SOURCE-FREE BINARY MUTATION FOR OFFENSE AND DEFENSE

by

Vishwath R. Mohan

APPROVED BY SUPERVISORY COMMITTEE:

Kevin W. Hamlen, Chair

Alvaro Cárdenas

Latifur Khan

Zhiqiang Lin

Dedicated to my parents, who encouraged without question.

To my wife, for lifting me far beyond where I could have flown myself.

To my grandfather, more technology-aware than most PhDs I know.

SOURCE-FREE BINARY MUTATION FOR OFFENSE AND DEFENSE

by

VISHWATH R. MOHAN, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2014

ACKNOWLEDGMENTS

This dissertation could not have been completed without the help of the author's advisor, Dr. Kevin Hamlen, who served as inspiration, role model and walking database of both ideas and knowledge.

Richard Wartell, the author's research partner and collaborator on some of the research presented in the dissertation, deserves a huge shout out. This dissertation owes a lot to not just his invaluable research assistance, but also his continued friendship and support. Because no one should have to delve the depths of x86 machine code alone.

The author wishes to thank Dr. Zhiqiang Lin, Dr. Latifur Khan, and Dr. Mehedy Masud, whose contributions and ideas, both defensive and offensive, greatly helped this dissertation achieve its goals.

Special thanks should also be given to Dr. Per Larsen, who provided the seed of the idea that eventually became Opaque CFI. He also proved to be a motivating collaborator and good friend, for which the author is grateful.

Finally, the author wishes to thank his wife, Sanjana Raghunath, for her patience and constant support.

The research reported in this dissertation was supported in part by the Air Force Office of Scientific Research (AFOSR) under Young Investigator Program (YIP) award FA9550-08-1-0044 and Active Defense award FA9550-10-1-0088, the National Science Foundation (NSF) under CAREER award #1054629, the Office of Naval Research (ONR) under award N00014-14-1-0030, and an NSF Industry-University Collaborative Research Center (IUCRC) award

from Raytheon Company. All opinions, recommendations, and conclusions expressed are those of the authors and not necessarily of the AFOSR, NSF, ONR, or Raytheon.

November 2014

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas." It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student's contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

SOURCE-FREE BINARY MUTATION FOR OFFENSE AND DEFENSE

Publication No. _____

Vishwath R. Mohan, PhD
The University of Texas at Dallas, 2014

Supervising Professor: Kevin W. Hamlen

The advent of advanced weaponized software over the past few years, including the Stuxnet, Duqu, and Flame viruses, is indicative of the seriousness with which advanced persistent threats (APTs) have begun to treat the cyber-realm as a potential theatre for offensive military action and espionage. This has coincided with a strong interest in creating malware obfuscations that hide their payloads for extended periods of time, even while under active search. Progress on this front threatens to render conventional software defenses obsolete, placing the world in dire need of more resilient software security solutions.

This dissertation underlines the seriousness of this threat through the design and implementation of two novel, next-generation malware obfuscation technologies that bypass today's widely deployed defenses. Unlike conventional polymorphic malware, which mutates randomly in an effort to evade detection, the presented attacks are *reactively adaptive* in the sense that they intelligently surveil, analyze, and adapt their obfuscation strategies in the wild to understand and defeat rival defenses. The dissertation then presents three novel software defenses that offer strengthened software security against both current and future offensive threats. Rather than attempting to detect threats statically (i.e., before

they execute), or introducing dynamic monitors that raise compatibility and performance penalties for consumers, the new defenses implement automated, source-free, binary software transformations that preemptively transform untrusted software into safe software. Experiments show that this *security retrofitting* approach offers higher performance, greater security, and more flexible deployment options relative to competing approaches. Thus, binary code transformation and mutation is realized as both a powerful offensive and a potent defensive paradigm for software attacks and defenses.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

One of the more observable consequences of our rapid technological progress is the increasing level of computerized automation used for previously mechanized or manual tasks. The surprising amount of software present in today's cars, the wealth of sensitive intellectual property that is stored on the cloud, and the wave of cyber-physical systems being used to control so much of a country's critical infrastructure are particularly illuminating examples that show how this phenomenon has manifested itself at the individual, corporate and national levels.

This increased reliance on computers at the corporate and national levels has also made cyber assets tactically valuable targets. In particular, the emergence of weaponized software, such as the Stuxnet (Sanger, 2012), Duqu (Guilherme and Szor, 2012), and Flame (Lee, 2012) viruses, points to a change in the deployment strategy and intended purpose of malware. What was once seen as the exclusive domain of cyber-criminals, used purely for monetary profit, is now being recognized for its potential as an effective reconnaissance and covert monitoring tool (Flame), or as a safer, cheaper way to preemptively strike at targets of strategic military value (Stuxnet).

In such a scenario, any competitive solution to cyber-security must include options for both offensive and defensive action, and approaches that focus purely on defense will be at a severe disadvantage. In essence, the clear distinction between attackers and defenders is being blurred as cyber-security begins to resemble a more traditional arms race.

To succeed against well-defended opponents, cyber-offensive malware must penetrate machines protected by real-time detection systems that rely heavily on static analysis. For

example, modern defenses typically scan untrusted binary software before it executes to determine whether it might be malicious, and the classification hinges on unique syntactic features of malware rather than its semantics. Static analysis is favored because it is faster and consumes fewer resources relative to purely dynamic methods (Kim and Karp, 2004; Kreibich and Crowcroft, 2004; Li et al., 2006; Newsome et al., 2005; Singh et al., 2004), and because it can sometimes spot suspicious syntactic anomalies associated with vehicles of *zero-day attacks* (Newsome et al., 2005; Li et al., 2006; Grace et al., 2010; Zhao and Ahn, 2013), which exploit vulnerabilities previously unknown to defenders. Resilience against static analyses is therefore a high priority for malware obfuscation technologies.

Oligormorphism, *polymorphism*, *virtualization-based obfuscation*, and *metamorphism* are the four main techniques modern malware uses to evade static analyses, and are explained in more detail in Section 2.1. Although these obfuscations can be temporarily effective against some modern static defenses, their reliance on random, undirected mutation makes them brittle against defenders who actively adapt their protections to target specific emerging threats and threat-families. In most cases, focused defender reverse-engineering efforts uncover mutation patterns or invariants that suffice for defenders to reliably detect and quarantine most or all variants of the malware. That is, malware employing one of these techniques is able to hide from a detection tool only so long as its signature is not known. Once a copy of the malware has been analyzed and a signature crafted, detection tools are able to correctly classify subsequently encountered copies.

In this context, it is reasonable to view current approaches to obfuscation as short-term solutions for attackers. They work on the element of surprise—send out a previously unseen malware that performs its task up until it is discovered and a fix released. The monetary benefit to exploiting this time lag is the major incentivising factor for malware authors who attack typical end-users.

However, the use of malware as a targeted espionage or counter-attack tool is severely hindered by the inability of obfuscation technologies to autonomously adapt their obfuscations once a signature has been crafted.

To overcome this limitation, next-generation cyber-weapons must employ more powerful, flexible, *reactively adaptive* obfuscation strategies that learn and adapt to changing defenses rapidly and autonomously. Such adaptation renders *signature-based defenses* moot, since as soon as defenders discover and deploy a new signature in response to the threat, reactively adaptive malware learns the new signature and adapts its obfuscation pattern to evade it. Thus, reactively adaptive mutation innovations will imbue weaponized software with true stealth capabilities rather than mere diversity.

Dually, next-generation cyber-defenders must adopt more powerful defensive paradigms that can cope with reactively adaptive malware threats posed by resourceful adversaries. Because today's heavy reliance upon static detection is so dependent on the syntax of untrusted binaries, this next wave of reactively adaptive malware is likely to overwhelm current defenses. Rather than relying on syntactic matching to discover malware, a more semantic-aware approach that predicts and selectively monitors the possible *behaviors* of untrusted binaries stands a far better chance of detecting and preventing malicious actions from occurring.

Traditionally, such *runtime monitors* have been implemented at the operating system or virtual machine level, where they capture all software activities and subject them to security checks. However, *In-lined Reference Monitors (IRMs)*—which instantiate the monitoring logic within the monitored applications themselves through automated binary program rewriting—offer numerous advantages to the traditional approach.

Firstly, because IRMs create self-monitoring applications, they do not require kernel modifications or administrative privileges to safely run. This makes them easier to deploy, and also allows their use in environments where such controls are unavailable (in low-resource

devices, for example). Secondly, IRMs can be customized to enforce organization-specific policies, or streamlined with platform-specific optimizations, allowing them to exhibit greater flexibility and lower overheads compared to traditional approaches. Lastly, because IRMs reside within the address space of the monitored application, they have ready access to a greater amount of its internal state, making them strictly more powerful than their traditional bretheren.

Recent defensive techniques attempt to capitalize on this idea. Systems like CFI (Abadi et al., 2009), XFI (Erlingsson et al., 2006), and SecondWrite (Smithson et al., 2010) statically transform and augment untrusted programs to include monitors that dynamically constrain the behavior of the program as it executes. Although they offer advantages over traditional system-level monitoring, these techniques either require code-producer cooperation (Abadi et al., 2009; Erlingsson et al., 2006) or introduce non-trivial overheads (Cheng et al., 2006; Lam et al., 2013). This severely limits the applicability of these techniques to the vast swath of *commercial off-the-shelf (COTS)* and legacy binaries in use today, which do not have source or debug information available, or for which consumers will not accept a significant performance degradation purely for the sake of improved security. There is thus a need for a source-free technique that can efficiently monitor and constrain an arbitrary application's behavior.

Code-reuse attacks (described in more detail in Section 2.2) are one of the most significant threat classes addressed by several of the approaches advanced by this dissertation. These attacks seek to subvert the control-flow of the running application and re-route it so that the executed sequence of instructions implements some malicious shell-code. Existing defenses have tried to mitigate this threat using techniques like *address space layout randomization (ASLR)*, binary instrumentation (Chen et al., 2009; Davi et al., 2011), or compile-time inserted protections (Onarlioglu et al., 2010). Unfortunately, none of these techniques achieves the triumverate of efficiency, security, and wide applicability. Compile-time approaches are efficient, but cannot be applied to COTS binaries, limiting their reach.

Binary instrumentation techniques offer greater support, but induce significant performance overheads—500% or more in certain cases (Chen et al., 2009)—while ASLR has been shown to be ineffective against code-reuse attacks like *Return-Oriented Programming (ROP)*. Here too, there is dire need of a source-free technique that can efficiently protect binaries against this class of attacks.

