expose practical constraints related to symbol visibility and whole-program optimization. These foundations are essential for understanding the challenges examined in Chapter 4 and for interpreting the enforcement analyses in Chapter 5. The remaining chapters build on this groundwork to investigate where current CFI mechanisms succeed, where they fall short, and how empirical evidence can inform more practical designs.

Chapter 4

Practical Challenges in Deploying CFI

Despite more than two decades of research, Control Flow Integrity has seen uneven adoption in production software systems. As discussed in Chapter 2, memory corruption vulnerabilities remain pervasive in modern systems, motivating defenses that constrain attackers' control over execution. Chapter 3 introduced CFI as a principled approach to limiting the impact of such vulnerabilities by restricting indirect control-flow transfers. While the core idea of CFI is well established, its practical deployment remains challenging in real-world environments. These challenges arise not from a lack of effective designs but from persistent tensions between security guarantees, performance constraints, compatibility requirements, and engineering complexity in real-world deployments. As a result, many deployed CFI mechanisms provide weaker protection than envisioned initially or are enabled only in restricted configurations [29, 7]. This chapter examines the primary obstacles that hinder widespread CFI deployment and situates them within the broader trajectory of real-world adoption studied throughout this dissertation.

4.1 Precision Versus Practicality

Strong CFI enforcement depends on the availability of a precise and sound control flow graph. Early formulations assume that such a graph can be computed statically and enforced at runtime [2, 1]. In practice, constructing a precise graph for large, modular software systems is difficult due to separate compilation, dynamic linking, function pointers, and the extensive use of virtual dispatch. Static analyses must therefore conservatively approximate control flow in the presence of incomplete information, leading to coarse equivalence classes that weaken security guarantees. Prior work demonstrates that attackers may exploit permissive classes to redirect execution to unintended but valid

targets, enabling expressive code reuse attacks despite CFI enforcement [10, 27, 17].

Increasing precision typically requires whole-program analysis or link-time optimization. However, these techniques conflict with common build practices and modular architectures, particularly in large and evolving code bases [65, 43]. As observed in large-scale adoption studies, this tension frequently leads developers to favor deployability over strict precision [29]. This tradeoff directly reflects the distinction between theoretical CFI models and deployable mechanisms discussed in Chapter 3. While precise control-flow graphs strengthen formal guarantees, real-world systems frequently sacrifice precision to preserve modularity, build scalability, and long-term maintainability.

4.2 Performance Overhead and Optimization Constraints

CFI enforcement introduces runtime checks for indirect control-flow transfers. Although individual checks are lightweight, their cumulative cost can be significant in performance-sensitive applications, especially those that rely heavily on indirect calls, including object-oriented C++ code and low-level system components [7]. Empirical evaluations generally show that fine-grained CFI incurs higher overhead than coarse-grained alternatives [2, 65]. To meet performance requirements, production deployments often reduce enforcement strength, selectively omit checks on hot paths, or restrict CFI to specific build configurations [39, 28].

Compiler optimizations further complicate enforcement. Transformations such as inlining, tail-call elimination, and control-flow restructuring affect the placement and correctness of CFI checks. Ensuring sound enforcement across optimization passes requires tight integration with the compiler backend, which increases engineering effort and limits portability across toolchains [65, 62].

4.3 Compatibility with Existing Software Ecosystems

Backward compatibility remains a central obstacle to strict CFI deployment. Many real-world applications rely on behaviors that are difficult to model precisely, including runtime code loading, custom calling conventions, and intentional violations of language abstractions. Strict CFI policies may therefore reject valid programs even in the absence of vulnerabilities [68]. Legacy C and C++ code frequently uses function pointers in ways that violate static type assumptions, including casting between incompatible function types or reusing function pointer fields for multiple purposes. While such practices constitute undefined behavior, they are widespread and often relied upon for performance

or flexibility [59]. Type-based CFI mechanisms must therefore weaken policies or introduce compatibility exceptions to accommodate these patterns [25].

Dynamic linking further complicates enforcement. Shared libraries may be loaded at runtime without full knowledge of the main program, thereby preventing complete control-flow graph construction at compile time. Modular and runtime techniques mitigate this limitation, but at the cost of increased complexity and overhead [43, 47].

