Control-Flow Security

by

William Patrick Arthur

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2016

For Allison and Marshall.

Acknowledgments

I would like to first thank my advisor Professor Todd Austin. This work would not be possible without his support and efforts. Serving as a role model, he was *always* personally motivating and made the work not only possible but a joy.

I also thank my committee members for their insight and encouragement; Professors Scott Mahlke, Dennis Sylvester, Prabal Dutta, and Jason Mars. They have proven to be mentors who exemplify what is most admirable about academia and research. Additionally, I thank Reetuparna Das, a person of great importance to this work. Thank you for challenging me to strive for excellence.

This work was not completed without the contribution and assistance of others, most notably from my collaborators. My undergraduate collaborators Ben Mehne and Sahil Madeka have proven invaluable. Additionally, my peers in graduate studies have provided invaluable insight and I am ever thankful for their time listening, challenging, and debating my ideas and this work. This includes Salessawi Ferede Yitbarek, Joseph Greathouse, Jason Clemons, Ricardo Rodriguez, Zelalem Birhanu Aweke, and too many others to mention in the space here. Most importantly, I thank my friend, colleague, and collaborator Biruk Wendimagegn Mammo.

Lastly, I thank my family for their patience and support. They have provided the encouragement, affirmation, and ultimately the purpose in completing this work.

Table of Contents

List of Tables

List of Figures

Abstract

Computer security is a topic of paramount importance in computing today. Though enormous effort has been expended to reduce the software attack surface, vulnerabilities remain. In contemporary attacks, subverting the control-flow of an application is often the cornerstone to a successful attempt to compromise a system. This subversion, known as a control-flow attack, remains as an essential building block of many software exploits.

This dissertation proposes a multi-pronged approach to securing software control-flow to harden the software attack surface. The primary domain of this dissertation is the elimination of the basic mechanism in software enabling control-flow attacks. I address the prevalence of such attacks by going to the heart of the problem, removing all of the operations that inject runtime data into program control. This novel approach, Control-Data Isolation, provides protection by subtracting the root of the problem; indirect control-flow. Previous works have attempted to address control-flow attacks by layering additional complexity in an effort to shield software from attack. In this work, I take a subtractive approach; subtracting the primary cause of both contemporary and classic control-flow attacks. This novel approach to security advances the state of the art in control-flow security by ensuring the integrity of the programmer-intended control-flow graph of an application at runtime. Further, this dissertation provides methodologies to eliminate the barriers to adoption of control-data isolation while simultaneously moving ahead to reduce future attacks.

The secondary domain of this dissertation is technique which leverages the process by which software is engineered, tested, and executed to pinpoint the statements in software which are most likely to be exploited by an attacker, defined as the Dynamic Control Frontier. Rather than reacting to successful attacks by patching software, the approach in this dissertation will move ahead of the attacker and identify the susceptible code regions before they are compromised.

In total, this dissertation combines software and hardware design techniques to eliminate contemporary control-flow attacks. Further, it demonstrates the efficacy and viability of a subtractive approach to software security, eliminating the elements underlying security vulnerabilities.

Chapter 1
Introduction

There are no solutions, only trade-offs.

Thomas Sowell

The rise of the information age has resulted in an increasing dependence on computing platforms. We rely on computer hardware and software for our daily communication, commute, work, and entertainment, both past and present [9, 132, 96, 122, 69, 115, 45, 16]. The increasing dependence on computers has correlated positively with increasing interest in compromising the security of systems [34]. Many high-profile software exploits have begun to motivate an intense interest in secure computing [32, 17, 72].

At the heart of many attacks today are control-flow attacks, including buffer overflows, code reuse attacks, return-to-libc, code gadgets, and Linux rootkits, among others [76, 106, 91, 25, 108]. As such, many mitigation techniques have been proposed and implemented to address the rising tide of exploitation [80, 2, 21, 24, 127, 126, 142, 149, 40, 113]. Consequently, many attempts to measure the software attack surface have also been made [6, 139, 105, 88, 148, 5]. However, despite the knowledge of these exploits for decades [25], control-flow attacks persist.

The software attack surface has been the interest of both researchers and adversaries extending for decades [119], starting in the 1960's with US government policy to address the potential existence of foreign threats to information systems [146] and the 1970's [145] with access control software in the computing industry. The struggle for secure systems continues today, with computing security evolving into a significant industry with a cost of billions USD every year [53]. The efforts of both security researchers and attackers has evolved over time and as protective measures have been implemented, commensurate effort from adversaries has answered [28, 114, 29, 49, 63, 64, 65].

1.1 The Software Attack Surface

The software attack surface encapsulates all avenues in which an adversary can gain access to a system and potentially cause damage or adverse effects to that system [94]. A pervasive building block involved in exploiting the attack surface is the control-flow attack, which either alters the sequence of program instructions executed and/or newly injected instructions by an adversary. The end goal of a control-flow attack is arbitrary code execution, which has the most powerful effect of any security defect as an attacker has full control of the executing application. Further, code injection remains one of the most common types of software attack on the Internet [3], which still pervades despite years of research [117].

Security in computing systems has long been an add-on, integrated after the development of software and systems has been completed [116, 48, 57]. Further, security has been considered as a non-functional requirement in software engineering, to the detriment of secure systems [133]. The culture of the industry with respect to security has a profound impact on the software attack surface, and a culture of security has been assessed to have a non-trivial impact on the integration of security in computing [93, 125]. Ultimately, security can be greatly benefitted from a top-down, pervasive, approach ultimately reducing the software attack surface [102].

1.1.1 The Cycle of Exploitation of the Software Attack Surface

As alluded to in Section 1.1, a longstanding struggle has occurred between system designers and their adversaries. Developers spend significant time and effort engineering systems. Indeed, over 90% of the effort in software engineering occurs outside of the actual coding of the software [112]. Further, the majority of the effort expended in software engineering occurs after the deployment of the system to the customers and users. After deployment of software and systems, failures begin to accrue, from small to total in scope [35]. These failures occur due to many factors including defects in design and implementation. As we will highlight in this work in Chapter 4, failures in practice originate from the complex interactions of systems and components.

When a failure manifests in the system, it has the potential for discovery by developers. Once discovered, the software maintainers then analyze the failure to determine the root cause. Unfortunately, this analysis is done without the depth of the engineering effort expended in designing the system. Additionally, once a defect manifests in the system, software engineers are under greater pressure to implement and release a fix, in the form of a software patch. This pressure to mitigate the effects of the defect are increased in the event the defect has been disclosed before a patch can be distributed. To further complicate the issue, the procedures of the initial software engineering effort such as extensive code reviews, design documents, and test suite development are often not present for the analysis, coding, and testing of the software patch. The consequence of the rushed timing and limited resources results in software patches which are of lesser quality than the original application code. It is no secret that many bug fixes spawn new exploits [97, 70]. One of the most notable cases of this issue is the Heartbleed bug identified in *OpenSSL* [32, 141]. The results of automated bug fixes are similar, where such fixes are often unverified and remain unfixed [79]. Regardless of the source, software patches are developed with less engineering support and rigorous testing than the original code, resulting in the higher likelihood of injecting more defects.

Many bug fixes are documented as Common Vulnerability and Exposures (CVE) [41] in the National Vulnerabilities Database (NVD) [100]. Reviewing these records reveals patches are often implemented as simple conditional branches, which guard the defect condition, a mere symptom of the manifestation of the defect. The rapid deployment and limited engineering of updates and fixes results in overall greater vulnerability in the software attack surface.

The Heartbleed bug discovered in *OpenSSL* [32] serves as an example for the cycle of software update and subsequent exploitation. An open source Transport Layer Security (TLS) implementation, *OpenSSL* provides encrypted communication between a user's web browser and a web server. The software implements Secure Sockets Layer (SSL) to encrypt the data transmitted between hosts on an internet. SSL is often the technology used to facilitate *https*, in this case representing Hypertext Transfer Protocol over SSL. Encrypting data transmitted over a network mitigates, among others, man-in-the-middle attacks. These attacks are accomplished when an adversary poses as the intended recipient of communication for both participants.

Occasionally, during lulls in communication between hosts, the connection is verified as active using signaling between hosts, referred to as "keep-alive". A security defect was injected into the codebase due to an incorrectly implemented patch to the OpenSSL code. This defect gave an attacker the ability to arbitrarily read up to 64 KB of the victim's memory, which could potentially include the private encryption key of the victim server.

The code enabling the vulnerability is shown in Figure 1.1. This line of code is made vulnerable by the lack of bounds checking on arguments to the function `memcpy`. This defect was not in the original OpenSSL implementation, but was added to implement RFC (Request For Comments) 6520 of the TLS specification. A period of two years passed between the addition of the defect and the discovery by the security community. The degree

```
memcpy(bp, pl, payload);
```

Figure 1.1 Heartbleed Bug Vulnerable Code. The Heartbleed bug [32] was a significant security vulnerability in the *OpenSSL* implementation of Transport Layer Security (TLS) encrypting two-thirds of all *apache*-based web servers. The defect, enabled by this single line of code, arose from the lack of bounds checking on an argument to the `memcpy` function call and persisted for two years.

to which the defect was exploited is unknown, but the vulnerability was widespread. Almost two-thirds of *apache*-based web servers, comprising a near majority of top web sites, were found to be vulnerable [50].

The continuous competition between the security community and attackers is played out in an ongoing cycle of exploitation. Software is engineered, developed, and deployed. Upon release of the software, adversaries begin probing for vulnerabilities which can be exploited for the gain of the attacker. As defects are discovered and exploited, they are addressed by the software maintainers who then patch the software. The newly patched software is released, whereupon the attackers seek and find new open avenues in the software attack surface. This cycle continues as long as the software remains in use and of value to compromise for an adversary.

Attack	Mitigating Defense	Attack Evolution	Persistent Threat
Stack Smashing	Non-Executable Stacks	Heap Sprays	Arbitrary Code Execution via Control-Flow Attack
Heap Sprays	Non-Executable Heap	Return-Oriented Programming	
Return-Oriented Programming	Address Space Layout Randomization (ASLR)	Code Gadgets	

Table 1.1 Escalation of Attacks and Defenses As defenses have been created to mitigate arbitrary code execution, adversaries have evolved attacks. The attacks and defenses shown are but a small sample of defense measures and the attacks they have inspired. Throughout the history of attacks and defenses, arbitrary code execution (the ability of an attacker to select the code to execute rather than the programmer) has remained. In this work, we break the cycle of exploitation by eliminating the root of attacks and moving ahead of adversaries.

An excellent example of this cycle is the control-flow attack, detailed below in Section 1.1.4. In short, control-flow attacks derail the execution of code from the programmer-intended code to attacker selected code. Over time, there have been many defenses created against such attacks. Once these defenses are in place, they are subsequently broken or circumvented by agile attackers. This is summarized in Table 1.1, which depicts the escalation in attack and defense. A critical enabling factor of this escalation is that each defense is designed to break the specific, current avenue of an attack. They do not address the underlying defect and make assumptions about the ability of an attacker to use other means to exploit the same inherent vulnerability. Table 1.1 is not a taxonomy of all exploits and defense mechanisms, but instead serves to demonstrate the resourcefulness of adversaries to evolve attacks to compromise the same underlying vulnerability.

1.1.2 The Evolution and Commercialization of Malware

To further compound the difficult task of hardening the software attack surface, efforts of adversaries have increased dramatically over time. Defined as any malicious software including viruses, Trojans, worms, adware, and spyware, malware represents the efforts of attackers to gain control over or cause detrimental effects to a system. Originally, software exploits were largely the domain of individuals, e.g., the Melissa virus [54], and government organizations [120]. However, today software exploitation and malware have increasingly shifted to the domain of professional hackers, becoming a corporatized endeavor. This shift is of paramount importance, as such a trend dramatically increases the resources available to adversaries. Recent attacks such as the carbanak APT (Advanced Persistent Threat) [74] demonstrate the scope and impact of contemporary software exploitation. This expansive attack was perpetrated across more than 30 nations and involved more than 100 financial banking institutions. Carried out over a two-year period, this attack has been estimated to have extracted up to one billion USD from the victim institutions. The attack, attributed to a criminal gang, embodies the shift in software attacks to a venture of organized crime. *Crimeware* has become ubiquitous in contemporary computing, replacing the old model of amateur hackers [8].

A consequence of the commercialization of the exploitation of the software attack surface is the dramatic increase of efforts to compromise systems. This effort can be seen in the significant increase in the number of new, unique, types of malware which have been discovered on the Internet. Extending in analysis from 1984 until today, the number of types of malware which have been discovered has eclipsed 500 million, with over 390,000 new malicious programs registered every day [13]. The number of new malware discovered each year has increased year over year, with a dramatic increase each year beginning in 2007[13].

1.1.3 Breaking the Cycle of Exploitation

The sheer volume of this increase has profound impact on the approach of defenders in the effort to harden the software attack surface. To date, the approach of researchers in addressing the rising tide of exploitation [80, 2, 21, 24, 127, 126, 142, 149, 40, 113] could be classified as one of mitigation, developing mechanisms to shield the software attack surface from the avenues previously exploited by attackers. The result of these attempts at mitigating the effects of malware begets the cycle of exploitation detailed in Section 1.1.1. Given the unyielding and resource-intensive efforts of adversaries, there is no expectation that time alone will yield increasing success in mitigating the effects of malware. To this end, a pivotal approach is necessary to counter the rising tide of exploitation, and more importantly to break the cycle of exploitation.

To achieve this goal, this work details two novel approaches to hardening the software attack surface: eliminating the underlying vulnerabilities which enable exploits, and moving ahead of attackers by identifying the vulnerabilities before they are exploited. As the software attack surface represents all avenues for an adversary to compromise a system, this work focuses on one of the most essential building blocks for many software exploits today: control-flow attacks. These attacks have pervaded for years and are used in attacks such as stack smashing, heap sprays, return oriented programming, and code gadgeting, as shown in Table 1.1 [76, 106, 91, 25, 108]. These attacks not only constitute a major contribution to the software attack surface, they have managed to evolve over time, circumventing the mitigation approaches of previous works [80, 2, 21, 24, 127, 126, 142, 149, 40, 113]. This dissertation details the methods necessary to break the ongoing cycle of exploitation for control flow attacks in order to establish control-flow security.

1.1.4 Control-Flow Attacks

Control-flow attacks are the subversion of the program execution to attacker-selected code. The realization of a control-flow attacks is manifested via code injected as user data, or as the unintended sequence of execution of existing code. At a finer granularity, these attacks violate, at runtime, the control-flow graph (CFG) of an application. The control-flow graph of an application is the collection of basic blocks of code (defined as a single-entry, single-exit block of instructions) and the edges which connect these blocks. The CFG encapsulates the programmer-intended order of execution for program instructions.

Contemporary control-flow attacks result from malicious user data injected into the program counter (PC). This is achieved when a malicious user corrupts data, typically achieved through the exploitation of a defect in some part of the executing code. The canonical example is the stack smashing attack. This attack leverages a buffer overflow (or buffer overrun), an element of exploitation first identified over four decades ago [25] and still pervasive today [36]. Buffer overflows occur when the boundary of a buffer is exceeded when writing data to the buffer. This overflow causes data in memory following the buffer boundary to be overwritten, corrupting other program data. Stack smashing occurs when a data buffer, which is a local variable contained on the program stack, is overrun. The result is a stack buffer overflow, and the consequences of the overflow is the overwriting of other data items on the stack with the overflow of data exceeding the buffer capacity. The intended consequence of the stack buffer overflow is the corruption of the return address on the program stack. If the overflow is crafted correctly, the data provided to the buffer will be used as the return address when the compromised function executes a `return` statement. Thus, rather than returning control to the calling function, the attacker has now selected the next instruction to be executed. Originally, the malicious data taken as the new return address would point directly to the overwritten buffer, which would be malicious code instead of the intended data though that is less common today as program stacks are generally made to be non-executable via memory protection. Stack smashing attacks have been focus of significant work, but remain a security concern today [135].

The cost of such attacks is notoriously difficult to estimate. However, recent reports indicate that, worldwide, corporations and individuals incurred $500 Billion USD cost in remediation of exploits arising from malware contained in pirated software [53].

1.2 Contemporary Control-Flow Attacks Rely on Mixing of Control with Data

The software attack surface constitutes a substantial threat to computer security. Software vulnerabilities facilitate a wide array of security exploits: buffer overflows, heap spray attacks, return-to-libc, integer underflow, code gadgets, and a host of others. In the commercialization of the malware industry [53], new and more serious threats have emerged such as *Crimeware*, which perpetrate identity theft for the purpose of monetary gain [75]. Control-flow attacks, which permit arbitrary code execution, have emerged as a primary means to exploit software.

This work drives to the heart of pervasive control-flow attacks by directly attacking the root of the problem: user-data derived control-flow. Contemporary research to protect control-flow has been focused on verifying the user data to be injected into the program counter [2, 21, 66, 80, 127, 142, 150, 149] in an effort to establish trusted user data for control-flow targets. Prior works approach control-flow security by layering additional complexity on top of user data in an effort to shield the vulnerability from attack. In this work we adopt a subtractive approach by removing the actual vulnerability. *We simply do not trust any user data, and instead remove all avenues for such data to be injected into the* *program counter.*

The mechanisms developed in this dissertation put forward the ideal of a *subtractive approach* to addressing control-flow security. As contemporary control-flow attacks are dependent on the pollution of control structure with user data, this work provides solutions and detailed analysis of eliminating the basic, essential building blocks of such attacks. These building blocks are the programming structures and mechanisms which allow user data to reach the program counter of a processor. Thus, the first accomplishment of this work is to eliminate the avenues which allow the corruption of control with data. However, in the face of removing ubiquitous building blocks of software the efficiency of a software-only solution remains a challenging task.

1.3 Hardware-Software Co-Design Can Greatly Enhance Control-Data Isolation

The effort of attackers to exploit the software attack surface continues to grow in the malware arms race [49]. Indeed, most attacks continue to be conducted within the application layer [55]. In this realm of exploitation, arbitrary code execution, generally achieved by control-flow attacks, pervade as a primary means to attack software. Control-data isolation (CDI), detailed within this dissertation in Chapter 2, eliminates contemporary control-flow attacks. However, barriers remain to the practical, wide-spread adoption of control-data isolated.

A software only solution, control-data isolation [12] implemented in Chapter 2, retains greater than desired runtime overheads for some applications. In the face of this, a more efficient implementation is desired. To that end this dissertation addresses the non-trivial performance impacts of CDI programs by introducing hardware that virtually eliminates the performance penalty associated with ensuring secure control flow at runtime. We demonstrate in Chapter 3 that through high-accuracy memoization of programmer-intended, compiler-selected indirect control transitions, we can nearly eliminate the runtime costs of eliminating contemporary control-flow attacks. In addition, we demonstrate that the structures we add to speed up CDI execution can easily perform the double duty of multi-way control point prediction, thereby providing even opportunities to speed up CDI program execution, compared to targets with simple indirect control predictors (e.g., BTB).

Additionally, architectural additions to computer hardware can increase the security guarantees of techniques like CDI. As contemporary control-flow attacks can be eliminated by the work in this dissertation, attackers will seek another avenue to gain control over systems. This never-ending cycle of exploitation is the hallmark of computer security to date. The cycle begins with the release of a piece of software. Malicious entities then compromise the software to exploit for their own gain. The developers of the software then analyze the attack, evaluate the vulnerability, and make changes to patch the system against the specific attack. This software is then released, where the cycle continues. A prominent feature of this dissertation is the inclusion of techniques to break the cycle of exploitation, getting ahead of the attackers before they strike.

To this end, control-data isolation is extended to protection of paths, addressing impossible paths of execution through the code. Analysis of paths of execution for security purposes has been the effort of research as well [80, 129, 4]. However, contemporary control-flow attacks, such as code gadgets and return-oriented programming, rely on a single compromised indirect control-flow instruction to derail execution to begin the attack. Eliminating the vulnerability of all edges will drive attackers to seek paths of exploitation.

1.4 Latent Software Defects are Control-Flow Path Dependent

The vast majority of security attacks are enabled by software bugs. Defects which escape detection of software quality assurance can have global impact, such as the Code Red and Sapphire/Slammer worms which utilized buffer overflows for system exploitation. Fueled by these and other high-profile exploits, buffer overflows remain a top security concern [101, 36]. Programs written in popular languages such as C and C++ are a rich source of buffer overflow bugs, as these languages cannot, without high overhead, systematically eliminate buffer overflow vulnerabilities [42]. This then places the burden on methods such as control-data isolation and software testing to find potential buffer overflow vulnerabilities before they are exploited and disallow their effects of derailing control flow.

As control-data isolation eliminates contemporary control-flow attacks, path-based control flow analysis and security becomes all the more pressing in importance. Though simple hardware additions can eliminate many impossible paths of execution, they are not a panacea to security. Thus, parallel effort into eliminating the software bugs which lead to exploits remains crucial. Understanding the way in which latent defects are exploited can reveal critical insight into their prevention.

The majority of security-related faults in software reside in the least likely to be executed code sequences, and by extension, the least tested portions of code [81]. Attackers use this information to reduce the effort in finding vulnerabilities. A malicious user will provide permutations of typical application inputs in an effort to cause slight (but expected) deviations from the well-travelled, and thus well-tested, path of normal execution. This exploit-rich code exists just beyond the well-trodden execution paths of testers and users, yet is readily reachable by attackers.

To break the cycle of exploitation, it is necessary to find defects before attackers to. The dynamic control frontier (DCF) [11] is a collection of paths rooted in dynamically executed paths. However, these paths are special in that, had the final control decision in these paths executed a different basic block, it would create a new, never-before-seen path. This defines the frontier of the path space executed by an application with respect to a set of inputs. Collectively, the DCF represents the most readily accessible paths of execution which are unlikely to be executed by end-users; consequently, these paths have a high degree of reachability for an attacker. Accordingly, any latent defects in the unexecuted portions of the dynamic control frontier paths are unlikely to be found by users and developers, but these bugs can be quickly uncovered by attackers.

It is interesting to look at the dynamic control frontier of an application arising from the test inputs of developers. Indeed, we show that this is valuable as we find real vulnerabilities at these locations. However, it is more intriguing to examine the dynamic control frontier for a non-trivial sized population of end users. An attacker is most interested in this frontier as it represents code paths which have not been tested nor executed with any frequency by any user of a particular program. In contrast, any paths frequently executed by users which are not represented in the test suites will probably be devoid of showstopper bugs, as users would otherwise complain. As such, in the construction of a system to profile the DCF, we must be mindful that such a system should analyze the DCF of a large population of users without imposing an unacceptable impact on individual user performance.

1.5 Contributions of this Work

In this dissertation, I detail a diversity of solutions to ensuring control-flow security. This includes methodologies for run time enforcement of the programmer-intended CFG of an application, and the discovery of software defects leading to exploits.

Software-Based Control-Flow Security. The first contribution to control-flow security in this dissertation is a novel technique for the dynamic enforcement of the programmer-intended control-flow graph of an application. This solution appears in Chapter 3 of this work. A fundamental building block of software exploitation today is the successful completion of a control-flow attack. These attacks violate, at runtime, the programmer specified CFG of an application. The flow of execution is then diverted from that which the programmer intended to attacker selected code. Contemporary control-flow attacks are propagated by an attacker injecting malicious runtime data into the program counter of an application. In Chapter 2, I present control-data isolation, a novel approach to enforce the CFG of dynamically executing software. This work directly addresses the root of the problem in contemporary control-flow attacks; indirect control flow. By eliminating the use of indirect control-flow instructions, the link between malicious runtime data and the program counter is severed. Specifically, in Chapter 2 this work makes the following contributions:

- Chapter 2 presents an effective, efficient, and scalable approach to enforcing the CFG of an application at runtime. We implement control-data isolation (CDI) as a compilation-based transformation to existing software applications and library code. This work advances the state-of-the-art in control-flow attack protection by targeting and eliminating the root cause: the injection of user data into the program counter.
- Presentation of an *llvm*-based compiler implementation that generates control-data isolated code for non-trivial programs and shared libraries, eliminating the use of indirect control flow in compiled programs.

Hardware-Based Control-Flow Security. Chapter 3 builds on the first contribution in control-data isolation, resulting in a hardware-software co-design which accelerates the validation of indirect control-flow edges at runtime. As the use of indirection in the program counter is pervasive in computing, eliminating its use can incur runtime overheads. This work details how simple hardware extensions to modern processors can eliminate nearly all overheads associated with the control-flow security methodology detailed in Chapter 2. Specifically, Chapter 3 of this work makes the following contributions:

- Chapter 3 shows how simple hardware additions for CDI support can guard a program from control-flow attacks.
- Demonstration that edge caching eliminates nearly all of the slowdowns associated with the execution of indirect jump validation sleds, the hallmark of CDI protection.
- Exploration of the use of the edge cache as an indirect branch predictor enables speedups for select CDI-compliant programs compared to architectures with simple

BTB-based indirection prediction.

- Development of an optimization technique which extends control-flow security to paths of execution. This optimization, leveraging the non-speculative Return Address Stack (RAS), addresses the future security concerns of path-based control-flow attacks.
- Further hardware optimization to extend Control-Data Isolation to address execution of path-based control-flow attacks. As CDI eliminates contemporary control-flow attacks by securing the CFG of software at runtime, the future of attacks may exploit control flow which adheres to the programmer-defined CFG. The final contribution of Chapter 3 directly addresses future threats by including a new hardware structure, the non-speculative Return Address Stack.

Software-Based Control-Flow Path Analysis. Leveraging the knowledge of how software is engineered, tested, and executed by end users, this dissertation details a novel approach to control-flow security for software. This approach changes the relationship between attackers, users, and developers. The work in Chapter 4 details the change of exploit mitigation from a cycle of reactive solutions, to a proactive approach. The current industry approach to software exploitation is reactionary: when attackers successfully compromise a software system, the exploit is evaluated and a mitigating fix is developed and deployed. Specifically, in Chapter 4 this work makes the following contributions:

- Chapter 4 presents the dynamic control frontier. This frontier defines the line of demarcation between heavily tested paths of execution and those which are untested, both by developers and end users. This frontier comprises a bridge between current industry standard of software testing such as code and branch coverage, and the next level of software testing in path-based testing coverage metrics. As it is currently infeasible to achieve complete testing coverage of all possible paths of execution in non-trivial software applications, the dynamic control frontier specifies the paths where software defects which lead to exploits are likely to hide.
- Presentation of an effective, scalable, and decentralized approach to identifying the dynamic control frontier for a program running across a large population of users.
- Presentation of a software implementation for harvesting dynamic control frontier

information from individual user machines. The approach utilizes dynamic code instrumentation to limit the impact to application execution while providing appropriate coverage of the dynamic control frontier in the aggregation of users.

- Demonstration of the value of the dynamic control frontier by showing that many known security vulnerabilities may be found there. We show that dynamic control frontier paths sensitize known exploits identified by the NIST National Vulnerabilities Database.
- Evaluation of the effectiveness of the approach by exploring the performancecost tradeoffs while harvesting DCF paths. We also developed a novel whole-path analysis technique that allows us to gauge the coverage of the approach (i.e., the total percent of dynamic control frontier paths found as a function of total population run time). We present results for a wide range of non-trivial software packages that show our approach achieves good coverage while keeping performance impacts low.

Lastly, Chapter 5 will offer conclusions on the insights of this work, along with future directions for research in control-flow security.

Summary. In summary, this dissertation targets three pillars of control-flow security, as shown in Figure 1.2. This dissertation directly addresses the longstanding vulnerabilities of control-flow security through exploit analysis and the adoption of a subtractive solution, removing the intrinsic mechanisms which enable control-flow exploits. This is accomplished through a multi-pronged approach to the pervasive use of control-flow attacks in compromising the software attack surface. First, the role control-flow plays in exploits is analyzed. This analysis then informs the approach of directly addressing the root of contemporary attacks, indirect control-flow. Rather than the historical approach of mitigation, this work subtracts the root mechanism enabling the vulnerability. This work demonstrates the feasibility, efficacy, and efficiency of the elimination of indirect control-flow in software. Subsequently, this work demonstrates how to eliminate the practical barriers to adoption of the techniques proposed through hardware-software co-design. Finally, this dissertation addresses the future of control-flow attacks after it eliminates their contemporary counterparts through path-based control-data isolation.

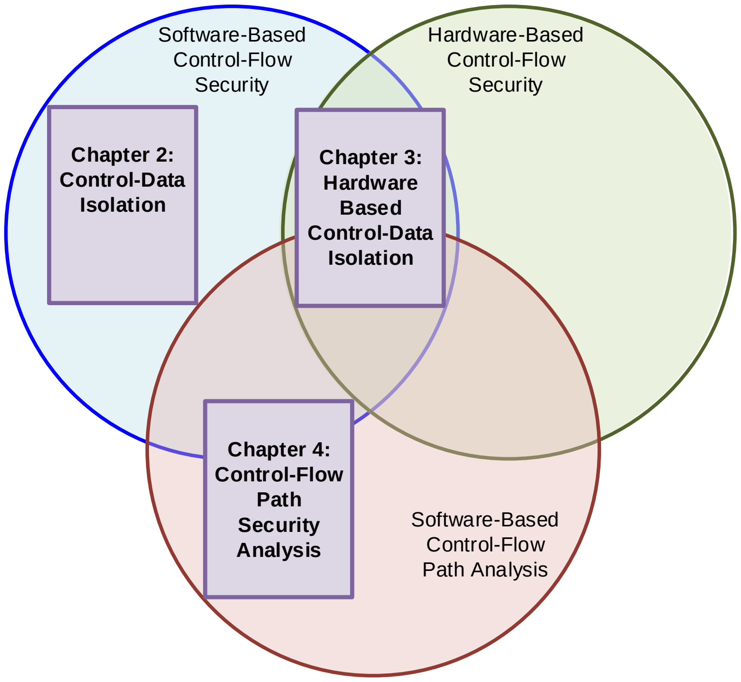

Software-Based Control-Flow Security
Hardware-Based Control-Flow Security
Chapter 2: Control-Data Isolation
Chapter 3: Hardware Based Control-Data Isolation
Chapter 4: Control-Flow Path Security Analysis
Software-Based Control-Flow Path Analysis

Figure 1.2 Elements of This Dissertation. This dissertation targets three pillars of control-flow security. Chapter 2 targets software solutions to eliminate the root vulnerability of contemporary control-flow attacks, through the introduction of control-data isolation (CDI). Chapter 3 develops efficient hardware solutions to eliminate any remaining barriers to adoption of CDI, including significant improvements in performance and security. Chapter 3 also details solutions to extend CDI protection from ensuring control-flow edges to securing paths of execution. Chapter 4 details execution path analysis to pinpoint where latent defects hide in software. This path analysis provides critical insight to heavyweight software defect analysis, narrowing the state space for comprehensive software testing.

Chapter 2
Control-Data Isolation

People who pride themselves on their “complexity” and deride others for being “simplistic” should realize that the truth is often not very complicated. What gets complex is evading the truth.