My Thesis. This dissertation draws attention to the fragile state of today's software defenses by introducing and exposing new reactively adaptive malware mutation approaches that imbue weaponized software with powerful stealth capabilities that reliably defeat today's static malware detection defenses. In response, it proposes new source-free binary code transformation defensive techniques that are better suited to securing applications against next-generation offensive threats. For both tasks, the dissertation leverages the power of automated source-free binary rewriting to propose offensive and defensive solutions that are both efficient and widely deployable.

The dissertation begins by detailing two obfuscation techniques that exploit current defenses' reliance on structural information to detect malware. Both techniques create directed obfuscations that are capable of bypassing today's defenses. It then describes three defensive techniques that leverage automated binary rewriting to retroactively secure binaries against (1) next-generation malware obfuscations, (2) return-oriented programming attacks, and (3) implementation-aware code-reuse attacks. All three techniques incur low overheads, and can be applied to legacy binaries with no need for source code or debug symbols—making them a practical option for real-world deployment.

The rest of this dissertation is laid out as follows. Chapter 2 presents an overview of currently used offensive and defensive techniques, as well the challenges associated with source-free binary rewriting.

Part I describes the two malware obfuscation techniques and evaluates their effectiveness against current defenses. Both techniques make directed changes to malware with respect

to a specific target system. They rely on information gleaned either from the automated analysis of the defenses in use (Chapter 3), or by the analysis of benign binaries on the target (Chapter 4). Doing so allows for the obfuscations to exhibit continued structural similarity with the defensive tools' notion of what benign binaries look like.

Part II discusses the creation of the first compiler-agnostic and source-free x86 binary rewriter that is robust to disassembly errors and compatible with advanced compiler techniques like interleaved code and data, or position independent code. We use the rewriter to create three security systems for x86 binaries: (1) Reins (Chapter 5) provides *Software Fault Isolation (SFI)* and arbitrary policy enforcement at the system API level. (2) STIR (Chapter 6) rewrites binaries so that the order of its basic blocks is randomized on each execution—protecting against code-reuse attacks like ROP. (3) O-CFI (Chapter 7) uses a combination of fine-grained randomization and control-flow integrity to protect against advanced code-reuse attacks that seek to nullify the advantages of load-time randomization by gaining knowledge about the runtime layout of a binary. All three systems are able to rewrite legacy COTS binaries while introducing negligible amounts of overhead.

Finally, relevant related work is presented in Chapter 8 and conclusions are presented in Chapter 9.

CHAPTER 2

BACKGROUND

2.1 Malware Detection and Obfuscation

The majority of malware detection systems employed on computing devices today rely significantly upon static detection heuristics. Static malware detection tools check and classify binaries as malicious or benign before they execute. This is in contrast to dynamic detection techniques that monitor the execution of a binary and classify it at run-time. Dynamic detection techniques have access to the behavior exhibited by a binary, and are able to utilize that information to make fairly accurate classifications. However, they also suffer from higher overheads (Kim and Karp, 2004; Singh et al., 2004; Kreibich and Crowcroft, 2004; Li et al., 2006; Newsome et al., 2005) than static approaches, which makes solely dynamic detection unsuitable for performance-critical environments.

Instead, static detection techniques are used to shortlist potentially malicious binaries which can then be subjected to a more expensive dynamic scrutiny if required. The weakness of static techniques derives from the undecidability of statically inferring arbitrary program behaviors. As a result, static detection usually relies upon the syntactic—rather than semantic—analysis of binary code. That is, it attempts to classify a binary as malicious by comparing its contents and structure for similarities to known malware. For this reason, static syntactic-based techniques are commonly referred to as static signature-based defenses.

The cat-and-mouse relationship between malware detection and obfuscation can be seen by looking at currently employed obfuscation strategies, most of which seek to attack weaknesses inherent in a static, syntax-based approach to detection. The most widely used of these are briefly detailed below (Ször, 2005).

- **Oligomorphism** uses simple invertible operations, such as XOR, to transform the malicious code and hide distinguishing features. The code is then recovered by inverting the operation to deploy the obfuscated payload.

- **Polymorphism** is an advancement of the same concept that encrypts most of the malicious code before propagation, leaving only a decryption routine, which unpacks the malicious code before execution.

  More advanced polymorphic techniques, such as polymorphic blending, strenthen the obfuscation by modifying statistical information of binaries via byte padding or substitution (Fogla et al., 2006). However, the malware's decryption routine (which must remain unencrypted) is often sufficiently unique that it can be used as a signature to detect an entire family of polymorphic malware. Semantic analysis techniques can therefore single out and identify the unpacker to detect malware family members (Kruegel et al., 2005).

- **Virtualization-based obfuscators** express malware as bytecode that is interpreted at runtime by a custom VM. However, this merely shifts the obfuscation burden to concealing the (usually large) in-lined, custom VM.

- **Metamorphism** is a more advanced approach to obfuscation that, in lieu of encryption, replaces its malicious code sequences with semantically equivalent code during propagation. This is accomplished using a metamorphic engine that processes binary code and modifies it to output a structurally different but semantically identical copy. Since the mutations all consist of purely non-encrypted, plaintext code, they tend to exhibit statistical properties indistinguishable from other non-encrypted, benign software.

Both oligomorphism and polymorphism are statically detectable with high probability using statistical or semantic techniques. Encrypting or otherwise transforming the code

significantly changes statistical characteristics of the program, such as byte frequency (Wang et al., 2006; Wang and Stolfo, 2004) and entropy (Lyda and Hamrock, 2007), prompting defenses to classify them as suspicious. Subsequent, more computationally expensive analyses can then be judiciously applied to these suspicious binaries to identify malware.

Current metamorphic engines focus on achieving a high diversity of mutants in an effort to decrease the probability that the mutants share any features that can serve as a basis for signature-based detection. However, diversity does not necessarily lead to indistinguishability. For example, malware signatures that whitelist features (i.e., those that classify binaries as suspicious if they *do not* contain certain features) actually become more effective as mutant diversity increases. Similarly, reverse-engineering current metamorphic engines often reveals patterns that can be exploited to derive a suitable signature for detection.

An additional weakness to all these techniques is that once a threat has been discovered, signatures that detect it can be crafted (usually with the aid of a manual analyst) after which the malware can easily be detected and eliminated from infected machines. Although this process offers little protection from zero-days or against undiscovered threats, they suffice to ensure that malware is only a threat for the period of time between its release and its eventual discovery.

2.2 Code-Reuse Attacks and Defenses

Subverting control-flows of vulnerable programs by hijacking function pointers (e.g., return addresses) and redirecting them to shell code is a widely used methodology underlying many software attacks. For such an attack to succeed, there are two conditions: (1) the targeted software is vulnerable to redirection, and (2) the attacker-supplied shell code is executable. Consequently, to stop these attacks, a great deal of research has focused on identifying and eliminating software vulnerabilities, either through static analysis of program source

code (Larochelle and Evans, 2001) or through dynamic analysis or symbolic execution of program binary code (Cadar et al., 2006; Godefroid et al., 2008).

Meanwhile, there is also a significant amount of work focusing on how to prevent the execution of shell code based on its origin or location. Initially, attackers directly injected malicious machine code into vulnerable programs, prompting the development of W⊕X (*write-xor-execute*) protections such as DEP (Andersen, 2004) and ExecShield (van de Ven, 2004) to block execution of the injected payloads. In response, attackers began to redirect control flows directly to potentially dangerous code already present in victim process address spaces (e.g., in standard libraries), bypassing W⊕X. Return-into-libc attacks (Solar Designer, 1997) and return oriented programming (ROP) (Shacham, 2007; Buchanan et al., 2008; Checkoway et al., 2010) are two major categories of such attacks. As a result, address space layout randomization (ASLR) (PaX Team, 2003; Bhatkar et al., 2005) was invented to frustrate attacks that bypass W⊕X.

ASLR has significantly raised the bar for standard library-based shell code because attackers cannot predict the addresses of dangerous instructions to which they wish to transfer control. However, a recent attack from Q (Schwartz et al., 2011) has demonstrated that attackers can alternatively redirect control to shell code constructed from *gadgets* (i.e., short instruction sequences) already present in the application binary code. Such an attack is extremely dangerous since instruction addresses in most application binaries are fixed (i.e., static) once compiled (except for position independent code). This allows attackers to create robust shell code for many binaries (Schwartz et al., 2011).

2.3 Binary Rewriting and In-lined Reference Monitors

Software is often released in binary form. There are numerous distribution channels, such as downloading from the vendor's web site, sharing through a P2P network, or sending via email attachments. All of these channels can introduce and distribute malicious code. Thus,

it is very common for end-users to possess known but not fully trusted binary code, or even unknown binaries that they are lured to run. To date, there are two major classes of practical mechanisms to protect users while running such binaries. One is a heavy-weight approach that runs the binary in a contained virtual machine (VM) (Ford and Cox, 2008; Scott and Davidson, 2002; Payer and Gross, 2011). The other is a lighter-weight approach that runs them in a sandboxing environment with an in-lined reference monitor (IRM) (Wahbe et al., 1993; Schneider, 2000; Yee et al., 2009).

The VM approach is appealing for several reasons. First, it avoids the problem of statically disassembling CISC binaries. Instead, VMs dynamically translate binary code with the aid of just-in-time binary translation (Ford and Cox, 2008; Scott and Davidson, 2002; Payer and Gross, 2011; Kiriansky et al., 2002). This allows dynamically computed jump targets to be identified and disassembled on the fly. Second, VMs can intercept API calls and filter them based on a security policy. Third, even if damage occurs, it can potentially be contained within the VM. Therefore, the VM approach has been widely used in securing software and analyzing malicious code.

However, production-level VMs can be extremely large relative to the untrusted processes they guard, introducing significant computational overhead when they are applied to enforce fine-grained policies. Their high complexity also makes them difficult to formally verify; a single bug in the VM implementation leaves users vulnerable to attack. Meanwhile, there is an air-gap if the binary needs to access host files, and VM services must also bridge the semantic-gap (Chen and Noble, 2001). While lighter-weight VM alternatives, such as program shepherding (Kiriansky et al., 2002), lessen some of these drawbacks, they still remain larger and slower than IRMs.

On the other hand, a large body of past research including SFI (Wahbe et al., 1993), PittSFIeld (McCamant and Morrisett, 2006), CFI (Abadi et al., 2009), XFI (Erlingsson et al., 2006), and NaCl (Yee et al., 2009), has recognized the many advantages of client-side, static,

binary-rewriting for securing untrusted, mobile, native code applications. Binary-rewriting boasts great deployment flexibility since it can be implemented separately from the code-producer (e.g., by the code-consumer or a third party), and the rewritten code can be safely and transparently executed on machines with no specialized security hardware, software, or VMs. Moreover, it offers superior performance to many VM technologies since it statically in-lines a light-weight VM logic directly into untrusted code, avoiding overheads associated with context-switching and dynamic code generation. Finally, safety of rewritten binaries can be machine-verified automatically—in the fashion of proof-carrying-code (Necula, 1997)—allowing rewriting to be performed by an untrusted third party.