4.4 Incomplete Coverage and Enforcement Gaps

Even when CFI is enabled, enforcement is often incomplete. Many implementations focus on indirect calls and returns, while excluding complex control flow mechanisms such as exception handling, `setjmp` and `longjmp`, or signal delivery [1]. Partial deployment is also common, with CFI enabled only for selected components, libraries, or build modes. Such gaps reduce the overall effectiveness of enforcement and may be exploitable to regain control over execution [10]. Large-scale empirical studies confirm that incomplete coverage is prevalent in real-world systems, particularly in long-lived and heterogeneous code bases [5, 29].

These limitations often reflect deliberate engineering tradeoffs rather than implementation oversights. Nevertheless, they underscore the difficulty of achieving comprehensive control flow protection in practice. From the attacker capabilities outlined in Chapter 2, these enforcement gaps are particularly relevant, as residual unprotected control-flow mechanisms can serve as entry points for code-reuse attacks even when core CFI checks are present.

4.5 Deployment Case Studies

The challenges discussed above manifest clearly in real-world deployments. LLVM-based CFI illustrates the tension between precision and compatibility. Although LLVM supports multiple CFI variants, developers frequently enable only coarse-grained policies to preserve compatibility with existing code and build workflows [62, 31], resulting in wide variation in enforcement strength across projects.

Android demonstrates the impact of performance and compatibility constraints at scale. While Android has progressively expanded CFI deployment across user space and kernel components, enforcement remains selective and is often combined with complementary mitigations such as shadow stacks or pointer authentication [47, 48]. Linux kernel CFI similarly highlights the challenges of partial coverage. Kernel CFI primarily focuses on forward-edge protection and excludes several complex control-flow mechanisms for practical reasons [21, 14]. Together, these case studies reinforce the observation that

real-world CFI deployments are shaped as much by engineering constraints as by security goals.

4.6 Security Guarantees and Threat Models

The threat models assumed by most CFI designs align with the attacker model introduced in Chapter 2, where memory corruption enables manipulation of control data but not arbitrary code modification [1]. In practice, however, real-world exploits often combine memory corruption with information leaks or other weaknesses, challenging the assumptions under which CFI guarantees are defined [56, 11]. Moreover, formal guarantees do not imply that remaining behaviors are benign. Prior work shows that coarse-grained CFI policies may still permit Turing-complete code reuse attacks [10, 24]. This gap between formal correctness and practical security complicates both evaluation and deployment decisions [7, 29].

4.7 Summary

Taken together, the challenges discussed in this chapter explain why CFI deployment in practice often diverges from the idealized mechanisms described in Chapter 3. Tradeoffs between precision, performance, compatibility, coverage, and threat modeling shape real-world enforcement decisions and directly influence the protection actually achieved. The next chapter builds on this analysis by summarizing the dissertation's contributions and situating them within the context of contemporary CFI deployment.

Chapter 5

Summary of Contributions

This dissertation examines Control Flow Integrity from the perspective of real-world deployment, adoption, and practicality. Although CFI has been studied extensively as a security mechanism, its use in production software systems remains uneven and often diverges from the assumptions made in early designs. The central contribution of this thesis is a systematic analysis of how CFI is deployed in practice, why deployment remains challenging, and how deployment choices shape the security guarantees that can realistically be expected. The dissertation consists of four peer-reviewed research papers that together form a coherent progression. Rather than proposing a new CFI mechanism in isolation, this work focuses on measuring adoption, identifying deployment barriers, providing actionable guidance, and enabling reproducible evaluation under practical constraints.

5.1 Paper I

"*Twenty years later: Evaluating the Adoption of Control Flow Integrity*"

The first paper presents a large-scale empirical sthe adoption of Control Flow Integrity doption across modern software ecosystems. By examining operating systems, system libraries, and widely deployed applications, this work shows that CFI adoption remains partial and highly heterogeneous, even decades after its introduction. The analysis reveals substantial variation in enforcement configurations, coverage, and policy granularity, with many deployments relying on coarse-grained policies or limited protection scopes [29]. Beyond quantifying adoption rates, this paper identifies systematic patterns in deployment decisions. It shows that CFI is often enabled selectively, restricted to specific components, or deployed with weakened policies to preserve compatibility and performance. These findings provide concrete evidence of the gap between theoretical CFI guarantees and real-world enforcement and motivate the need to

study CFI as a deployed system rather than an abstract defense.