Barbarians inside the Gates and Other Controversial Essays
Thomas Sowell

Computer security has become a central focus in the information age. Though enormous effort has been expended on ensuring secure computation, software exploitation remains a serious threat. The software attack surface provides many avenues for hijacking; however, most exploits ultimately rely on the successful execution of a control-flow attack. This pervasive diversion of control flow is made possible by the pollution of control flow structure with attacker-injected runtime data.

Many control-flow attacks persist because the root of the problem remains: runtime data is allowed to enter the program counter. In this paper, we propose a novel approach: Control-Data Isolation. Our approach provides protection by going to the root of the problem and removing all of the operations that inject runtime data into program control. While previous work relies on CFG edge checking and labeling, these techniques remain vulnerable to attacks such as heap spray, read, or GOT attacks and in some cases suffer high overheads. Rather than addressing control-flow attacks by layering additional complexity, our work takes a subtractive approach; subtracting the primary cause of contemporary control-flow attacks. We demonstrate that control-data isolation can assure the integrity of the programmer’s CFG at runtime, while incurring average performance overheads of less than 7% for a wide range of benchmarks.

2.1 Introduction

The software attack surface constitutes a substantial threat to computer security. Software vulnerabilities facilitate a wide array of security exploits: buffer overflows, heap spray attacks, return-to-libc, integer underflow, code gadgets, and a host of others. Today, the risk of software exploitation has escalated beyond DDOS attacks and amateur attacks such as the *Melissa* virus [54]. In the commercialization of the malware industry, new and more serious threats have emerged such as *Crimeware*, which perpetrate identity theft for the purpose of monetary gain [75]. As most attacks are conducted within the application layer [56]. Control-flow attacks, which permit arbitrary code execution, have emerged as a primary means to exploit software.

Our work drives to the heart of pervasive control-flow attacks by directly attacking the root of the problem: user-data derived control-flow. Contemporary research to protect control-flow has been focused on verifying the user data to be injected into the program counter (PC) [2, 21, 68, 80, 127, 142, 150, 149] in an effort to establish trusted user data for control-flow targets. These previous works approach control-flow security by layering additional complexity on top of user data in an effort to shield the vulnerability from attack. In this work we adopt a subtractive approach by removing the actual vulnerability. *We simply do not trust any user data, and instead remove all avenues for such data to be injected into the program counter.*

2.1.1 Control-Flow Attacks

Control-flow attacks implement the redirection of program execution to attacker-selected code, either injected as user data or existing code in the form of code gadgets. These attacks violate, at runtime, the control flow graph (CFG) of an application by corrupting the PC with user-injected data, thereby allowing a program to execute a control edge not defined by the programmer.

As introduced in Chapter 1, Control-flow attacks exploit an inherent weakness ubiquitous in software development: determination of control-flow target addresses at runtime. It is the enmeshed relationship between the Program Counter and runtime data which creates the fundamental weakness of software to control-flow attacks. The classic example of such an attack is the stack buffer overflow. When input to a buffer exceeds the pre-allocated size on the program stack, the return address in the stack frame may be overwritten. In this case, the user data is used as the target of a return instruction, which can then jump to malicious code including the input buffer on the stack.

As a critical element of software exploitation, considerable effort has been expended to address control-flow attacks. Countermeasures such as stack protection, Address Space Layout Randomization (ASLR), and Non-Executable Data (NXD) have been widely adopted. Though many counter-measures have been devised [2, 21, 24, 127, 126, 142, 149], control-flow attacks remain a pervasive threat to computer security [101] due to the persistence of mixing runtime data with program control. Recently, mitigating techniques such as Control Flow Integrity (CFI) [2] and its descendants [150, 149], Program Shepherding [80], and taint analysis [67] have been proposed. These techniques, which propose increased security through verification of runtime data, retain several vulnerabilities. Some are susceptible to CFG forgery attacks or allow the PC to target the middle of a basic block (or even the middle of an instruction). They also place constraints on their threat models that weaken their protections, such as the requirement of non-executable data or the assumption that an attacker cannot read or infer the contents of data memory. Additionally, most works do not address call-graph based control flow (i.e., dynamic library calls and returns). In this work, we relax the constraints of previous work, by assuming that the attacker has free reign over all of data memory (read, write, and execute), while also addressing the important issues of call-graph protection and dynamically introduced code such as shared libraries. The limitations of previous works are discussed further in Section 2.8 and Table 2.2.

2.1.2 Control-Data Isolation

Previous works attempt to mitigate control-flow attacks through *verification* of the runtime data which enters the program counter. Though this additional layer infers increased security, it nevertheless leaves the original, fundamental vulnerability: user data is injected directly into the PC. By contrast, this work eliminates arbitrary control flow by *eliminating the connection that exists between the PC and user data*, a technique which we call Control-Data Isolation (CDI). By disallowing the use of runtime data as control-flow targets, the programmer can ensure that all executions adhere to their specified control-flow graph (CFG).

In this paper, we implement CDI by generating code without the use of return and indirect jump/call instructions, the two types of instructions in modern architectures that connect user data and the PC. This creates some challenges in creating arbitrary code, in particular for calls/returns, indirect function calls, and shared libraries, but we show in Sections 2.2 and 2.3 how to implement (and subsequently optimize) these code sequences without the use of indirect control-flow instructions. The programs we create *completely sever the link between the PC and user data, and if the entire system adheres to the principles* *of control-data isolation, all control changes are limited to valid CFG edges, eliminating the way attackers execute control-flow attacks today.*

2.1.3 Contributions of this Chapter

Primary accomplishment of this chapter is the identification the common thread of software exploitation and the direct elimination of the root cause: the direct injection of user data into the control structure of software. The majority of this chapter is derived from the published work "Getting in Control of Your Control-Flow with Control-Data Isolation" [12]. This chapter makes the following contributions:

- We present an effective, efficient, and scalable approach to enforcing the CFG of an application at runtime. We implement control-data isolation (CDI) as a compilation-based transformation to existing software applications and library code. We advance the state-of-the-art in control-flow attack protection by targeting and eliminating the root cause: the injection of user data into the program counter.
- We present an *llvm*-based compiler implementation that generates control-data isolated code for non-trivial programs and shared libraries, eliminating the use of indirect control flow in compiled programs.
- We analyze a diverse set of programs and design and evaluate targeted, profile-guided optimizations to improve the performance of control-data isolated code.
- We evaluate the efficiency of CDI, showing through detailed experiments that the performance and storage costs are minimal, less than many of the previously proposed control-flow attack mitigation techniques.

This chapter, Chapter 2, encapsulates the following work. Section 2.2 provides an in-depth analysis of CDI. Section 2.3 details our implementation approach of eliminating all indirect control flow, while Section 2.4 addresses dynamic code from shared libraries. Section 2.6 provides detailed analysis of our *llvm* compiler-based implementation, PitBull. Experiments testing our method and a full analysis of results are delivered in Section 2.7. Finally, Section 2.8 evaluates related works, and Section 2.9 highlights chapter conclusions.

2.2 Protecting Control Flow with Control-Data Isolation

Control-flow attacks work by injecting malicious runtime data into the program counter of a susceptible target process. They are a divergence from the programmer-defined CFG of an application, occurring when an attacker creates new control-flow edges from user data at runtime. This can take many forms such as return-oriented programming, heap spray attacks, stack smashing, and even hijacking calls to library functions.

2.2.1 Threat and Trust Model

The goal of a control-flow attack is to subvert the control flow of a vulnerable process and execute code of the attacker's choosing. In this work, we consider the attacker to play a powerful role. An adversary is assumed to possess arbitrary read, write, and execute privilege to data memory, including the stack and heap. That is, we start from the position that an attacker controls all of data memory. In traditional compilation techniques, many control-flow target addresses are derived from or stored in data memory; hence, once an attacker gains some level of read/write/execute control over data memory, there are typically many avenues to direct program flow to code of their choosing. This is precisely how control flow attacks are currently accomplished.

We do make the assumption that the attacker cannot arbitrarily overwrite executing code segments at runtime. We see this assumption of non-writable code (NWC) as a fundamental element of security. Without this one protection, the attacker could simply substitute their own code for that of the application, obviating the need for control-flow attacks. Similarly, the program loader is trusted, as a compromised loader could simply replace system code with malicious code at load time. It is important to note, however, that the loader can be protected against attacks with CDI, in the same way as other applications.

An important aspect of our relaxed threat model is the assumption that data segments, specifically the heap, may contain executable code. As long as the non-writable code requirement is met, an application may execute code in the heap with full CDI protections. Previous works including all works based on CFI [2], expressly forbid the execution of code on the heap. This requirement is due to their susceptibility of forgery attacks. As they rely on labels placed at target locations, a heap spray attack could create forged labels which fraudulently identify malicious code as acceptable targets for an indirect call or jump. Our work is not susceptible to this attack, as all targets are embedded into the existing programmer-specified and loader-blessed instructions, eliminating the need to trust destination labels.

The key element of both our threat model and CDI principle is that user data expressly cannot be trusted. An important distinction between CDI and previous works such as CFI [2], and its descendants [24, 150], is their use of a shadow stack [111] to secure all `return` instructions. As the shadow stack is resident in data memory, it is inherently susceptible to attack and requires additional protection measures, increasing the potential attack surface. CDI provides the same protection against control-flow attack for all indirect instructions, obviating the need to trust or shield user data.

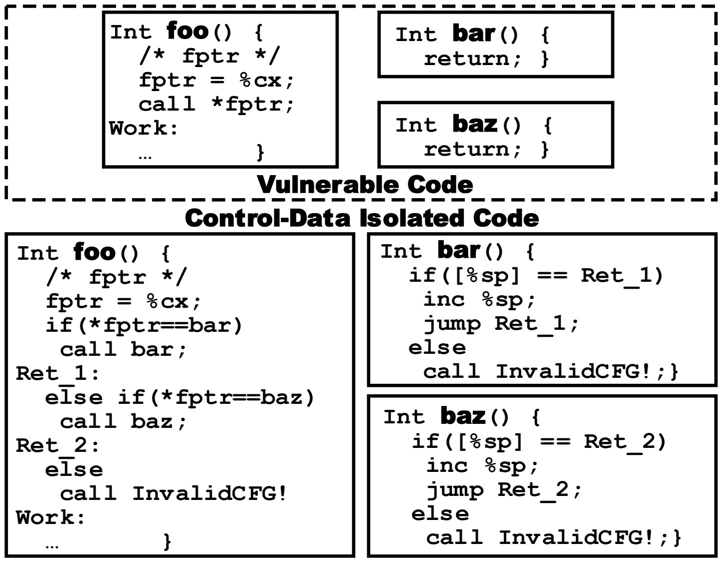
2.2.2 CDI Threat Protection

The implementation of CDI eliminates the possibility of any runtime data being used as a control-flow target address. In this work we accomplish this goal by disallowing the execution of indirect control-flow instructions. Simply put, an indirect jump, call, or return will never be executed. This eliminates the critical element pervasive to control-flow attacks. Without these instructions, stack smashing, heap spray, buffer-overflows, return-to-GOT, and return-to-libc attacks are crippled, as they all rely on the ability to derail the control-flow of a process, currently achieved by polluting the data value of indirect control-flow targets. Further, the availability of useful code gadgets and any remaining control-flow attacks are diminished to legal traversals of the program's CFG, since it is not possible to jump to the middle of a basic block (or instruction). By addressing, and removing, the root of the problem we can significantly reduce the software attack surface by limiting control to the programmer-specified CFG. The extent of the protection is determined by the degree to which the code running on the machine adheres to CDI principles. If all code running utilizes CDI, then user-injected data cannot find its way into the PC, and the system is hardened against control-flow attacks. To facilitate this ultimate goal, we focus on CDI-based compilation for applications, libraries, and dynamically introduced code objects, such as shared libraries.

In our relaxed threat model, we enable code to be executed in data space. This supports the use of a prevalent technology previous works have not: just-in-time compilation (JIT) and dynamically-generated code. JITted code presents challenges to CDI implementation, such as jump tables for loop unrolling. However, problems analogous to this have already been addressed by our work for similar structures such as the global offset table (GOT). In essence, CDI compliant code does not inhibit just-in-time compilation. The process of integration for dynamically-compiled code is found in Section 2.5.

2.2.3 Achieving Higher Levels of Protection by Isolating Control and Data

The threat model defined above creates many opportunities for ambitious attackers to achieve arbitrary code execution. Some instructions, namely indirect control flow instructions, derive control flow, in whole or part, on runtime data. When an attacker gains some level of control over data memory, this runtime data can be manipulated in a malicious manner, permitting an attacker to use (and abuse) the programmer's indirect jumps at will. This can be observed in attacks such as code gadgets, heap sprays, and buffer overflows. These attacks must, at some point, rely on a control-flow target derived from user data which may be injected by an attacker.



```
Int foo() {
  /* fptr */
  fptr = %cx;
  call *fptr;
Work:
  ...       }
```

```
Int bar() {
  return; }
```

```
Int baz() {
  return; }
```

```
Int foo() {
  /* fptr */
  fptr = %cx;
  if(*fptr==bar)
   call bar;
Ret_1:
  else if(*fptr==baz)
   call baz;
Ret_2:
  else
   call InvalidCFG!
Work:
  ...       }
```

```
Int bar() {
  if([%sp] == Ret_1)
   inc %sp;
   jump Ret_1;
  else
   call InvalidCFG!;}
```

```
Int baz() {
  if([%sp] == Ret_2)
   inc %sp;
   jump Ret_2;
  else
   call InvalidCFG!;}
```

Figure 2.1 CDI Control Flow Protection. Indirect branches are converted to direct conditional branches, severing the link between potentially malicious runtime data and the program counter.

To assure that the program's execution adheres to the CFG defined by the programmer, we isolate control-flow from runtime data. To achieve this end, we focus on all control flow decisions at runtime including those which are encapsulated in the executable code objects and control transfers in between. Thus with CDI, all valid edges in the CFG of an application are encoded in the programmer-specified and loader-blessed instructions of an application. This CFG functions as the golden model which completely defines the valid control flow of an application. That is to say, any dynamic paths which adhere to the CFG are potentially secure, but any paths which violate the CFG are explicitly insecure. By embedding all control-flow targets within programmer-written instructions, rather than derived from user data, we eliminate the weakness in software which enables control-flow attacks.

Figure 2.1 depicts a simple code sequence vulnerable to control-flow attacks, and an equivalent code sequence constructed with CDI that is protected from control-flow attacks (the full details of this process are discussed in Section 2.3). *To prevent exploitation of indirect control flow instructions, we simply remove them from software.* The indirect branches are replaced by direct branches, which only allow predetermined know-valid edges. The permissible targets of these instructions, i.e., `Ret_1`, `Ret_2`, `bar`, and `baz`, are identified via CFG discovery.

The control-data isolated code has no avenue for potentially malicious runtime data to be injected into the PC. As such, all target addresses of control-flow come from the programmer-specified text segment of an application. By eliminating the use of indirect instructions, attacks such as Stack Smashing become impossible to implement directly on the programmer's CDI-protected code. Similarly, attacks such as Heap Spray attacks rely on the execution of a control-flow instruction which derives its target from data memory. Even rootkits, where 96% of Linux rootkits integrate control-flow attacks [108], rely on subverting data which is injected into the program counter. Additionally, return-oriented programming (ROP) attacks, including those without any function calls, are defeated as these attacks rely on an initial derailment of the control-flow from the CFG by user data injected into the PC.

Implementing CDI requires validation of all control targets, which in turn requires complete knowledge of the CFG. Indirect control flow instructions such as function pointers make control flow graph discovery a challenge. In spite of this, previous works have demonstrated that the task of CFG discovery is achievable [2, 24, 134, 144, 150, 149]. Our CFG discovery approach is addressed in-depth in Section 2.3. Another key challenge, often overlooked by previous works, is control flow transfer between dynamically-linked objects such as shared libraries. Our work solves this issue, as detailed in Section 2.4.

Indirect control flow is an intrinsic part of modern software, so its removal has the potential to adversely impact the performance of programs. We address this concern by leveraging profile-guided code generation to efficiently select validated targets, which is detailed in Section 2.3. We develop an efficient, effective CDI software implementation which assures the runtime integrity of a program's CFG, demonstrated in Section 2.6. At first glance, it may appear that eliminating indirect control flow will inherently result in program slowdowns. However, previous research into *devirtualization* demonstrates that such a process is utilized to facilitate program speedups [15, 73, 78]. Devirtualization is the process by which dynamic virtual function calls are replaced with object test and direct calls, similar to the process depicted for `fptr` in Figure 2.1. By leveraging superior branch pre-

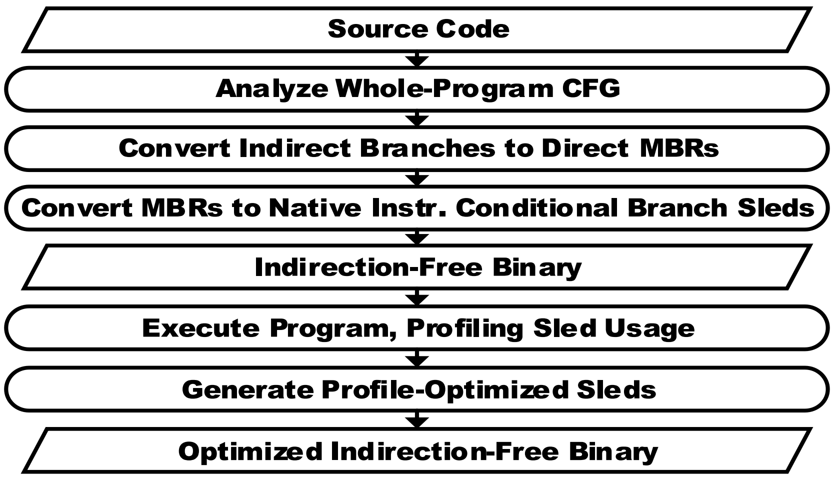

Source Code
Analyze Whole-Program CFG
Convert Indirect Branches to Direct MBRs
Convert MBRs to Native Instr. Conditional Branch Sleds
Indirection-Free Binary
Execute Program, Profiling Sled Usage
Generate Profile-Optimized Sleds
Optimized Indirection-Free Binary

Figure 2.2 CDI Compilation Flow. CDI-protected, indirection-free code is generated from application source code. This process converts indirect control-flow to direct branching, which is then profiled to optimize runtime performance of CDI-hardened code.

diction, devirtualization has been proven to improve execution speed in the object-oriented languages to which it has been applied.

2.3 Control-Data Isolation via Elimination of Indirect Control Flow

The work of creating software free of indirect control flow can be accomplished at varying stages in software development. In this work we propose a combination of a compile-time and load-time solutions that eliminate the use of indirect instructions in binaries. To achieve this, we must discover the CFG of an application and from it identify the indirect branching instructions and their control-flow targets. This information is used in eliminating indirection by substituting hard-coded, direct control flow into the target application. We also implement and identify several optimizations to apply when creating applications free of indirect control-flow.

An overview of this approach is shown in Figure 2.2. The CDI process begins by discovering the CFG of an application, and subsequently identifying all indirect control flow instructions, i.e., returns and indirect jumps and calls. These are then converted to multi-way branches (MBRs) and a complete target set for each MBR is then identified. A ***sled*** of conditional branch/direct jump pairs, one for each target, is substituted for each MBR. The sled does the work of converting indirect jumps to direct ones, by comparing the proposed target one-by-one with all of the validated potential targets of the indirect jump. An example of a sled is depicted in Figure 2.1 by the instructions:

```
if(*fptr==bar) call bar;
else if(*fptr==baz) call baz;
```

which are substituted for the vulnerable indirect call. When a matching target is found, a direct jump is made to the validated target; otherwise, an invalid control-flow decision is declared. The resulting code is dynamically profiled and optimized for performance. This process is studied in further detail in Section 2.3.3.

2.3.1 CFG Discovery

To enforce the golden-model CFG at runtime, a complete CFG which encapsulates all possible paths through a program for non-trivial software applications must be determined. Lifting binary code to determine the CFG of an application is both an active and well researched topic [7, 19, 18, 134]. However, such a task is made difficult specifically due to indirect control flow. Previous works such as CFI have been able to determine the precise CFG from binary analysis, while in this work we obtain such information from our *llvm*-based compilation flow.

We shall only consider indirect control flow instructions for CFG analysis, as direct control flow instructions are both trivial for building a CFG and they are not subject to code injection attacks (given the non-writable code assumption of our threat model). The key issue, then, to constructing a runtime invariant CFG is to determine the set of all possible targets for each and every indirect control flow instruction, as shown in Figure 2.3.

Considering software at a low level, indirect control flow may be categorized into three groups: jumps, calls, and returns. Indirect jumps, such as those arising from switch statements, are implemented for performance when the `case` set for a `switch` statement is large. At compile time, the target set of basic blocks for the `case` statements is known, making resolution of control flow edges simple. Other indirect jumps often have but a single target, e.g., process linkage table (PLT) entries. These must be resolved at load time for shared library linking. In any case, the exact address will be known at least by load time, thus, the potential targets of indirect jumps are knowable before execution begins.

Direct function calls and returns may be resolved from the call graph for an application. Indirect calls and their returns, however, are a special challenge which arises from programming constructs like function pointers. Pointer analysis in general is difficult for compilers, limiting optimization possibilities. However, in terms of CFG construction, function pointer analysis has distinct advantages over conventional pointer analysis. Most

```
for each instruction inst in application
 if type(inst) == return
  target_set(inst) == all instruction
     after call_sites
 elseif type(inst) == indirect_call
  target_set(inst) = all function
     where function_type(function)
           == call_type(inst)
 elseif type(inst) ==indirect_jump
  target_set(inst) = all instruction
     where instruction == target(inst)
 elseif type(inst) == virtual_call
  target_set(inst) = all function
     where vptr(inst) ∈ vtable(function)
 elseif type(inst) == optimized_switch
  target_set(inst) = all instruction
     where instruction == case(inst)
 elseif type(inst) == function_pointer_call
  target_set(inst) = all function_ptr
     where function_type(function_ptr)
           == function_type(inst)
 replace inst with multi-way_branch mbr
     where target_set(mbr) ==target_set(inst)
```

Figure 2.3 Indirect Instruction Target Set for CFG Construction. For each individual indirect jump, indirect `call`, or `return`, all allowable control flow edges must be determined prior to executing the code.

compilers, including *gcc* and *g++*, enforce function pointer assignment by argument and return types. We leverage this knowledge for greater precision in call-graph CFG analysis. There are special conditions which can work to defeat efficient function pointer analysis, such as function pointer casting and return type casting, using data types such as `void *`. However, a complete and correct (but perhaps conservatively constructed) CFG remains determinable. In the worst-case analysis, a function pointer may be assumed to reach any function. Performing function pointer analysis provides a more concise CFG, which further reduces potential code gadgets. Concurrently, this also improves runtime performance by reducing the size of conditional branch sleds for indirect function calls.

Virtual functions are implemented as indirect calls via the `vptr` attribute. Previous work has shown that these may be converted to direct calls by source code rewriting [83]. During compilation however, the same essential information for vtable implementation, i.e. class inheritance and overriding, is leveraged to derive a valid target set for a `vptr` directed call.

Returns are the most prevalent of all indirect instructions. In theory, the potential set of targets for any return can be determined by identifying all call sites for a function. In practice this does not always hold true, as programming constructs such as tail calls must be detected to reveal the true target. By reverse CFG walking, all reachable paths are found to determine possible return targets.

Position independent code (PIC) are code objects where the resolved address of any instruction is not known until the library is loaded. This presents a special challenge to discovering the CFG when considering objects compiled with PIC. However, the CFG for this code is fully discoverable at compile-time, as the underlying information about target sets for multi-way branches is available without dependence on addressing information.

2.3.2 Indirection Elimination

Elimination of indirect control-flow is the heart of this work. This severs the link between potentially insecure data and the program counter. Once a complete CFG has been constructed for an application, indirect control flow is no longer necessary for correct execution.

Indirection elimination is the process by which indirect control flow is replaced by direct control flow. The most straightforward approach is to replace an indirect branch with an equivalent set of conditional branches. This construct, called a sled, tests a potential target address against the known set of valid targets identified by CFG discovery. For example, a return statement would be replaced by a series of `if...then` statements, where each `if` statement tested a potential known-valid return address, which if matched would lead to a direct jump to the valid target. This process is depicted in returning from functions `bar()` and `baz()` in Figure 2.1. After complete indirection conversion has been achieved, all targets are reached by direct jumps or calls. Consider the event where an attacker is able to corrupt the data for a return, i.e., stack smash. All potential valid targets will be tested against the tainted value, which will fail to redirect control flow. At the end of any sled, a direct call to an abort function is inserted. This allows for the graceful exit of the program under attack, which can also be used to collect information on the attack.

Though elegant, CDI may introduce inefficiencies to runtime performance. Some instructions, particularly returns, may have a large set of valid control transfer targets. Performance implications are explored in detail in Section 2.7.

2.3.3 CDI Performance Optimizations

Assessing potential runtime implications of CDI, there are two major elements which may contribute to a degradation in performance. The first is the number of targets for each multi-way branch. A large set of valid targets will generate a correspondingly large sled of conditional branches. This creates both a larger binary and the potential to execute a greater number of instructions before taking the intended edge. There are several ways to address this concern.

Multi-Way Branch Target Ordering. A significant optimization is the profile-guided ordering of conditional branches in multi-way branch sleds, the process of which is shown in Figure 2.4. Dynamic profiling of edge counts can dictate insertion order of conditional branches. Complex orderings could be envisioned, such as tuning for branch prediction accuracy. However, the simple method of ordering edges by descending execution frequency provides a highly effective way to minimize the average number of not-taken branches which must be executed before arriving at the correct edge.

Single Target Set Reduction. The simplest optimization is the reduction of single-target indirect instructions to unconditional, direct jumps.

Frequent Function Cloning. Another straightforward optimization is function duplication for frequently called functions, which can proportionately reduce the set of valid return targets for each individual function clone. This optimization works well for small functions with many call sites.

Large Target Set Resolution. This optimization replaces a series of conditional branches with another mechanism which has either constant or logarithmic time complexity, e.g., a binary search tree. Any search method would incur some overheads, creating a minimum threshold to seek an alternate for a series of conditional branches. For example, a long series of conditional branches where the first is almost exclusively taken will execute faster than a search over the same targets in the average case.

The second major performance factor in indirection elimination is branch prediction performance for the inserted conditional branches. Branch mispredictions have non-trivial impact on runtime performance of applications. As such, addressing the predictability of the extra branches inserted to eliminate indirection is a concern. The only controllable dimension to conditional branch insertion is their ordering. Choosing an ordering by execution

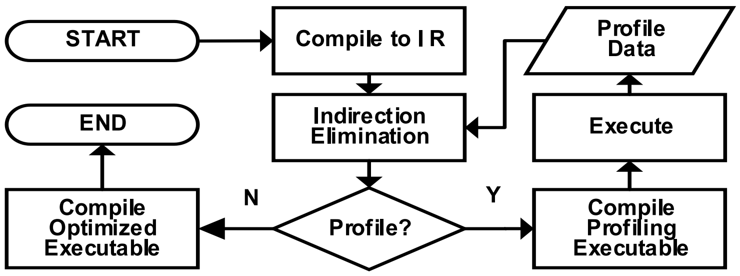


Figure 2.4 Dynamic Profiling for CDI Optimization. The runtime performance of multi-way branches implemented with direct conditional branches is greatly impacted by target ordering. By profiling target execution counts, we leverage inherent branch bias and order conditional branches by execution frequency.

frequency, ascending or descending, provides the average-case performance benefit of executing the most predictable branches first. Both the overwhelmingly taken and never taken branches will be nearly perfectly predictable. However, ordering with the most oft-taken branches first provides the added benefit of executing less untaken branches in the best and average case.

2.3.4 Detecting Attacks in CDI Protected Programs

When a control flow attack occurs on a CDI protected program, the realized effect is to exhaust the list of allowable targets in a conditional branch sled without taking any edge. This will also happen in the event of a non-malicious data corruption bug affecting a potential control-flow target. When this happens, the application will instead directly call a handler routine which gracefully exits the program. This handler can aid in debug/diagnosis by obtaining information about the crash, in the form of a unique ID for the call and the offending target address. This data can then be analyzed to determine the nature of the unexpected control edge.

To prevent control-flow attacks, it is essential to disallow any control flow which violates the predetermined CFG. All control flow is classified as either authorized or illegal, to facilitate our relaxed attack model (only a single illegal edge is needed to perform a heap spray attack). By disallowing all illegal CFG edges, we remove the essential element of control-flow attacks, thereby hardening software against them.

2.4 CDI Implementation for Shared Libraries

Not all control flow edges originate and terminate within a target binary. Many applications make calls to functions in dynamically-linked libraries at runtime. In order to provide protection for any application, the library code it calls should also adhere to the principle of CDI. To achieve this, we extend the use of indirection elimination to shared libraries.

2.4.1 Dynamic Nature of Shared Libraries

Shared libraries are referred to as such because a single copy of the library can be loaded once into physical memory and shared at multiple start addresses by multiple processes running concurrently. Further, they are dynamically linked when an application is loaded into memory. The dynamic nature of shared libraries make them a natural match for indirect control-flow. However, this also creates a natural vulnerability to control-flow attacks as well. An example of this is the return-to-libc attack [29], which circumvents non-executable stack protection to call attacker-desired functions in libc.

The dynamic nature of shared libraries, and their pervasive use of indirect control flow, presents new challenges for implementing CDI. These challenges include position independent code (PIC), the use of indirect jumps in the PLT in conjunction with the global offset table (GOT), and returns to potentially many different applications from a shared function in a library. Here we demonstrate the process of CDI in the context of shared libraries on Linux systems, though similar methods would be applicable to other approaches such as Dynamically Linked Libraries (DLL's) for *Windows*.

The current implementation of dynamically-linked shared libraries on Linux operating systems works as follows. Shared library code is compiled separately from application code and linked together when the application is executed. This linking is accomplished by the resolution of shared symbols in the symbol table of all linked objects. Each function call to a shared library is facilitated by the PLT and the GOT. When a function is called, the application executes a direct call to the PLT entry in the application code associated with the shared library function. The PLT entry then executes an indirect jump to the function, the target of which is stored in the GOT. When the library function completes execution, control returns to the original call site.

To facilitate the sharing of libraries, the address of a shared library function in the virtual address space must be resolved, as this is typically a randomized location in the memory space due to ASLR. When a function is called for the first time, the target address of the PLT jump in the GOT will not target the desired library function, but instead the next instruction in the PLT entry. This is a direct jump to a helper function which will determine the actual address of the desired function, via the program loader using the symbol tables of the code objects. Once the target address is established, the corresponding entry in the GOT is overwritten with the actual address of the desired function. This process is called *binding*, typically seen as *lazy binding* where the binding between objects is done at runtime upon the first invocation of a library function. This introduces an inherent weakness to control-flow attack, as the GOT table of function addresses could be overwritten with data at runtime which is then directly injected into the PC at the next shared library function invocation. Attacks on the GOT due to this weakness have been demonstrated [28, 114].

2.4.2 Enforcing CDI for Shared Libraries

Elimination of indirect control-flow removes the need to establish trust in user data. Target set resolution for MBRs remains the same process regardless of whether code is static, relocatable or position-independent. However, PIC code cannot contain absolute address references. To remedy this, all conditional branch/direct jump sleds are comprised of PC-relative address references. This allows all jumps and calls within PIC code to be implemented as direct jumps and calls.