Unfortunately, all past approaches to rewriting native binary code require some form of cooperation from code-producers. For example, Google's Native Client (NaCl) (Yee et al., 2009) requires a special compiler to modify the client programs at the source level and use NaCl's trusted libraries. Likewise, Microsoft's CFI (Abadi et al., 2009) and XFI (Erlingsson et al., 2006) requires code-producers to supply a *program database* (PDB) file (essentially a debug symbol table) with their released binaries. Earlier works such as PittSFIeld (McCamant and Morrisett, 2006) and SASI (Erlingsson and Schneider, 1999) require code-producers to provide `gcc`-produced assembly code. Code that does not satisfy these requirements cannot be rewritten and is therefore conservatively rejected by these systems. These restrictions have prevented binary-rewriting from being applied to the vast majority of native binaries because most code-producers do not provide such support and are unlikely to do so in the near future.

2.4 Challenges with Source-Free Disassembly

To rewrite binaries in a way that preserves intended functionality, a rewriter must correctly handle instructions that are shifted to new locations—ensuring not only that all relative data references are updated, but also that any relevant control flows are correctly repointed.

This in turn relies on obtaining an accurate disassembly of the binary; without it, neither data references nor control flows can be recovered. However, doing so statically and without access to source code or debug information entails numerous challenges.

1. **Disassembly undecidability:** It is not possible in general to fully disassemble arbitrary x86 binaries purely statically. All static disassemblers rely on heuristics to find the reachable code amidst the data, and even the best disassemblers frequently guess incorrectly even for non-malicious, non-obfuscated binaries (Wartell et al., 2011). Solutions that assume fully correct disassemblies are therefore impractical for real-world, legacy, COTS binaries.

2. **Interleaved Code and Data:** Modern compilers aggressively interleave static data within code sections in both PE and ELF binaries for performance reasons. In the compiled binaries there is generally no means of distinguishing the data bytes from the code. Inadvertently randomizing the data along with the code breaks the binary, introducing difficulties for instruction-level randomizers. Viable solutions must somehow preserve the data whilst randomizing all the reachable code.

3. **Computed jumps:** Native x86 code often dynamically computes jump destinations from data values at runtime. Such operations pervade almost all x86 binaries; for example, binaries compiled from object-oriented languages typically draw code pointers from data in method dispatch tables. These pointers can undergo arbitrary binary arithmetic before they are used, such as logic that decodes them from an obfuscated representation intended to thwart buffer overrun attacks.

   Preserving the semantics of computed jumps after stirring requires an efficient means of dynamically identifying and re-pointing all code pointers to the relocated instruction addresses. Prior work, such as static binary rewriters for software fault isolation (Wahbe et al., 1993; Small and Seltzer, 1996; Erlingsson and Schneider, 1999;

McCamant and Morrisett, 2006), relies upon compile-time support to handle this. However, randomizing legacy code for which there is no source-level relocation or debug information requires a new solution.

4. **Callbacks:** A callback occurs when the OS uses a code pointer previously passed from the program as a computed jump destination. Such callbacks are a mainstay of event-driven applications. Unlike typical computed jumps, callback pointers are not used as jump targets by any instruction visible to the randomizer. The only instructions that use them as jump targets are within the OS. This makes these code pointers especially difficult to identify and re-point correctly.

5. **Position-dependent instructions:** Instructions whose behavior will break if they are relocated within the section that contains them are said to be *position-dependent*. Ironically, position-dependent instructions are typically found within blocks of *position independent code* (PIC)—code sections designed to be relocatable *as a group* at load-time or runtime (Oracle Corporation, 2010). The position independence of such code is typically achieved via instructions that dynamically compute their own addresses and expect to find the other instructions of the section at known offsets relative to themselves. Such instructions break if relocated within the section, introducing difficulties for more fine-grained, instruction-level randomization.

PART I

MALWARE OFFENSE

CHAPTER 3

EXPLOITING AN ANTIVIRUS INTERFACE[1]

Static signature-based malware detectors identify malware by scanning untrusted binaries for distinguishing byte sequences or *features*. Features unique to malware are maintained in a *signature database*, which must be continually updated as new malware is discovered and analyzed.

Signature-based malware detection generally enforces a static approximation of some desired dynamic (i.e., behavioral) security policy. For example, access control policies, such as those that prohibit code injections into operating system executables, are statically undecidable and can therefore only be approximated by any purely static decision procedure such as signature-matching. A signature-based malware-detector approximates these policies by identifying syntactic features that tend to appear only in binaries that exhibit policy-violating behavior when executed. This approximation is both unsound and incomplete in that it is susceptible to both false positive and false negative classifications of some binaries. For this reason signature databases are typically kept confidential, since they contain information that an attacker could use to craft malware that the detector would misclassify as benign, defeating the protection system. The effectiveness of signature-based malware detection thus depends on both the comprehensiveness and confidentiality of the signature database.

Traditionally, signature databases have been manually derived, updated, and disseminated by human experts as new malware appears and is analyzed. However, the escalating

[1]This chapter contains material previously published as: Kevin W. Hamlen, Vishwath Mohan, Mohammad M. Masud, Latifur Khan, and Bhavani Thuraisingham. Exploiting an Antivirus Interface. Computer Standards & Interfaces Journal, 31(6):1182–1189, April 2009.

rate of new malware appearances and the advent of self-mutating, polymorphic malware over the past decade have made manual signature updating less practical. This has led to the development of automated data mining techniques for malware detection (Kolter and Maloof, 2004; Schultz et al., 2001; Masud et al., 2008), that are capable of automatically inferring signatures for previously unseen malware.

In this chapter we show how these data mining techniques can also be applied by an attacker to discover ways to obfuscate malicious binaries so that they will be misclassified as benign by the detector. Our approach hinges on the observation that although malware detectors keep their signature databases confidential, all malware detectors reveal one bit of signature information every time they reveal a classification decision. This information can be harvested particularly efficiently when it is disclosed through a public interface. The classification decisions can then be delivered as input to a data mining malware detection algorithm to infer a model of the confidential signature database. From the inferred model we derive feature-removal and feature-insertion obfuscations that preserve the behavior of a given malware binary but cause it to be misclassified as benign. The result is an obfuscation strategy that can defeat any purely static signature-based malware detector.

We demonstrate the effectiveness of this strategy by successfully obfuscating several real malware samples to defeat malware detectors on Windows operating systems. Windows-based antivirus products typically support Microsoft's `IOfficeAntivirus` interface (Microsoft Developer Network (MSDN) Digital Library, 2009), which allows applications to invoke any installed antivirus product on a given binary and respond to the classification decision. Our experiments exploit this interface to obtain confidential signature database information from several commercial antivirus products.

The rest of this section is organized as follows. Section 3.1 provides an overview of our approach, Section 3.2 describes a data mining-based malware detection model, and Section 3.3 discusses methods of deriving binary obfuscations from a detection model. Section 3.4 then

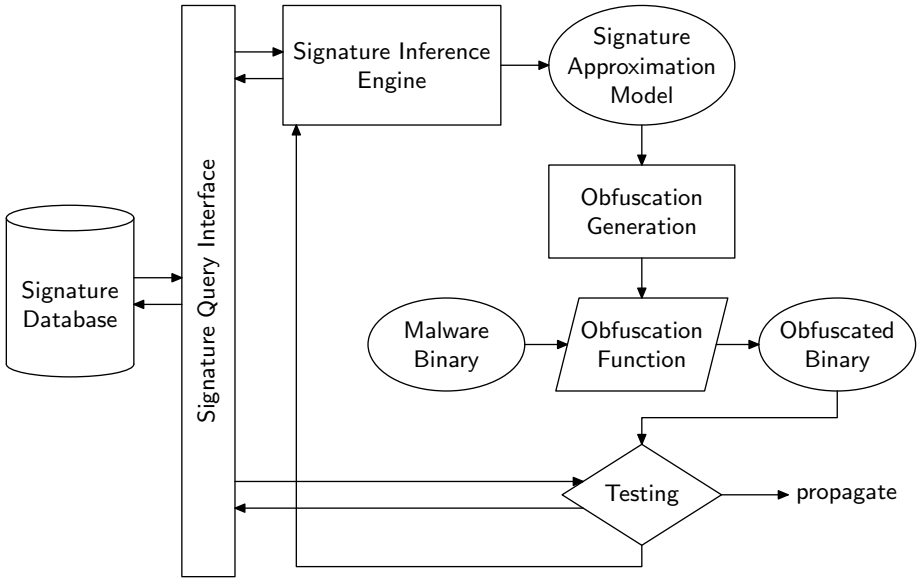

Figure 3.1. Binary Obfuscation Architecture

describes experiments and evaluation of our technique. Section 3.5 concludes with discussion and suggestions for future work.

3.1 Overview

The architecture of our binary obfuscation methodology is illustrated in Figure 3.1. We begin by submitting a diverse collection of malicious and benign binaries to the victim signature database via the signature query interface. The interface reveals a classification decision for each query. For our experiments we used the `IOfficeAntivirus` COM interface that is provided by Microsoft Windows operating systems (Windows 95 and later) (Microsoft Developer Network (MSDN) Digital Library, 2009). The `Scan` method exported by this interface takes a filename as input and causes the operating system to use the installed antivirus product to scan the file for malware infections. Once the scan is complete, the method returns a success code indicating whether the file was classified as malicious or benign. This allows applications to request virus scans and respond to the resulting classification decisions.

We then use the original inputs and resulting classification decisions as a training set for an inference engine. The inference engine learns an approximating model for the signature database using the training set. In our implementation, this model was expressed as a decision tree in which each node tests for the presence or absence of a specific binary n-gram feature that was inferred to be security-relevant by the data mining algorithm.

This inferred model is then reinterpreted as a recipe for obfuscating malware so as to defeat the model. tree encodes a set of binary features that, when added or removed from a given malware sample, causes the resulting binary to be classified as malicious or benign by the model. The obfuscation problem is thus reduced to finding a binary transformation that, when applied to malware, causes it to match one of the benignly-classified feature sets. In addition, the transformation must not significantly alter the behavior of the malware binary being obfuscated. Currently we identify suitable feature sets by manual inspection, but we believe that future work could automate this process.

Once such a feature set is identified and applied to the malware sample, the resulting obfuscated sample is submitted as a query to the original signature database. A malicious classification indicates that the inferred signature model was not an adequate approximation for the signature database. In this case the obfuscated malware is added to the training set and training continues, resulting in an improved model, whereupon the process repeats. A benign classification indicates a successful attack upon the malware detector. In our experiments we found that repeating the inference process was not necessary; our obfuscations produced misclassified binaries after one round of inference.