5.2 Paper II

"*SoK: A Practical Guideline and Taxonomy to LLVM's Control Flow Integrity*"

The second paper addresses the lack of practical guidance for deploying CFI in complex and heterogeneous software systems. Based on an in-depth analysis of LLVM's compiler-based CFI mechanisms, this work introduces a systematic taxonomy that maps CFI variants to classes of memory corruption vulnerabilities. This mapping makes explicit the tradeoffs between enforcement precision, performance overhead, compatibility constraints, and the level of protection implied by different CFI configurations [31]. Building on this taxonomy, the paper derives a practical guideline to help developers reason systematically about CFI configuration and incremental deployment decisions. Rather than treating CFI as a defense mechanism, this contribution frames it as a spectrum of enforceable policies whose effectiveness depends on both vulnerability characteristics and deployment context. This perspective enables more informed engineering decisions and reduces the risk of misinterpreting the protection provided by a given configuration.

5.3 Paper III

"*Lessons Learned and Challenges of Deploying Control Flow Integrity in Complex Software: the Case of OpenJDK's Java Virtual Machine*"

The third paper investigates the challenges of enabling CFI in real-world codebases. Through hands-on deployment experience and systematic analysis, it identifies recurring obstacles related to build systems, language features, compiler optimizations, and legacy coding patterns. The study shows that many enforcement gaps and policy weakenings arise from deliberate engineering tradeoffs rather than implementation errors [28]. This paper provides a realistic account of the effort required to deploy CFI at scale and explains why strong enforcement is difficult to achieve in practice. By documenting concrete failure modes and compatibility issues, it complements the empirical findings of Paper I and the conceptual framework of Paper II.

5.4 Paper IV

"CFIghter*: Automated Control Flow Integrity Enablement and Evaluation for Legacy C/C++ Systems*"

The fourth paper presents CFIghter, an automated framework for enabling, configuring, and evaluating Control Flow Integrity in legacy C and C++ software systems. The paper addresses the practical challenge that many real-world codebases were not designed with CFI in mind and cannot be instrumented reliably with default compiler configurations. CFIghter systematizes the process of building, instrumenting, and evaluating software with compiler-based CFI, reducing the manual effort typically required to deploy and analyze CFI in complex projects [30]. Rather than proposing a new CFI mechanism, this contribution focuses on operationalizing existing CFI implementations. CFIghter exposes configuration limitations, unsupported language constructs, and build system constraints that prevent full enforcement, making these obstacles explicit and measurable. By applying the framework to a diverse set of real-world programs, the paper demonstrates that automated CFI enablement is often feasible under realistic constraints, while also showing that coverage and policy strength vary significantly across projects.

Taken together, these four papers advance the understanding of Control Flow Integrity as a practical defense rather than a purely theoretical construct. They provide empirical evidence of current deployment practices, actionable guidance for engineers, and tooling support for reproducible evaluation under realistic constraints. More broadly, this dissertation emphasizes the importance of evaluating security mechanisms not only by their formal guarantees, but also by how they are deployed, configured, and maintained in production systems.

Chapter 6

Conclusion & Future Work

This dissertation examined Control Flow Integrity through the lens of real-world adoption and deployment. While CFI has long been established as a principled mitigation against control-flow hijacking, its practical security impact depends on how it is selected, integrated, and sustained in production software systems. Across four complementary papers, this thesis investigated the state of CFI adoption in practice, the guidance available to developers during deployment, the engineering challenges encountered in complex systems, and the role of automation in enabling consistent enforcement.

Taken together, the four papers that constitute this dissertation provide a consolidated view of why Control Flow Integrity remains challenging to deploy despite its conceptual maturity. Rather than treating CFI as an abstract enforcement mechanism, this work grounds its analysis in empirical measurement, developer-facing guidance, and production engineering constraints that shape real-world systems.

Paper I established an empirical baseline for CFI adoption in practice. By measuring effective enforcement in deployed binaries across major software platforms, it showed that LLVM-CFI remains unevenly adopted and is frequently enabled only in partial or restricted forms. These findings confirm earlier observations that exploit mitigations often diverge from their intended designs once deployed at scale [59, 7]. Despite extensive research following the original CFI proposals [2, 1], strong and comprehensive enforcement remains the exception rather than the norm in production systems.

Paper II addressed the lack of practical guidance faced by developers attempting to adopt CFI. It introduced a systematic taxonomy that maps LLVM-CFI variants to classes of memory-corruption vulnerabilities, providing a structured basis for informed, incremental deployment decisions. Rather than evaluating attacker behavior, this taxonomy clarifies what different CFI mechanisms are designed to protect, where their limitations lie, and how compatibility constraints influence their applicability. This perspective aligns with prior work showing that enforcement precision, performance overhead, and maintainabil-

ity are tightly coupled in large code bases [65, 38, 5, 31].