In order to enforce CDI for shared library calls, our work eliminates the use of all indirect jumps implemented in the current structure using the PLT and GOT. An overview of our shared library implementation is depicted in Figure 2.5. Shared libraries remain separately compiled and linked by the program loader when an application is executed. As before, the PLT is used to invoke the library function. However, with CDI the program loader will overwrite the indirect jump instruction in the PLT entry with a direct call to the address of the library function, which was previously being written into the GOT as an indirect target. This is depicted in the application in Figure 2.5. We enforce dynamic linking at load time (i.e., non-lazy binding) before any runtime data is encountered. Thus, all control transfer targets are derived from programmer-specified instructions and the program loader, side-stepping any need to trust runtime data.

The task of returning from a shared library call is the last challenge in eliminating indirect control flow, and it requires eliminating the use of the `return` instruction. Here we leverage the same mechanism used to call the library function: the PLT. The return instruction is replaced with a direct, PC-relative jump to a new PLT entry whose purpose is to return control-flow back to the calling code object. This PLT entry then contains a direct jump to one of two locations. In the case where only one dynamically-linked code object in a process address space may call a given function, the PLT of the called function contains a

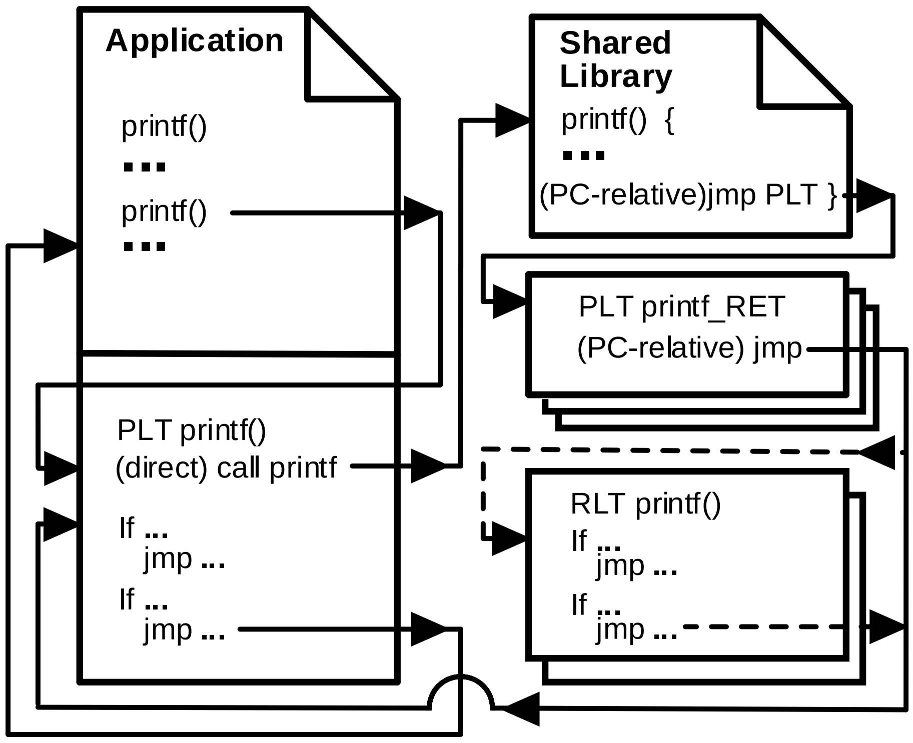


Figure 2.5 CDI for Shared Library Control Flow Transfer. All indirect control flow is replaced by direct calls and jumps, resolved at load time and written to the PLT, obviating the need for the GOT in function calls. In the case where more than one object may invoke a library function within the same process, an RLT entry is created, which executes a sled to return execution to the calling code object PLT entry. This then selects the correct return point in the application. Each process has a unique copy of the PLT and RLT while continuing to share the library code.

direct jump back to the PLT of the calling function. If there is more than one code object in a process address space which may call the library function (e.g., `malloc()` is called by both the application and library other than libc) then the single direct jump from the PLT will prove insufficient. In this case, a new code object is defined, referred to as the return linkage table (RLT). An RLT entry holds a conditional branch/direct jump sled which contains the return target addresses for all of the possible calling code objects within the address space of the process calling the library function. The PLT entry in the called function will then directly jump to its respective RLT entry. When the prospective return address is compared to the list of allowable targets and a match is found, the RLT then executes a direct jump to the target. The RLT is depicted in the shared library object in Figure 2.5.

The inclusion of direct jumps in the PLT and RLT require that they not be shared in memory (as they will differ for each application). Thus, they are aligned on page boundaries immediately following the shared PIC code of the library. This facilitates the ability to reach the PLT for each application by the same instruction in the library function. Consequently, all the benefits of shared libraries are retained such as dynamic linking and a single copy of large libraries like *glibc*. Additionally, the elimination of indirection in the implementation of shared libraries effectively removes the ability to perpetrate GOT-based attacks and any attack which exploits a `return` instruction, as not a single `return` instruction will remain in any code executed by a process.

It should be noted that the elimination of indirection in PIC is greatly aided by PC-relative instructions in the x86-64 and ARM ISAs. In other ISAs such as 32-bit x86, PIC implementation is more complicated by lack of PC-relative jump instructions. In such a case, CDI can still be readily achieved. To accomplish this, sharing would be disallowed, and libraries would be implemented as relocatable code, which is identical in implementing CDI as application code.

2.5 CDI Compliant Code for Dynamic Compilation

As discussed in Section 2.2.2, the threat and trust model for CDI is fully compatible with dynamically generated code or just-in-time compilation. Previous works based on CFI [2] are subject to label spoofing attacks, making additional assumptions about memory security such as disallowing code execution from the heap [10]. By contrast, CDI does not rely on labels at the targets of indirect jumps but instead verifies targets at the source of the indirect edge, as discussed in Section 2.2.1. This significant difference enables the integration of dynamically generated code.

2.5.1 Dynamically Generated Code

The previous sections of this chapter have dealt with applications and software code objects which are compiled in advance of execution, referred to as ahead of time compilation. Alternate solutions, such as interpreted languages like *Python* and virtual machines like the *Java Virtual Machine* (JVM) [131] allow code to be executed without being previously compiled. While application of CDI to interpreted code is outside of the scope of this work, the underlying applications which execute the interpreted code, such as *CPython* and JVMs, are precompiled applications and thus can be compiled to be CDI compliant. In this section, we will analyze the applicability of CDI to dynamically generated code for the Java Virtual Machine, though such an analysis could be generalized to other dynamically generated code.

Early implementations of JVMs were criticized for slow execution [87]. To accelerate execution of Java bytecode, frequently executed code can be compiled to machine code, which can then be directly executed on hardware avoiding the overhead of the virtual machine. This is also known as just-in-time compilation (JIT). Over time, compilation for JIT has become much more sophisticated, employing advanced compilation optimizations [62]. The application of control-data isolation to just-in-time compilation is straightforward. All intra-object control-flow would be made CDI compliant through the same compilation strategy outlined in Section 2.3. Similarly, control-flow between interpreted code such as the JVM and compiled code would be implemented as described in Section 2.4.

2.5.2 Threat and Trust Model for Dynamically Generated Code

Enabling control-data isolation for dynamically generated code requires revisiting the security model detailed previously in Section 2.2.1. The result is the identification of two additional potential security concerns. The first is the attestation of dynamically generated code to assure CDI compliance. This is achieved through verification that the code statements comply with the intrinsic properties of CDI compliant code, as detailed in Section 2.5.3. The second is the assurance that once the dynamically generated code is written to memory, it can not be overwritten by an attacker. For this second issue, memory protection mechanisms such as `mprotect` on Linux or `VirtualProtect` on Windows operating systems can be used to protect code at the page granularity from arbitrary modification. This is precisely the same assumption as made in Section 2.2.1.

2.5.3 Properties of CDI Compliant Code

To assess the compliance of a given code object, we must first identify the properties of CDI compliant code. As detailed in Section 2.2.3, control-data isolation is realized in this work by the removal of all indirect control flow. However, simply scanning a disassembled binary for indirect instructions is insufficient to determine whether code is CDI compliant. The removal of indirect instructions during the CDI process results in some essential properties of an application or code object. Here we detail those properties, and it is these properties that render CDI-compliant code immune to control-flow attacks such as code gadgets, heap sprays, buffer overflows, and stack smashing, among others.

Complete absence of indirect control-flow instructions. The first property of control-data isolated code is the complete absence of indirect control-flow instructions. That is, there are no instructions which modify the instruction pointer that reference registers or memory locations. The only allowable control-flow altering instructions contain targets which are hard-coded within the instruction.

All control-flow altering instructions target the beginning of a valid instruction. Another property is that no control transfer will ever target an arbitrary byte in the code, i.e., the middle of an instruction.

All `call` instructions target the first instruction of a valid function. As a result of the CDI compilation flow is that only valid function targets of a call instruction are allowable for any function call.

All control-flow altering instructions target a valid instruction within the same code object. As all indirect instructions are eliminated, all control-flow must originate and terminate within the code object. The exception to this is the inter-object control flow detailed in Section 2.4. The example of this is the use of the global offset table and process linkage table. These instructions are explicitly identified and handled in the manor shown in Figure 2.5.

2.5.4 CDI-compliant Code Attestation

Given the identification of properties of CDI compliant code in Section 2.5.3, we now detail the process for attestation of CDI compliant machine code, the algorithm for which is shown in Figure 2.6. Whenever a code object is dynamically compiled, it is first verified to comply with the properties of all CDI compliant code. The first step of this process is the disassembly of the code object, identifying all of the instructions in the code object. The disassembled code is then scanned for any indirect control-flow instructions, failing the verification if any exist in the code.

The control-flow graph for the code object is then constructed. Unlike the heavyweight process of determining a CFG from the original code, the lack of indirection makes the determination of the precise CFG straightforward. Constructing a CFG from compiled code has long been accomplished with static binary rewriting techniques. Static binary rewriting has been employed in Control-Flow Integrity [2] and derivative works, such as that by Zhang and Sekar [150]. Once the CFG is constructed, all edges can be verified as compliant to the properties identified in Section 2.5.3. The inter-object control-flow, such as the mechanisms to return to the JVM once execution completes, is implemented in the precise fashion identified for linking dynamically-linked libraries as shown in Section

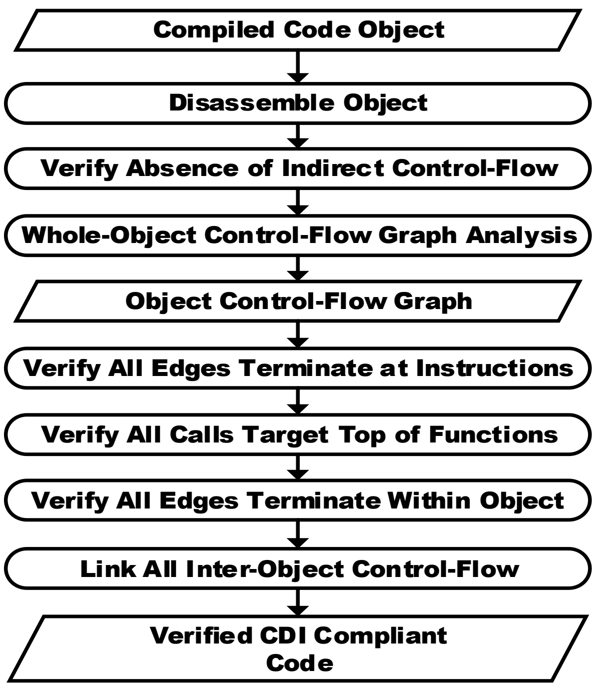

Compiled Code Object
Disassemble Object
Verify Absence of Indirect Control-Flow
Whole-Object Control-Flow Graph Analysis
Object Control-Flow Graph
Verify All Edges Terminate at Instructions
Verify All Calls Target Top of Functions
Verify All Edges Terminate Within Object
Link All Inter-Object Control-Flow
Verified CDI Compliant Code

Figure 2.6 CDI Code Attestation]. Dynamically generated code is attested to be CDI compliant in accordance with the threat and trust model additions detailed in Section 2.5.2. The process begins with a code object which has been compiled with the CDI principle, as identified in Section 2.3.2. This object is then disassembled to identify all instructions, which are examined to verify no indirect instructions are present. The control-flow graph of the object is then discovered. The properties inherent to all control-data isolated code, detailed in Section 2.5.3 are each then verified against the list of instructions and CFG. In the case that all properties hold for the code object, it is considered verified to be CDI compliant.

2.4.2. Following the above procedure for CDI compliance attestation and object linking as detailed above and in Figure 2.6, the JIT compiled-code pages are marked as non-writable code, completing the verification process for ensuring dynamically compiled code is CDI compliant.

2.6 PitBull: Compiler-Based Control-Data Isolation

To validate our control-data isolation enforcement via indirection conversion, PitBull (**P**ositive **I**ndirection elimination **B**y **l**llvm) was built. PitBull is a compiler optimization utilizing the *llvm* compiler infrastructure [86]. The set goal was to establish feasibility for indirection-free executables. Of equal importance, this also facilitates the analysis of runtime performance implications of CDI.

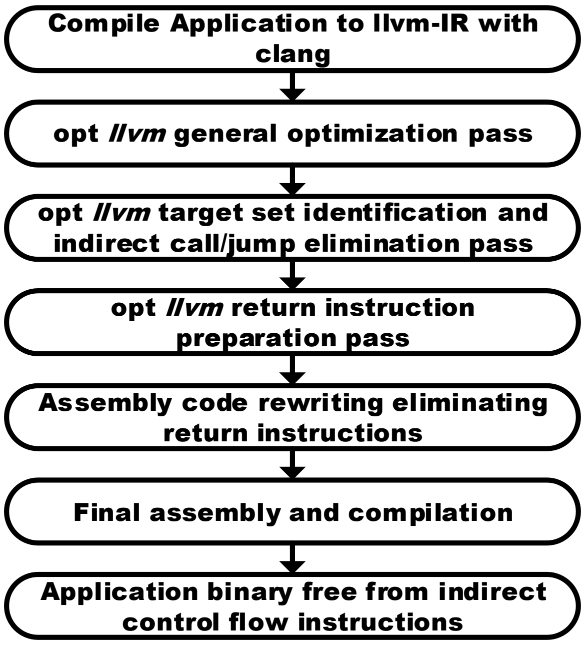

Compile Application to llvm-IR with clang
opt *llvm* general optimization pass
opt *llvm* target set identification and indirect call/jump elimination pass
opt *llvm* return instruction preparation pass
Assembly code rewriting eliminating return instructions
Final assembly and compilation
Application binary free from indirect control flow instructions

Figure 2.7 PitBull Compilation Flow. Applications are compiled to be free of indirect control flow instructions. Leveraging the *llvm* compiler infrastructure, optimization passes identify all valid targets of indirect control flow and insert conditional branches in replacement.

The *llvm* optimization-based implementation of PitBull, shown in Figure 2.7, works as follows. The target applications are first compiled to *llvm*-IR with the clang compiler. All IR files are then linked by the *llvm*-link tool. A standard optimization pass is then performed by the *llvm* tool *opt*. At this point the target executable has been compiled into *llvm*-IR and is ready for our indirection conversion optimization passes, invoked again with the *llvm* tool *opt*. The primary pass first identifies the nodes and edges of the CFG relevant to indirect calls, jumps, and returns. Function pointer analysis is performed to identify control flow edges not readily available from the standard dot-callgraph *llvm opt* pass. Once the targets of indirect control flow instructions have been identified, indirect call and jump instructions are replaced with a series of `if..then` (`icmp..br`)statements. For each allowable target, a compare is made to the candidate target, followed by a direct jump to the allowable target. A second pass then aggregates call and return data in preparation for the ensuing assembly-level rewriting passes. The transformed *llvm*-IR is then compiled to assembly via the *llvm llc* tool.

At this point, indirect calls and jumps have been eliminated from the target application. Returns are then handled by assembly code rewriting. First, a label is placed after each call of the program. Next, all return statements are replaced with a series of compares and direct jumps. The set of valid return targets, provided by the first *opt* pass, have their corresponding newly inserted labels compared to the stack pointer. Each compare is followed by a conditional direct jump to the compared label. After all compares and conditional jumps are inserted, a terminating call to the attack handling routine is inserted. Since we substitute direct jumps for returns, the stack pointer is incremented before any compares, which then compares to the stack pointer minus one word.

All indirect instructions have now been replaced by compares and direct jumps. The code is then assembled and an executable is produced. We note that the process for relocatable code is exactly the same as that for static application code. When the code is relocated at load time, the relocation table will update all absolute address references with their new location, for both the compares and direct jump instructions.

To facilitate shared libraries, we also eliminate indirect control flow instructions in PIC code. This requires a more nuanced approach than static or relocatable code. The target sets for indirect branching instructions are identified exactly as before. However, the inserted sleds must be PIC-compliant and may not contain absolute address references. To implement this, our system instead utilizes PC-relative instructions. For each allowable target, a comparison is made to the candidate target, followed by a PC-relative direct jump.

In our framework, we do not implement load-time functionality. This means that our current *llvm* compiler-based implementation does not provide our proposed CDI protection against GOT-based attacks. However, we do model overheads associated with dynamic library implementation, including non-lazy binding. Further, the additional sled added to the calling application PLT, depicted in Figure 2.5, is simulated by adding the sled as padding to the return location of the called library function. The execution of library calls would be nearly identical in performance, save the last step would be a direct call from the PLT. As stated for devirtualization in Section 2.2.3, this is expected to improve execution time for calls. Returns from libraries would potentially suffer from additional RLT entry traversal. However, this impact would be minimal, as in our benchmarks only 2% of control transfer from library calls require an RLT entry, while the remainder would jump directly from the PLT of the shared library to the PLT of the application.

2.7 Experimental Evaluation

To fully understand the runtime implications of CDI, the performance of our compiler implementation was evaluated. The testing platform consists of 64-bit x86 workstations running *Ubuntu* 12.04 LTS *Precise Pangolin* with Linux kernel 3.5.0-39-generic. Compilation and optimization is accomplished with clang and *llvm*, both release version 3.3. All optimization passes are registered *llvm* passes, while the assembly code rewriting is performed using *Perl*.

2.7.1 Benchmark Applications

Several security-sensitive and network-facing applications were chosen to evaluate runtime performance. These include sha1sum, sha256sum, sha512sum, and md5sum from the *GNU Coreutils suite*, as well as *tcpdump*, a popular network packet analyzer and *bftpd*, an ftp server. The *SPECINT2000* benchmarks were also included to allow a direct comparison between our work and earlier works such as CFI [2]. We further implemented CDI for the *musl* libc library [99], due to a known lack of compatibility between *clang* and *glibc*.

2.7.2 Performance Evaluation

SPEC benchmarks were executed with the standard runspec interface. Other benchmarks were executed while processing as input large, 45GB network capture files. Results are timed and averaged over 5 runs, shown in Table 2.1. The runtime overheads shown reflect the increase in runtime for benchmarks relative to the original, unmodified applications. Default compilation parameters are held constant for both original and modified binaries. The nave runtime overheads represent the performance overhead without any subsequent optimizations. Ranging from almost zero to nearly 2× slowdown, the nave implementation averages about 45% for all benchmarks. When optimizations are applied (as detailed in Section 2.3.3), we see a dramatic decline in the execution overheads for all benchmarks, where over half have no perceivable overhead at all. There is also a noticeable difference in runtime overheads between SPEC benchmarks and the network-facing applications. SPEC benchmarks, by design, are generally compute-intensive workloads. However, the remaining applications, such as tcpdump, typically have performance which is I/O bound. For these workloads, which are a prime candidate for CDI protection, the cost for such protection is hidden by I/O overhead.

2.7.3 Impact of Optimization

As observed in Table 2.1, optimization has a considerable impact on performance. There are two main factors which influence this; the heavily biased nature of dynamic branch execution, and the execution frequency of indirect control flow instructions. In our experiments, we implemented an optimization based on execution frequency of indirect branch targets.

				Valid Control Flow Edges per Indirect Instruction							
				Static				Dynamic			
Benchmark	Optimized Runtime Overhead	Naïve Runtime Overhead	Binary Size Increase	Max	Mean	Median	Mode	Max	Mean	Median	Mode
gzip	0.7%	4.3%	14%	45	5.7	2	2	15	3.0	2	1
vpr	1.0%	2.5%	33%	116	5.1	3	2	20	2.7	2	1
gcc	34.4%	59.2%	112%	194	30.3	8	2	648	5.5	2	0
mcf	0%	0%	10%	2	1.0	1	1	2	1.0	1	1
crafty	4.0%	10.0%	29%	33	5.8	3	2	16	2.4	2	2
parser	5.3%	39.3%	46%	216	8.3	3	2	151	6.1	2	2
eon	22.8%	51.2%	73%	114	7.2	2	0	33	0.5	0	0
perlbmk	20.9%	189.1%	148%	537	31.4	18	18	134	3.8	1	0
vortex	20.1%	143.0%	42%	192	11.7	4	3	158	8.7	2	1
bzip2	0.9%	1.5%	9%	3	2.1	2	2	3	1.4	1	1
twolf	0.7%	1.1%	24%	18	3.8	2	2	8	1.5	1	1
md5sum	0%	0%	25%	8	2.6	2	2	3	1.0	1	0
sha1sum	0%	0%	22%	8	2.6	2	2	3	1.0	1	0
sha256sum	0%	0%	20%	8	2.6	2	2	3	1.0	1	0
sha512sum	0%	0%	16%	8	2.6	2	2	3	1.0	1	0
bftpd	0%	0%	81%	109	6.8	2	2	41	1.5	0	0
tcpdump	0%	1.2%	174%	400	75.5	65	65	14	0.1	0	0
SPEC Avg.	10%	45.6%	49%	134	10.2	4.4	2	108	3.7	1.5	1
Average	6.5%	29.6%	52%	293	10.2	4.4	2	74	2.5	1.2	0

Table 2.1 Control-Data Isolation Performance. The optimized runtime overhead from CDI appears in the first column. The last 8 columns detail indirect control flow edges metrics for benchmarks. The static details the properties of the CFG related to indirect control flow instructions. Dynamic data reflects the runtime control edges seen during execution. Together, these contrast dynamic and static properties of control flow. Runtime overhead can be seen to positively correlate with control flow edges.

Benchmark applications were profiled to collect edge counts for the MBR edges. This information is then fed back into a second compilation. Edge counts are utilized to order conditional branch insertion for indirection conversion, by descending order of execution frequency. This yields the optimized performance shown in Table 2.1.

When considering the dynamic behavior of branches, it has long been known that branches are heavily biased to one particular branch direction during execution [147]. This biased property strongly facilitates the high accuracy of modern branch prediction.

Though indirect branches are more difficult to predict [41], they remain highly biased as well [142]. To assess this bias, we profiled execution of benchmarks to determine the distribution of dynamically executed targets, shown in Figure 2.8. When indirect control flow instructions are broken down into binary branch decisions, the resulting control flow points, taken individually, become more easily predicted than the original indirect branch. As shown in Table 2.1, the dynamic target set is considerably smaller than the static set. The data in Figure 2.8 strongly supports our MBR optimization, previously identified by

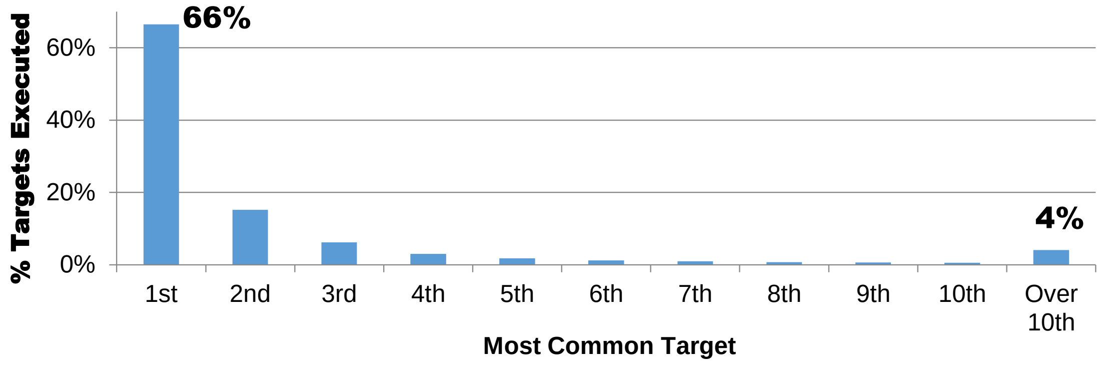


Figure 2.8 Indirect Branch Bias. Branching instructions are heavily biased in execution. The percentage of instructions as a function of the most common targets for each indirect branch/call/return are binned by execution frequency. For all benchmarks combined, the most commonly executed target of each MBR accounts for over 66% of all edges executed.

Chang and Hwu [33] for multi-way branches implemented for `switch` statements. In their work, any `switch` statement with less than ten branches would be implemented as sleds of conditional branches and direct jumps, ordered by profiled execution frequency. This highlights the crucial factor for runtime overheads; dynamic branching properties, not static, are the driving force behind runtime performance.

The second property of software which heavily dictates the performance of indirection conversion is the relative execution frequency of indirect control flow. For all benchmark applications, indirect instructions accounted for 1% of the total instructions executed. Coupling this with highly biased branches, it is no surprise that indirection-free transformations incur minimal overheads.

As shown in Table 2.1, runtime overheads generally tend to be positively correlated with the size of target sets for indirect instructions. One indirect instruction in the *gcc* benchmark had 648 different valid targets executed at runtime (a `return` instruction). That implies that at least once, a function was forced to execute 647 not-taken conditional branches before finding the correct edge for a return. This highlights the opportunity for low time-complexity alternatives to conditional branch insertion, as discussed in Section 2.3.3., which is left for future work.

2.8 Related Work

This work is conducted in light of many techniques which have been devised in an attempt to address control-flow attacks. An abridged list is presented here, divided into software and hardware approaches. Direct comparisons are summarized in Table 2.2.

Work	Explicit Dependencies				Susceptible to these Attacks			Relies on Data Memory Security	Eliminates Usage of Indirect Control
	NWC	ASLR	W⊕X	Shadow Stack	Heap Spray	GOT	Read		
This work	Yes	No	No	No	No	No	No	No	Yes
Abadi et al. [1]	Yes	No	Yes	Yes	No	Yes	No	Yes	No
Xia et al. [142]	Yes	No	No	No	Yes	Yes	No	Yes	No
Budiu et al. [24]	Yes	No	Yes	Yes	No	Yes	No	Yes	No
Zhang, Sekar [150]	Yes	No	No	Yes	No	No	No	Yes	No
Kiriansky et al. [80]	Yes	No	No	No	No	No	Yes	Yes	No
Cowan et al. [43]	Yes	No	No	No	Yes	Yes	Yes	Yes	No

Table 2.2 CDI Related Works. The first set of columns details what system dependencies are explicitly required to maintain the purported security benefits of a work. The next set details which vulnerabilities a work provides no hardening against. The final column states whether a work eliminates the root cause of contemporary control flow attacks: indirect control-flow. Our work remains as the single one to harden against all control-flow attacks while maintaining only the most fundamental dependency of NWC. NWC=non-writable code, Shadow Stack=separate stack in memory to verify return address targets, GOT=Attacks on calls to libraries, Read=Technique is weakened if attacker can read or infer any data memory contents or locations.

2.8.1 Software Mechanisms

An important work in this area is Control Flow Integrity (CFI) by Abadi et al. [2], which spurred an avalanche of interest in the dynamic enforcement of software CFGs at runtime. The relatively low overhead, simple solution set a high bar for all subsequent efforts. In their work, the authors utilized a labeling system to verify the authenticity of a target address. Return instructions are guarded by a shadow stack, with the code segment register functioning as the shadow stack pointer. Though CFI is an elegant solution, it relies on a more restrictive attack model than our work while incurring greater execution overheads. Further, with reliance on a shadow stack located in data memory, and not addressing shared library calls and returns, there are continued concerns about control-flow attacks. In contrast, CDI's threat model assumes that the attacker fully owns data memory, with read, write, and execute privilege.

A recent work which expands upon CFI is Control Flow Integrity for COTS Binaries [150] by Zhang and Sekar. This work provides a solid implementation for CFI instrumentation for stripped binaries. However, it was not extended to protect control transfer between shared library and application code (e.g., GOT attack [28]). It also retains limitations from binary-rewriting such as not handling dynamic code generation.

G-Free [104] is a compiler-based approach to eliminating ROP attacks. This is a two-pronged approach of excising unintended return or return-like instructions, along with encryption-based verification of the context in which indirect branches are executed (e.g., a `return` instruction is executed only after the first block of the function in which it resides has been executed). Though this appears to constrain code gadgets, the approach offers no protection from non-code gadget attacks such as heap spray and return-to-GOT.

Another foundational work is Secure Execution Via Program Shepherding by Kiriansky et al. [80]. Utilizing dynamic binary instrumentation, Program Shepherding enforces a security policy by monitoring control flow transfer at runtime. Though Program Shepherding could enforce a policy similar to CDI, it still cannot determine all valid indirect branching targets without nontrivial compilation support (such as what we propose in this work). The CFG as enforced by program shepherding is emblematic of the actual CFG, and therefore cannot offer the same level of protection as CDI. When an application's CFG is discovered at runtime, the targets of indirect jumps cannot be known before they execute, and therefore cannot be verified dynamically. This allows a jump to the middle of an x86-64 instruction, permitting unfettered code gadgets within an application.

Recently, compiler-based solutions have also been proposed. One such work is Enforcing Forward-Edge Control-Flow Integrity in *gcc* & *llvm* [137], which instruments applications with CFI checks and labels at compile time. Though Tice *et. al.* achieve low runtime overheads, they do not address return instructions, which constitute the majority of indirect control flow. Their approach remains vulnerable to code gadgets, as the range verification for jumping to compiled executables allows jumping to the middle of instructions.

Another compiler-based approach to CFI is Control-Flow Restrictor: Compiler-based CFI for iOS [109]. In their work, Pewny and Holz share the most commonality with our work, applying a similar MBR conversion approach. However, focusing on iOS on an ARM platform, they do not explore large, complex applications with many functions or topics such as PIC or shared libraries.

2.8.2 Hardware Solutions

A variety of hardware-based solutions have been proposed to address control flow security. One example is Architectural Support for Software-Based Protection [24]. That work is an extension of the original CFI [2] work, with a proposed ISA extension to move CFI label checking into hardware. This work carries with it the same weaknesses as CFI. Another hardware solution is offered in the work CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters by Xia et al. [142]. This work leverages existing hardware in the form of performance counters. Though their solution has low overheads, it has to contend with deficiencies such as false positives and negatives, as well as allowing suspicious branch targets to execute, making the technique readily susceptible to heap spray attacks.