3.2 A data mining based malware detection model

A data mining-based malware detector first trains itself with known instances of malicious and benign executables. Once trained, it can predict the proper classifications of previously

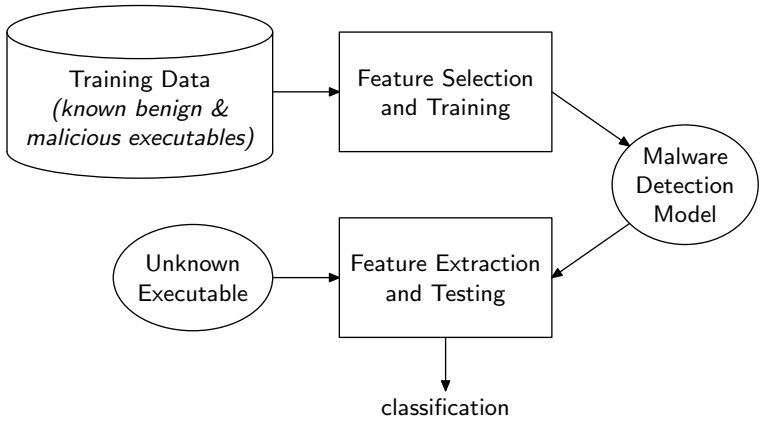

Figure 3.2. A data mining-based malware detection framework

unseen executables by testing them against the model. The high-level framework of such a system is illustrated in Figure 3.2.

The predictive accuracy of the model depends on the given training data and the learning algorithm (e.g., support vector machine, decision tree, naïve bayes, etc.) Several data mining-based malware detectors have been proposed in the past (Kolter and Maloof, 2004; Schultz et al., 2001; Masud et al., 2008). The main advantage of these models over the traditional signature-based models is that data mining-based models are more robust to changes in the malware. Signature-based models fail when new malware appears with an unknown signature. On the other hand, data mining-based models generalize the classification process by learning a suitable malware model dynamically over time. Thus, they are capable of detecting malware instances that were not known at the time of training. This makes it more challenging for an attacker to defeat a malware detector based on data mining.

Our previous work on data mining-based malware detection (Masud et al., 2008) has developed an approach that consists of three main steps:

1. feature extraction, feature selection, and feature-vector computation from the training data,

2. training a classification model using the computed feature-vector, and

3. testing executables with the trained model.

These steps are detailed throughout the remainder of the section.

3.2.1 Feature extraction

In past work we have extracted three different kinds of features from training instances (i.e., executable binaries):

1. **Binary n-gram features:** In order to extract these features, we consider each executable as a string of bytes and extract all possible n-grams from the executables, where n ranges from 1 to 10.

2. **Assembly n-gram features:** We also disassemble each executable to obtain an assembly language program. We then extract n-grams of assembly instructions.

3. **Dynamic link library (DLL) call features:** Library calls are particularly relevant for distinguishing malicious binaries from benign binaries. We extract the library calls from the disassembly and use them as features.

When deriving obfuscations to defeat existing malware detectors we found that restricting our attention only to binary n-gram features sufficed for our experiments reported in Section 3.4. However, in future work we intend to apply all three feature sets to produce more robust obfuscation algorithms.

Binary n-gram feature extraction: To extract features, we first apply the UNIX `hexdump` utility to convert the binary executable files into textual *hexdump files*, which contain the hexadecimal numbers corresponding to each byte of the binary. This process is performed to ensure safe and easy portability of the binary executables. The feature extraction process consists of two phases: (1) feature collection, and (2) feature selection.

The feature collection process proceeds as follows. Let the set of hexdump training files be $\mathcal{H} = \{h_1, ..., h_b\}$. We first initialize a set L of n-grams to empty. Then we scan each hexdump file h_i by sliding an n-byte window over its binary content. Each recovered n-byte sequence is added to L as an n-gram. For each n-gram $g \in L$ we count the total number of positive instances p_g (i.e., malicious executables) and negative instances n_g (i.e., benign executables) that contain g.

There are several implementation issues related to this basic approach. First, the total number of n-grams may be very large. For example, the total number of 10-grams in our dataset is 200 million. It may not possible to store all of them in computer's main memory. Presently we solve this problem by storing the n-grams in a large disk file that is processed via random access. Second, if L is not sorted, then a linear search is required for each scanned n-gram to test whether it is already in L. If N is the total number of n-grams in the dataset, then the time for collecting all the n-grams would be $O(N^2)$, an impractical amount of time when $N = 200$ million. In order to solve the second problem, we use an Adelson-Velsky-Landis (AVL) tree (Goodrich and Tamassia, 2005) to index the n-grams. An AVL tree is a height-balanced binary search tree. This tree has a property that the absolute difference between the heights of the left sub-tree and the right sub-tree of any node is at most one. If this property is violated during insertion or deletion, a balancing operation is performed, and the tree regains its height-balanced property. It is guaranteed that insertions and deletions are performed in logarithmic time. Inserting an n-gram into the database thus requires only $O(\log_2(N))$ searches. This reduces the total running time to $O(N \log_2(N))$, making the overall running time about 5 million times faster when N as large as 200 million. Our feature collection algorithm implements these two solutions.

Feature selection: If the total number of extracted features is very large, it may not possible to use all of them for training. Aside from memory limitations and impractical computing times, a classifier may become confused with a large number of features because

most of them would be noisy, redundant, or irrelevant. It is therefore important to choose a small, relevant and useful subset of features for more efficient and accurate classification. We choose information gain (IG) as the selection criterion because it is recognized in the literature as one of the best criteria isolating relevant features from large feature sets. IG can be defined as a measure of effectiveness of an attribute (i.e., feature) in classifying a training data (Mitchell, 1997). If we split the training data based on the values of this attribute, then IG gives the measurement of the expected reduction in entropy after the split. The more an attribute can reduce entropy in the training data, the better the attribute is for classifying the data.

The next problem is to select the best S features (i.e., n-grams) according to IG. One naïve approach is to sort the n-grams in non-increasing order of IG and select the top S of them, which requires $O(N \log_2 N)$ time and $O(N)$ main memory. But this selection can be more efficiently accomplished using a heap that requires $O(N \log_2 S)$ time and $O(S)$ main memory. For $S = 500$ and $N = 200$ million, this approach is more than 3 times faster and requires 400,000 times less main memory. A heap is a balanced binary tree with the property that the root of any sub-tree contains the minimum (maximum) element in that sub-tree. First we build a min-heap of size S. The min-heap contains the minimum-IG n-gram at its root. Then each n-gram g is compared with the n-gram at the root r. If $IG(g) \leq IG(r)$ then we discard g. Otherwise, r is replaced with g, and the heap is restored.

Feature vector computation: Suppose the set of features selected in the above step is $\mathcal{F} = \{f_1, ..., f_S\}$. For each hexdump file h_i, we build a binary feature vector $h_i(\mathcal{F}) = \{h_i(f_1), ..., h_i(f_S)\}$, where $h_i(f_j) = 1$ if h_i contains feature f_j, or 0 otherwise. The training algorithm of a classifier is supplied with a tuple $(h_i(\mathcal{F}), l(h_i))$ for each training instance h_i, where $h_i(\mathcal{F})$ is the feature vector and $l(h_i)$ is the class label of the instance h_i (i.e., positive or negative).

3.2.2 Training

We apply SVM, Naïve Bayes (NB), and decision tree (J48) classifiers for the classification task. SVM can perform either linear or non-linear classification. The linear classifier proposed by Vapnik (Boser et al., 1992) creates a hyperplane that separates the data points into two classes with the maximum margin. A maximum-margin hyperplane is the one that splits the training examples into two subsets such that the distance between the hyperplane and its closest data point(s) is maximized. A non-linear SVM (Cortes and Vapnik, 1995) is implemented by applying a kernel trick to maximum-margin hyperplanes. This kernel trick transforms the feature space into a higher dimensional space where the maximum-margin hyperplane is found, through the aid of a kernel function.

A decision tree contains attribute tests at each internal node and a decision at each leaf node. It classifies an instance by performing the attribute tests prescribed by a path from the root to a decision node. Decision trees are rule-based classifiers, allowing us to obtain human-readable classification rules from the tree. J48 is the implementation of the C4.5 Decision Tree algorithm. C4.5 is an extension of the ID3 algorithm invented by Quinlan (2003). In order to train a classifier, we provide the feature vectors along with the class labels of each training instance that we have computed in the previous step.

3.2.3 Testing

Once a classification model is trained, we can assess its accuracy by comparing its classification of new instances (i.e., executables) to the original victim malware detector's classifications of the same new instances. In order to test an executable h, we first compute the feature vector $h(\mathcal{F})$ corresponding to the executable in the manner described above. When this feature vector is provided to the classification model, the model outputs (predicts) a class label $l(h)$ for the instance. If we know the true class label of h, then we can compare the prediction with the true label, and check the correctness of the learned model. If the

model's performance is inadequate, the new instances are added to the training set resulting in an improved model, and testing resumes.

In the next section, we describe how the model yielded by the above process can be used to derive binary obfuscations that defeat the model.

3.3 Model-reversing Obfuscations

Malware detectors based on static data-mining attempt to learn correlations between the syntax of untrusted binaries and the (malicious or benign) behavior that those binaries exhibit when executed. This learning process is necessarily unsound or incomplete because most practically useful definitions of "malicious behavior" are Turing-undecidable. Thus, every purely static algorithm for malware detection is vulnerable to false positives, false negatives, or both. Our obfuscator exploits this weakness by discovering false negatives in the model inferred by a static malware detector.

The decision tree model inferred in the previous section can be used as a basis for deriving binary obfuscations that defeat the model. The obfuscation involves adding or removing features (i.e., binary n-grams) to and from the malware binary so that the model classifies the resulting binary as benign. These binary transformations must be carefully crafted so as to avoid altering the runtime behavior of the malware program lest they result in a policy-adherent or non-executable binary.

A simple example will illustrate. Figure 3.3 shows a simple decision tree model for malware detection. Each internal node in the tree denotes a feature test. For example, the root has feature test f_1. A test instance x (i.e., executable) is first tested against the root. If x has feature f_1, then the detector follows the left branch (True); otherwise it follows the right branch (False). This process continues until the detector arrives at a leaf node. Leaf nodes denote classification decisions, expressed as a minus sign (benign) or plus sign

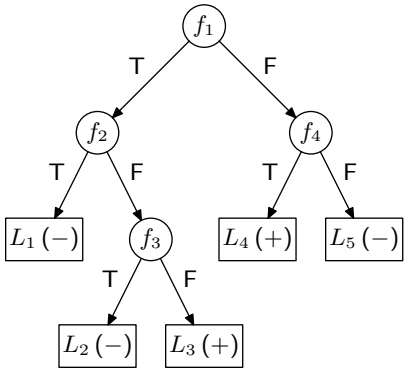

Figure 3.3. An example of a decision tree-based malware detection model

(malicious). For example, instances that cause the detector to arrive in leaf node L_1 will be classified as benign.