Paper III examined the practical challenges of deploying LLVM-CFI in a complex, production-grade software system. Through a detailed case study of integrating LLVM-CFI into a modern Java Virtual Machine, it demonstrated that compatibility issues, undefined behavior, legacy code patterns, and toolchain limitations are central obstacles to effective enforcement. The study showed that substantial manual effort, including selective exclusions and iterative debugging, is often required to achieve a stable deployment. These findings illustrate that weakened or partial enforcement is frequently not a design choice but an engineering necessity [7, 28].

Paper IV was built directly on these observations by addressing the root causes of fragile and inconsistent CFI deployment. It introduced an automated framework for LLVM-CFI enforcement, designed to reduce manual effort and mitigate compatibility barriers. By automating policy derivation and enforcement decisions, this approach demonstrates how stronger and more consistent CFI deployment can be made practical in large, evolving codebases. This contribution reframes automation as a prerequisite for scalable adoption of CFI rather than an optional optimization.

Taken together, the results of this dissertation suggest that Control Flow Integrity should be understood as a spectrum of enforcement strategies rather than as a binary security property. Deployed systems routinely occupy intermediate points along this spectrum, balancing security benefits against performance, compatibility, and maintenance constraints. Empirical measurement and deployment-aware analysis are therefore essential for understanding the effective guarantees provided by CFI in practice, as design intent alone often diverges from deployed reality [59, 5].

At the same time, the findings indicate that partial enforcement remains valuable. Even incomplete or coarse-grained CFI policies can meaningfully reduce the exploitability of memory corruption vulnerabilities when applied consistently [65, 10]. This supports incremental deployment strategies that prioritize broad coverage and robustness over maximal policy precision, especially in complex legacy systems.

6.1 Future Work

The findings of this dissertation point to several directions for future research.

One promising direction is improved tooling for deployment transparency. Developers and system integrators currently lack standardized mechanisms to inspect, validate, and reason about the effective CFI policies enforced in compiled binaries. Improved visibility into real enforcement could reduce misconfigurations and strengthen practical guarantees [27, 68].

Another direction concerns adaptive enforcement strategies. Rather than relying on a single static policy, future systems could adjust CFI precision based on performance budgets, execution context, or code maturity. Adaptive

approaches have been explored in related work on control-flow protection and may offer a path toward stronger guarantees without prohibitive overhead [44, 4].

A third avenue is deeper integration between CFI and complementary mitigations. In practice, CFI interacts with mechanisms such as address space layout randomization, shadow stacks, and hardware-supported control-flow protections. Understanding how these defenses compose, and where their assumptions conflict, remains an important open problem [8, 12].

Finally, future work should continue to ground CFI research in real-world constraints. As long as performance-critical systems rely on memory-unsafe languages, enforcement mechanisms must contend with legacy code, undefined behavior, and evolving toolchains. Bridging the gap between principled security models and deployable mitigations will require sustained attention to engineering realities alongside theoretical advances [29].

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. 2005, pp. 340–353. DOI: 10.1145/1102120.1102165.

[3] James P Anderson. *Computer security technology planning study*. Tech. rep. 1972.

[4] Erick Bauman, Jun Duan, Kevin W Hamlen, and Zhiqiang Lin. "Renewable Just-In-Time Control-Flow Integrity". In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 2023, pp. 580–594.

[5] Lucas Becker, Matthias Hollick, and Jiska Classen. "{SoK}: On the Effectiveness of {Control-Flow} Integrity in Practice". In: *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*. 2024, pp. 189–209.

[6] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. "Smashing the stack protector for fun and profit". In: *ICT Systems Security and Privacy Protection (IFIP SEC 2018)*. Springer. 2018, pp. 293–306.

[7] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. "Control-flow integrity: Precision, security, and performance". In: *ACM Computing Surveys (CSUR)* 50.1 (2017), pp. 1–33.

[8] Nathan Burow, Xinping Zhang, and Mathias Payer. "SoK: Shining light on shadow stacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 985–999.

[9] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 2012, pp. 133–143.

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. "{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 161–176.

[11] Nicholas Carlini and David Wagner. "{ROP} is still dangerous: Breaking modern defenses". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 385–399.

[12] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. "HCFI: Hardware-enforced control-flow integrity". In: *CODASPY*. 2016, pp. 38–49.

[13] chromium.org. *Memory safety*. `https://www.chromium.org/Home/chromium-security/memory-safety/`. Accessed 2025-12-11.