It should be noted that all previous works may be classified as mitigation techniques. That is, they all seek to guard, verify, or otherwise shield the root of the problem for control-flow attacks: indirect branching. As such, they all rely heavily on a host of security assumptions, which are in turn susceptible to attack. Regardless of the proposed solutions, of which there are many, control-flow attacks persist. In contrast, our work directly addresses the root issue and permanently removes it by isolating control from user data.

2.9 Chapter Conclusions

Computer security has become a dominant topic in the information age. The software attack surface has remained as a chief area of security exploit for years. Though vulnerabilities have been well studied, exploitations persist. Given the continuing nature of these attacks, this work directly addresses and eliminates the prevailing root of the problem: indirect control flow.

In this work we presented a novel approach to software security, called control-data isolation, which eliminates the link between potentially malicious runtime data and program control by eliminating the use of indirect control in generated software. We have shown that eliminating the root cause giving rise to the predominant mode of control-flow attacks is not only feasible, but has minimal impact on runtime performance. Control-data isolation provides a greater level of security than previous proposals while experiencing overheads that are comparable or better. We feel strongly that by directly addressing control-flow attacks, rather than mitigating them, the overall software attack surface can be greatly diminished.

Chapter 3

Locking Down Insecure Indirection with Hardware-Based Control-Data Isolation

Matter is plastic in the face of Mind.

Valis

Philip K. Dick

Arbitrary code injection pervades as a central issue in computer security where attackers seek to exploit the software attack surface. A key component in many exploits today is the successful execution of a control-flow attack. Control-Data Isolation (CDI) has emerged as a work which eliminates the root cause of contemporary control-flow attacks: indirect control flow instructions. These instructions are replaced by direct control flow edges dictated by the programmer and encoded into the application by the compiler. By subtracting the root cause of control-flow attack, Control-Data Isolation sidesteps the vulnerabilities and restrictive threat models adopted by other solutions in this space (e.g., Control-Flow Integrity). The CDI approach, while eliminating contemporary control-flow attacks, introduces non-trivial overheads to validate indirect targets at runtime. In this work we introduce novel architectural support to accelerate the execution of CDI-compliant code. Through the addition of an edge cache, we are able to cache legal indirect target edges and eliminate nearly all execution overhead for indirection-free applications. We demonstrate that through memoization of compiler-confirmed control flow transitions, overheads are reduced from 19% to 0.5% on average for Control-Data Isolated applications. Additionally, we show that the edge cache can efficiently provide the double-duty of predicting multi-way branch targets, thus providing even speedups for some CDI-compliant executions, compared to an architecture with unsophisticated indirect control prediction (e.g., BTB).

3.1 Introduction

Computer security has become an ever-rising concern in the modern world. At the heart of security lies the software attack surface. This surface provides attackers with a wide range of opportunities to exploit computing systems. Well known vulnerabilities run the range from code gadgets, return-to-libc, heap spray attacks, rootkits, the classic yet still pervasive buffer overflows, and a plethora of other attacks. The effort of attackers to exploit the software attack surface continues to grow in the malware arms race [49]. Indeed, most attacks continue to be conducted within the application layer [55]. In this realm of exploitation, arbitrary code execution, generally achieved by control-flow attacks, pervade as a primary means to attack software.

We build on a novel control-flow attack elimination method, Control-Data Isolation (CDI) [12], which protects programs from control-flow attacks by eliminating all indirect control transfers. We address the non-trivial performance impacts of CDI programs, by introducing hardware that virtually eliminates the performance penalty associated with ensuring secure control flow at runtime. We demonstrate that through high-accuracy memoization of programmer-intended, compiler-selected indirect control transitions, we can nearly eliminate the runtime costs of eliminating contemporary control-flow attacks. In addition, we demonstrate that the structures we add to speed up CDI execution can easily perform the double duty of multi-way control point prediction, thereby providing even opportunities to speed up CDI program execution, compared to targets with simple indirect control predictors (e.g., BTB).

3.1.1 Control-Flow Attacks

As we have shown in Chapter 1, control-flow attacks persist as a primary building block for software exploitation. These attacks target the program counter (PC) of a system, redirecting program execution to code of the attackers choosing. This code can be existing code in the application space, or injected by the attacker as user data. In any case, ***all forms of control-flow attacks corrupt the PC of an executing application with the injection of malicious user data.*** These attacks violate the programmer-intended control-flow graph of an application. A classic example of control-flow attacks is a buffer overflow. In this attack, a buffer located on the stack (i.e., local function variable) receives malicious input, overflowing the bounds of the array, overwriting the return address on the stack and directing execution to attacker-injected code on the stack. This chapter will continue to build upon Chapter 2 in the elimination of control-flow attacks.

As detailed in Chapters 1 and 2, control-flow attacks and code injection have grown as a major threat to computing, many works have sought to address them. These include mechanisms such as Stack Guard [44], Point-Guard [43], Address Space Layout Randomization (ASLR), Read or Execute memory page protections, and others. Additionally, these attacks have received considerable attention within the research community. Earlier works such as Program Shepherding [80] have attempted to address control-flow attacks, as well as Control Flow Integrity (CFI)[1] which has inspired many descendant works [37, 52, 98, 107, 127, 142, 149, 110]. Adoption of these countermeasures serves to make control-flow increasingly difficult, but ultimately, these countermeasures only represent stumbling blocks for attackers, as they have repeated devised more sophisticated attacks (e.g., heap spray attacks), and blended attacks (e.g., canary key read attacks), such that control flow attacks remain a dire software vulnerability to this day.

At the heart of all contemporary control flow attacks lies an inherent weakness pervasive in software: indirect control flow. That is, attackers exploit the operation of determining control flow targets at runtime from user-injected data. The mixing of runtime data and the program counter continues to enable such attacks.

3.1.2 Control-Data Isolation

Control-Data Isolation (CDI) [12], detailed in full in Chapter 2 of this dissertation, has been proposed to eliminate contemporary control-flow attacks. CDI, reviewing from Chapter 2, takes the approach of removing the inherent weakness software has to contemporary control-flow attacks: indirect control flow. ***By stopping the mixing of control information and user data, through the elimination of all indirect control transfer instructions, CDI removes the one tool attackers have to force a program off the programmer-intended CFG.*** As such, programs that enforce CDI inherently enforce the control-flow graph (CFG) of an application. That is, all dynamic control flow decisions are hard-coded directly into the instructions of an application text segment protected as executable and non-writable in memory.

Reviewing the concepts of Chapter 2, CDI works by compiling the program without indirect control instruction, e.g., jumps through register, returns, indirect calls, and replacing them with multiway branch code sequences that validate any attempted multiway branch target is to an expected and valid control target. The compiler, using whole-program control and call graph analysis, generates the complete list of valid indirect targets, which is then embedded into the program using a “sled” code sequence construct. Previous work has shown that this approach works for regular programs as well as shared libraries, and even dynamically generated code with a limited degree of operating system support.

The major limitation to approaches such as CDI (as well as CFI and other such works) is the performance penalty associated with executing these conditional branch sleds in lieu of indirect instructions. This arises from the relevant frequency of indirect instructions such as `return`, as well as the potentially long sleds for popular functions. As well, such sleds can inflate the binary of applications, causing added pressure to instruction caches. Additionally, executing more direct control-flow instructions also increases load on prediction mechanisms such as the BTB and conditional branch predictor of a processor.

3.1.3 Advancing CDI with Architecture Support

To facilitate the adoption of Control-Data Isolated code, this chapter introduces a novel, conditional branch sled-backed, capability to safely allow indirection. Our work couples an indirect jump (such as `jreg`, indirect call, and `return`) with a sled of direct compares and jumps. When indirect instructions are first encountered in a program execution they are not yet validated. Thus, they will not be executed and the sled comprising all allowable control-flow transitions for the indirect instruction is instead executed. If the sled indicates that an indirect jump target is valid, it will result in a taken conditional branch. This valid control flow edge is comprised of the indirect PC address and the target address of the taken conditional branch. This valid CFG edge is then stored in an edge cache. As indirect control-flow instructions are encountered in program execution, we first probe the edge cache by the indirect PC address and register-derived target address. If this edge is represented in the edge cache it was formerly generated by executing the direct control flow encoded in the sled. In this case, the sled is skipped, and the indirect control instruction is allowed to execute. If the indirect instruction and its proposed target cannot be validated by the edge cache, the sled is executed to its valid edge and this new edge consisting of source and target addresses is cached.

Using this approach, we ensure that all control-flow edges are validated against the whitelist of targets represented in the sleds. As our results show in Section 3.5, a trivial fraction of sleds is actually executed resulting in nearly no slowdown for this powerful control-flow protection mechanism. Additionally, we observe that the information contained in the edge cache, combined with history information, could provide double-duty as an indirect branch predictor. We propose additions to the indexing mechanism of the edge cache for prediction of indirect branches. Accurate indirect branch prediction remains an important problem in computing and continues to be the subject of contemporary research efforts [51, 118, 124]. Works such as ITTAGE [124] continue as even small improvements to prediction rates can yield worthwhile performance gains. As such, we show that, compared to architectures with only simple BTB-based indirection prediction, using the edge cache additionally as an indirect predictor results in a modest speedup for CDI-compliant programs, with some applications reaching up to over 6% performance improvement.

3.1.4 Contributions of this Chapter

The primary achievements of this chapter are two-fold. The first is the elimination of the potential overheads of Control-Data Isolation remaining from the previous work of Chapter 2. The second purpose of this chapter is to extend the principles of CDI to paths of execution. The majority of this chapter is derived from the publication "Locking Down Insecure Indirection with Hardware-Based Control-Data Isolation" [10] This chapter makes the following contributions:

- We show how simple hardware and software for CDI support can guard a program from control-flow attacks.
- We demonstrate that edge caching eliminates nearly all of the slowdowns associated with the execution of indirect jump validation sleds.
- We explore how the use of the edge cache as an indirect branch predictor enables speedups for select CDI-compliant programs compared to architectures with simple BTB-based indirection prediction.
- We detail a novel hardware mechanism to extend the principal of control-data isolation toward the elimination of path-based control-flow attacks. As shown in Chapter 2, CDI eliminates contemporary control-flow attacks. In this chapter, we show how simple hardware support can eliminate future attacks in control-flow security which violate the programmer-intended *paths* of execution for applications.

This chapter, Chapter 3, encapsulates the following work. Section 3.2 reviews how control flow is protected by Control-Data Isolation and details how to accelerate execution of CDI-compliant code while Section 3.3 details our proposed edge cache and indirect prediction mechanisms. Section 3.4 then provides details of our implementation and Section 3.5 shows and discusses results. Section 3.7 highlights related works in control-flow security and we conclude in Section 3.8.

3.2 Protecting Control Flow with Control-Data Isolation

Arbitrary code injection attacks are the end goal for many attackers. At the heart of code injection are contemporary control-flow attacks. CDI protects against the subversion of control flow at runtime by eliminating the use of indirection.

3.2.1 Threat and Trust Model

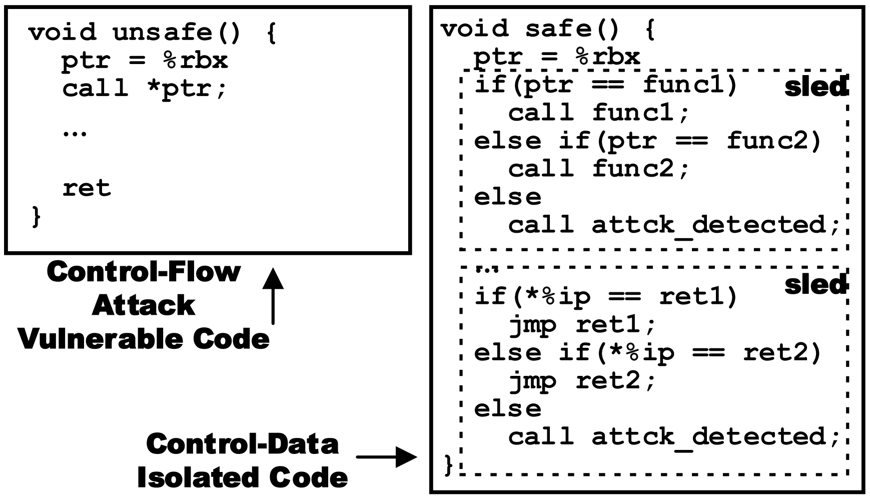
In this work, we adopt a relaxed threat model and minimal trust model, similar to previous CDI work [12]. In our threat model, an attacker is presumed to possess great influence over a system. In this, we assume that any attacker has arbitrary and complete control over data memory. That is, an attacker is assumed to be allowed read, write, and even execute privilege over data memory.

Delineating our trust model is simple. We only presume that an attacker does not possess the ability to overwrite information from the code segment of an application at runtime. That is, we rely on memory protection of write or execute for the code of a target application. We consider this a basic tenet of security. Violation of this trust obviates the need for a control-flow attack, as arbitrary code injection would be replaced by arbitrary code replacement.

3.2.2 Control-Data Isolation Mechanism

Contrary to prior works, CDI takes a subtractive approach to preventing code injection. In prior works, control-flow attacks have been addressed by layering additional complexity onto the software stack in an effort to shield software from exploitation. These additional layers of complexity are intended to protect the underlying indirect control instructions from malicious user data. However, the additional layers rely on supplemental, restrictive, security assumptions and may be subject to attack themselves, such as the Rio platform supporting Program Shepherding [80]. Further, recent work by Davi and Sadeghi [47] have demonstrated that the policy of recent course-grained CFI works [37, 52, 98, 107, 150] are still subject to control-flow attacks. Control-Data Isolation has emerged as a novel approach in addressing and eliminating control-flow attacks by subtracting the root cause of indirect control flow. Specifically, control-data isolated code completely removes indirect control flow from software, thereby severing the path between user data and the program counter. Without this capability, it is not possible to exit the programmer specified CFG.

Control-Data Isolation is achieved by a compilation process, which replaces unsafe, indirect control-flow, instructions with direct control-flow instructions. First, the entire control flow and call graph of an application must be determined. This enables the encoding of the programmer-intended edges of a CFG into the direct branches of an application. Once the CFG is fully discovered, all indirect instructions such as `return`'s, indirect calls and indirect jumps are converted into Multi-Way Branches (MBRs), where each allowable target of an MBR is defined and encoded by an edge in the CFG. These MBRs are subsequently converted to what are referred to as conditional branch ***sleds*** consisting of a sequence of compares and direct jumps. In essence, ***the sled constitutes a whitelist of allowable targets for any particular indirect jump; any transfer not contained in the whitelist of targets results in termination of the program.*** Figure 3.1 demonstrates how validated sleds with direct branches can replace an indirect call instruction. Rather than trust the register or memory indirect reference of the instruction, these sleds completely define all allowable control targets. At each point in the original application where an indirect instruction would appear, control instead falls through to the sled, which compares the potential target contained in the register (the stack pointer in the case of a `return` instruction) to each allowable target embedded in the instructions. A direct jump or call is then taken when an offered target matches. In the event all possible direct control-flow instructions are exhausted for a given sled, the target is considered malicious, and the program execution is terminated with a message. It is important to note that, though CDI is implemented via compilation, the potential to implement CDI on existing binaries. Operations such as discovering the complete CFG of an application, can be achieved by static binary rewriting [1].


sled
sled
Control-Flow Attack Vulnerable Code
Control-Data Isolated Code

```
void unsafe() {
  ptr = %rbx
  call *ptr;
  ...

  ret
}
```

```
void safe() {
  ptr = %rbx
  if(ptr == func1)
    call func1;
  else if(ptr == func2)
    call func2;
  else
    call attck_detected;
  ...
  if(*%ip == ret1)
    jmp ret1;
  else if(*%ip == ret2)
    jmp ret2;
  else
    call attck_detected;
}
```

Figure 3.1 Control-Data Isolated Code [12]. CDI replaces all indirect control flow instructions such as indirect calls, jumps, and returns with direct, conditional control flow instructions. For each indirect instruction, all allowable targets are determined by the compiler. This encodes the CFG of an application within the binary, represented by the conditional branch *sleds*.

3.2.3 Runtime Overheads Executing CDI-Compliant Code

Previous work by Arthur et al. [12] has demonstrated the benefits and viability of indirection-free software execution. However, performance concerns persist. Runtime overheads remain well above average for important applications, even with a number of compiler optimization proposed in the previous work. The runtime overheads of executing CDI compliant code arise from these sources.

The first cause of runtime overhead from control-data isolated code stems from an increase in dynamic instruction execution counts, due to executing indirect call validation sleds. Instructions such as the `return` instruction, a single 1-Byte instruction in the *x86-64* ISA results in multi-byte compare and jump instructions for each potential return address target. For functions with many potential return sides, the associated sleds can become very large. In the case of *176.gcc*, we observed this expansion to double the total number of instructions executed over the native application. The second cause of runtime overheads is directly related to the first; the increase in binary size from the inserted sleds leads to increased pressure on the L1 instruction cache, leading to increased instruction cache misses and program stalls. The third source of overheads from CDI-protected code is related to the conditional branch prediction accuracy. With the introduction of slides, there can be a significant increase in the number of branch instructions, which places additional pressure on already overprescribed branch predictor resources. In the previous example of *176.gcc*, half of the additional dynamic instructions are direct conditional branches, dramatically increasing demand on branch prediction resources.

In previous efforts that implemented software-only CDI frameworks, runtime overheads of program execution were reported as relatively low on average, about 7% for a range of benchmarking applications and 10% average performance overheads for the *SPECINT2000* benchmark suite. However, some applications incur particularly high individual overheads, with *176.gcc* incurring a 34.4% slowdown, as well as *253.perlbmk*, and *255.vortex* suffer from over 20% slowdown each.

Optimization is essential for reducing overheads from CDI-compliant code. The primary optimization for CDI compilation is the ordering of conditional branches in sleds. Since the sled will execute until an outgoing edge matches the potential target, CDI software is profiled for execution counts for the edges of each sled. This information is then used to order the sled branches by execution frequency, most frequent first. This assures, for profile-represented workloads, the desired target will be reached sooner on average. It has been demonstrated that optimization strongly impacts runtime overheads, as *253.perlbmk* and *255.vortex* suffer 189% and 143% runtime overhead, respectively for un-optimized sled execution. This serves to demonstrate that workloads which are non-representative of the profiled execution will likely suffer unacceptably high runtime overheads, and thus experience increased benefit from memoization of sled edges.

3.3 Architectural Acceleration of CDI

To directly address these important concerns with control-data isolated code, we propose integration of the CDI principle with secure and efficient processor support via novel architectural additions.

3.3.1 Caching Indirect Control Transitions

CDI-compliant code identifies, before runtime, all allowable control-flow transitions of an application. These control-flow transitions define the entire space of the allowable edges of the control-flow graph (CFG) of an application. This CFG is then encoded in the binary of an application as direct control flow edges for all control transitions. We enable the determination, at runtime, of the allowable targets of indirect branch of an application, without always executing the conditional branch sled which replaced it. This property of knowing the allowable control, at the beginning of a conditional branch sled, can allow skipping the execution of the sled and directly transferring control to the predetermined, allowed target.

In this work, we propose the inclusion of a new structure to be added to the processor: the *edge cache*. The goal of the edge cache is simple: when queried, verify a proposed target of an indirect instruction as adhering to the programmer-defined CFG encoded in a CDI-compliant application binary. This would enable the immediate selection of the allowable target without the overhead of executing the associated sled. The system integration of the edge cache is shown in Figure 3.2 with functionality shown in Figure 3.3.

In the operation of the edge cache, the application executes as normal until an indirect instruction is reached. At that point the edge cache is accessed for the potential control-flow transfer. If the cache verifies the control flow edge, the instruction is allowed to execute and the program continues as expected. However, if the edge cannot be found in the edge cache, the edge is considered to be unverified. As all indirect instructions in CDI-compliant code are backed by a sled of compares and direct jumps which validate all legal targets, execution simply advances to this sled.

When an indirect instruction is committed, two outcomes are possible. In the event that the source and target pair was verified by the edge cache, the instruction commits as usual.

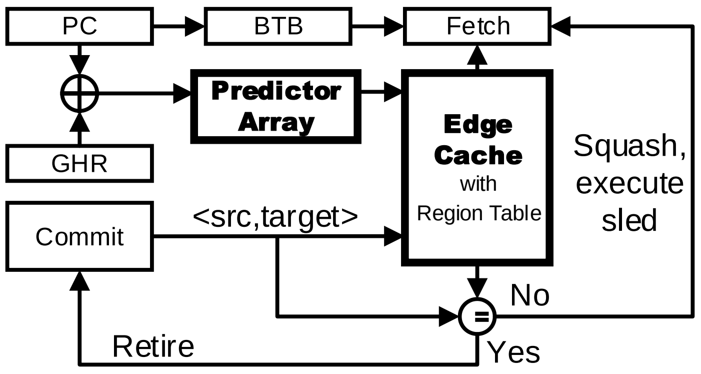


Figure 3.2 Edge Cache System Integration. The predictor array is accessed in parallel with the Branch Target Buffer (BTB) for predicting the targets of indirect control flow. Before an indirect instruction is committed, the edge cache is referenced. If the edge has been executed previously, it has been derived from the Control-Flow Graph (CFG) and is allowed to commit. If the edge cache misses on the edge the instruction is squashed and execution is directed to the sled for the indirect instruction.

In the event that the target was not verified, the address of the instruction is retained. In this case, whenever the next taken branch of the executed validation sled is subsequently committed, the address of that direct branch target is also retained. This pair of addresses then defines a validated control-flow edge for the indirect instruction, as the target must arise from a taken branch in the sled following the instruction. This edge (indirect PC and target address) is then added to the edge cache for verification of future instances of the edge.

3.3.2 Edge Cache Architecture

The cache is designed as a set-associative cache, and is indexed with the virtual address comprised of the `xor` of the source and target addresses of an attempted control-flow edge. Owing to the nature of the cache to assure secure control flow of an application, all address fields in the cache must contain the full 64 bits for each of the source and target addresses for an entry, shown in Figure 3.3. This is to prevent aliasing of malicious, attacker-injected addresses with valid entries contained within the cache. For replacement within a set of the cache, whenever an entry is accessed, the *useful bit* is incremented. Whenever a new entry is added, a victim is randomly chosen from the set of ways whose useful bit is not set. In the event all useful bits are set, all bits are reset and a victim is chosen at random. Whenever the edge cache is accessed, the full source and target addresses comprise the tag for matching the query. The size of the edge cache and associated structures is demonstrated in Table 3.1.

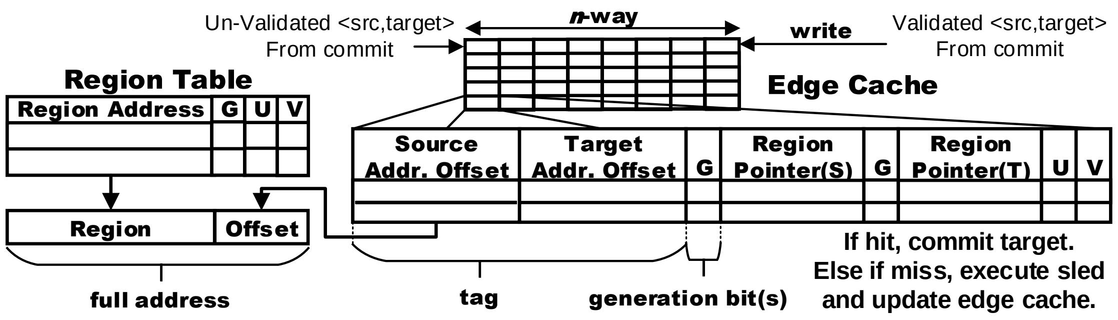


Figure 3.3 Edge Cache Validation of Indirect Edges. The edge cache enables memoization of indirect control flow edges to accelerate CDI-compliant code execution. The edge cache is an n-way, set-associative cache which, in conjunction with the region table, stores the source and target addresses of previously seen edges comprised of indirect instruction source addresses and direct, conditional branch targets executed from the sled. Whenever an indirect instruction (e.g., return or indirect call) instruction reaches commit, the source and target addresses are verified against control edge information stored in the edge cache and region table. The highest n bits of an address are stored in the region table to exploit address locality, while the lowest 64-n address bits remain in the edge cache. If the edge cache misses on the access, execution falls through to the conditional branch sled backing the indirect instruction. Otherwise the instruction is allowed to commit.

The edge cache is accessed whenever an indirect instruction is committed (e.g., `return` instruction). Since the potentially large conditional branch sled will be executed whenever verification fails, it is imperative to achieve a high hit rate when polling the edge cache. Factors which influence the hit rate of the edge cache include cache parameters such as size, associativity, and replacement policy. In the case of the edge cache, the point in the pipeline at which the cache is accessed can also have an impact on the rate of verification.

The immediate intuition is to poll the edge cache when an instruction is fetched, much like would be done in the case of a Branch Target Buffer (BTB). However, this policy can lead to excessive misses for the edge cache. At the time an indirect instruction is decoded, the instruction address is known but the target may only be speculated. This gives rise to targets which are not valid control-flow targets for a given branch. Such an instruction can never be confirmed by the edge cache, as it would not be derived from the conditional branch sled executed whenever an indirect instruction is not verified. This situation leads to more frequent executions of conditional branch sleds, as the mispredicted instruction will result in a fall through to the sled and the sled is guaranteed to complete execution (i.e., if the mispeculation could be discovered later, the sled will continue to execute). Further, when an incorrect target supplied by prediction is verified by the edge cache, but subsequently identified as mispredicted, the resolved target must still be verified.

To avoid the default of executing the conditional branch sleds on unverifiable edges, we instead poll the edge cache when an instruction reaches commit. A mispredicted indirect instruction is generally identified as such after the execute stage when the actual target becomes known. At the commit stage, the resolved target is always known and can be verified by the edge cache. This eliminates the execution of sleds when prediction mechanisms alias.

However, it also is the case that at commit, any unverified edge will result in execution of the conditional branch sled for the indirect instruction as well as result in squashing all instructions speculatively executed after the unverified edge. This increases the penalty per sled executed, but reduces the incidence of such executions. Given the length of longer sleds can exceed two thousand targets for some applications (such as *403.gcc*) we observed placing the verification of edges at commit to be more efficient.

To minimize the performance impact of squashing while executing CDI-enabled code, the edge cache is polled immediately at the beginning of the commit stage. This allows identification of mispredicted indirect instructions before the fetch stage is notified of the misprediction. At this point the resolved target is known and the edge cache can be polled for the instruction and target addresses. In the event that the edge is verified, the resolved target is allowed to execute and fetching begins at the target address. In the event that the edge cannot be verified, the instruction is not allowed to direct execution flow and the program counter is updated to the fall through to the first instruction of the associated sled. For instructions which are correctly predicted, the edge cache is also polled. If the instruction and target address pair are verified, then the instruction is allowed to commit as normal. However, if the edge cache cannot verify the edge the instruction is treated as a mispredicted instruction. Program control will once again be directed to the fall through address to execute the sled, identical to the procedure when a mispredicted instruction cannot be verified.

As highlighted earlier in Section 3.2, whenever an edge cannot be verified and a sled is executed, the edge cache must be updated. Execution of the sled initiates the recording of the PC of the instruction that initiated the sled, plus the PC of the valid target that the sled selected. When ultimately committed, this ¡source, target¿ address pair are used to update the edge cache. Note that, so long as instructions are committed in order, only a single pair may be in flight at any one time assuring that the target will come from the sled corresponding to a given indirect instruction.

3.3.3 Minimizing Edge Cache Storage

To eliminate any aliasing in the edge cache, the entire 64-bit virtual address for each source and target must be stored. To mitigate storage size, we include the addition of the region table. As identified by Seznec [124], targets of indirect branches possess address locality. We exploit this knowledge by sharing the highest ***n*** bits of addresses in the region table. Each source and target address field in the edge cache are replaced by a region pointer and region offset, moving the region address to the region table, as shown in Figure 3.3.

Due to the finite size of the region table, it is possible to have a stale region pointer which references an evicted entry in the region table. To avoid reconstructing an invalid address, we must flush the edge cache, region table, and predictor array whenever such an event would happen. To mitigate flushing, we add a generation bit to the region table and edge cache. Whenever an entry in the region table must be replaced, the generation bit is checked. If the bit is unset (0), the entry is replaced and the generation bit is set. If the bit is already set, the flush occurs. Whenever the region table is accessed, the generation bit is compared to that in the edge cache. If they match, then the region address is used and edge validation proceeds as normal. Otherwise, the entry in the edge cache is invalidated and execution is directed to the sled. The addition of a single generation bit greatly reduces the number of flushes for our benchmark applications by two orders of magnitude, with flushing occurring about every half-billion instructions.

We add a 32-entry, 4-way set associative region table. Each entry stores a 46-bit region address, a generation bit, a useful bit for replacement, and a valid bit. This reduces source and target storage requirements in the edge cache from 64 bits each to 18 bits for the region offset, 5 bits for the region pointer, and one generation bit. As shown in Table 3.1, this reduces the size for a 1,024-entry edge cache from 16kB in storage to 6.25kB, *a reduction of 61%*. The number of bits for the region offset is chosen by the resulting number of regions which must be cached. A region offset of 14 bits (matching typical page granularity) results in the number of regions exceeding 1,000 for some of our benchmark applications shown in Section 3.2. However, for a region offset of only 18 bits, this is reduced to 63. The final configuration with resulting storage sizes can be seen in Table 3.1 in Section 3.5.1.

3.3.4 Indirect Branch Prediction with the Edge Cache

Branch prediction has significant impact on runtime performance. Further, indirect branch misprediction constitutes a disproportionate share of the total cost of branch misprediction relative to conditional branches [77]. Unlike conditional branches where only two outcomes are possible, indirect branches can have large target sets. Work in indirect branch prediction is continually evolving and improving [56, 118, 124, 143, 51].

The edge cache is extremely efficient at verification of previously executed, programmer-intended control flow edges. This also creates an opportunity to use the edge information to enhance indirect branch prediction. Since the edge cache may only contain edges which are known to be valid (i.e., obey the compiler-encoded CFG of an application) these edges may be used as secure predictions for indirect control flow instructions. We leverage this in our implementation to predict targets for our sled-backed indirect instructions. We retain the usage of the *Return Address Stack* (RAS) to predict the targets of sled-backed `return` instructions, as the RAS is the most accurate prediction mechanism available for this task. Note that all indirect target predictions are ultimately subject to verification by the edge cache.

Whenever a potentially control-flow altering instruction is fetched the branch predictor is queried to determine the next instruction address to be fetched. At that time, the RAS is used for `return` instructions while the directional branch predictor and BTB are referenced for other instructions. For our sled-backed indirects, the RAS is used for the `return` instructions and the edge cache is referenced for predictions otherwise. The edge cache has an advantage over the traditional BTB prediction in that multiple targets can be stored for each source instruction address, with each potentially differentiated for prediction by the addition of history information. We leverage the *Global History Register* (ghr) available in existing branch prediction resources to incorporate path history information to enhance edge cache provided predictions.