According to the model in Figure 3.3, malware having features f_1 and f_2 will be classified as benign. Likewise, malware lacking both f_1 and f_4 will also be classified as benign. Thus, an obfuscation that inserts features f_1 and f_2 into a binary, or removes both f_1 and f_4 from the binary, without altering the runtime behavior of the binary, suffices to conceal malware from the detector.

3.3.1 Path Selection

We begin the obfuscation process by searching for a candidate path through the decision tree that ends in a benign leaf node. Our goal will be to add and remove features from the malicious executable x so as to cause the detector to follow the chosen decision tree path during classification. Since the path ends in a benign-classifying decision node, this will cause the malware to be misclassified as benign by the detector.

Each path from the root to a leaf node in a decision tree can be thought of as a classification rule composed of the conjunction of the conditions encoded by each node in the path. For example, in Figure 3.3 the path from the root to leaf node L_2 encodes the following rule (R_2):

$$R_2 : f_1 \wedge \neg f_2 \wedge f_3 \implies \mathit{benign}$$

which says that if a test instance has feature f_1 and does not have feature f_2, and has feature f_3, then the instance is benign. Here the negated term $\neg f_2$ indicates that this pattern must not be present in the test instance.

While we believe that it is possible in theory to obfuscate any executable binary to satisfy any given rule at classification-time, some rules are significantly easier to realize than others for any given binary. In general, feature removal tends to be more difficult to implement than feature addition. Thus, to minimize the number of feature additions, for each benign-classifying rule in the decision tree we count the number of negated conjunct terms corresponding to features that are present in x. The path with the fewest such terms is likely to be easiest to implement and is therefore chosen as the candidate rule.

Once a candidate rule R is selected, we evaluate each conjunct term of R against binary x. Each such test may either succeed or fail for x. Negated feature tests $\neg f$ may fail because feature f is present in x, and non-negated feature tests f may fail because feature f is not present in x. Let $\mathcal{F}_r$ be the set of features f such that $\neg f$ is a term in rule R and test $\neg f$ fails for x, and let $\mathcal{F}_a$ be the set of features f such that f is a term in rule R and test f fails for x. Adding all features in $\mathcal{F}_a$ to x and removing all features in $\mathcal{F}_r$ from x would cause the resulting binary to satisfy rule R. If these feature-additions and feature-removals can be implemented without changing the runtime behavior of x, then these transformations suffice to successfully obfuscate x so that its malicious behavior is not detected. In the following sections we discuss strategies for implementing feature-removals and feature-additions to successfully obfuscate x in this way.

3.3.2 Feature Insertion

Inserting new features into executable binaries without significantly altering their runtime behavior tends to be a fairly straightforward task. We discuss several strategies for adding features to x86 Portable Executable (PE) binary files. While the discussion is specific to PE files, we believe these strategies can also be extended to other binary formats.

The x86 PE binary format is composed of sections of binary data. The starting file offset of each section is linked from a header at the beginning of the file, or from fields within other sections reachable via the header. PE files can therefore be thought of as tree data structures rooted at the file header; the system loader does not typically process them sequentially from beginning to end. Bytes appearing outside of any section are therefore completely ignored by the system loader.

Thus, one easily implementable strategy for transparently inserting a new feature into a PE file is to simply append the feature bytes to the end of the file, or to insert them between existing sections. These bytes will be ignored by the system loader and will not be present in the process memory image when the binary is loaded. They will therefore have no effect upon the runtime behavior of the process.

In our tests reported in Section 3.4 we found that this simple feature-insertion sufficed to defeat the detectors we tested. However, a more sophisticated detector might limit its feature-selection process to reachable PE sections in order to defeat this attack. To counter this defense, we could have introduced the features to existing sections in one of the following ways:

- *Add the feature to a non-loaded section.* Each section in a PE file includes a flag that specifies whether the section is loaded into memory at runtime. contain meta-data that is useful for tools such as debuggers but that is not used during execution. Features can therefore be safely added to new or existing non-loaded sections without affecting the program's runtime behavior.

- *Append the feature to a dynamically-sized section.* Some loaded sections, such as the heap, grow at runtime. These sections have two different length specifiers in PE files—one specifies the size of any statically initialized data that is loaded from the PE file into the segment at process start, and the other specifies the amount of memory to allocate (but not initialize) for the entire section at load time.

Feature data can be safely added to these growable sections by appending it to the statically initialized data for the section. This feature data will be loaded into the section at process start but will be overwritten as the section grows. It will only affect runtime behavior of programs that read uninitialized heap memory before it is allocated. Since no standard memory manager does this, this is a simple and safe way to add features to loaded sections without changing the behavior of most programs.

- *Insert the feature into the code segment as dead (i.e., unreachable) code.* This strategy involves inserting the feature bytes into unused portions of the code segment, possibly by shifting existing code blocks to make room. We discuss this technique in more detail below.

Dead code insertion is more difficult to implement than the other feature insertion strategies, but has the advantage that it is provably undecidable for any purely static detector to reliably isolate these features from the rest of the code. Dead code identification is a well known undecidable problem for any architecture that includes conditional or computed jumps. Thus, the feature will not be safely discardable during feature selection, and will therefore be included in the decision tree by any classifier that depends on a static analysis for feature selection.

Dead code insertion cannot be computably implemented for arbitrary binaries, but it can be implemented as long as a control-flow graph for the binary code is known. This is a reasonable assumption if we assume that the attacker has access to the malware source code. Most existing compilers for x86 architectures already insert small blocks of static data or padding between methods using this information, and assemblers have directives for doing this as well. Dead code can be safely inserted using either of these established techniques.

3.3.3 Feature Removal

Removal of a feature from an executable binary is more difficult to implement without changing the program's runtime behavior. There are two major techniques for doing this implemented by existing malware:

- encryption (polymorphic malware), and
- code mutation (metamorphic malware).

Polymorphic malware encrypts the majority of its code and data using a random key. This payload is then decrypted at runtime and executed. The payload can be re-encrypted using a different key during propagation, creating many syntactically different but functionally identical variants.

Features found in the encrypted payload cyphertext of a polymorphic virus or worm can typically be removed simply by choosing a different encryption key. With a large enough key space and a sufficiently diverse collection of cyphertexts, the probability of finding a cyphertext that includes none of the disallowed features can be raised arbitrarily high. Non-polymorphic malware can be made polymorphic by wrapping it in a polymorphic propagation system.

Thus, polymorphic malware propagation reduces the feature space available to a detector to the two remaining file portions that cannot be encrypted: the file meta-data, and the decryption kernel. File meta-data cannot be encrypted because the system loader must be able to parse it in order for the binary to be executable. The decryption kernel that decrypts the payload cannot be encrypted without introducing a new decryption kernel, hence reintroducing the issue.

Although PE meta-data cannot be safely encrypted, the meta-data of typical x86 PE files can easily be crafted to be identical to that of known benign executables. A detector that does not reject important benign programs cannot, therefore, reliably distinguish malicious

instances from non-malicious instances based on features found in this standard meta-data. Thus, we henceforth limit our attention to removing features from the decryption kernel.

The decryption kernel of a polymorphic worm is a relatively small code stub at or near the program entrypoint that decrypts the encrypted payload and then branches to the newly decrypted code. Removing a feature from the decryption kernel requires replacing it with a functionally equivalent byte sequence. Metamorphic malware engines achieve this by randomly applying a set of known code equivalence transformations to the decryption code to produce syntactically different but functionally identical code. One of the simplest such transformations is to randomly insert `nop` (no-operation) instructions between various instructions, which will be ignored by the processor at runtime.

While `nop`-insertion is an effective feature-removal strategy for some malware detectors, more sophisticated detectors can defend against such an attack by disregarding all `nop` instructions during feature selection.[2] To defeat such a detector, we could have resorted to a more powerful metamorphic obfuscator such as the MetaPHOR system; cf. (Walenstein et al., 2006). MetaPHOR disassembles x86 binary code to a simplified intermediate language in which common sequences of instructions are expressed as single operations. It then re-assembles new x86 binary code from the intermediate representation pseudo-randomly. That is, for any given intermediate code there exist many possible equivalent x86 instruction sequences, which are chosen randomly to create a syntactically different but functionally equivalent instruction sequence.

While polymorphism and metamorphism are powerful existing techniques for obfuscating malware against signature-based detectors, it should be noted that existing polymorphic and metamorphic malware mutates randomly. Our attack therefore differs from these existing

[2]instructions in arbitrary x86 binary code is not decidable in general due to the non-aligned nature of the instruction set and the resulting instruction sequence aliasing decision problems. However, a feature-selector based on n-grams could simply disregard all 0x90 bytes (the `nop` op-code) to defeat `nop`-insertion at the expense of losing a marginal amount of decision information.

approaches in that we choose obfuscations that are derived directly from signature database information leaked by the malware detector being attacked. Our work therefore builds upon this past work by showing how antivirus interfaces can be exploited to choose an effective obfuscation, which can then be implemented using these existing techniques.

3.4 Experiments

To test our approach, we conducted two sets of experiments. In the first experiment we attempted to collect classification data from several commercial antivirus products by querying their public interfaces automatically. In the second experiment we obfuscated a malware sample in order to defeat the data mining-based malware detector we developed in past work (Masud et al., 2008), and that is described in Section 3.2. In future work we intend to combine these two results to test fully automatic obfuscation attacks upon commercial antivirus products.

3.4.1 Dataset

We have two non-disjoint datasets. The first dataset (dataset1) contains a collection of 1,435 executables, 597 of which are benign and 838 are malicious. The second dataset (dataset2) contains 2,452 executables, having 1,370 benign and 1,082 malicious executables. The distribution of dataset1 is hence 41.6% benign and 58.4% malicious, and that of dataset2 is 55.9% benign and 44.1% malicious. This distribution was chosen intentionally to evaluate the performance of the feature sets in different scenarios. We collect the benign executables from different Windows XP, and Windows 2000 machines, and collect the malicious executables from VX Heavens (VX Heavens, 2009), which contains a large collection of malicious executables. The benign executables contain various applications found in the Windows system folder (e.g. `C:\Windows`), as well as other executables drawn from the default program installation directory (e.g., `C:\Program Files`) of various machines. Malicious

executables contain viruses, worms, trojans, and back-doors. We select only the Win32 Portable Executables (PE) in both the cases. We would like to experiment with other executable formats (e.g., ELF) in the future.