[14] Jonathan Corbet. *Control-flow integrity in 5.13 [LWN.net]*. `https://lwn.net/Articles/856514/`. Accessed 2025-12-02. May 2021.

[15] The MITRE Corporation. *CWE - 2025 CWE Top 10 KEV List Insights*. `https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html`. Accessed 2026-01-01.

[16] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. *Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks*. 1998.

[17] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection". In: *USENIX Security*. 2014, pp. 401–416.

[18] Solar Designer. *Getting around non-executable stack (and fix)*. 1997.

[19] ARM Developers. *Armv8.1-M PACBTI Extensions*. `https://developer.arm.com/documentation/109576/0100/Pointer-Authentication-Code/Introduction-to-PAC`. Accessed 2025-12-02.

[20] Windows App Development. *Data Execution Prevention - Win32 apps | Microsoft Learn*. `https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention`. Accessed 2025-12-05.

[21] Jake Edge. *Control-flow integrity for the kernel [LWN.net]*. `https://lwn.net/Articles/810077/`. Accessed 2025-12-02. Jan. 2020.

[22] MITRE - Common Weakness Enumeration. *CWE-416: Use After Free*. `https://cwe.mitre.org/data/definitions/416.html`. Accessed 2025-12-02.

[23] MITRE - Common Weakness Enumeration. *CWE-843: Access of Resource Using Incompatible Type (Type Confusion)*. https://cwe.mitre.org/data/definitions/843.html. Accessed 2025-12-02.

[24] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. "Control jujutsu: On the weaknesses of fine-grained control flow integrity". In: *ACM CCS*. 2015, pp. 901–913.

[25] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. "On the effectiveness of type-based control flow integrity". In: *ACSAC*. 2018, pp. 28–39.

[26] OWASP Foundation. *Buffer Overflow | OWASP Foundation*. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow. Accessed 2025-12-02.

[27] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. "Out of control: Overcoming control-flow integrity". In: *IEEE S&P*. 2014, pp. 575–589.

[28] Sabine Houy and Alexandre Bartel. "Lessons Learned and Challenges of Deploying Control Flow Integrity in Complex Software". In: *IEEE Secure Development Conference (SecDev)*. 2024, pp. 153–165.

[29] Sabine Houy and Alexandre Bartel. "Twenty years later: Evaluating the Adoption of Control Flow Integrity". In: *ACM TOSEM* 34.4 (2025), pp. 1–30.

[30] Sabine Houy, Bruno Kreyssig, and Alexandre Bartel. "CFIghter: Automated Control-Flow Integrity Enablement and Evaluation for Legacy C/C++ Systems". In: *Under review: 11th IEEE European Symposium on Security and Privacy*. https://arxiv.org/abs/2512.22701. 2026.

[31] Sabine Houy, Bruno Kreyssig, Timothée Riom, Alexandre Bartel, and Patrick McDaniel. "SoK: A Practical Guideline and Taxonomy to LLVM's Control Flow Integrity". In: *IEEE Secure Development Conference (SecDev)*. 2025, pp. 129–141. DOI: 10.1109/SecDev66745.2025.00024.

[32] *ISO/IEC 9899:201x - n1570.pdf*. Programming Languages — C. 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[33] *Itanium C++ ABI*. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. Accessed 2025-12-08.

[34] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. "Return-oriented programming on RISC-V". In: *ACM Asia CCS*. 2020, pp. 471–480.

[35] Billy Lau Kevin Deus Joel Galenson and Android Security & Privacy Team Ivan Lozano. *Google Online Security Blog: Data Driven Security Hardening in Android*. https://security.googleblog.com/2021/01/data-driven-security-hardening-in.html. Accessed 2025-12-02. Jan. 2021.

[36] Michael Larabel. *Linux Kernel Rust Code Sees Its First CVE Vulnerability - Phoronix*. https://nvd.nist.gov/vuln/detail/CVE-2024-34391. Accessed 2025-12-02. Dec. 2025.

[37] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. "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*. 2020, pp. 1821–1835.

[38] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. "CCFI: Cryptographically enforced control flow integrity". In: *ACM CCS*. 2015, pp. 941–951.

[39] Nicoló Maunero, Paolo Prinetto, and Gianluca Roascio. "CFI: Control flow integrity or control flow interruption?" In: *IEEE EWDTS*. 2019, pp. 1–6.

[40] MITRE - Common Weakness Enumeration. *CWE - CWE-121: Stack-based Buffer Overflow (4.15)*. https://cwe.mitre.org/data/definitions/121.html. Accessed 2024-11-18.