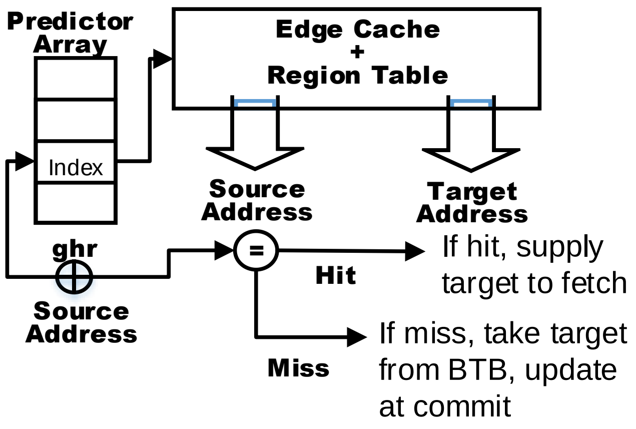

Predictor Array
Edge Cache + Region Table
Index
Source Address
Target Address
ghr
Source Address
=
Hit
If hit, supply target to fetch
Miss
If miss, take target from BTB, update at commit

Figure 3.4 Predictor Array-Based Indirect Prediction. The edge cache is leveraged to provide indirect branch prediction via the predictor array. This array is a direct-mapped cache which contains indices to the edge cache. When an indirect instruction is fetched, the predictor array is indexed with the xor of source address and global history register. The resultant edge is tag matched based on the source address of the indirect instruction and that of the source stored in the edge cache. Whenever an indirect instruction is retired, the predictor array is updated.

The use of the edge cache as a prediction mechanism, however, proves to be problematic for indexing the cache. The edge cache must provide a very high rate of verification, as we will highlight in Section 3.5. To achieve this, the edge cache is indexed using the simple `xor` of the source and target addresses of an edge, seen in practice to have both effective dispersion and low aliasing. Prediction is the tendering of the target, however, and thus the target address cannot be used to index the edge cache for the prediction. Source indexing, even coupled with history information, has lower dispersion and higher aliasing, which is intolerable as the penalty for executing sleds can be high.

We sidestep this issue with the addition of a predictor array. This structure can be indexed separately from the edge cache, satisfying the disparate requirements of verification and prediction. The predictor array is a direct-mapped cache which holds pointers to the potential predicted targets in the edge cache. As return instructions are over 85% of the dynamic occurrences of indirect instructions in our benchmarks, the number of entries in the predictor array can be much less than that of the edge cache. The storage requirement for each entry is also minimal, consisting of enough bits to point to any entry in the edge cache. For example, a 2,048-entry edge cache would require 11 bits of information for each predictor array entry. When accessing the set-associative edge cache, the source address is used as the tag. The predictor array is indexed by the simple `xor` of the instruction address and *ghr*, using the lowest ***n*** bits. That is a 256-entry predictor array would use the lowest 8 bits of the address `xor` *ghr* history. Performance of the predictor array is discussed in Section 3.5.

To maximize available resources, the edge cache and BTB are accessed in parallel for prediction. If the edge cache provides a target as described above, then that target is used for speculative execution. However, should the prediction request miss in the edge cache, then any target offered by the BTB will be taken. This reduces load on the BTB while still providing a second opportunity to predict the outcome of an indirect instruction. Since every control transition involving an indirect instruction must be validated at commit, the edge cache and predictor array can be update whether the target was predicted by the edge cache or the BTB. Both correctly predicted and mispredicted instructions prompt an update to the edge cache and predictor array, to accommodate both missed and correct predictions offered by the edge cache and BTB. Note that this does not apply to `return` instructions, which are predicted exclusively by the RAS. Consequently, `return` instructions only update the edge cache when they cannot be validated, and never update the predictor array.

3.4 Methodology

To evaluate the efficiency and efficacy of Control-Data Isolation-enabled hardware, we modeled the operation of an edge cache using the *gem5* simulator infrastructure [20]and the *SPECINT2000* and *SPECINT2006* benchmark applications.

3.4.1 Implementation Platform

We leverage the gem5 simulator to assess the impact of CDI-compliant applications and their acceleration using the edge cache. We use the O3 detailed CPU model in *gem5*, in conjunction with the simple memory model and execute applications in isolation using the syscall emulation mode. The processor is configured to match an *Intel Haswell* processor. That is, L1 instruction and data caches are both 32kB, 8-way associative, a 256kB L2 cache, and an 8MB 8-way associative L3 cache are also included. Cache line size is 64 bytes, with a 2.4 GHz processor. The target architecture is x86-64.

3.4.2 Benchmark Applications

To evaluate the performance of accelerating CDI-protected code, we evaluated the *SPEC2000* and *SPEC2006* Integer benchmark applications [130]. All applications were first compiled to be CDI-compliant using the *llvm* compiler infrastructure [86]. Some of the benchmarking applications, however, are not present in our analysis due to compilation issues with our *llvm* infrastructure.

3.4.3 Hardware/Software Co-Design

To develop the hardware-enabled memoization of valid control-flow edges we implement the CDI algorithm in the *llvm* [86] compiler infrastructure. Indirect control flow instructions are removed during the compilation flow, substituting conditional branch sleds and direct jumps and calls instead. In our work we follow the procedure outlined in the previous work by Arthur et al. [12] to eliminate indirection from our benchmark applications.

First, an application is compiled to a single llvm-IR using the *clang* compiler. The CFG for the application is then discovered, including function pointer analysis to determine a complete CFG for an application. This enables the determination of all permissible targets for each individual indirect control-flow instruction. At this point, each indirect control flow instruction is backed by a sled of compares and direct control-flow transition instructions which define the set of allowable edges for that instruction. However, unlike software-based CDI, we do not remove the indirect instructions. Instead, the sled is placed as the fall through for the indirect instruction, and is executed in the event the edge cache cannot validate the prospective edge for the indirect instruction. To handle the return from an indirect call instruction, a direct jump is placed between the call and the sled, which is also bypassed when execution falls through to the sled.

3.5 Experimental Evaluation

We evaluated our CDI-compliant implementation with several metrics to determine the efficacy of CDI-acceleration. These include runtime performance (i.e., overhead of executing CDI-compliant code), edge cache-enabled indirect branch prediction, and security analysis of the Control-Data Isolation policy.

3.5.1 Runtime Performance Analysis

The impact of executing the conditional branch sleds associated with control-data isolated code are the chief concern in achieving indirection-free, secure application execution. This is a primary purpose of this work: to eliminate any performance penalty arising from execution of conditional branch sleds substituted for indirect instructions. The results of the edge cache on mitigating sled execution are shown in Figure 3.5. A 1024 entry, 8-way set associative edge cache was used in conjunction with a 32-entry, 4-way set associative region table and a 256-entry predictor array to obtain the depicted results, the storage size of which is depicted in Table 3.1.

It can be seen in Figure 3.5 that the addition of the edge cache almost eliminates the runtime overheads of software-based CDI protection. The average speed up for all benchmarks is 0.995 from native, non-CDI compliant code (with indirection and without sleds). This means that executing CDI-compliant code, on average for all benchmarks, results in a slowdown of 0.5%. This is possible due to very high memoization rates for the edge cache coupled with edge cache-based branch prediction. The greatest challenge is represented by *176.gcc* and *403.gcc*. These applications result in slowdowns of 3.1% and 4.0% respectively. However, on some benchmarks we observe a non-trivial speedup over native execution, notably *400.perlbench* with a speedup of 0.983, an improvement of almost 2% over native execution. This happens due to the high density of indirect calls coupled with a non-trivial improvement in edge cache prediction over simple BTB-based prediction for this benchmark.

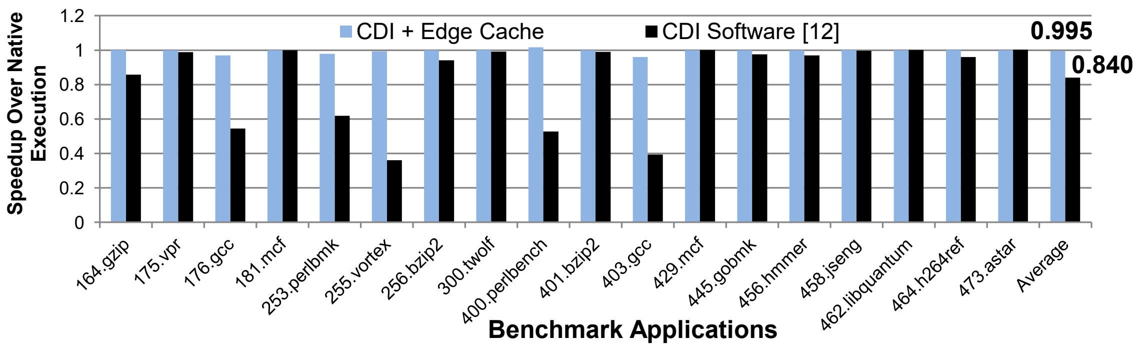


Figure 3.5 Speedup Over Native Execution. The performance of edge cache-accelerated CDI is shown, normalized to the baseline of native application execution (no sleds are executed). We see that caching control flow edges is highly effective at eliminating the overheads of sled execution, with an average of 0.995 times the application execution speed of non-CDI compliant code. Without the use of the edge cache, performance suffers with an average 19% slowdown. These results are achieved with the configuration depicted in Table 3.1: a 1,024 entry, 8-way set associative edge cache with a 32-entry 4-way set associative region table and a 256 entry predictor array.

Attribute	**Size in Bits**
----Edge Cache Size----	
Source, Target Address Offset	18 x 2
Region Table Ptr. for Source, Target Address	5 x 2
Region Table Generation Bit (Source,Target)	1 x 2
Useful Bit, Valid Bit	1 x 2
Total Bits per Entry	50
Number of Entries	1024
Total Size of Edge Cache	**6.25 kB**
----Region Table Size----	
Region Address	46
Generation Bit, Useful Bit, Valid bit	1 x 3
Total Bits per Entry	49
Number of Entries	32
Total Size of Region Table	**196 Bytes**
----Predictor Array Size----	
Pointer to Edge Cache	10
Number of Entries	256
Total Size of Predictor Array	**320 Bytes**
Total Size of All Components	**6.75 kB**

Table 3.1 Storage Size of Edge Cache and Components. The edge cache has 1024 entries at 50 bits per entry with a tag size of 32 bits. The region table contains 32 entries with 49 bits per entry and a tag size of 48 bits. The predictor array contains 256 entries with 10 bits per entry, direct mapped.

Without the edge cache, performance for software-enabled CDI protection results in an average speed up of 0.84, a slowdown of 19% for the average of all benchmarks.

Edge cache validation rates, shown in Table 3.2, strongly affects the overheads from

Benchmark	Edge Cache Validation Rate	Benchmark	Edge Cache Validation Rate
164.gzip	0.9999	401.bzip2	0.9999
175.vpr	0.9999	403.gcc	0.9964
176.gcc	0.9959	429.mcf	0.9999
181.mcf	0.9999	445.gobmk	0.9993
253.perlbmk	0.9998	456.hmmer	0.9999
255.vortex	0.9996	458.sjeng	0.9995
256.bzip2	0.9999	462.libquantum	0.9999
300.twolf	0.9999	464.h264ref	0.9996
400.perlbench	0.9999	473.astar	0.9999
		Average	**0.9994**

Table 3.2 Edge Cache Validation Rate. The edge cache validation rate has a large impact on the performance of CDI-compliant code. As such, a very high rate of validation is desired. For all benchmark applications, validation rate exceeds 99%.

CDI-compliant code. We can see an impact on *176.gcc* from a validation rate of 0.9959 results in program slowdowns of 3.1%. It was observed that for the same application, a drop in edge cache validation rate to 0.9795 arising from a direct-mapped cache with 1,024 entries results in a slowdown of 6.1% for the application execution. This is due to the high penalty of executing sleds, partly arising from the validation time at commit, which results in squashing speculated instructions at each fall through to a sled. Further reducing the edge cache size to 512 entries (8-way) incurs a slowdown of over 10% for the same benchmark.

Predictor Array Size	mpki	Runtime Speedup
32 entries	18.94	0.9237
64 entries	18.69	0.9464
128 entries	17.14	0.9987
256 entries	14.55	1.0170
512 entries	14.40	1.0179

Table 3.3 Predictor Array Sensitivity on Predicting Indirect Branches. Increasing the resources available to the predictor improves the prediction rate and subsequent application performance. Indirect branch mispredicts decrease appreciably until reaching a predictor array size of 512 entries. The benchmark shown is *400.perlbench*.

3.5.2 Indirect Branch Prediction Analysis

To further mitigate runtime overheads, we also leverage the edge cache using the predictor array. Since not all benchmark applications contain instructions such as indirect calls, we show prediction performance for a subset of applications which contain a non-trivial number of dynamic indirect call instructions in Table 3.4. Some benchmarks, such as *253.perlbmk* contain non-trivial quantities of indirect jumps arising from `switch` statements. These are eliminated at compile time using the `lowerswitch` *llvm* `opt` tool compiler flag, and thus we do not evaluate the prediction capability for such instructions. As a result, the mispredictions per thousand instructions (mpki) is naturally diminished merely by our compilation flow.

Benchmark	**Native mpki**	**edge cache mpki**	**CDI +BTB Pred. Speedup**	**CDI +edge speedup**
176.gcc	2.592	1.046	0.968	0.969
253.perlbmk	2.429	0.517	0.960	0.981
400.perlbench	19.226	14.549	0.953	1.0170
403.gcc	2.696	1.103	0.960	0.960
458.sjeng	1.291	0.340	0.989	1.000
464.h264ref	0.827	0.483	0.998	1.001

Table 3.4 Indirection Misprediction Rate of CDI-Compliant Code. Indirect instructions can be predicted leveraging the edge cache. In the process of creating CDI-compliant code, indirect jumps, such as those arising from switch statements, are converted to direct control flow automatically and are removed. Thus they are considered regular, direct control flow for CDI-compliant code, rather than sled-backed indirects. This changes the volume of indirect predictions, reducing mispredictions per thousand instructions (mpki) automatically. Benchmarks shown are those with non-trivial dynamic executions of indirect calls, which are the instructions predicted by the edge cache and predictor array. One particular benchmark, *400.perlbench*, has large dynamic counts of indirect call instructions, and thus benefits from the enhanced prediction of the edge cache.

We perform sensitivity analysis on the size of the predictor array using the *400.perlbench* benchmark application. We see in Table 3.3 that prediction accuracy continues to improve until moving from 256 to 512 entries, where doubling of predictor slots has a marginal effect. Note that prediction is achieved through the addition of a 320-byte predictor array (256 entries). For the same benchmark, increasing the number of BTB entries by 4X still underperforms compared to the predictor array.

3.5.3 Security Analysis

Control-Data Isolation reduces the software attack surface via the elimination of the weakness which enables contemporary control-flow attacks. However, the eliminated indirect control-flow instructions are then replaced with direct control transitions. Previous works have endeavored to assess the reduction in potential target space of Control-Flow Integrity compliant applications. An interesting measure for target space reduction was introduced by Zhang and Sekar, who proposed the simple metric average indirect target reduction (AIR)[150]. This metric evaluates the average number of allowable targets for individual indirect instructions. We adopt this metric for the CDI policy and compare our implementation to that of Zhang and Sekar, as shown in Table 3.5.

Benchmark	Bin CFI: CFI for COTS Binaries [150]	CDI Indirect Target Reduction	CDI Architectural Target Set Reduction
400.perlbench	97.89%	100.00%	99.999%
401.bzip2	99.37%	100.00%	99.999%
403.gcc	98.34%	100.00%	99.999%
429.mcf	99.25%	100.00%	99.999%
445.gobmk	92.20%	100.00%	99.999%
456.hmmer	98.61%	100.00%	99.999%
456.sjeng	99.10%	100.00%	99.999%
462.libquantum	98.89%	100.00%	99.999%
464.h264ref	99.52%	100.00%	99.999%
473.astar	98.95%	100.00%	99.999%
Average	**98.21%**	**100.00%**	**99.999%**

Table 3.5 Average Indirect Target Reduction. Zhang and Sekar [150] introduced the metric Average Indirect Target Reduction (AIR) which measures the percentage reduction of the average target space for instructions of an application. That is, if no policy exists to restrict indirect control flow, the average indirect instruction can target any byte in a binary, or 0% reduction. The AIR value reflects what part of the binary space cannot be targeted for an indirect instruction on average. This metric effectively represents the granularity of control-flow protection policies. CDI-compliant code completely eliminates indirect instructions, resulting in a 100% AIR value. However, in our work we allow indirects to execute edges encoded in their respective sleds. The target reduction for our work can be seen in the third column. This clearly is a reduced space, on the order of tens of edges per indirect on average as opposed to thousands in the case of course-grained CFI seen in [150].

3.5.4 Area and Power Analysis

We leverage the McPat [89] and CACTI [136] tools from Hewlett-Packard Labs to evaluate the area, power, and timing overheads from utilizing the edge cache and associated components. Estimation included output of the *gem5* simulation, in conjunction with the information contained in Table 3.1 for 32nm technology. Overall, the design yielding the performance shown in Figure 3.5 added an extra area of 0.443 mm2, an addition of 1% over the base design. The worst-performing benchmark, *403.gcc*, incurred less than 1% additional estimated power for the application execution. This low area and power overhead results from the small edge cache size in conjunction with the relatively rare dynamic occurrence of indirect instructions, about 1 in 100 on average for all benchmarks.

3.6 Extending Control-Data Isolation to Advance Path-Based Control-Flow Security

Control-Data Isolation (CDI) enforces, at runtime, the control-flow graph (CFG) of an application. Detailed in Chapters 2 and 3 of this dissertation, CDI eliminates contemporary control-flow attacks by subtracting the root cause: indirect control-flow. Due to the continual evolution of attacks, the research community must continue to pursue the elimination of vulnerabilities. Both recent [31] and past [80] works have identified the potential of attacks subverting the path of code execution while strictly adhering to the CFG of an application. As the underlying vulnerability in contemporary control-flow attacks is eliminated with CDI, the specter of path-based attacks advances to the forefront in reducing the software attack surface.

In anticipation of the future evolution of attacks, we introduce the non-speculative Return Address Stack (nRAS). This novel mechanism dramatically reduces the reliance on CDI compliant code to verify the target address of `return` instructions. Using a secure, non-speculative RAS, the precise edge of the CFG can reliably be determined. Leveraging this information, trust in the static CFG information encoded in the binary of the application, along with the memorization of CFG edges in the edge cache, can be avoided in 99.84% of all dynamic occurrences of the return instruction, as shown in Table 3.6. Thus hardening the software attack surface against path-based control-flow exploits like those detailed in [31].

3.6.1 Path-Based Attacks

Due to the relentless efforts of attackers in compromising software systems, researchers must continue to advance existing defense mechanisms. Control-data isolation is a defense mechanism which eliminates contemporary control-flow attacks, thereby closing the door on many software exploits. However, security research has identified that control-flow attacks could potentially advance to path-based attacks which do not violate the control-flow graph (CFG) of an application. The most recent example of this is a work by Carlini et al. [31] which demonstrates that an application could potentially be compromised without violating the programmer defined CFG. The revelation that some avenue of attack may be present within the confines of the CFG points to the next logical step in the evolution of control-flow attack defense: *control-data isolation which enforces programmer intended* *paths of execution.*

Whenever an attacker corrupts the target return address on the program stack, the sled of conditional branches and direct jumps used in CDI enforces the CFG of an application. However, an attacker may still force the execution of an edge in the CFG which is not the programmer intended edge in a given dynamic instance, so long as that edge is valid within the context of the CFG. This type of attack is outline in detail by Carlini et al. [31]. Thus, eliminating indirect instructions can not, taken alone, eliminate this type of attack. Essentially, an attacker could still exploit the *paths* of execution while still constrained to executing *edges* intended by the programmer. As such, the protections of CDI would not suffice in addressing such attacks.

3.6.2 Impossible Paths

Control-data isolation eliminates these attacks by eliminating the mechanism necessary to initiate such an attack, indirect control-flow instructions. However, both recent [31] and previous [80] works have detailed the likely evolution of control-flow attacks: path-based attacks which do not violate the CFG of an application, but still provide value to attackers. An example of such an attack is demonstrated by a simple call and return sequence of program functions. A given function may be called by, and subsequently return to, multiple caller functions. However, at runtime it should only return to the calling function for each given dynamic instance of the callee. An attacker may seek to subvert an application by corrupting data for the return instruction, but in such a way as to still target a valid edge of the control-flow graph. This results in the execution of an impossible path, as shown in Figure 3.6.

3.6.3 Advancing Control-Data Isolation with the Non-Speculative Return Address Stack

The impossible path shown in Figure 3.6 is permitted by the white-list of targets for the return instruction in the function baz in CDI-compliant code. This constitutes an attack on the sled of conditional branches and direct jumps which compromise the software implementation of CDI. The CFG of an application encodes all programmer-intended control edges in software, but does not exclude the impossible path seen in Figure 3.6, nor does it encode the valid path. To address this newly important vulnerability, we will detail a novel approach to the attempted execution of impossible paths and prohibit such paths whenever discovered. This is achieved through the addition of modest architectural extensions which

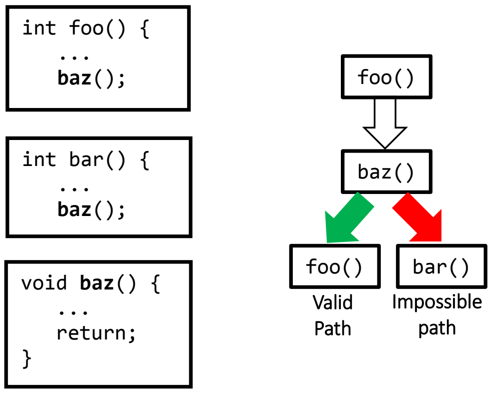


Figure 3.6 Impossible Paths. An attacker may subvert control-flow of an executing application by targeting an invalid address while still respecting the CFG of an application. Both functions `foo` and `bar` may call `baz`. However, in a specific instance of `baz`, control-flow must return to the calling function.

track the history of function calls and returns, at the instruction level, which is then used to verify control transfer from `return` instructions. Through the addition of a simple non-speculative Return Address Stack (nRAS), function return target addresses can be verified to be the correct, single allowable target. Utilized in conjunction with the existing CDI mechanisms, this can narrow the allowable target set from edges statically determined by the CFG to but a single allowable target at runtime.

3.6.4 Contributions of the Non-Speculative Return Address Stack

The goal of this extension to CDI is to reduce the software attack surface through the identification, and subsequent elimination, of infeasible paths at runtime. Thus, with the addition of the non-speculative RAS, the following further contributions are made:

- We demonstrate that simple hardware additions, along with existing CDI mechanisms, extends CDI protections to infeasible path executions.
- We demonstrate that keeping a non-speculative stack of return addresses can eliminate over 99% of all infeasible paths.
- Further, we demonstrate that the security of an application, as measured by the AIR metric proposed by Zhang and Sekar [150], is improved by more than four orders of magnitude for common benchmarking applications.

3.6.5 The Return Address Stack

To accelerate the execution of instructions, many processors use speculative execution. Rather than wait for the target address of a control-flow altering instruction, a speculative address is used to immediately continue fetching instructions. One mechanism to support speculative fetch of instructions is the Return Address Stack (RAS). The RAS is implemented as a fixed size stack. Whenever a call instruction is executed, the return address (the next instruction after the call) is pushed onto the stack. When a return instruction is fetched, the subsequent fetching of instructions is directed to the address popped from the return address stack. When the target is eventually determined from the program stack, it is compared to the speculative address supplied by the RAS. In the event of an incorrect prediction the speculative instructions are squashed, and execution is directed to the address indicated by the program stack. Various program events can cause the RAS to hold an incorrect value. These include limitations to the size of the RAS and programming paradigms which violate the matching of calls and returns. Examples are recursion which overflows the limited capacity of the RAS, and setjump longjump, which unwinds the program stack to an earlier program stack frame.

3.6.6 The Non-Speculative Return Address Stack

Earlier in Section 3.3.1, the edge cache [10] was introduced to enforce the CFG of an application at runtime. The edge cache is referenced at commit to validate prospective indirect control-flow edges, and the software“sled” was executed whenever the edge cache could not verify the to-be-committed control edge. Since in any case all control edges must be derived from the program instructions, all control-flow is determined *in advance* by the programmer. This, then, eliminates contemporary control-flow attacks.

However, future attackers may seek to exploit an application while adhering to the CFG of an application. In this event, the sled of conditional branches and direct jumps is subverted. The attacker, rather than choosing a target outside of the sled, chooses an alternate target within the sled. As our threat model allows the attacker complete control of data memory, we must assume this type of attack will eventually emerge. Recent literature [31] has established this type of attack as feasible and potentially desirable to attackers.

To directly address future attacks on the sled, and by extension the edge cache, the non-speculative Return Address Stack is introduced. Though the speculative RAS for target prediction has a high prediction accuracy, it does not offer any guarantee of a given target address. Thus, we introduce the non-speculative return address stack. With this minimal hardware addition to the design of a processor, we can conservatively guarantee the validity of the return addresses. That is, given a dynamic occurrence of a return instruction, we can determine whether the target is valid without the need to consult the edge cache. When this ability reaches high probability, we can proportionally eliminate attacks on the sled and edge cache of CDI.

3.6.7 Eliminating Control Path Attacks with the Non-Speculative Return Address Stack

The non-speculative RAS (nRAS) records the only allowable target address for a matching call/return instruction pairs. Whenever the target of a return instruction does not match the value recorded on the non-speculative RAS, it is a potential an attack on the control-flow of an application. With the non-speculative RAS, we can address such attacks.

However, the non-speculative RAS, just as the speculative return address stack, can not be guaranteed accurate *at all times*. To address this, we conservatively assert when the non-speculative RAS will be absolutely correct. To this end, there are three factors to consider in ensuring accuracy. These include speculative execution, overflow of a finite stack, and program control which violates the strict paring of calls and returns.

Speculative execution has a non-trivial impact on the accuracy of the RAS for return target prediction [128]. To address this, we implement the non-speculative RAS. Simply put, this is a RAS which is employed at the commit stage, when the edge cache is also interfaced. By checking the nRAS at commit, we eliminate changes to the stack by speculatively executed instructions.

The correctness of the stack can also be compromised by call stack depths which exceed the fixed capacity of the stack. The canonical example of this is the use of recursion. When the depth of recursion exceeds stack capacity, the stack can no longer guarantee validity of return addresses. To account for this, we conservatively consider that in the face of recursion which overflows the nRAS, the target addresses contained can no longer be guaranteed. In such cases we fall back on CDI, which still enforces the CFG of an application.

Not all programming paradigms adhere to strict call and return matched pairs. An example of this is the setjump, longjump pair. This programming concept, implemented in the C standard library with the `setjmp` and `longjmp` functions, is used to return to a stack frame arbitrarily distant in the program stack. Essentially, it serves as an indirect jump to a previous execution point in the instruction stream, so long as it is to a function with a valid frame on the program stack. Whenever this happens, the program stack is "unwound", and the intervening stack frames are eliminated. However, the RAS (and likewise nRAS) is not updated and thus is out of sync with the subsequent return instructions. It is important to note that programming constructs like setjump/longjump are not implemented special instructions, and can not be detected by observance of instructions at the commit stage. To address such stack unwinding, we propose the addition of the Stack pointer Address Stack (SAS) to detect when control flow violates call/return pairs, detailed below.

3.6.8 Detecting Stack Unwinding with the Stack Pointer Address Stack

To determine, with certainty, when the non-speculative RAS will contain the only allowable target of a return instruction, we introduce the Stack pointer Address Stack (SAS). A companion to the nRAS, the SAS will contain stack pointer addresses. Whenever a call instruction is committed, the return address target will be pushed on the nRAS. In parallel to this, the content of the stack pointer register is pushed on the SAS. This is now considered, within the context of the currently executing function, the line of demarcation between the current stack frame and the previous stack frame. Whenever an instruction is committed which modifies the stack pointer, the new address will be compared against the value on the top of the SAS, determined by peaking at the top of the stack without popping the address. In normal execution, the stack will grow downward in addresses, pushing and popping from the program stack. The stack pointer, generally, will not point to an address greater than that at the beginning of the currently executing function. Whenever an update to the stack pointer results in an address greater than that on the top of the SAS, it is conservatively assumed that the stack is being unwound, and that the non-speculative RAS will no longer contain values which are certain to match dynamic occurrences of return targets.

3.6.9 Dynamic Enforcement of the Valid Control-Flow Graph Edge

The non-speculative RAS is considered to contain the only allowable target address of a return instruction at execution. As identified in Section 3.6.7, accuracy of the non-speculative RAS can not be always guaranteed. Thus, we conservatively determine at what times the non-speculative RAS will be considered accurate. If, at any time, the previously detailed events undermine the accuracy of the RAS, the non-speculative RAS and SAS are flushed. This makes utilization of the nRAS simple: whenever a value exists on the nRAS, it is considered to be irrefutably accurate. If the target address of a return instruction, at commit, does not precisely match the address popped from the non-speculative RAS it is considered to be an attack. In such an event, an exception is raised and the program is halted to prevent the control-flow attack. The algorithm for the nRAS is shown in Figure 3.7.

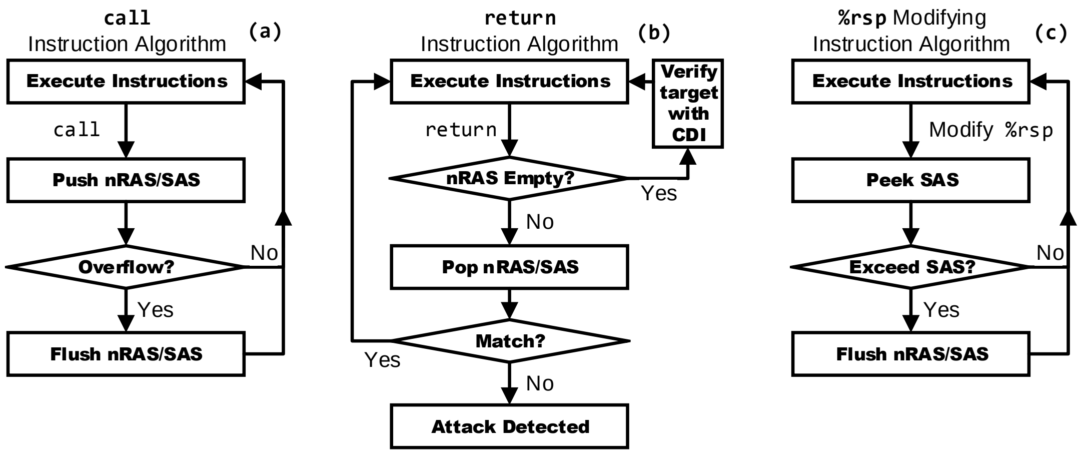


Figure 3.7 Algorithm for Non-Speculative Return Address Stack Operation. During execution the non-speculative Return Address Stack (nRAS), in conjunction with the Stack Pointer Stack(SAS), verifies return instruction address targets at the commit stage. (a) Whenever a call instruction is executed, the return address is pushed on the nRAS, along with the current stack pointer on the SAS. In the event the nRAS/SAS overflow, both structures are flushed and execution continues. (b) Whenever a `return` instruction is executed, the non-speculative RAS is checked. If the nRAS is empty, the potential target is verified using CDI. If the RAS contains an entry, it is considered the only valid target for the `return`. If the top of the RAS matches the potential target, the instruction is allowed to commit and execution continues as normal. If the top of the stack does not match the target, an attack is declared, and program execution halts. (c) When any other instruction which modifies the stack pointer (`%rsp`) is executed, the new stack value is compared to the top of the SAS. If the stack pointer exceeds the value on the SAS, the instruction is assumed to unwind the stack, and the nRAS and SAS are flushed.