3.4.2 Interface Exploit Experiment

To test the feasibility of collecting confidential signature database information via the antivirus interface on Windows operating systems, we wrote a small utility that queries the `IOfficeAntivirus` (Microsoft Developer Network (MSDN) Digital Library, 2009) COM interface on Windows XP and Vista machines. The utility uses this interface to request virus scans of instances in dataset1. We tested our utility on four commercial antivirus products: Norton Antivirus 2009, McAfee VirusScan Plus, AVG 8.0, and Avast Antivirus 2009.

In all but Avast Antivirus we found that we were able to reliably sample the signature database using the interface. Our utility required no elevated privileges to successfully harvest this data on Windows XP and Windows Vista systems. Benign classifications had no observable effect (other than to solicit the appropriate return code from the interface), while malicious classifications had the side-effect of quarantining the executable named in the query. The quarantining process typically involved GUI activity (e.g., a pop-up window warning the user) and file activity (e.g., moving the file to a safe location), which slowed down the detection process slightly. However, we found that this activity did not prevent ongoing scan requests. On average we were able to obtain classification decisions at a rate of 2 MB/sec (4.6 files per second) on a 2Ghz Windows Vista desktop machine with a standard 5400 RPM SATA harddrive. We therefore expect that a large amount of classification data could be gathered in this way fairly easily on victim systems.

In the case of Avast Antivirus 2009 we found that the return code yielded by the interface was not meaningful—it did not distinguish between different classifications. Thus, Avast

Antivirus 2009 was not vulnerable to our attack. However, in Section 3.5 we discuss possible methods of circumventing this limitation that could be implemented in future work.

3.4.3 Model-driven Obfuscation Experiment

We next used the techniques described in Section 3.3 to obfuscate a malicious executable so as to conceal it from a data mining-based malware detector (Masud et al., 2008). To train the classifier we used the two datasets described above. Each dataset has different sizes and distributions of benign and malicious executables. We select the classification-relevant binary n-gram features using the techniques explained in Section 3.2. Then we build decision tree classifiers using the selected feature sets. Our implementation is developed in Java with JDK 1.5. We use Weka ML toolbox (Witten and Frank, 2005) for training the decision tree classifier (the C4.5 algorithm).

In order to evaluate our technique on malware obfuscation, we chose to obfuscate the *Win32.Navidad.a* virus using our technique. Our malware detection model M sucessfully classified this as malware. In order to defeat the model, the malware was obfuscated via the following steps:

1. Generate the binary feature vector corresponding to the malware x using our technique described in Section 3.2.3. Let the feature vector be $\mathcal{F}(x) = \{f_1(x), ..., f_n(x)\}$, where $f_i(x)$ is either 0 or 1 depending on whether the feature (i.e., n-gram) f_i is absent or present in x.

2. Analyze the decision tree model M and identify a candidate rule R as described in Section 3.3.1.

3. Identify the features that must be inserted or remove to satisfy R.

4. Insert and/or remove the necessary features to/from the malware using a hexadecimal editor.

In the case of *Win32.Navidad.a*, we only needed to insert features in order to successfully defeat the model. The resulting obfuscated binary was misclassified by the detector as benign. We also verified through informal testing that the obfuscated malware still had identical functionality to the original malware.

Although we performed this obfuscation manually, we believe it would be fairly easy to fully automate this process. One obvious approach would be to apply existing polymorphic and metamorphic malware obfuscation engines such as MetaPHOR; cf. (Walenstein et al., 2006) to feature-containing portions of the malware using a succession of randomly chosen cryptographic keys and seed values until the unwanted features are removed. We intend to investigate such an approach in future work.

3.5 Conclusion

In this chapter we have outlined a technique whereby antivirus interfaces that reveal classification decisions can be exploited to infer confidential information about the underlying signature database. These classification decisions can be used as training inputs to data mining-based malware detectors. Such detectors will learn an approximating model for the signature database that can be used as a basis for deriving binary obfuscations that defeat the signature database. We conjecture that this technique could be used as the basis for effective, fully automatic, and targeted attacks against signature-based antivirus products.

Our experiments justify this conjecture by demonstrating that classification decisions can be reliably harvested from several commercial antivirus products on Windows operating systems by exploiting the Windows public antivirus interface. We also demonstrated that effective obfuscations can be derived for real malware from an inferred model by successfully obfuscating a real malware sample using our model-reversing obfuscation technique. The obfuscated malware defeated the detector from which the model was derived.

Our signature database inference procedure is not an effective attack against one commercial antivirus product we tested because that product does not fully support the antivirus interface. In particular, it returns the same result code irrespective of its classification decision for the submitted binary file. However, we believe this limitation could be overcome by an attacker in at least two different ways:

First, although the return code does not divulge classification decisions, the product does display observably different responses to malicious binaries, such as opening a quarantine pop-up window. These responses could be automatically detected by our query engine. Determining classification decisions in this way is a slower but still fully automatic process.

Second, many commercial antivirus products also exist as freely distributed, stand-alone utilities that scan for (but do not necessarily disinfect) malware based on the same signature databases used in the retail product. These light-weight scanners are typically implemented as Java applets or ActiveX controls so that they are web-streamable and executable at low privilege levels. Such applets could be executed in a restricted virtual machine environment to effectively create a suitable query interface for the signature database. The execution environment would provide a limited view of the filesystem to the victim applet and would infer classification decisions by monitoring decision-specific system calls, such as those that display windows and dialogue boxes.

From the work summarized in this chapter, we conclude that effectively concealing antivirus signature database information from an attacker is important but difficult. Current antivirus interfaces such as the one currently supported by Windows operating systems invite signature information leaks and subsequent obfuscation attacks. Antivirus products that fail to support these interfaces are less vulnerable to these attacks, however they still divulge confidential signature database information through covert channels, such as graphical responses and other side-effects.

Fully protecting against these confidentiality violations might not be feasible; however there are some obvious steps that defenders can take to make these attacks more computationally expensive for the attacker. One obvious step is to avoid implementing or supporting interfaces that divulge classification decisions explicitly and on-demand through return codes. While this prevents benign applications from detecting and responding to malware quarantines, this reduction in functionality seems reasonable in the (hopefully uncommon) context of a malware attack. Protecting against signature information leaks through covert channels is a more challenging problem. Addressing it effectively might require leveraging anti-piracy technologies that examine the current execution environment and refuse to divulge classification decisions in restrictive environments that might be controlled by an attacker. Without such protection, attackers will continue to be able to craft effective, targeted binary obfuscations that defeat existing signature-based malware detection models.

CHAPTER 4

FRANKENSTEIN[1]

The technique described in Chapter 3 is able to bypass feature based defenses by first constructing a classification model that approximates the feature detection capabilities of a target defensive tool, and then using that model to create directed changes in a malware binary. Our second technique, detailed in this chapter, pursues an alternative strategy to create directed mutations. It exploits the reliance of feature-based tools on structural information by constructing malware entirely out of parts of benign binaries. Because these binaries are classified as benign by defensive tools on the target system, any program created purely out of sequences harvested from these binaries cannot be reliably classified as malicious by static feature-based defenses.

Mutants generated in this manner will use different code sequences to implement the same malicious behavior, and in this sense Frankenstein can be thought of as an extension of metamorphism. However, while traditional metamorphic engines focus on achieving a high diversity of mutants in an effort to decrease the probability that the mutants share any features that can serve as a basis for signature-based detection, we observe that such diversity does not necessarily lead to indistinguishability. For example, malware signatures that whitelist features (i.e., those that classify binaries as suspicious if they *do not* contain certain features) actually become more effective as mutant diversity increases. Similarly, reverse-engineering current metamorphic engines often reveals patterns that can be exploited to derive a suitable signature for detection.

[1]This chapter contains material previously published as: Vishwath Mohan and Kevin W. Hamlen. Frankenstein: Stitching Malware from Benign Binaries. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, pp. 77–84, August 2012.

Our system therefore adopts a different approach to metamorphism that is inspired by recent advances in return-oriented programming. Return-oriented programming searches the address spaces of victim binaries for *gadgets*—instruction sequences that end with the return instruction (Shacham, 2007). Previous work has shown that a sufficiently large code base (such as the standard C library *libc*), suffices to find a Turing-complete set of gadgets that can be used to exploit known vulnerabilities (Roemer et al., 2012). Others have also shown that searching for gadgets is an automatable task (Schwartz et al., 2011).

We apply the idea of harvesting instructions to obfuscate malicious code. Rather than using a metamorphic engine to mutate, we stitch together harvested code sequences from benign files on the infected system to create a semantically equivalent binary. By composing the new binary entirely out of byte sequences common to benign-classified binaries, the resulting mutants are less likely to match signatures that include both whitelisting and blacklisting of binary features.

Our main contribution is a new method to obfuscate malware that works by synthesizing copies entirely from byte sequences that have already been classified as benign by local defenses. In doing so, we demonstrate a heretofore unrecognized synergy between research on metamorphic obfuscation and that on return-oriented programming.

As a proof-of-concept, we present a toy implementation consisting of a binary obfuscator that generates stand-alone x86 native code mutants from a specification (described in Section 4.1), but that does not self-propagate. Experiments focus on obfuscating code whose size and functionality is representative of the small, unencryptable portion of malware (e.g., unpackers) rather than full malware payloads, to which alternative approaches, such as mimimorphism (Wu et al., 2010), are usually applicable. Thus, we envision our approach as a complement to these alternatives rather than a replacement.

The rest of this chapter is laid out as follows. In Section 4.1 we present a high-level overview of Frankenstein and discuss its constituent components. Section 4.2 contains

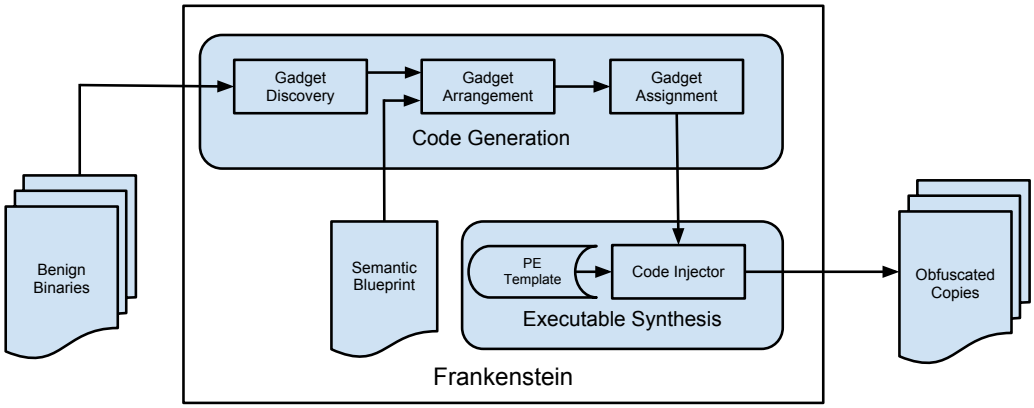

Figure 4.1. High-level architecture of Frankenstein

the details of our prototype implementation, and Section 4.3 reports experimental results. Finally, Section 4.4 summarizes and discusses opportunities for future work.