[41] MITRE - Common Weakness Enumeration. *CWE - CWE-122: Heap-based Buffer Overflow (4.15)*. https://cwe.mitre.org/data/definitions/122.html. Accessed 2024-11-18.

[42] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavy-weight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[43] Ben Niu and Gang Tan. "Modular control-flow integrity". In: *PLDI*. 2014, pp. 577–587.

[44] Ben Niu and Gang Tan. "Per-input control-flow integrity". In: *ACM CCS*. 2015, pp. 914–926.

[45] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (Nov. 1996). URL: http://www.phrack.com/issues.html?issue=49&id=14.

[46] PaXTeam. *pax.grsecurity.net/docs/aslr.txt*. https://pax.grsecurity.net/docs/aslr.txt. Accessed 2025-12-02. 2003.

[47] Android Open Source Project. *Control Flow Integrity | Android Open Source Project*. https://source.android.com/devices/tech/debug/cfi. Accessed 2025-12-02. 2020.

[48] Android Open Source Project. *ShadowCallStack - Android Open Source Project*. https://source.android.com/docs/security/test/shadow-call-stack. Accessed 2025-12-02. 2022.

[49] The Chromium Projects. *Control Flow Integrity*. https://www.chromium.org/developers/testing/control-flow-integrity/. Accessed 2025-12-02. Nov. 2023.

[50] rcvalle. *LLVM Control Flow Integrity (CFI) Support for Rust Design Doc*. https://rcvalle.com/docs/rust-cfi-design-doc.pdf. Accessed 2025-12-05. Oct. 2021.

[51] rcvalle. *Tracking Issue for LLVM Control Flow Integrity (CFI) Support for Rust #89653*. https://github.com/rust-lang/rust/issues/89653. Accessed 2025-12-05. Oct. 2021.

[52] Alex Rebert and Christoph Kern. "Secure by design: Google's perspective on memory safety". In: *Technical report, Google Security Engineering* (2024).

[53] Stephen Röttger. *Control Flow Integrity in V8*. https://v8.dev/blog/control-flow-integrity. Accessed 2025-12-02. Oct. 2023.

[54] secuirty.apple.com. *Towards the next generation of XNU memory safety: kalloc_type*. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/. Accessed 2025-12-11. Oct. 2022.

[55] Google Play services. *Factory Images for Nexus and Pixel Devices*. https://developers.google.com/android/images. Accessed 2025-12-02. 2022.

[56] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. 2007, pp. 552–561.

[57] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. *On the effectiveness of address-space randomization*. https://doi.org/10.1145/1030083.1030124. Accessed 2025-12-02. 2004.

[58] Jeff Vander Stoep and Stephen Hines. *Google Online Security Blog: Rust in the Android platform*. https://security.googleblog.com/2021/04/rust-in-android-platform.html. Accessed 2025-12-02. 2021.

[59] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal war in memory". In: *IEEE S&P*. 2013, pp. 48–62.

[60] GNU Compiler Collection team. *Vtable Verification Feature Proposal*. Accessed 2025-12-02. URL: https://docs.google.com/document/d/1EsIEkOUHMrTmgRnFNEe9ZupXPYq1wPliuFQB3GPJtzs/pub.

[61] GNU Compiler Collection team. *Vtable Verification User's Guide*. Accessed 2025-12-02. URL: https://docs.google.com/document/d/1wN-uygC0hicLe1dyAGCvtn_tJhnwFer0Nsy56b84doY/pub.

[62] The Clang Team. *Control Flow Integrity – Clang documentation*. https://clang.llvm.org/docs/ControlFlowIntegrity.html. Accessed 2025-12-02.

[63] The Clang Team. *Control Flow Integrity – Clang documentation (#fsanitize-kcfi)*. https://clang.llvm.org/docs/ControlFlowIntegrity.html#fsanitize-kcfi. Accessed 2025-12-02.

[64] The Clang Team. *Control Flow Integrity Design Documentation*. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. Accessed 2025-12-02.

[65] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Ulfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". In: *USENIX Security*. 2014, pp. 941–955.

[66] Jonathan Wakely. *vtv - GCC Wiki*. https://gcc.gnu.org/wiki/vtv. Accessed 2025-12-02. Nov. 2014.

[67] GCC Wiki. *vtv - GCC Wiki*. https://gcc.gnu.org/wiki/vtv. Accessed 2025-12-02.

[68] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. "CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software". In: *USENIX Security*. 2019, pp. 1805–1821.