3.6.10 Methodology of the Non-Speculative Return Address Stack

To evaluate the efficacy of the non-speculative Return Address Stack, we modeled the operation of the nRAS and the SAS using the *Pin* dynamic binary instrumentation platform [92] and the *SPECINT2000* and *SPECINT2006* benchmarking applications [130].

We leverage the Pin dynamic binary translation platform, using a custom Pintool to inspect all instructions and implement the algorithm defined in Figure 3.7. The benchmarking applications are compiled using gcc/g++, enabling the use of all benchmark applications which were incompatible with our *llvm* compiler infrastructure [86] used in Section 3.4.2. The applications were executed to completion using the reference input set, monitoring the performance of the non-speculative return address stack and the stack pointer address stack.

3.6.11 Experimental Evaluation of the Non-Speculative Return Address Stack

We evaluated our non-speculative RAS implementation for security and performance for CDI-compliant execution while reducing the attack surface for impossible paths. This includes the ratio of `return` instructions verified by the nRAS and the reduction in attack surface as determined by the reduction in the AIR metric [150] as shown in Section 3.5.3.

Table 3.6 shows the accuracy of the non-speculative RAS. For all benchmarks, there is a 99.84% reduction in the usage of the edge cache and sled to verify indirect edges for CDI compliance. This is a substantial reduction of the attack surface for exploiting impossible paths of execution. This is demonstrated in Table 3.7, where the Average Indirect target set Reduction is shown. Here, we have expanded the information found in Table 3.5 to include the number of targets, on average, which our nRAS optimized approach allows per indirect instruction. Here, we see an improvement of four orders of magnitude over related work [150]. The result is *an average of only 3.0 valid targets per indirect instruction.* It is important to note that not only are these targets only valid CFG edges, they are also a subset of the CDI sled, reducing the software attack surface by eliminating 99.84% of all impossible paths.

3.7 Related Work

Our efforts in accelerating Control-Data Isolated code are related and influenced by earlier works in control-flow security and branch prediction. A short list is presented herein and comprises software, hardware, and hybrid approaches.

3.7.1 Software Mechanisms

The primary related work is that of Arthur et al. in Getting in Control of Your Control Flow with Control-Data Isolation [12]. In this work, the authors introduce the central idea of control-data isolation. As described earlier in Section 3.2, this approach to control-flow security is a departure from previous approaches. As opposed to works such as CFI[1] and its descendants [37, 46, 110, 142, 150, 149], CDI advocates for the eradication of the root cause of contemporary control-flow attacks: indirect control flow. Key advantages of this work are the relaxed security model which assumes that an attacker may establish arbitrary control of data memory and the obviation for the need of associated, additional, protection

Benchmark	Number sled bypass	Number sled runs	nRAS overflow	SAS detect frame escape	Percent nRAS accuracy	Probability to avoid execution of sled for return
164.gzip	410,581,174	0	0	0	100%	1.0000
175.vpr	141,492,333	0	0	0	100%	1.0000
176.gcc	350,432,523	237,301	13,959	0	100%	0.9993
181.mcf	6,062,555	0	0	0	100%	1.0000
186.crafty	2,290,979,532	516,866	30,404	0	100%	0.9998
197.parser	4,264,999,987	23,147,046	1,361,591	0	100%	0.9946
252.eon	1,497,965,400	15,877	934	0	100%	0.9999
253.perlbmk	1,143,125,183	19,548	1150	1	100%	0.9999
254.gap	3,508,781,456	6,813,972	400,822	0	100%	0.9980
255.vortex	1,990,941,766	2,349,484	138,205	0	100%	0.9988
256.bzip	849,459,603	0	0	0	100%	1.0000
300.twolf	2,065,113,192	0	0	0	100%	1.0000
SPEC2000 avg.	**1,502,635,721**	**2,758,341**	**162,255**	**0.83**	**100%**	**0.9982**
400.perlbench	4,647,481,951	71,827	4,228	52	100%	0.9986
401.bzip2	1,328,875,062	0	0	0	100%	1.0000
403.gcc	977,241,911	755,258	44,427	0	100%	0.9992
429.mcf	399,208,096	8,584	505	0	100%	0.9998
445.gobmk	3,544,928,759	40,772,544	2,398,385	0	100%	0.9863
456.hmmer	1,395,740,543	0	0	0	100%	1.0000
458.sjeng	22,160,089,625	5,180,424	304,731	0	100%	0.9998
462.libquantum	971,815,011	0	0	0	100%	1.0000
464.h264ref	26,835,787,507	2,906	171	0	100%	0.9999
471.omnetpp	16,393,081,883	415	40	2	100%	0.9999
473.astar	9,376,154,736	0	0	0	100%	1.0000
483.xalancbmk	24,693,188,486	391,157,521	23,009,266	0	100%	0.9749
SPEC2006 avg.	**9,393,632,798**	**36,495,790**	**2,146,812**	**5**	100%	**0.9975**
Average (all)	**5,468,480,345**	**19,627,066**	**1,154,534**	**2**	100%	**0.9984**

Table 3.6 Non-Speculative RAS Performance. The performance of the non-speculative RAS (nRAS) and the Stack pointer Address Stack (SAS) is shown to be effective at the determination of a single target for return instructions. The number of sled runs and sled bypasses reflect the probability that the nRAS is considered to be accurate, and thus the only allowable target for the dynamic instance of the return instruction. The number of times the RAS was exceeded is found in the fourth column. The fifth column shows the count of the SAS detecting a change in stack pointer which indicates a potential stack unwind (e.g., longjump). It is important to note that one-hundred percent of the times the non-speculative RAS is considered to be the only source of the allowable target, it is correct, shown in the sixth column. Lastly, the methodology allows the precise, single allowable target to be determined 99.84% of the time, dramatically reducing the attack space from impossible paths. For all measurements, a nRAS and SAS size of 16 entries was used.

mechanisms. We seek to advance that work by establishing a secure platform to accelerate performance of CDI-compliant code.

A seminal work in the area of control flow security is Control Flow Integrity [1] by Abadi

Benchmark	Bin CFI: CFI for COTS Binaries [150]	CDI Indirect Target Reduction	CDI Architectural Target Set Reduction	CDI Architectural Target Set Reduction, with nRAS	Bin CFI: CFI for COTS Binaries[150], Average Target Set Size	CDI Architectural Target Set Reduction, with nRAS, Average Target Set Size
400.perlbench	97.89%	100.00%	99.999%	99.9999%	26,083	6
401.bzip2	99.37%	100.00%	99.999%	99.9999%	466	1.3
403.gcc	98.34%	100.00%	99.999%	99.9999%	61,954	11
429.mcf	99.25%	100.00%	99.999%	99.9999%	174	1.0
445.gobmk	92.20%	100.00%	99.999%	99.9999%	315,695	1.4
456.hmmer	98.61%	100.00%	99.999%	99.9999%	4,534	1.6
456.sjeng	99.10%	100.00%	99.999%	99.9999%	1,415	1.1
462.libquantum	98.89%	100.00%	99.999%	99.9999%	574	1.0
464.h264ref	99.52%	100.00%	99.999%	99.9999%	2,859	2.6
Average	98.13%	100.00%	99.999%	99.9999%	45,972	**3.0**

Table 3.7 Non-Speculative RAS Security Performance. Compared to previous work on course-granularity CFI [150] by Zhang and Sekar, the use of the nRAS in conjunction with CDI reduces the average allowable targets from tens of thousands to 3.0. Further, all targets in this work are valid edges of the CFG of an application. For all measurements, a non-speculative RAS and SAS size of 16 entries was used.

et al. The reportedly low overhead solution employs labels and checks to protect indirect calls and jumps. Whenever an indirect control flow instruction is executed, the label is read from the target location and checked against the label at the source. If the two match, the indirect instruction is allowed to execute. Return instructions are protected using a shadow stack which is used to verify the prospective return address from the normal program stack. In practice, CFI has a prohibitive runtime overhead. This resulted in subsequent works based on CFI to relax CFG edge constraints (expanding aliasing of target sets) resulting in course-grained CFI. Further, CDI relies on the shadow stack which must reside in data memory making it subject to potential attack.

Another important work in this area is Control Flow Integrity for COTS Binaries [150]. This paper by Zhang and Sekar demonstrated a robust implementation of course-grained CFI. This work brought overheads to marginal levels for SPEC benchmarks and was shown to work on more substantial applications as well. However, this work inherits the original issues relevant to the original CFI, including a more restrictive threat model. Additionally, as shown in Table 3.5, the cost of reducing overhead is greatly increasing the allowable edges for each indirect instruction. The reduction in runtime overhead in [150] is achieved by adopting a course-grained approach to CFI. Table 3.5 shows the large difference in the allowable targets, and thereby protection level, of our work in contrast to the work of Zhang and Sekar. Whereas our work averages tens of valid targets for each indirect instruction, their work allows thousands to tens of thousands of valid targets on average for indirect instructions.

Recently, other works have endeavored to demonstrate exploits which directly challenge CFI policy. Stitching the Gadgets: On the Ineffectiveness of Course-Grained Control-Flow Integrity Protection [47] demonstrates vulnerabilities of course-grained CFI policies of various works, including [37, 52, 98, 107, 150]. The authors are able to demonstrate the existence of a set of Turing-complete code gadgets, even under the most restrictive combined policy of the CFI works scrutinized. By enforcing a fine-grained control-flow security policy and eliminating the use of indirect control flow, both CDI and our work are not subject to the attacks delineated in [47].

3.7.2 Hardware Solutions

Our work is not the first to propose hardware-assisted protection mechanisms. A recent work in this space is Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation by Davi, Koeberl, and Sadeghi [46]. This work targets embedded systems, enforcing a CFI policy which uses labels to verify potential indirect targets. In their algorithm, call instructions are only allowed to transfer program execution to labeled call sites. Similarly, returns are only allowed to target call labels which are active (i.e., have been called but have not yet returned). Though lightweight, the solution is still more course grained than our work (e.g., any call may target any function). Also, in common with all CFI-type works, the methodology is subject to label spoofing attacks, which require additional assumptions about memory security (e.g., no code may execute from the heap). Fundamentally, all works which derive from CFI rely on the target of an indirect control flow instruction. As our work is based on CDI, rather than CFI, our approach does not suffer this weakness.

3.7.3 Control-Data Isolation in Contrast to CFI

The related works above [1, 37, 52, 110, 142, 150, 149] derive from the policy inheriting from Control Flow Integrity [1]. An essential differentiation between CFI and our work in Control-Data Isolation is the reliance on the targets of indirect control transitions to verify their integrity. That is, the target provides self-reported information (labels) which are matched with the indirect instruction. This has an inherent weakness to label spoofing which requires CFI to use a restrictive attack model to address. This cannot be mitigated merely by code protection, which could potentially protect existing code. Any code injected into the address space of the application could contain a spoofed label. That is, in CFI-related works, *a system must **never** allow execution of code which the programmer does not expect to exist.* In contrast, the security guarantees with CDI are fully contained and encoded in the indirect instruction, via the sled. In CDI no target is trusted and, so long as the application binary is not modified, any injected code cannot be reached. Simply put *CFI relies on no unexpected code appearing in the system while CDI makes all unexpected code unreachable.*

3.7.4 Indirect Branch Prediction

Due to the impact on performance, indirect branch prediction remains a current topic of research. Value based BTB indexing by Farooq et al. [51] uses compiler support to help increase indirect branch prediction accuracy. This work stores multiple targets for a given indirect instruction in the BTB, which are used in conjunction with hint instructions placed in the application binaries during compilation. Whenever an indirect instruction is executed, it is preceded by the hint instruction, which is used to access the correct target from the BTB. The authors demonstrate improvement over BTB prediction as well as tagged target cache prediction. Though our compiler infrastructure encodes the CFG into the binary with direct control flow, no hints are provided from the CDI compilation process.

In the work by Kim et al. [77], indirect calls are predicted as edges of multi-way branches, leveraging existng prediction mechanisms to model indirect calls as a collection of conditional branches. Though the authors demonstrate a performance improvement, the dynamic discovery of edges prohibits such a mechanism from being repurposed for security guarantees.

3.8 Chapter Conclusions

Control-Data Isolation presents a novel approach to eliminating contemporary control-flow attacks. In this work, we demonstrate that a fine-grained protection policy can be implemented with virtually no overhead. By eliminating all indirect control flow, program control must adhere to the programmer-intended CFG. Using the edge cache, memoization of safe control edges eliminates the overhead associated with executing direct control-flow sleds while mitigating the need for accurate profiling. We believe that accelerating CDI-compliant execution through architectural support can tear down the key remaining barrier to widespread adoption of CDI-enabled code that was execution overheads.

Chapter 4
Scalable Profiling for Likely Security Bug Sites

Adding manpower to a late software project makes it later.

The Mythical Man Month: Essays on Software Engineering
Frederick P. Brooks Jr.

Software bugs comprise the greatest threat to computer security today. Though enormous effort has been expended on eliminating security exploits, contemporary testing techniques are insufficient to deliver software free of security vulnerabilities. Control-data isolation eliminates contemporary control-flow attacks by removing a key component of the attack: indirect control flow. However, adversaries still retain the ability to control and corrupt data memory.

This chapter proposes a novel approach to security vulnerability analysis: dynamic control frontier profiling. Security exploits are often buried in rarely executed code paths hidden just beyond the path space explored by end-users. Therefore, this chapter details Schnauzer, a distributed sampling technology to discover the dynamic control frontier, which forms the line of demarcation between dynamically executed and unseen paths. This frontier may then be used to direct tools (such as white-box fuzz testers) to attain a level of testing coverage currently unachievable. Further, in this chapter demonstrates that the dynamic control frontier paths are a rich source of security bugs, sensitizing many known security exploits.

4.1 Introduction

In Chapter 2, we detailed Control-Data Isolation (CDI) [12] which severs the link between user data and the program counter (PC). Accomplished by the elimination of indirect control-flow, CDI eliminates contemporary control-flow attacks. However, the underlying defect which allowed an adversary to mount an attack, e.g., by subverting user data, still remains. An example is the stack smashing attack, detailed in Chapter 1 and Chapter 2. Though such an attack can not achieve arbitrary code execution in CDI-compliant code, the data on the program stack may still be corrupted by an adversary. This could result in a Denial-of-Service (DOS) attack, which denies service to a legitimate user when the program crashes due to the corrupted data on the program stack. Though much less powerful than arbitrary code execution, enabled by control-flow attacks, DOS and other attacks remain a potential threat in CDI-compliant code.

The tools of an adversary, e.g., the corruption of data memory, remain due to the underlying defects in software. These defects, whether injected inadvertently by design or fabrication, pervade in software and provide the root of the software attack surface. This chapter goes an important step further in the elimination of control-flow attacks; addressing the defects in software which provide adversaries the avenues to mount attacks.

The vast majority of security attacks are enabled by software bugs. Defects which escape detection of software quality assurance can have global impact, such as the *Code Red* and *Sapphire/Slammer* worms which utilized buffer overflows for system exploitation. Fueled by these and other high-profile exploits, buffer overflows remain a top security concern [101]. Programs written in popular languages such as C and C++ are a rich source of buffer overflow bugs, as these languages cannot, without high overhead, systematically eliminate buffer overflow vulnerabilities [42]. This then places the burden on test to find potential buffer overflow vulnerabilities before they are exploited.

Commercial software is heavily tested before deployment. Indeed, coding consumes only a small percentage of development effort [112], while studies have shown that testing comprises greater than fifty percent of the cost of software development [26, 84]. Regardless, software defects continue to escape detection.

Understanding the way in which latent defects are exploited can reveal critical insight into their prevention. The majority of security-related faults reside in the least likely to be executed code sequences, and by extension, the least tested portions of code [81]. In an effort to heighten initial customer satisfaction, developers tend to focus their limited test resources on the code paths they anticipate users will execute most often, creating significant overlap in developer test and user execution. This in turn shapes a common discovery model used by attackers to locate defects. A malicious user will provide permutations of typical application inputs in an effort to cause slight (but expected) deviations from the well-travelled, and thus well-tested, path of normal execution. Given the combined nature of testing and exploitation discovery models, the location of defects most likely to be exploited can be identified. This exploit-rich code exists just beyond the well-trodden execution paths of testers and users, yet is readily reachable by attackers. We identify these locations as the ***dynamic control frontier (DCF)***[11].

The dynamic control frontier is a collection of paths rooted in dynamically executed paths. However, these paths are special in that, had the final control decision in these paths executed a different basic block, it would create a new, never-before-seen path. This defines the frontier of the path space executed by an application with respect to a set of inputs. Collectively, the DCF represents the most readily accessible paths of execution which are unlikely to be executed by end-users; consequently, these paths have a high degree of reachability for an attacker. Accordingly, any latent defects in the unexecuted portions of the dynamic control frontier paths are unlikely to be found by users and developers, but these bugs can be quickly uncovered by attackers.

It is interesting to look at the dynamic control frontier of an application arising from the test inputs of developers. Indeed, in this Chapter it will be shown that this is valuable as we find real vulnerabilities at these locations. However, it is more intriguing to examine the dynamic control frontier for a non-trivial sized population of end users. An attacker is most interested in this frontier as it represents code paths which have not been tested nor executed with any frequency by any user of a particular program. In contrast, any paths frequently executed by users which are not represented in the test suites will probably be devoid of showstopper bugs, as users would otherwise complain. As such, in the construction of a system to profile the DCF, we must be mindful that such a system should analyze the DCF of a large population of users without imposing an unacceptable impact on individual user performance.

4.1.1 Contributions of this Chapter

The goal of this work is not to fix software bugs which drive security exploits; existing tools will be utilized for this purpose. Our goal is to instead show such tools, which often suffer from exponential path explosion, where they can best focus their efforts to find real-world, mission-critical security exploits. This goal merits the works namesake: schnauzer. Utilized by law enforcement, emergency responders, and medical professionals the schnauzer is a working dog that is exceptionally capable of locating critically important items (illegal drugs, missing persons, etc.). The schnauzer does not actually find the desired item; it instead zeroes in on the locations where its human partner should search – the perfect metaphor for our work.

The value of the DCF is not to identify code paths with the highest density of bugs. The value of the DCF is to identify the code paths which are least tested by developers and users, while also most readily accessible to attackers. We will demonstrate that there is mounting evidence that bugs hidden within the DCF are more likely to be exploited, and therefore are of the greatest merit to discover.

This chapter builds upon the works in Chapter 2 and Chapter 3 in leveraging control-flow to eliminate contemporary attacks. Previous work in CDI, both software [12] and hardware [10], eliminate malicious control-flow essential to contemporary control-flow attacks. However, the defects in software which give adversaries the requisite control over data memory, also a necessity for control-flow attacks, still remain. This chapter directly addresses the underlying software defects which give rise to the powerful adversary identified in the threat model detailed in Chapter 2. Whereas CDI eliminates the direct injection of malicious data into the program counter, this chapter details a novel mechanism to identify the defects facilitating the corruption of data.

The primary accomplishment of this chapter is the development of a low-overhead, efficient software mechanism that effectively identifies the dynamic control frontier over a large population of users of an application. This frontier is the line of demarcation between frequently executed code paths, and those paths which are untested, pinpointing the paths of execution most likely to harbor future security exploit-enabling bugs. The majority of the work in this chapter is derived from the publication “Schnauzer: Scalable Profiling for Likely Security Bug Sites” [11]. In this work, we make the following novel contributions:

- Presentation of an effective, scalable, and decentralized approach to identifying the dynamic control frontier for a program running across a large population of users.
- Presentation of a software implementation for harvesting dynamic control frontier information from individual user machines. The approach utilizes dynamic code instrumentation to limit the impact to application execution while providing appropriate coverage of the dynamic control frontier in the aggregation of users.
- Demonstration of the value of the dynamic control frontier by showing that many known security vulnerabilities may be found there. We show that dynamic control frontier paths sensitize known exploits identified by the NIST National Vulnerabilities Database.
- Evaluation of the effectiveness of the approach by exploring the performancecost tradeoffs while harvesting DCF paths. Also, herein is developed a novel whole-path

analysis technique that allows us to gauge the coverage of the approach (i.e., the total percent of dynamic control frontier paths found as a function of total population run time). We present results for a wide range of non-trivial software packages that show our approach achieves good coverage while keeping performance impacts low.

This chapter, Chapter 4, encapsulates the following work. Section 4.1.2 provides an in-depth overview of the dynamic control frontier. Section 4.2 details our DCF profiler, Schnauzer. Experiments conducted to evaluate the benefits and costs of DCF profiling, and a full analysis of the results are delivered in Section 4.3. Finally, Section 4.4 lists related works while Section 4.5 gives chapter conclusions.

4.1.2 Dynamic Control Frontier Discovery

Security exploits arise from bugs which escape detection by the developer. Often, hidden bugs only appear when sensitized by the proper path [26]. For example, attempting to free a pointer after already doing so previously (double free). The predominance of path-sensitized bugs follows from the observation that commercial software generally achieves both branch and code coverage but remains deficient with respect to path coverage. Unfortunately, achieving path coverage is currently an intractable problem for applications of any appreciable size. This is due to the explosion in the number of paths, ultimately limiting path testing to a tiny subset [103].

Given the combination of path explosion and the need for path sensitization to activate bugs, it is inherent that exhaustive testing to locate bugs is an infeasible approach [152]. Thus, software testers are forced to constrain the path space to some feasible subset [26]. The quandary of test allocation, or the optimal test resource allocation problem (OTRAP) [103], is generally approached from the perspective of software reliability and cost [71, 140], rather than security. Identifying the subset of paths which are likely to contain bugs, which are in turn likely to be exploited, would yield the highest productivity in test relevant to potential exploit detection.

This subset of paths, deemed highly likely to result in exploits, is encompassed by the dynamic control frontier. The dynamic control frontier represents the border of dynamic execution between dynamically seen paths and those which are unseen. The first unexecuted basic block of these dynamic control frontier paths represents a location that is likely to hide a security exploit.

Consider the known security bug modeled in Figure 4.1, a high-level representation of an exploit discovered in *OpenSSL*. The bug, documented in the National Vulnerability Database

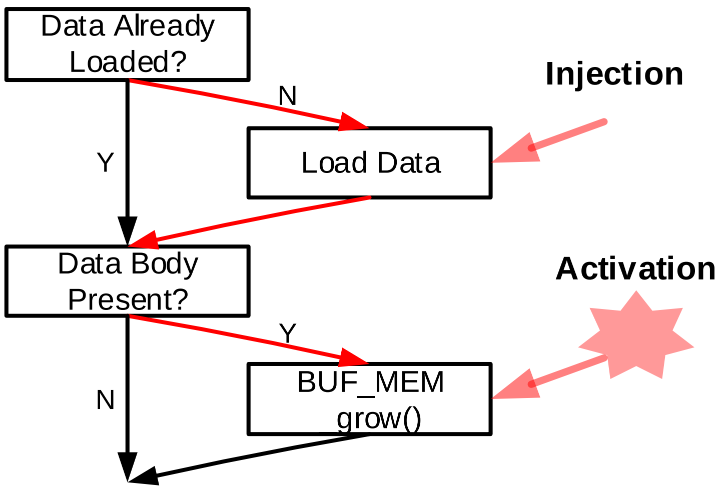


Figure 4.1 Path Sensitization Represented is the high-level overview of a security bug from the National Vulnerability Database (CVE-2012-2110 and CVE-2012-2131) for the OpenSSL application. In experimentation, this vulnerability was found to be sensitized by the dynamic control frontier. Here, a buffer overflow attack results from crafted data of an RSA public key. Note that the vulnerability is only sensitized by a single path (N, Y), indicated in red.

[100] and which enables a buffer overflow attack, is only sensitized by a specific path of execution. When handling DER encoded data, maliciously crafted data can activate the vulnerability. Note that the (N, N) (Y, N) and (Y, Y) paths were seen with some frequency while running *OpenSSL ssltest* and do not sensitize the bug, but the path (N, Y) was not seen, and hence represents the DCF where the bug was sensitized by the untested path.

Paths which remain unexecuted, that are not comprehensively tested, will continue to harbor latent bugs. This space, all un-executed paths, is still far too large for comprehensive testing. However, the code paths which are immediately outside the dynamic execution are the ones which are the most readily reachable by attackers. Because the dynamic control frontier is unlikely to ever be executed, the security bugs in this code will typically only be fixed when an active exploit is exposed. Debugging the DCF will force any attacker to probe deeper into the code. This will raise the bar in terms of the amount of effort required to attack programs, and make it much more difficult for an attacker to find good security bugs. In Section 4.3 we show that the number of dynamic control frontier paths is relatively few and quite rich in security exploits.

As defined in Figure 4.2, the dynamic control frontier of an application are the sets of length-*n* paths, comprised of basic blocks, where the first *n-1* blocks have been seen to be executed, but the full series of *n* basic blocks has not been seen to execute. Thus, the dynamic control frontier is a path in which the last control decision to basic block bb_n creates

$$DCF(P) = \{p_i, p_j, \ldots\ p_m\}$$
$$p_i = \langle bb_1, bb_2, \ldots, bb_{n-1}, bb_n \rangle \mid$$
$$\langle bb_1, \ldots, bb_{n-1} \rangle \in EX(P)$$
$$\wedge \langle bb_1, \ldots, bb_{n-1}, bb_n \rangle \notin EX(P)$$
$$EX(P) = \{\ \ldots all\ paths\ executed \ldots \}$$

Figure 4.2 Dynamic Control Frontier. The dynamic control frontier of an application P, DCF(P), is the set of paths *p*, comprised of a series of *n* basic blocks. These paths consist of basic blocks, where the first *n-1* basic blocks were in fact executed in EX(P), but the whole series of *n* was not.

a never-before-seen path of execution. Basic block bb_n is the likely site of a security exploit, sensitized by the path leading to it (bb_1,..., bb_{n-1}). More formally, the dynamic control frontier is defined as follows. The dynamic control frontier DCF(P), of an application P, is the set of paths p comprised of a series of n basic blocks. These paths consisting of n basic blocks, where the first n-1 basic blocks form a path (of length *n-1*) in the set of executed paths, EX(P), but the full length-*n* path of basic blocks is not contained in the set of executed paths EX(P).

4.1.3 Computing the Dynamic Control Frontier

Determining the exact dynamic control frontier, which we call the ***ground truth DCF***, for a given application execution can be accomplished by analyzing its execution trace. The ground truth DCF computation method is given in Figure 4.3. First, a trace of basic blocks is collected for an application in execution with respect to a set of inputs. This trace is then scanned for all length-*n* paths of basic blocks. These paths are sorted into sets by their length-*n-1* path prefixes. For each path within the set of paths with common path prefixes, if there exists any control exit from the *n-1* block (the last block of the path prefix) which is not represented in the set, then this path prefix, along with the unseen control exit block, is a member of the set of ground truth DCF paths.

4.1.4 Profiling the Dynamic Control Frontier

Establishing the ground-truth set of dynamic control frontier paths, needed to provide good coverage of the dynamic control frontier paths for an application, is prohibitively expensive

$$EX(P) = \{ \dots all\ paths\ executed \dots \}$$
$$GTDCF(EX(P)) = \{ \dots all\ ground\ truth\ DCF\ paths \dots \}$$
$$p_x = \langle bb_1, bb_2, \dots, bb_{n-1}, bb_n \rangle$$
$$pp_y = \langle bb_1, bb_2, \dots, bb_{n-1} \rangle$$
$$PREFIX(p_x) = \langle bb_1, bb_2, \dots, bb_{n-1} \rangle \in p_x$$
$$EQ(p_x, pp_y) = PREFIX(p_x) \leftrightarrow pp_y$$
$$(\forall pp_s \in EX(P))[\exists p_t[EQ(p_t, pp_s) \wedge p_t \notin EX(P)] \rightarrow p_t \in GTDCF(EX(P))]$$

Figure 4.3 Ground Truth Dynamic Control Frontier. The ground truth DCF of an execution instance EX(P) of application P, GTDCF(EX(P)), is a set of paths *p*, comprised of a series of *n* basic blocks. These ground truth paths are those where their length-*(n-1)* path prefix was executed, EX(P), but their entire length-*n* paths were not.

to do widely. Analyzing an execution trace assumes a finite application run. Also, such a trace grows to unmanageable size after long execution periods. For example, collecting a trace consisting of purely conditional branch information, limited to instruction address and branch direction, while executing the *SQLite* test suite *quick test* accrues over 300 GB of data during ground truth analysis of a 154 billion instruction length execution. Further, keeping track of all potential DCF paths during a programs execution is a significant performance overhead; for example, the ground truth DCF analysis of *SQLite* using *Pin*-based instrumentation [92] resulted in an average application slowdown of 26×.

Thus, a practical method must be developed to profile an application for the ground truth DCF. This can be achieved by sampling a small subset of the paths executed by an individual user and combining these samples over a large user population. While observing the execution of an application, at occasional intervals, a path is selected for profiling. A hypothesis is made from the length-*(n-1)* prefix seen in execution and the length-*n* path derived from this prefix which is not seen (i.e., the hypothesis is constructed by taking the opposite branch direction out of the last control decision seen to execute). We then hold this hypothesis for an extended period of time, waiting to see if the path is executed, and thus the hypothesis refuted. If the hypothesis path is not seen to be executed for this holding period, it is considered a good candidate for a DCF path. If, however, the hypothesis is seen to be executed, the hypothesis is refuted and not considered further.

The dynamic control frontier can be established for any single execution of an application. However, this frontier will vary depending on the inputs to an application for a given instance. Consequently, the DCF discovered for a single user is of limited value. A user may run the application with inputs which ultimately refute a hypothesis considered a good candidate by another user. Potential DCF candidates are therefore collected into a single ***global path filter*** database which is shared with all users over time.

Initially, dynamic control frontier path hypotheses will be sampled by multiple users. If a path is refuted, it will be removed from the global path filter. Otherwise, as hypotheses in the global path filter age they come to represent true dynamic control frontier paths. These venerable DCFs can then be used to filter hypothesis creation on individual hosts as profiling these high-confidence DCF paths would provide no benefit. Figure 4.4 depicts an overview of DCF sampling.