4.1 Design

Franksentein searches programs on the local machine for gadgets, which it composes to form a semantically identical copy of itself. The different components of Frankenstein are diagrammatically represented in Figure 4.1. Below, we describe our notion of a gadget, which differs from its usual definition in the context of return-oriented programming, and then discuss each of the components in more detail.

4.1.1 Gadgets

Our definition of a gadget is a more relaxed version of that needed for return-oriented exploits (Shacham, 2007; Schwartz et al., 2011). Return-oriented programming, which is a form of control-flow hijacking, relies on the `ret` instruction to transfer control from one sequence of instructions to the next. Thus, only sequences that end with a `ret` constitute viable gadgets. Since we statically stitch gadgets together, we are not bound by this

constraint. For our purposes, a gadget is any sequence of bytes that are interpretable as valid x86 instructions.

A second difference from return-oriented programming is that, as a purely static approach, we need not restrict the search to the address space of a currently running process. Gadgets are therefore harvested from executable file images on victim systems. Both these facts afford us a much larger pool of potential gadgets from which to construct mutants. This is advantageous since we would like each copy of Frankenstein to differ as much as possible from all other copies.

Gadgets are categorized by their *type* (a semantic abstraction of the kind of task they perform) and by a set of *parameters* (instantiated with values that specialize the gadget to a particular task). For example, the **MovReg** type represents any gadget that moves a value from one register to another. All **MovReg** gadgets have two parameters, *InReg* and *OutReg*, that represent the operation $\mathit{OutReg} \leftarrow \mathit{InReg}$.

A complete set of gadget types, their associated parameters, and the semantic task that each encodes is given in Table 4.1. The symbols $\bowtie_{cmp}$ and $\diamond_{aop}$ represent integer comparison and modular arithmetic operations, respectively. The collection is Turing-complete, and therefore suffices to build arbitrary computations. In contrast to gadget types for return-oriented programming (Schwartz et al., 2011), our collection includes types for conditional and non-conditional branches. These are unnecessary for return-oriented programming since in that context every gadget is reached via an appropriate return instruction injected into the stack. To better resemble benign software, Frankenstein uses more conventional control-flows that include standard branching instructions.

Every gadget is also associated with a *clobber list*, which represents secondary register and memory locations whose values the gadget modifies. The clobber list is used to find a sequence of gadgets that do not interfere with one another.

Table 4.1. Gadget types

Gadget Type (t)	**Input** (ℓ)	**Parameters** (p)	**Semantic Definition**
NoOp	—	—	No change to memory or registers
DirectBranch	*Offset*	—	$EIP \leftarrow EIP + Offset$
DirectConditionalBranch	*Offset*	$\bowtie_{cmp}$, Reg_1, Reg_2	$EIP \leftarrow EIP + Offset$ if $Reg_1 \bowtie_{cmp} Reg_2$
LoadReg	*OutReg*, *InReg*	—	$OutReg \leftarrow InReg$
LoadConst	*OutReg*, *Value*	—	$OutReg \leftarrow Value$
LoadMemAddr	*OutReg*, *Addr*	—	$OutReg \leftarrow [Addr]$
LoadMemReg	*OutReg*, *AddrReg*	*Scale*, *Disp*	$OutReg \leftarrow [AddrReg * Scale + Disp]$
StoreMemAddr	*InReg*, *Addr*	—	$[Addr] \leftarrow InReg$
StoreMemReg	*InReg*, *AddrReg*	*Scale*, *Disp*	$[AddrReg * Scale + Disp] \leftarrow InReg$
Arithmetic	*OutReg*, $InReg_1$, $InReg_2$	$\diamond_{aop}$	$OutReg \leftarrow InReg_1 \diamond_{aop} InReg_2$

Table 4.2. Examples of logical predicates

Predicate	**Semantic Definition**	**Suitable Gadgets**
noop	—	NoOp
move(L_1,L_2)	$L_1 \leftarrow L_2$	All Loads/Stores
add(L_1,L_2,L_3)	$L_1 \leftarrow L_2 + L_3$	Arithmetic
sub(L_1,L_2,L_3)	$L_1 \leftarrow L_2 - L_3$	Arithmetic
jump(n, *Why*)	Jump n blueprint steps if *Why* holds	DirectBranch, ConditionalBranch

The types defined in the table are sufficient to carry out our initial experiments, but future work should consider extending the table to support obfuscation of more complex tasks.

4.1.2 Semantic Blueprint

Typical metamorphic malware recompiles itself from a bytecode intermediate language during propagation. Each mutant carries a freshly obfuscated intermediate form of itself for this purpose. (The intermediate form is data, which is easier to obfuscate than code.) In contrast, Frankenstein propagates by re-synthesizing itself from a more abstract *semantic blueprint*. The semantic blueprint is a sequence of abstract machine states, where each step in the sequence is represented as a logical predicate. Each predicate is a combination of an atomic term and zero or more *locations*. A location can be a specific register, memory address,

```
hypotenuse_squared :-
    mov(L1, '[0x401248]'),
    mov(L2, '[0x40124C]'),
    mul(L3, L1, L1), mul(L4, L2, L2)
    add(L5, L3, L4), mov('EAX', L5).
```

Figure 4.2. A semantic blueprint to compute the square of a triangle's hypotenuse

immediate value, or a variable that refers to an arbitrary register or memory location. The **jump** predicate instead has arguments consisting of a relative offset into the blueprint's list of states and an optional condition.

A subset of logical predicates used by Frankenstein is shown in Table 4.2. For example, the **move** predicate is an abstraction of the movement of a value from one location L_2 to another L_1. Thus, depending on the values of its locations, a **move** predicate might be satisfiable by any of the **Load*** or **Store*** gadget types. The flexibility of a predicate to match multiple gadget types allows the semantic blueprint to more abstractly encode *what* is computed rather than *how* the computation is carried out. This in turn allows for a diverse set of gadgets to match a given portion of the semantic blueprint.

Figure 4.2 shows how predicates can be chained together to form a clause. In the example, the memory locations `0x401248` and `0x40124C` contain the values of two sides of a right-angled triangle, and the calculated length of the square of the hypotenuse is stored in the `EAX` register. Variables L1–L5 can refer to memory locations or registers.

The level of abstraction (and with it the diversity of mutants) can be tuned by adjusting the granularity of the predicates in the blueprint. It is possible to create layers of predicates that can each be expressed as clauses of lower-layer predicates, each layer effectively abstracting higher-level operations. For example, two consecutive predicates that increment register r and then multiply it by 2 could be replaced by a single predicate that computes $2(r + 1)$. The resulting predicate would be satisfied by new gadget sequences, such as one that first multiples by 2 and then adds 2. The trade-off is the greater search time required

to discover implementations of more abstract gadgets. If the predicates are too abstract, the search becomes intractable.

Since gadget discovery is based on search, our proof-of-concept implementation expresses semantic blueprints as predicates written a logic programming language (Prolog). While a full Prolog search engine is obviously too heavy-weight for inclusion in real malware, effective gadget search does not require the full capabilities of logic programming. We expect a combination of unification and simple depth-first search to suffice, and consequently believe that a much slimmer implementation is possible.

4.1.3 Gadget Discovery

The majority of the obfuscation process involves finding a suitable set of gadgets that can be used to implement the semantic blueprint. The search process proceeds similarly to gadget searches for return-oriented programming (Schwartz et al., 2011), but with several variations reflecting our different focus (obfuscation as opposed to finding one viable sequence from a limited code base).

In the discovery phase, Frankenstein searches the local file system for binaries. From our experiments, our experience has been that 2–3 binaries from the *system32* folder suffices to provide a code base from which to harvest a diverse, Turing-complete set of gadgets on Microsoft Windows systems. Frankenstein starts by collecting byte sequences from the code sections of these binaries using a variable-length sliding window. The sliding window approach is simpler than implementing (and obfuscating) a full disassembler, and it increases the pool of available gadgets by including for consideration the many misaligned instruction sequences that all benign programs contain (but rarely execute).

Each byte sequence is passed through an instruction decoder to produce an instruction sequence. Sequences containing invalid op-codes or undesirable branches (such as calls or returns) are discarded, and the remaining sequences are tested for gadget viability using Algorithm 1.

Algorithm 1 Gadget discovery

Input: σ_0 (initial symbolic machine state), and
$[i_1, \ldots, i_n]$ (instruction sequence)
Output: $G \subseteq T \times \Phi$ (matching gadget types)

```
for j = 1 to n do
  σ_j ← E⟦i_j⟧σ_{j−1}
end for
G ← ∅
for all t ∈ T do
  if U(t, σ_n) is defined then
    φ ← U(t, σ_n)
    G ← G ∪ {(t, φ)}
  end if
end for
return G
```

Frankenstein performs gadget discovery with the aid of a small abstract evaluator $\mathcal{E} : I \to \Sigma \to \Sigma$ that defines the effect of an instruction $i \in I$ upon a symbolic machine state $\sigma \in \Sigma$. Notation $\mathcal{E}[\![i]\!]\sigma$ denotes the resulting symbolic state, where states $\sigma : \ell \to e$ map locations ℓ (*viz.*, registers, flags, and memory addresses) to symbolic expressions e. For example, each register's initial content is encoded as a fresh symbol in the initial abstract state: $\sigma(eax) = EAX$, and so on.

After composing the effects of all instructions in a candidate sequence, the final symbolic output state is *unified* with each possible gadget type $t \in T$. A gadget type t is conceptually a state predicate, possibly containing uninstantiated parameters p. Unification $\mathcal{U}(t, \sigma')$ succeeds if there exists an instantiation $\phi : \Phi = p \to \mathit{ParamVals}$ of the parameters such that substituting t according to ϕ yields a concrete predicate satisfied by symbolic state σ'. In that case the unification returns the parameter instantiation ϕ. Otherwise $U(t, \sigma')$ is undefined (and the search continues). Two instruction sequences that match a particular gadget type are considered equivalent if they have identical instantiations of all parameters, excluding the clobber list.