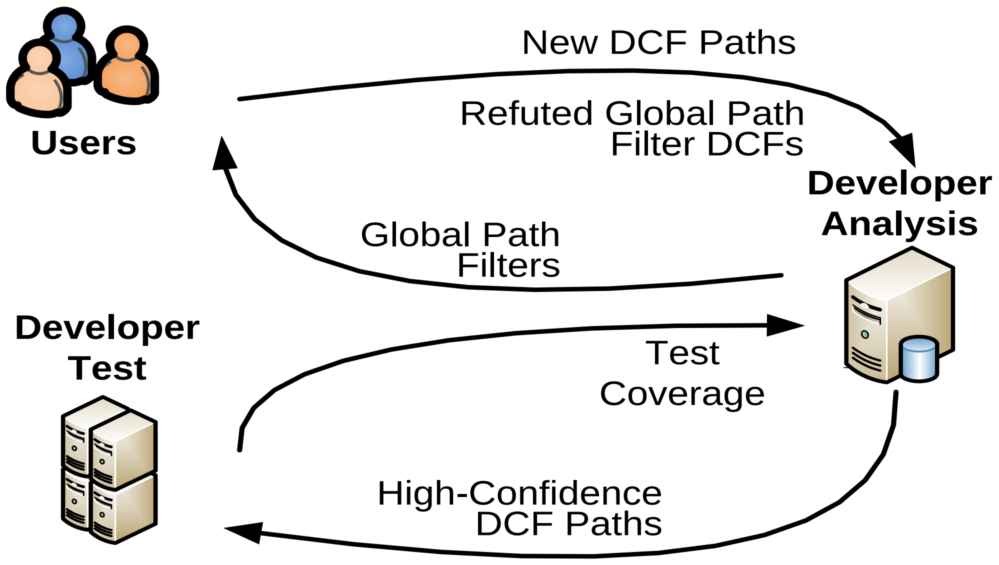


Figure 4.4 Dynamic Control Frontier Sampling System. Users profile application execution while sampling to discover dynamic control frontier paths. These DCFs drive developer analysis, which directs testing methods. Global path filters coordinate work between users.

4.1.5 Leveraging the Dynamic Control Frontier

Once dynamic control frontier paths begin to materialize, they must be harnessed to find security vulnerabilities. We can use white-box testing to deeply analyze DCF paths for security vulnerabilities. ***White-box testing*** has emerged as an effective testing approach to overcome this limitation [60]. White-box testing is designed to fully explore the dynamic control flow within a code module. The approach essentially is the reverse of fuzz testing. Rather than buffeting the code with random inputs in the hope of exposing new code paths, the approach instead selects a specific code path for testing and then uses SAT-based tools to deduce the inputs to the program or function that would cause the path to execute.

White-box testing has offered the ability to improve fuzz testing by a considerable margin [61], however, the approach still has limitations. For any non-trivial program, the number of paths that must be explored by white-box testing quickly overwhelms the computational capability of existing tools. For example, if the code in Figure 2.1 is embedded within a loop, the number of paths will be exponential with the number of loop iterations, e.g., at 1000 iterations the number of unique paths is 2^{1000}.

DCFs have the potential to become a divining rod for white-box testing tools, showing them where to spend their efforts to search for vulnerabilities. The DCF instructs which path to follow to reach the likely bug site; the SAT engine typically found in white-box testing tools can determine the inputs necessary to execute the DCF path (or determine that it is an infeasible path of execution). It is interesting to note that a key insight from white-box testing is that bugs are not far from the path of execution; they are just out of reach. Dynamic control frontier profiling leverages this same insight by identifying code just beyond the demarcation of executed code. To effectively expose bugs, attackers must explore code that is not executed ***by any user***. As such, ***there is much promise to improve security vulnerability analysis via white-box testing by identifying dynamic control frontier code paths over a large population of users machines.***

4.2 Schnauzer: A Distributed DCF Profiler

To validate our distributed approach to profiling the dynamic control frontier, Schnauzer was built. Our profiler was implemented as a client tool utilizing the *DynamoRIO* dynamic instrumentation tool platform [23]. The goal in developing Schnauzer was to push DCF analysis into the user space by using sampling to demonstrate the potential to minimize runtime overheads associated with DCF profiling, all the while achieving coverage of the ground truth DCF.

Efficiency is critical when profiling in the user space. Thus, profiling at the abstract level of the basic block is undesirable. As such, the conditional branches of an application are preferred to model paths for the dynamic control frontier. Conditional branches are chosen as they may be observed directly from the execution stream, unlike basic blocks. Furthermore, they provide an elegant representation of the control flow of an application, reducing the amount of information necessary when compared to basic block analysis. ***A dynamic control frontier path is simply then a path derived from a length-n executed path***

of conditional branches in which the trailing conditional branch only goes one direction in this case the length-*n* DCF path is the same path that exits in the opposite (and yet unseen) direction.

The *DynamoRIO* implementation of Schnauzer, shown in Figure 4.5, works as follows. An application begins unmodified execution through *DynamoRIO*. At random, bounded sampling intervals, the next *n* conditional branch edges are instrumented. At each branch edge seen during execution, a few assembly-level path tracking instructions are added, as shown in Figure 4.6, to record the occurrence of the path to a memory location when the edge is re-executed. Upon reaching the last branch in the length-*n* path, the unseen edge of this last branch becomes a potential node on the dynamic control frontier. A function call, referred to as the refuting instrumentation, is inserted at this edge which, when called, invokes a routine in our *DynamoRIO* client to evaluate the path leading to the edge. This path/node combination constitutes a DCF hypothesis, as it has not yet been seen during execution and the path formulation information for this hypothesis is recorded.

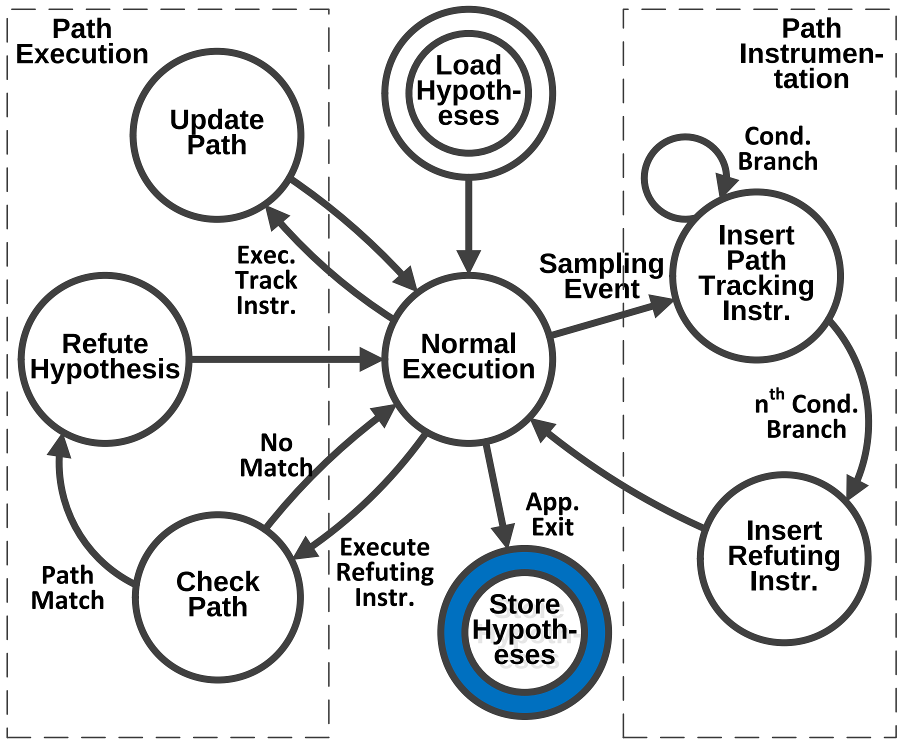


Figure 4.5 DynamoRIO DCF Profiling Client. The application executes unmodified until a path is selected for profiling, at which time lightweight instrumentation is added only to the selected path. This instrumentation updates the path history when executed. In the event the last edge of a hypothesis is executed (the refutation instrumentation), the path history is checked for a match. Hypotheses may persist across application executions.

The application then continues to execute uninterrupted. If at any time the refutation instrumentation call is invoked (i.e., the previously unseen branch edge from the last branch in the hypothesis is taken), the function will compare the recorded incoming path to the hypothesis' path prefix to determine if the path leading up to the edge matches that of the current hypothesis. If the dynamic path matches the hypothesis, then that hypothesis is refuted, and the *DynamoRIO* code cache is flushed to remove the potential DCF hypothesis. By only instrumenting paths which are hypothesized to be DCF paths, overheads remain low.

If after some long period of aging time a hypothesis has not been seen in the execution trace, this hypothesis is considered confirmed. At that time, it is added to the set of dynamic control frontier hypotheses, which will be reported en masse to the developers at a later time. Before any new hypothesis is formed, it is checked against the global path filter plus the internal list of recently recorded DCFs to avoid duplication of effort. Our client also loads and stores hypothesis and global path filter state whenever profiling is invoked. Accordingly, profiling persists across an arbitrary number of application executions. The work of confirming hypotheses, as well as initiation of random sampling, is performed by a separate thread of execution created within *DynamoRIO*. This allows such work to be completed without slowing the target application.

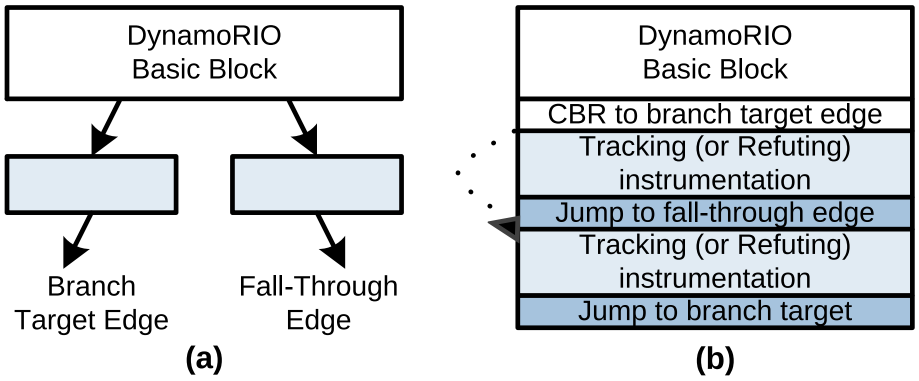


Figure 4.6 DynamoRIO DCF Profiling Dynamic Instrumentation. DynamoRIO basic blocks (a) are instrumented with assembly-level instructions inserted only on branch edges for paths being actively sampled. (b) Shows the layout of new basic blocks with instrumentation. Note that for a single active hypothesis, only the relevant subset (tracking or refuting) would occur, and only on a single edge for each conditional branch in the hypothesis.

4.3 Experimental Evaluation and Results

To fully understand the benefit of the DCF, both the cost and accuracy of DCF profiling were evaluated.

4.3.1 Benchmark Applications

Benchmarks were carefully selected to represent commonly used programs. These programs are popular, network facing applications which increases their profile to attack. Additionally, we sought out programs that had access to high-quality test suites, especially fuzz testers, such that DCF path profiling could run for extended periods of time to locate the code that developers (knowingly or not) chose not to test. The *OpenSSL* (1.0.1c) toolkit, *Python* interpreter (2.7.1), *Tor* (The Onion Router 0.2.2.37), *InspIRCd* Internet Relay Chat server (1.1 and 2.0), and *Pidgin*(2.10.4) executed the regression test suites with their respective distributions. The *SQLite* (3.7.7) benchmark was executed with the fuzz testing components of the standard tcl test library. The *tshark* network analysis tool (1.6.0) was tested with the fuzz test generation tool included with the *tshark* distribution.

4.3.2 Experimental Framework

The testing platform consists of 64-bit x86 servers running *Ubuntu* 11.04 *Natty Narwhal* with Linux kernel 2.6.38-10-generic. All path information was gathered using either the *DynamoRIO* [23] or the *Pin* [92] binary instrumentation tools to instrument benchmark applications. There are four major variables relevant to DCF profiling; path length-n, sampling interval, hypothesis age threshold, and the number of concurrent hypotheses for a given analysis. Of these, path length has a direct relationship with the DCF, while the other three are sampling parameters.

Since bugs are often sensitized by a particular path, the DCF has an important relationship with path length. The bug represented in Figure 4.1 would not be sensitized by a path length of 1 (branch coverage), as all branches involved see both edges in normal execution. This yields no ground truth DCF paths, as described by Figure 4.3, and the bug would therefore escape detection by DCF profiling. This observation motivates the desire for longer DCF paths. However, as path length grows, the odds of the same path executing again reduces, potentially resulting in the DCF becoming the set of all paths. To determine the optimal path length for DCF profiling, the relationship between path length and known security defects was explored. This analysis, shown in Section 4.3.6, determined that a path length of 4 was most effective. For this reason, the subsequent experiments were conducted with a path length of 4 conditional branches.

To further reduce the runtime overhead due to instrumentation, long intervals of time can elapse between hypothesis formulation and the aging threshold. In all overhead and coverage experimental results shown, the sampled hypothesis formulation period is randomly distributed between 1 and 100 milliseconds of instrumented program run time while the hypothesis aging period is 10 seconds. These values were found to facilitate an effective coverage rate while maintaining accuracy of profiled dynamic control frontier paths with respect to the ground truth DCF.

The number of concurrent path hypotheses is limited to a single hypothesis. While imposing the lowest overhead, a single hypothesis also limits sampling capacity. Later in Section 4.3.5, we show that a single hypothesis is virtually as effective as multiple concurrent hypotheses in establishing coverage of the DCF ground truth.

4.3.3 Ground Truth Dynamic Control Frontier

A custom pintool was created to perform whole-path analysis of a program to discover all of the dynamic control frontier paths. The whole-path analyzer generates the entire conditional branch trace for all of the programs test inputs. We then scan this trace for all unique length-n paths, and then rescan the trace to determine which of the discovered paths exit in only one direction. The opposite exit of the paths' prefix constitutes the complete set of DCF paths that our sampling system could discover, and these paths form the ground truth necessary to gauge coverage of the proposed sampling mechanism. Table 4.1 shows the application trace and ground truth DCF set size for all benchmarks. The number of ground truth DCF paths is seen to be very few when compared to the potential path space arising from the large execution traces.

To assess the reduction in path space, we statically analyzed the potential number of length-n paths which could be executed for an application. A conservative estimate was made based on extending the cyclomatic complexity measure (CCM) [95] to include inter-procedure paths. Developed by McCabe, CCM is a simple metric to assess path complexity for a function. Leveraging CCM, we estimated the number of length-n paths within a given function. We then extended this to inter-procedure paths by identifying the length-n paths which may extend beyond the function, both leading into and exiting from the function, for all call sites within the code base. This measure, though an estimate, is considered quite conservative as it does not consider the path space expansion arising from loops. This inter-procedure complexity measure adapted from CCM is shown in the third column of

Application	# Instructions Profiled	# Potential Length-*n* Paths	# Ground Truth DCF Paths
SQLite	16,948,864,926	13,642,304	17,351
OpenSSL	5,014,034,838	23,221,696	10,086
tshark	684,000,546	38,467,136	178
Python	656,068,272	12,175,712	35,206
Tor	118,310,256	1,191,280	10,639
InspIRCd	46,246,206	11,165,696	3,950
Pidgin	4,762,914	6,833,360	3,641

Table 4.1 Benchmark Applications. Profiled instruction trace size for ground truth analysis is shown in the second column. The third column represents the potential number of length-*4* paths, measured from an inter-procedure cyclomatic complexity measure. The final column shows the number of length-4 DCF paths within the profile trace.

Table 4.1 for all benchmarks.

4.3.4 Analysis of DCF Sampling

We evaluated the runtime overhead from profiling with Schnauzer as well as the accuracy of the coverage with respect to the ground truth DCF. Figure 4.7 details the runtime overhead experienced when profiling applications with our *DynamoRIO client*.

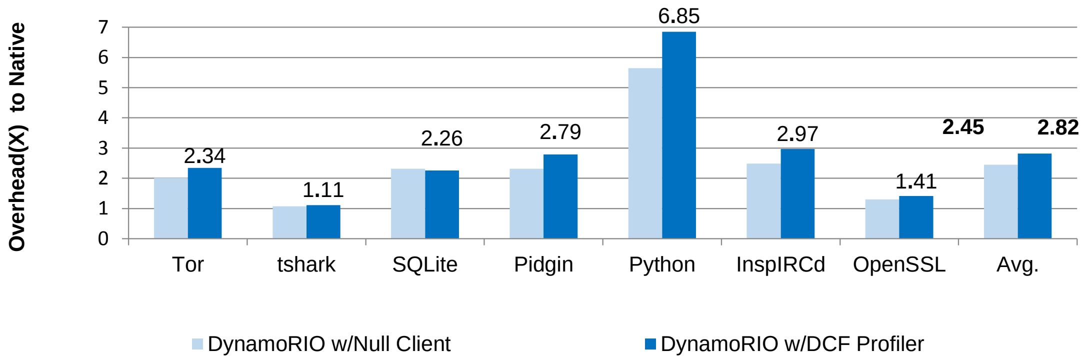


Figure 4.7 Sampling Overhead. Runtime overheads for applications are minimally above the slowdown experienced from the *DynamoRIO* core with a NULL client.

In all cases the majority of execution slowdown (2.82× average application runtime overhead) is attributed to the *DynamoRIO* core, which averages 2.45× runtime performance penalty compared to native execution. This small Schnauzer instrumentation overhead is due to a lightweight approach of only instrumenting code paths which are being actively profiled, which results in a 15% overall increase in execution time relative to *DynamoRIO* with a NULL client. The slight improvement in overhead experienced by SQLite from our client is attributed to the alteration of fundamental *DynamoRIO* operating mechanisms (e.g., code cache) which affects performance, in this case positively.

Given the general-purpose nature and powerful flexibility of *DynamoRIO*, a lighter-weight DCF path-specific dynamic instrumentation tool could potentially significantly improve DCF profiling performance. Indeed, custom tools have been shown to be highly effective when compared to binary instrumentation platforms like DynamoRIO and Pin. Zhao et al. demonstrate a low-overhead tool for shadow memory translation with Umbra [151], while Bosman et al. develop a dynamic taint analysis tool, Minemu [22], which is significantly faster than any competing general-purpose solution. Minemu demonstrated that, for such dynamic analyses, slowdown was not a fundamental property but instead arose from non-specialized implementations.

DynamoRIO was chosen as an initial development platform for power and flexibility combined with rapid accurate prototyping of DCF profiling. Although runtime overheads demonstrated generally remain higher than desired, we believe initial deployment is certainly possible (and planned) with the current framework for a range of applications.

Because we locate DCFs with sampling, there is legitimate concern as to whether or not the technique will observe all of the possible (ground truth) DCFs, and moreover, will all of the DCFs be identified in a reasonable amount of run time. As shown in Figure 4.8, our profiler locates the vast majority of DCFs in a short period of time. Larger applications, with billions of instructions, necessitate trillions of instructions of execution to receive good profiling coverage of all possible DCFs. This translates to at most ten thousand users profiling the application a single time each, certainly within reach of a modest user population.

Additionally, because sampling may deem a path a DCF, which in fact both directions were executed (but only one was observed), the accuracy of sampling must also be measured. Figure 4.9 shows the accuracy with which DCF paths are selected while profiling. Accuracy is given as the percentage of likely DCF paths, discovered by sampling, which are in the set of ground truth DCF paths for the application trace. Some applications achieve perfect accuracy while sampling, and overall Schnauzer is almost 99% accurate in profiled DCF paths with respect to the ground truth DCF.

4.3.5 Schnauzer Profiling Scalability

The dynamic control frontier is most valuable when it is derived from a sizeable population of end-users. Further, it is expected to profile an application for its entire life cycle. Schnauzer must therefore scale with application size, duration of execution, and population

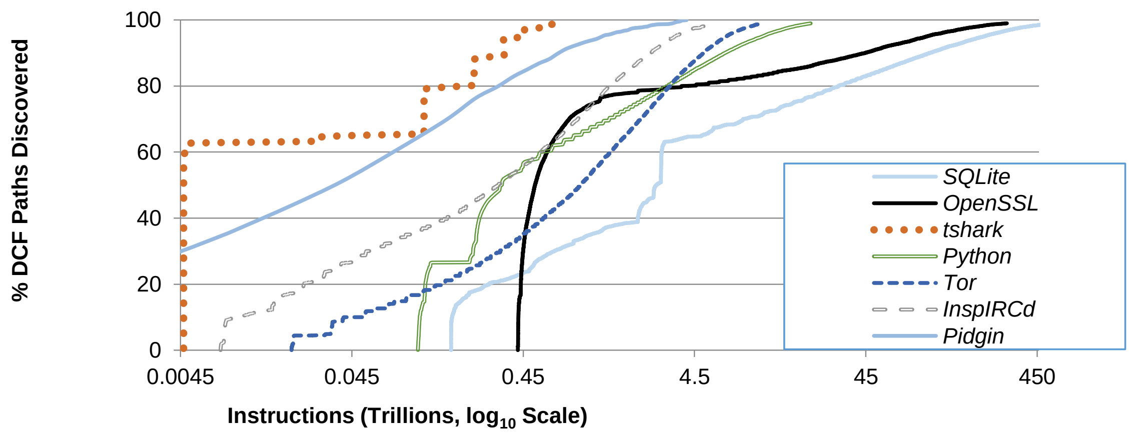


Figure 4.8 Path Coverage via Sampling. All benchmarks attain 100% dynamic control frontier path coverage. Even application traces of billions of instructions achieve coverage within trillions of instructions. Thus, a user population of less than ten thousand can profile a trace in a single run. Profiling is done while utilizing only a single active hypothesis at any time, and with a path length of 4 conditional branches.

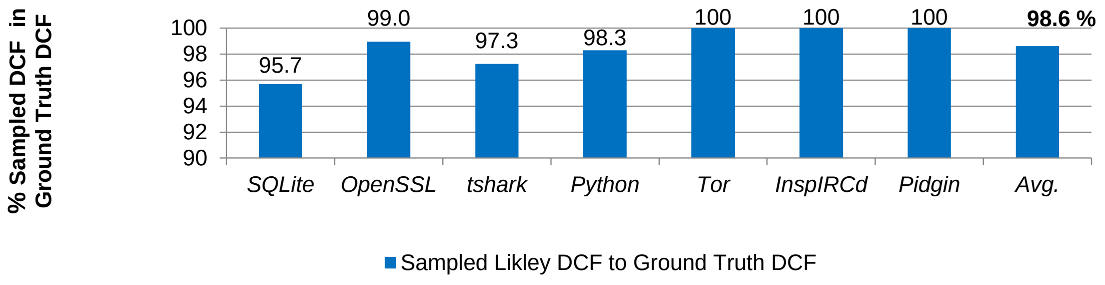


Figure 4.9 Sampling Accuracy. The percentage of likely DCF paths discovered by sampling which appear in the set of ground truth DCF paths.

of users.

As shown in Table 4.1, the number of DCF paths for an application is quite small when compared to the potential path space of such a long execution trace, greatly narrowing the domain for test. It must be considered, however, to what extent the dynamic control frontier path space will grow as an application execution continues unbounded. Figure 4.10 demonstrates that as trace length grows ever larger, the ground truth DCF path space grows linearly. This gives confidence that the path space for test, the number of ground truth paths which must be discovered while sampling, and the incidental work such as updating the global path filter, will all remain within a bounded, manageable range.

Schnauzer scales very well with increasing path length. As shown in Figure 4.11, to facilitate the highest degree of path sensitivity, ***path length has no appreciable effect on***

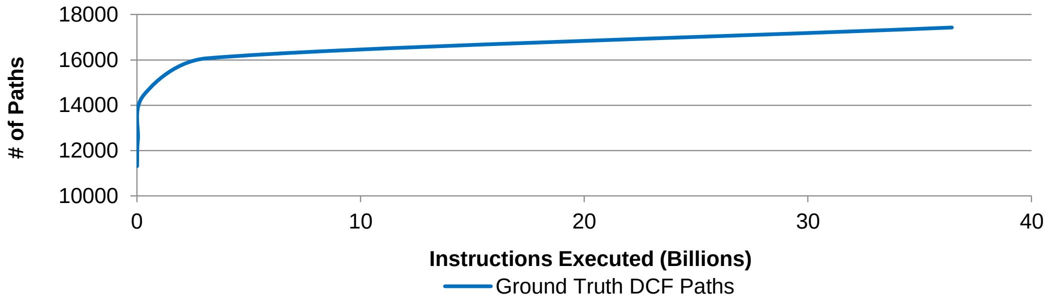


Figure 4.10 DCF Path Growth. As the number of executed instructions grows, ground truth DCF path space remains small. The application shown is SQLite, executing increasing durations of the fuzz testing component of the test suite.

sampling overhead for paths ranging from 1 to 64 conditional branches. This is due to the lightweight approach for path instrumentation, as only a few assembly-level instructions are added to the path. As well, the number of DCF paths will increase linearly with path length. Given this, paths of up to a length of 64 conditional branches may be analyzed with little impact to performance, should the need for greater path sensitivity arise.

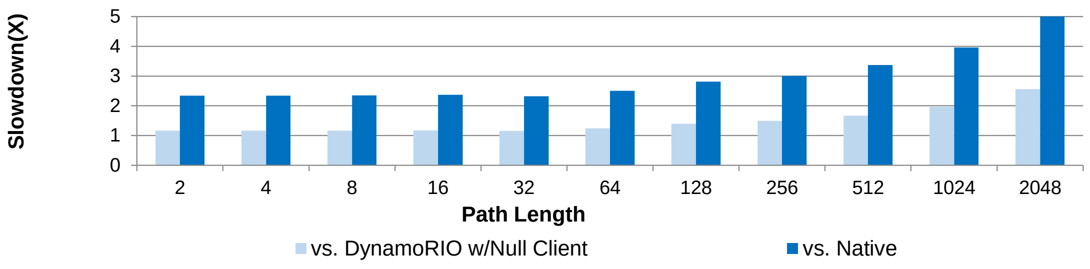


Figure 4.11 Path Length Scaling. As the profiled path length increases in Schnauzer, the performance overhead rises slowly. For paths up to 64 conditional branches, little difference is seen. The benchmark shown is *Tor*.

Profiling overheads are kept low by limiting sampling frequency and the number of paths concurrently being sampled. Figure 4.12 reveals ***only a single path need be actively profiled at any time***. The utilization of a global path filter and local list of recently sampled DCF paths eliminates redundant work and allows all DCF paths to be discovered in an acceptably similar amount of time, regardless of the number of concurrent hypotheses.

Scalability at the system level is achieved as well. Given the rate of dynamic control frontier path discovery while profiling *SQLite*, the overall bandwidth requirement from a population of users to the aggregation point at the developer is under 5 bytes/second per user. Such a result suggests that a single central server shard could likely serve 10,000s of individual user machines performing DCF path profiling. As the path space of an application

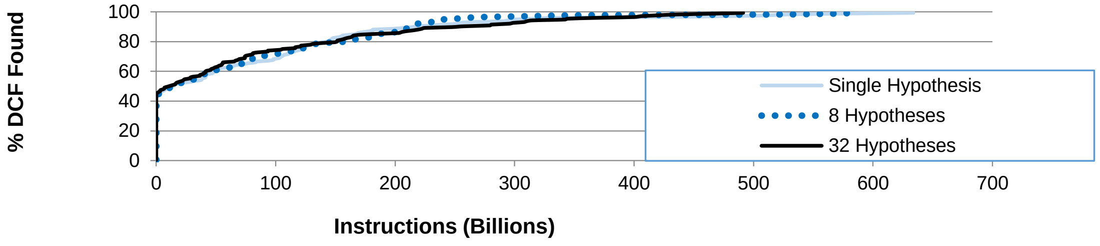


Figure 4.12 Concurrent Hypotheses. The number of active hypotheses has minimal impact on sampling coverage. This is due to the inclusion of the global path filter and local sampled path list to eliminate redundant sampling. Benchmark shown is *tshark*.

is explored, the influx of new path information will decrease. To enhance profiling over the entire life cycle of an application, the aging time for a DCF hypothesis can be increased. Increasing this age threshold brings profiled DCF paths closer to the ground-truth set of DCF paths for the entire lifecycle of an application.

4.3.6 DCF Correlation with Real Vulnerabilities

It has been shown that a large execution trace can contain a tractable number of dynamic control frontier paths for comprehensive test. However, it is necessary to demonstrate that this information delineating the frontier of dynamic execution is also a fertile source of real security exploits. To establish the relationship between the dynamic control frontier and security exploits, we sought to find if profiled DCF paths indeed sensitized important security bugs. The DCF paths gleaned from ground-truth analysis were compared to bug reports from fixed security bugs. Fixed bugs were chosen so as to know the precise location of an exploited bug within the source code. These bug locations could then be compared to the profiled DCF paths. If the location of a known bug is found to be sensitized by and located directly at the end of a DCF path, then the bug can be said to have been effectively hidden behind the dynamic control frontier.

As seen in Table 4.2, known security bugs are sensitized by the dynamic control frontier. ***A total of 14 security exploits were found at the dynamic control frontier*** for the profiled benchmark applications. The security exploits are drawn from the National Vulnerability Database (NVD) [100], which is maintained by the National Institute of Standards and Technology (NIST). The database was searched for Common Vulnerability Exposures (CVEs) [41] existing in benchmark applications. Not all vulnerabilities listed in the NVD for our benchmark applications were sensitized by DCF paths. Some, such as configuration errors, are beyond the scope of DCF path analysis. Others were simply not sensitized by the set of

Application	Vulnerability	Security Advisory
OpenSSL	Buffer Overflow	CVE-2012-2110
	Buffer Overflow	CVE-2012-2131
	Integer Underflow	CVE-2012-2333
SQLite	Buffer Overflow	CVE-2007-1888
Tor	DoS	CVE-2011-0492
	Buffer Overflow	CVE-2011-1924
Pidgin	DoS	CVE-2011-4939
tshark	Format String	CVE-2009-0601
	DoS	CVE-2011-0538
	DoS	CVE-2012-2394
Python	DoS	CVE-2010-2089
	DoS	CVE-2012-2135
InspIRCd	Buffer Overflow	CVE-2008-1925
	Heap Overflow	CVE-2012-1836

Table 4.2 Software Vulnerabilities Sensitized by Dynamic Control Frontier Paths. Known software vulnerabilities identified in the NIST National Vulnerabilities Database (NVD) were shown to be sensitized by DCF paths.

DCF paths profiled from our test inputs. However, these results are a strong affirmation that the control frontier indeed harbors bugs which are likely to be exploited.

It is interesting to note that profiling the dynamic control frontier is not only fruitful for finding security bugs. We also have early evidence that it is a prime target to search for software bugs in general. To this end, a separate analysis of the SQLite application was performed. In this analysis the ground-truth DCF was compared to the most recently fixed bugs in the SQLite code base. We found that ***12 of the most recent 20 bugs fixed in SQLite lay on code paths sensitized by the dynamic control frontier***. Of those 12 bugs, 5 were clearly enabling security vulnerabilities.

To determine an optimal path length for our experiments, the benchmarks were profiled for DCF paths of varying length, as shown in Figure 4.13. These sets of DCF paths were then analyzed to determine which vulnerabilities, listed in Table 4.2, would be sensitized by the set of DCF paths for a given path length. Within the scope of our experiments, the number of DCF paths increases roughly linearly with path length. More vulnerabilities are identified by the growing set of DCF paths. All vulnerabilities shown in Table 4.2 were discovered with a path length of 4 branches, with no other CVE entries indicated by longer paths. Therefore, this path length was selected our experiments. This coincides with the observation that bugs may be more likely to be found with shallow control flow activation rather than being correlative with path coverage [24]. It is important to note that even in the event that this path length is not optimal for another application, Schnauzer is amenable to longer paths as well.

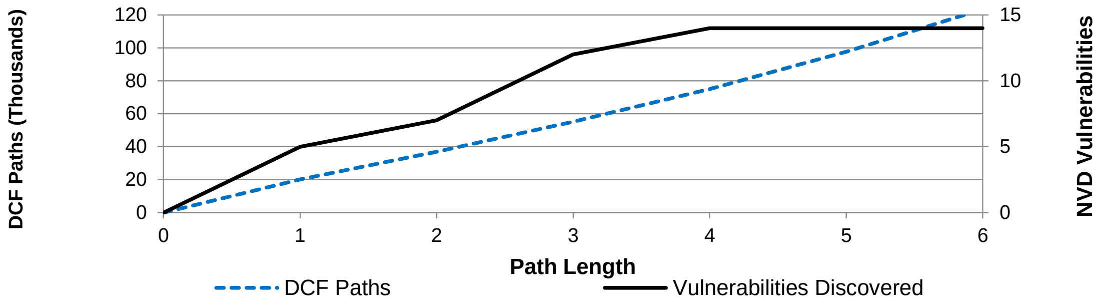


Figure 4.13 Impact of DCF Path Length on Vulnerability Discovery. Shown is the relationship between path length and vulnerabilities discovered, for the sum of all benchmarks listed in Table 4.1 and vulnerabilities identified in Table 4.2. Benchmarks were profiled for paths of varying lengths. As path length increases, the number of DCF paths increases, with more bugs sensitized. All vulnerabilities in Table 4.2 are discovered by the set of DCF paths profiled for a path length of 4.

4.4 Introduction

Much work has been done in the pursuit to identify and fix security vulnerabilities. Even more effort has been expended to deliver comprehensive testing of applications. Some related works are entirely complementary to Schnauzer. Other efforts assist in building a foundation for finding vulnerabilities but are not entirely sufficient themselves to accomplish the central goal of identifying code paths likely to be exploited, and thus DCF paths could be a powerful mechanism to focus analysis effort.

4.4.1 Hot Path Analysis

The preponderance of path analysis has historically been performed to identify hot, or heavily executed, paths. This is common in compiler optimizations but is also used for testing purposes. The work of Vaswani et al. [138] defines a hardware-based programmable path profiling mechanism. This work focuses primarily on solutions for hot path analysis, limiting its adaptability to dynamic control frontier profiling. Buse and Weimer [27] utilize static analysis to identify hot paths which are determined to be over 50% of total runtime of an application and generated by only 5% of feasible paths. This work highlights the difficulty of path profiling before application deployment.

4.4.2 Path Analysis and Distributed Sampling

The concept of distributed sampling and end-users performing testing tasks has become a more prevalent topic. Greathouse et al. have demonstrated the feasibility of distributed sampling for otherwise heavyweight security vulnerability analyses. The applications are, however, dataflow analyses [67, 68]. Ko et al. extensively investigate the concept of End User Software Engineering, which highlights the changing mindset of end-users playing a more involved role in the software life cycle [82]. Chilimbi et al. [38] have proposed a method to determine which paths were dynamically executed by deployed software that had never been tested, termed Efficient Path Profiling. This may be quite useful, but it focuses on finding latent bugs which are likely to directly impact users, thus focusing on software reliability. This is in contrast to DCF profiling, which seeks to enhance software security. A key assertion in this work was that edge profiling is sorely inadequate in comparison to path profiling. This built upon the previous work of Chilimbi et al. for Residual Path Profiling [39] which also focused solely on highly executed paths. Path-based data has been proposed by Liblit et al. to generate useful information on program crashes, specifically paths defined by conditional branches [90]. While this lends credibility to the usefulness of conditional branch-based path information, the purpose is strictly limited to post-mortem analysis of application failures. Ayers et al. [14] employ a different methodology to achieve these same ends.

4.4.3 Complementary Works

Testing technology has evolved along with software engineering techniques. Many useful tools exist which identify an ever-increasing ratio of bugs before deployment.

Godefroid et al. have implemented *DART* [59], a tool to automatically generate random tests to explore all possible code. This is a highly useful tool that could likely be made more effective with DCF profiling. Though it seeks to explore all sections of code, it cannot test all potential paths. A key challenge is that *DART* may never complete execution, making the determination of when to cease testing difficult.

The practice of fuzz testing supplies a software unit under test with a random generation of inputs in an attempt to break the unit, in the form of failed assertions and core dumps. The technique is sometimes called black-box testing because it creates inputs without regard to the internal structure of the software under test. This approach is very good in theory; however, in practice the probability of generating the correct set of inputs to achieve all possible paths within a given unit under test is effectively zero for non-trivial codes. Despite limitations, the approach has been effective at exposing security flaws. For example, Googles *cross_fuzz* tool generates random web pages for testing browsers, and it has exposed hundreds of potential security flaws in all major browsers [123]. When coupled with dynamic program analysis tools that can identify security vulnerabilities without active exploits, such as taint analysis [121] or input bounds checking [85], fuzz testing becomes a power tool in the war against attackers.

While effective, pure random fuzz testing has limited penetration on complex program control sequences. Another important work related to DCF profiling is Microsofts white-box fuzz testing tool *SAGE* [60]. This tool developed by Godefroid et al. strongly advances white-box fuzz testing of enterprise-level software. *SAGE* has become a primary tool for bug detection within Microsoft. The tool takes a test suite, with hand-generated and fuzz-generated tests, and then uses SAT-based techniques to derive new program inputs to change the direction of one branch in an existing dynamic code path. The newly derived code path is then subjected to symbolic execution analysis that includes input bounds checking, taint analysis and overflow checking. Approximately one-third of all Windows 7 security bugs found have been identified by *SAGE*. A highly representative example is a bug identified by *SAGE* which affected code that parsed ANI-format animated cursors [58]. The bug had escaped detection by extensive black-box testing over many years and generations of the Windows operating system. Using modest desktop hardware, *SAGE* was able to detect the bug within a few hours.

Random fuzz testing comprised the basis for testing four out of seven of our benchmark applications. Even so, we find vulnerabilities sensitized by DCF paths for these fuzz tested executions. The reality is that random fuzz testing does not provide deep code penetration [26, 30, 60]. This work is just another demonstration of the limitation of random fuzzing.

Even in light of such strong performance, many bugs are left undetected. A key challenge to any testing platform is the path space associated with a software application. Testing every path which may be executed remains infeasible for the foreseeable future. The infeasibility of complete path analysis is what makes DCF path analysis useful. Our work is to distill path data which may direct existing testing technologies. Applications such as *DART* and *SAGE* suffer the inadequacy of limited path exploration. The implementation of DCF path analysis can assist by directing such tools to high-value paths that likely contain security vulnerabilities.

Concolic execution tools allow deeper penetration of application code. However, these tools (such as *KLEE* [30]) have no path preference, including DCF paths. Indeed, in achieving code coverage, KLEE will execute the basic block where the defect lies, but not necessarily with the path required to sensitize the bug. We fully expect this to be the case, as industry has currently moved into an era of full code coverage for test. This property of concolic execution, however, does not preclude discovering DCF paths anyway. Table 4.1 shows Schnauzer identified 17,351 length-*4* DCF paths for *SQLite*, one of which sensitized the buffer overflow vulnerability identified in Table 4.2. This significantly narrows the field of discovery from the 13.6 million paths facing *KLEE*. As path lengths increase, the path space increases dramatically. The same measure for *SQLite* estimated almost 200 billion length-*16* paths.

This further highlights how contemporary test can benefit from DCF analysis. Even when code coverage is achieved, vulnerability-enabling defects still remain. Current white-box testing attempts to brute-force application code to provide deeper penetration. DCFs provide a heuristic to narrow the path space faced by code penetration testing.

4.5 Chapter Conclusions

Bugs in software remain the greatest security threat in programs today. There is much compelling evidence in the testing literature (e.g., analysis of *Windows 7* security bugs [60]) which suggest that the key to finding and fixing security vulnerabilities is to analyze code paths at the dynamic control frontier. In this work we presented a comprehensive technique for profiling an application to discover the dynamic control frontier. We have shown that by using a distributed profiling approach, such profiling can be achieved efficiently for a substantial population of users. Furthermore, we have demonstrated the high value of DCF paths by correlating our discovered paths to 14 known security advisory vulnerabilities documented in the National Vulnerabilities Database. We feel strongly that efficient user-based dynamic control frontier path profiling, combined with existing white-box testing techniques and heavyweight dynamic security vulnerability analysis tools, will be a powerful weapon in the future fight against attackers.

Chapter 5
Conclusion

Systems program building is an entropy-decreasing process, hence inherently metastable. Program maintenance is an entropy-increasing process, and even its most skillful execution only delays the subsidence of the system into unfixable obsolescence.

The Mythical Man-Month: Essays on Software Engineering
Frederick P. Brooks Jr.

Security has come to the forefront of computing in the information age. Despite continuing effort to harden the software attack surface, control-flow attacks have persisted as a fundamental building block in the exploitation of computing systems.

5.1 Dissertation Summary

This dissertation represents a novel approach to addressing contemporary control-flow attacks. Taking a subtractive approach, where the fundamental building blocks of attacks are removed from software, this work moves beyond the mitigation approach employed by competing works to date.

The novel approaches detailed within this dissertation are enabled by three main insights. First, isolation of control and data eliminates the mechanisms attackers have relied on for decades. Second, design for security is a fundamental principle which can eliminate barriers to adoption for techniques to harden the software attack surface, such as CDI. This design for security principle is demonstrated in the hardware-software co-design detailed in this dissertation in Chapter 3. Lastly, software exploits are heavily control-flow dependent. With this insight it is possible to foreshadow attacks, eliminating the root defects before they are exploited.

Chapter 2 detailed the process of control-data isolation, a novel approach to ensuring the programmer-intended control-flow graph of software at runtime. The primary accomplishment of CDI is to eliminate the direct injection of malicious user data into the program counter. This is achieved in practice through the elimination of the fundamental building blocks of control-flow attacks; indirect control flow. The result is the elimination of contemporary control-flow attacks, an essential element of many software exploits.

Chapter 3 proposes efficient hardware extensions to eliminate the remaining barriers to adoption for control-data isolation proposed in Chapter 2. Through the memoization of programmer-intended control-flow graph edges, the edge cache virtually eliminates runtime overheads for control-data isolation compliant code. In addition to the performance optimization for control-data isolation, Chapter 3 also presents a novel method to address potential future control-flow attacks, the non-speculative return address stack. This stack can conservatively determine the single, precise target of return instructions at runtime. This minimal hardware addition extends the control-data isolation principle to paths of execution.

In Chapter 4, the intersectionality of control-flow paths and software defects was established. The technique developed in Chapter 4 leveraged the way software is engineered, tested, and used to determine where latent bugs reside in the control-flow of an application. By profiling the execution frontier of a wide user base, software developers can focus finite testing resources for heavyweight path testing. The result is a methodology to get ahead of attackers. This in turn makes software security a proactive, rather than reactive, endeavor.

5.2 Future Control-Flow Security Research

Although this dissertation represents a non-trivial pivot in the approach to control-flow security, the pursuit in hardening the software attack surface is never-ending. Control-Data isolation, detailed in this work, has been implemented as proof of concept. However, in the security community full implementation is highly valued. To this end, the exploration of a CDI compliant, whole-system implementation would be effective in proving the efficacy of CDI in the face of existing and future attacks. Thus, extending the compilation framework in Chapter 2 to a full LAMP stack would be a positive direction for future research in control-flow security.

An interesting topic arising from this work is the form future control-flow attacks will take. As the mechanisms used in contemporary control-flow attacks are eliminated, attackers will seek other ways to exploit a system. In the context of control-flow, the likely candidate for future exploitation will be attacks that adhere to the control-flow graph of an application. That is, as control edges are made secure through CDI attackers may seek to compromise the paths of execution at runtime. The addition of the non-speculative return address stack in Chapter 3 is a step in the extension of CDI to path-based security. However, much work remains. This work has demonstrated that the control-flow graph of software can, with high efficiency, *always* be assured at runtime. However, it is yet unknown if the paths of execution within the bounds of the control-flow graph can be protected in all cases, and if so, at what cost.

A key element of this dissertation is the elimination of underlying mechanisms which create the software attack surface. This has been proposed as a subtractive approach to security, as detailed in Chapter 2. This subtraction has arisen as a consequence of avoiding the same pitfalls of the many attempts to mitigate control-flow attacks for decades. However, future work involves the generalization of the subtractive approach for the software attack surface in general. In this dissertation application of the subtractive approach has been *ad hoc* and has grown out of the acceptance of the failure of mitigation approaches to address control-flow attacks. However, no generalization of the subtractive approach has yet been proposed. This is a strong area for future work as attackers have long demonstrated the ability to circumvent additive security measures.

Given the ubiquity of control-flow attacks, many countermeasures have been adapted. However, these have been concentrated on resource-rich domains like servers and laptops. With the rise of the Internet of Things (IoT), low cost devices are set to become more connected, and likewise more vulnerable to attack. These systems lack many of the protections developed for more expensive computing platforms. This in turn lowers the bar for adversaries to successfully compromise a system and execute a control-flow attack. The adoption of CDI for embedded devices which will drive the IoT in the future is an essential avenue for future work. Development of a whole-system implementation for this target domain represents a significant future application of Control-Data Isolation.

Owing to the threat model detailed in Chapter 2, the robust solutions in this dissertation lift arbitrary restrictions on an adversary assumed in other works. This in turn accommodates a wider domain of application, including such technologies as dynamically generated code and just-in-time compilation. The enabling of control-flow security in turn elevates the threat and trust model of future systems. A primary example of this is the attestation of CDI compliant code. Chapter 2 detailed the properties of CDI compliant code and the algorithm for verification of these properties. However, work remains in the integration of CDI compliant attestation for systems and the mechanisms necessary to implement CDI attestation in practice.

5.3 Conclusion

Within this dissertation, a novel approach to control-flow security has been presented in detail. The realization of the techniques presented in this work represents a pivot in the approach to assuring the control-flow of an application at runtime, in the adoption of a subtractive approach to eliminating control-flow attacks. Through the analysis of the role of control-flow in the software attack surface, to the elimination of the building blocks of attacks, this dissertation demonstrates the key in the elimination of contemporary control-flow attacks.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, number MSR-TR-2005-18, pages 340–353, Alexandria, VA, November 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[3] D. Ahn and G. Lee. Countering code injection attacks with tlb and i/o monitoring. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 370–375, Oct 2010.

[4] Erdem Aktas, Furat Afram, and Kanad Ghose. Continuous, low overhead, run-time validation of program executions. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 229–241, Washington, DC, USA, 2014. IEEE Computer Society.

[5] M. Almorsy, J. Grundy, and A. S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 662–671, May 2013.

[6] Jim Alves-Foss and Salvador Barbosa. Assessing computer security vulnerability. *SIGOPS Oper. Syst. Rev.*, 29(3):3–13, July 1995.

[7] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, New York, NY, USA, 2013. ACM.

[8] M. Anderson. Update - crimeware pays. *IEEE Spectrum*, 45(7):13–13, July 2008.

[9] Kathleen Andreoli and Leigh A Musser. Computers in nursing care: the state of the art. In *Nursing and Computers*, pages 176–186. Springer, 1985.

[10] William Arthur, Sahil Madeka, Reetuparna Das, and Todd Austin. Locking down insecure indirection with hardware-based control-data isolation. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 115–127, New York, NY, USA, 2015. ACM.

[11] William Arthur, Biruk Mammo, Ricardo Rodriguez, Todd Austin, and Valeria Bertacco. Schnauzer: scalable profiling for likely security bug sites. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–11, Feb 2013.

[12] William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. Getting in control of your control flow with control-data isolation. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 79–90, Feb 2015.

[13] AV-TEST. Av-test malware. Technical report, AV-TEST GmbH, 2016.

[14] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. Traceback: First fault diagnosis by reconstruction of distributed control flow. *SIGPLAN Not.*, 40(6):201–212, June 2005.

[15] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[16] David W Bates, Michael Cohen, Lucian L Leape, J Marc Overhage, M Michael Shabot, and Thomas Sheridan. Reducing the frequency of errors in medicine using information technology. *Journal of the American Medical Informatics Association*, 8(4):299–308, 2001.

[17] Hal Berghel. Cyber chutzpah: The sony hack and the celebration of hyperbole. *Computer*, 48(6):350–366, 2015.

[18] A. R. Bernat and B. P. Miller. Structured binary editing with a cfg transformation algebra. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 9–18, Oct 2012.

[19] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[21] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[22] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World's Fastest Taint Tracker. In *the Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[23] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[24] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 42–51, New York, NY, USA, 2006. ACM.

[25] Z. Budrikis, J. Hullett, and D. Q. Phiet. Transient-mode buffer stores for nonuniform code tv. *IEEE Transactions on Communication Technology*, 19(6):913–922, December 1971.

[26] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 443–446, Sept 2008.

[27] Raymond P. L. Buse and Westley Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 144–154, Washington, DC, USA, 2009. IEEE Computer Society.

[28] c0ntext. How to Hijack the Global Offset Table with pointers. `http://www.exploit-db.com/papers/13203/`, 2011.

[29] c0ntext. Bypassing non-executable stack during exploitation using return-to-libc. `http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf`, 2012.

[30] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *the Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[31] 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)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[32] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE Security Privacy*, 12(4):63–67, July 2014.

[33] Pohua P. Chang and Wen-mei W. Hwu. Control flow optimization for supercomputer scalar processing. In *Proceedings of the 3rd International Conference on Supercomputing*, ICS '89, pages 145–153, New York, NY, USA, 1989. ACM.

[34] Ian M. Chapman, Sylvain P. Leblanc, and Andrew Partington. Taxonomy of cyber attacks and simulation of their effects. In *Proceedings of the 2011 Military Modeling & Simulation Symposium*, MMS '11, pages 73–80, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[35] R. N. Charette. Why software fails [software failure]. *IEEE Spectrum*, 42(9):42–49, Sept 2005.

[36] K. Chen, Y. Zhang, and P. Liu. Dynamically discovering likely memory layout to perform accurate fuzzing. *IEEE Transactions on Reliability*, PP(99):1–15, 2016.

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

[38] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective Statistical Debugging via Efficient Path Profiling. In *the Proc. of the International Conference on Software Engineering (ICSE)*, 2009.

[39] Trishul M. Chilimbi, Aditya V. Nori, and Kapil Vaswani. Quantifying the effectiveness of testing via efficient residual path profiling. Technical report, Microsoft Research, 2007.

[40] Wang Chunlei, Zhao Gang, and Dai Yiqi. An efficient control flow security analysis approach for binary executables. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 272–276, Aug 2009.

[41] The Mitre Corporation. Common Vulnerabilities and Exposures. `http://cve.mitre.org/`.

[42] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 227–237, 2003.

[43] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[44] Crispin Cowan, Carlton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Berkeley, CA, USA, 1998. USENIX Association.

[45] Darren Dalcher. Why the pilot cannot be blamed: a cautionary note about excessive reliance on technology. *International Journal of Risk Assessment and Management*, 7(3):350–366, 2007.

[46] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6, June 2014.

[47] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[48] Y. Demchenko, C. d. Laat, O. Koeroo, and D. Groep. Re-thinking grid security architecture. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 79–86, Dec 2008.

[49] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 559–570, New York, NY, USA, 2013. ACM.

[50] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[51] M.U. Farooq, K. Khubaib, and L.K. John. Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 59–70, Feb 2013.

[52] Ivan Fratrić. Ropguard: Runtime prevention of return-oriented programming attacks. 2012.

[53] John F. Gantz, Richard Lee, Alejandro Florean, Victor Lim, Biplap Sikdar, Logesh Madhaven, Sravana Kumar Sristi Lakshmi, and Mangalam Nagappan. The link between pirated software and cybersecurity breaches. how malware in pirated software is costing the world billions. Technical report, IDC, The National University of Singapore, 2013.

[54] L. Garber. Melissa virus creates a new type of threat. *Computer*, 32(6):16–19, June 1999.

[55] Gartner. Gartner [online]. `http://www.gartner.com/technology/home.jsp`, 2013.

[56] W.J. Ghandour and N.J. Ghandour. Leveraging dynamic slicing to enhance indirect branch prediction. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 292–299, Oct 2014.

[57] D. P. Gilliam. Security risks: management and mitigation in the software life cycle. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004. WET ICE 2004. 13th IEEE International Workshops on*, pages 211–216, June 2004.

[58] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, Sept 2008.

[59] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[60] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[61] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, 2011. ACM.

[62] Brian Goetz. Java theory and practice: Dynamic compilation and performance measurement. Technical report, IBM Corporation, 2004.

[63] Max Goncharov. Russian underground 101.

[64] Max Goncharov. Russian underground 2.0.

[65] Max Goncharov. Russian underground revisited.

[66] J. L. Greathouse, I. Wagner, D. A. Ramos, G. Bhatnagar, T. Austin, V. Bertacco, and S. Pettie. Testudo: Heavyweight security analysis via statistical sampling. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 117–128, Nov 2008.

[67] Joseph L. Greathouse, Chelsea LeBlanc, Todd Austin, and Valeria Bertacco. Highly Scalable Distributed Dataflow Analysis. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2011.

[68] Joseph L. Greathouse, Ilya Wagner, David A. Ramos, Gautam Bhatnagar, Todd Austin, Valeria Bertacco, and Seth Pettie. Testudo: Heavyweight Security Analysis via Statistical Sampling. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2008.

[69] Mark D Griffiths and Nigel Hunt. Dependence on computer games by adolescents. *Psychological reports*, 82(2):475–480, 1998.

[70] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 55–64, May 2010.

[71] Chin-Yu Huang and M. R. Lyu. Optimal testing resource allocation, and sensitivity analysis in software development. *IEEE Transactions on Reliability*, 54(4):592–603, Dec 2005.

[72] informationisbeautiful. World's biggest data breaches[online]. `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/`.

[73] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 294–310, New York, NY, USA, 2000. ACM.

[74] Kaspersky. Carbanak apt. Technical report, Kaspersky Lab, 2015.

[75] Kaspersky Corporation. Computer threats[online]. `http://www.kaspersky.com/threats`.

[76] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 94–105, June 2012.

[77] Hyesoon Kim, J. Joao, O. Mutlu, Chang Joo Lee, Y.N. Patt, and R. Cohn. Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. volume 58, pages 1153–1170, Sept 2009.

[78] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. Vpc prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 424–435, New York, NY, USA, 2007. ACM.

[79] M. Kim, S. Sinha, C. Grg, H. Shah, M. J. Harrold, and M. G. Nanda. Automated bug neighborhood analysis for identifying incomplete bug fixes. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 383–392, April 2010.

[80] Vladimir Kiriansky, Derek Bruening, Saman P Amarasinghe, et al. Secure execution via program shepherding. In *USENIX Security Symposium*, volume 92, 2002.

[81] B. Kitchenham and S. Linkman. Validation, verification, and testing: diversity rules. *IEEE Software*, 15(4):46–49, Jul 1998.

[82] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, April 2011.

[83] Bradley M. Kuhn and David W. Binkley. An enabling optimization for c++ virtual functions. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, SAC '96, pages 420–428, New York, NY, USA, 1996. ACM.

[84] Dharmender Singh Kushwaha and A. K. Misra. Software test effort estimation. *SIGSOFT Softw. Eng. Notes*, 33(3):6:1–6:5, May 2008.

[85] Eric Larson and Todd Austin. High Coverage Detection of Input-Related Security Faults. In *the Proc. of the USENIX Security Symposium*, 2003.

[86] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004.

[87] Young-Min Lee, Byung-Chul Tak, Hye-Seon Maeng, and Shin-Dug Kim. Real-time java virtual machine for information appliances. *IEEE Transactions on Consumer Electronics*, 46(4):949–957, Nov 2000.

[88] D. J. Leversage and E. J. Byres. Estimating a system's mean time-to-compromise. *IEEE Security Privacy*, 6(1):52–60, Jan 2008.

[89] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480, Dec 2009.

[90] Ben Liblit and Alex Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical report, Berkeley, CA, USA, 2002.

[91] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 37–44, Nov 2011.

[92] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[93] J. Malcolmson. What is security culture? does it differ in content from general organisational culture? In *43rd Annual 2009 International Carnahan Conference on Security Technology*, pages 361–366, Oct 2009.

[94] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011.

[95] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[96] Daniel McNeill and Paul Freiberger. *Fuzzy logic: The revolutionary computer technology that is changing our world*. Simon and Schuster, 1994.

[97] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74, Oct 2013.

[98] Microsoft Corporation. Microsoft enhanced mitigation experience toolkit [online]. `https://www.microsoft.com/emet`, 2014.

[99] musl libc. musl-libc[online]. `http://www.musl-libc.org/`.

[100] National Institute of Standards and Technology. National Vulnerability Database. `http://nvd.nist.gov/`.

[101] S. Neuhaus and T. Zimmermann. Security trend analysis with cve topic models. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 111–120, Nov 2010.

[102] F. J. B. Nunes, A. D. Belchior, and A. B. Albuquerque. Security engineering approach to support software security. In *2010 6th World Congress on Services*, pages 48–55, July 2010.

[103] H. Ohtera and S. Yamada. Optimal allocation and control problems for software-testing resources. *IEEE Transactions on Reliability*, 39(2):171–176, Jun 1990.

[104] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM.

[105] R. Ortalo, Y. Deswarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, Sep 1999.

[106] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615, May 2012.

[107] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security*, pages 447–462, 2013.

[108] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM.

[109] Jannik Pewny and Thorsten Holz. Control-flow restrictor: Compiler-based cfi for ios. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 309–318, New York, NY, USA, 2013. ACM.

[110] Jannik Pewny and Thorsten Holz. Control-flow restrictor: Compiler-based cfi for ios. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 309–318, New York, NY, USA, 2013. ACM.

[111] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the Usenix Tech. Conference*, pages 211–224, 2003.

[112] R. Pressman. *Software Engineering, A Practitioner's Approach*. McGraw-Hill, 2009.

[113] O. Qingyu, H. Kai, and W. Xiaoping. Research on the embedded security architecture based on the control flow security. In *Computer Science and Engineering, 2009. WCSE '09. Second International Workshop on*, volume 1, pages 133–137, Oct 2009.

[114] Marco Ramili. Global Offset Table Injection Procedure. `http://marcoramilli.blogspot.com/2011/11/global-offset-table-injection-procedure.html`, 2011.

[115] Michael A Rappa. The utility business model and the future of computing services. *IBM Systems Journal*, 43(1):32, 2004.

[116] A. Rein, C. Rudolph, J. F. Ruiz, and M. Arjona. Introducing security building block models. In *BioMedical Computing (BioMedCom), 2012 ASE/IEEE International Conference on*, pages 132–139, Dec 2012.

[117] R. Riley, X. Jiang, and D. Xu. An architectural approach to preventing code injection attacks. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 30–40, June 2007.

[118] E. Rohou, B.N. Swamy, and A. Seznec. Branch prediction and the performance of interpreters; don't trust folklore. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 103–114, Feb 2015.

[119] D. Ruiu. Learning from information security history. *IEEE Security Privacy*, 4(1):77–79, Jan 2006.

[120] Bruce Schneier. Stuxnet. `http://www.schneier.com/blog/archives/2010/10/stuxnet.html`, October 2010.

[121] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *the Proc. of the IEEE Symposium on Security and Privacy*, 2010.

[122] Simon Schwartzman. *High Technology vs. Self-Reliance: Brazil Enters the Computer Age*. Instituto Universitário de Pesquisas do Rio de Janeiro, Sociedade Brasileiro de Instrução, 1985.

[123] Larry Seltzer. Microsoft, Google Clash Over IE 0-Day Leaked to Chinese Hackers. `http://www.pcmag.com/article2/0,2817,2375029,00.asp`.

[124] André Seznec. A 64-kbytes ittage indirect branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.

[125] E. Sherif, S. Furnell, and N. Clarke. Awareness, behaviour and culture: The abc in cultivating security compliance. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 90–94, Dec 2015.

[126] Y. Shi and G. Lee. Augmenting branch predictor to secure program execution. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 10–19, June 2007.

[127] Yixin Shi, S. Dempsey, and Gyungho Lee. Architectural support for run-time validation of control flow transfer. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 506–513, Oct 2006.

[128] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Microarchitecture, 1998. MICRO-31. Proceedings. 31st Annual ACM/IEEE International Symposium on*, pages 259–271, Nov 1998.

[129] M. L. Soffa. Path sensitive analysis for security flaws. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 3–3, Dec 2008.

[130] Standard Performance Evaluation Corporation. Standard performance evaluation corporation[online]. `http://www.spec.org`.

[131] Sun Microsystems, Inc. The java virtual machine specification[online]. `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`.

[132] John K. Swearingen, Boyd L. Alexander, and John J. Boyle. Congressional reliance on and use of computers. In *Proceedings of the 1978 Annual Conference*, ACM '78, pages 130–, New York, NY, USA, 1978. ACM. Chairman-Chartrand, Robert L.

[133] A. K. Talukder, V. K. Maurya, B. G. Santhosh, E. Jangam, S. V. Muni, K. P. Jevitha, S. Saurabh, and A. R. Pais. Security-aware software development life cycle (sasdlc) - processes and tools. In *2009 IFIP International Conference on Wireless and Optical Communications Networks*, pages 1–5, April 2009.

[134] H. Theiling. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 23–30, 2000.

[135] T. Thomas, A. Pouraghily, K. Hu, R. Tessier, and T. Wolf. Multi-task support for security-enabled embedded processors. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 136–143, July 2015.

[136] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. Cacti 5.1. *HP Laboratories, April*, 2:24, 2008.

[137] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[138] Kapil Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 217–228, March 2005.

[139] J. Voas, A. Ghosh, G. McGraw, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*, pages 250–263, Jun 1996.

[140] Z. Wang, K. Tang, and X. Yao. Multi-objective approaches to optimal testing resource allocation in modular software systems. *IEEE Transactions on Reliability*, 59(3):563–575, Sept 2010.

[141] D. A. Wheeler. Preventing heartbleed. *Computer*, 47(8):80–83, Aug 2014.

[142] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2012.

[143] Zichao Xie, Dong Tong, Mingkai Huang, Xiaoyin Wang, Qinqing Shi, and Xu Cheng. Tap prediction: Reusing conditional branch predictor for indirect branches with target address pointers. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 119–126, Oct 2011.

[144] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries, 2009.

[145] J. Yost. Old doi 10.1109/mahc.2015.56 eh - good doi 10.1109/mahc.2015.21 - ehthe origin and early history of the computer security software products industry. *IEEE Annals of the History of Computing*, PP(99):1–1, 2015.

[146] J. R. Yost. Computer security [guest editors' introduction]. *IEEE Annals of the History of Computing*, 37(2):6–7, Apr 2015.

[147] Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 276–286, New York, NY, USA, 1995. ACM.

[148] A. A. Younis, Y. K. Malaiya, and I. Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 1–8, Jan 2014.

[149] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573, May 2013.

[150] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX.

[151] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and Scalable Memory Shadowing. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2010.

[152] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.