It is common for a large instruction sequence to be recognized as multiple valid gadget types when considering its effect on different machine state variables. For example, the instruction sequence

```
mov ebx, dword ptr [eax*4 + 0xc]
mov ecx, eax
inc ecx
```

can be used as a **LoadReg** gadget representing $ecx \leftarrow ebx$, a **loadMemReg** gadget representing $ebx \leftarrow [eax * 4 + \texttt{0xc}]$, or as an **Arithmetic** gadget representing $ecx \leftarrow ecx + 1$. In each case, the clobber list includes all other state variables that are modified. We also include additional constraints for sequences where memory indirection is involved. In our example gadget above, the value of $eax * 4$ must be a valid memory address to ensure that it does not cause a crash when executed. These constraints are expressed in the form of logical clauses as part of the arrangement layer (described below), which ensures that we only find valid solutions.

4.1.4 Gadget Arrangement

The next step involves finding a suitable combination of gadget types that match the semantic blueprint. In Frankenstein, gadget arrangement is a natural consequence of the way that the semantic blueprint is defined. The logical predicates that we define in Table 4.2 also happen to be the lowest level in the layered approach to constructing predicates described previously. We call this the *arrangement level* because all possible gadget arrangements are expressed in terms of these predicates.

Given a clause defined in terms of higher-level predicates, logic programming can be used to reduce them to multiple clauses composed entirely of arrangement layer predicates, such that the definition of each clause in turn represents one or more potential gadget arrangements. We assume that malware authors have access to arbitrary high-level representations

of the code, including requirements, design, and implementation of the payload. They can therefore use this information to express the malware in terms of the higher-level predicates.

At present, our prototype of Frankenstein does not have support for higher level predicates, and expresses blueprints using predicates from the arrangement level only. However, adding higher-level predicates is not conceptually difficult, and we plan to include this feature in future versions of our system.

4.1.5 Gadget Assignment

In the last phase, we use the discovered gadgets to find satisfiable assignments for each generated gadget arrangement. We leverage the unification process of logic programming for this purpose, which is well-suited to this problem. Frankenstein begins by converting each discovered gadget into an extended version of one of the predicates defined in Table 4.2. The extension adds two terms to each predicate: a list of clobbered locations and an identification number. The identification number associates each of these predicates with the instruction sequence that the gadget represents, while the clobber list facilitates discovery of sequences of non-interfering gadgets.

Next, the predicates that form the definition of each of the reduced clauses obtained in the gadget arrangement phase above are also extended to include variables that represent a clobber list and identification numbers. To each definition, we also add a generated list of constraints that prevent the parameters of predicates from interfering with one another. Finally, we use constraint logic programming to solve each clause. The full set of solutions obtained represent all the possible gadget assignments that implement the original semantic blueprint.

4.1.6 Executable Synthesis

For each successful gadget assignment, Frankenstein masks all external calls in the code by converting them into computed jumps. As a result, Frankenstein's mutants have no

noteworthy system calls in their import address tables, concealing them from detectors that rely upon such features for fingerprinting.

The last step is injecting the finished code into a correctly formatted binary so it can be propagated. Frankenstein has a binary parsing component and a seed binary that it uses as a template. For each mutant, it injects the code into the template file and updates all relevant metadata in the header. At this point the new mutants are natively executable programs.

4.2 Implementation

To test the viability of our approach, we created a prototype stand-alone obfuscator that takes a gadget arrangement as input and produces a working portable executable (PE) file as output. The prototype searches the local system for programs, mines them to discover gadgets, finds a suitable gadget assignment, and realizes it as a PE file. The prototype was implemented in a combination of Python and Prolog. The experiments were performed on a quad-core virtual machine with 3 GB RAM running 64-bit Windows 7. The host machine is an Intel i7 Q6500 quad-core laptop running 64-bit Windows 7.

The gadget discovery, gadget assignment, and duplication phases implement the algorithms described in the previous section. However, the abstract evaluator that analyzes and discovers gadgets currently supports only a limited subset of instructions—about 8 different instructions excluding branches. Even though this greatly reduces the number of gadgets available for incorporation into mutants, it nevertheless suffices to find more than enough gadgets to implement our sample programs.

The discovery module, implemented in Python, takes a set of binaries and a semantic blueprint as input. It outputs a series of Prolog predicates that define each discovered gadget, as well as a Prolog query that represents a viable combination of the gadget types specified in the arrangement. This is delivered to the assignment module, implemented in Prolog, which outputs all discovered solutions to the query. Each solution is then converted into its

equivalent byte sequence by the executable synthesis module, implemented in Python, which injects the byte code into a template PE file. For ease of testing, the duplication module contains pre-fabricated templates for function prologues and epilogues, which are used to modularize the synthesized byte sequence as a stand-alone function. We note that function prologues and epilogues could easily be synthesized using the gadget discovery mechanism instead, if so desired.

4.3 Experimental Results

We tested our prototype by discovering gadgets in some common Windows binaries. For our results, we only chose gadgets that contained 2–6 instructions. Our results are tabulated in Table 4.3. We recorded the number of gadgets found and time taken both with and without using the sliding window protocol discussed in Section 4.1. Surprisingly, we found that using the sliding window protocol to discover misaligned sequences increased the gadget count by only 34% on average but increased discovery time by 794%, a trade-off that does not seem worthwhile. We conjencture that increasing the number of instructions supported by the abstract evaluator will help balance these ratios somewhat, but that a better strategy is likely to be one that searches for gadgets using a simple fall-through disassembly while increasing the number of binaries mined. All results we discuss hereafter are based on the numbers for the non-sliding window algorithm.

The results show that even with the limited capacity of our prototype, 2–3 binaries are sufficient to bring the number of gadgets above 100,000. On average we discovered about 46 gadgets per KB of code, finding approximately 2338 gadgets per second.

Next, we tested the prototype's ability to synthesize working code. We chose two algorithms for our experiments: insertion sort and a loop that XORs an array of bytes using a one-time pad. Both programs contain operations commonly found within the packers

Table 4.3. Gadget discovery statistics for some Windows binaries

		Without Sliding Window		With Sliding Window	
Binary Name	**File Size** (KB)	**Gadgets Found**	**Time Taken** (s)	**Gadgets Found**	**Time Taken** (s)
gcc.exe	1327	82885	29.70	97163	172.24
calc.exe	758	41914	22.09	60390	189.86
explorer.exe	2555	89617	40.31	127859	429.56
cmd.exe	295	17514	7.17	25008	88.34
notepad.exe	175	4512	1.82	6974	24.39

Input:

L1 = address of data,
L2 = address of one-time pad,
L3 = array length,
L4 = address of encrypted output

Blueprint:

```
xor_encryption :-
  move(L5, 0),
  jump(7, L5 = (L3-1)),
  move(L6, [L1+L5*4]),
  move(L7, [L2+L5*4]),
  xor(L8, L6, L7),
  move([L4+L5*4], L8),
  add(L4, L4, 1),
  add(L5, L5, 1),
  jump(-7, always).
```

Figure 4.3. Semantic blueprint for a simple XOR oligomorphism

used by conventional malware. The semantic blueprints for these programs are shown in Figures 4.3 and 4.4.

The semantic blueprints were reproduced as Prolog queries, with extensions to predicates and added constraints to ensure non-interference between gadgets as detailed previously. In both cases, only gadgets harvested from `explorer.exe` were used. The queries produced over 10,000 viable gadget assignments each, with an average speed of 3 assignments per second.

Input:

L1 = address of array
L2 = length of array

Blueprint:

```
insertion_sort :-
  move(L3, 1),
  jump(14, L3 = L2),
  move(L4, L3),
  move(L5, [L4*4+L1]),
  jump(8, L4 = 0),
  jump(7, [L1+(L4-1)*4] < L5),
  sub(L4, L4, 1),
  move(L6, [L4*4+L1]),
  add(L4, L4, 1),
  move([L4*4+L1], L6),
  sub(L4, L4, 1),
  jump(-7, always),
  move([L4*4+L1], L5),
  add(L3, L3, 1),
  jump(-13, always).
```

Figure 4.4. Semantic blueprint for insertion sort

This high diversity can be attributed to multiple satisfiable sub-arrangements of gadgets, which can each be combined with every variation of all other sub-arrangements, leading to a combinatorially high number of unique overall arrangements. Although this might appear to produce a large number of similar variants, diversity can be ensured by harvesting gadgets from different sets of binaries and additionally by only selecting assignments that have no gadgets in common with each other.

To better understand the size increase induced by our approach, we compared the sizes of 100 mutants generated by Frankenstein for the one-time pad XOR algorithm against its corresponding compiler generated code. The compiled code was generated with C++ using Visual Studio 2010 with basic security checks turned on and optimization set to full. The mean of the sizes of the generated mutants was 48 bytes compared to the 25 bytes produced by Visual Studio. The variance in size between the generated samples was 16. This shows

Table 4.4. The number of fresh n-grams shared by at least m mutants

	mutant subset size (m out of 20)				
n	3	4	5	6	7
2	0	0	0	0	0
3	5	4	4	2	0
4	14	5	4	2	0
5	19	8	4	2	0
6	23	11	4	1	0
7	26	12	3	1	0
8	26	9	1	1	0
9	24	9	1	1	0
10	23	9	1	1	0
11	0	0	0	0	0
total	160 (2.3%)	67 (1.0%)	22 (0.3%)	11 (0.2%)	0 (0%)

that the size of a mutant can be expected to be slightly less than double the size of its optimized compiler-generated version, an increase that we feel is an acceptable cost for the benefit of obfuscation.

To assess the binary distribution of the generated mutants, we generated 20 implementations of the XOR blueprint after mining donor program `explorer.exe` for gadgets, and counted the number of n-grams that do not appear in the donor program and were shared by at least m mutants. Our results are tabulated in Table 4.4. Only about 20 such n-grams are common across 25% of our mutant population, and no n-grams are common across more than 35% of the population. In addition, all the common n-grams are relatively short; no n-grams of length $n \geq 11$ were shared. These are encouraging results because they indicate that few binary n-gram features are relevant for distinguishing malware instances from the benign programs used for gadget harvesting.

Our experimental results are promising, and suggest that developing a more comprehensive Frankenstein tool is a worthwhile endeavor. Specifically, we conjecture that a more comprehensive abstract evaluator that can analyze a greater number of instructions can

potentially find far more gadgets, and thus produce mutants that exhibit even greater diversity.

4.4 Conclusion

We presented a new way of obfuscating malware that is fundamentally different from existing metamorphic malware approaches. Rather than recompiling the code purely randomly during propagation, which leads to diverse but potentially distinguishable binary features, our system searches non-malicious programs on the local system for byte sequences that function as the building blocks for semantically equivalent but syntactically new copies. Our experiments showed that mining a few files is both sufficient to obtain high mutant diversity, and fast enough to be a practical mutation strategy.

By creating new copies entirely from byte sequences obtained from benign files, we argue that it becomes significantly more difficult for defenders to infer adequate signatures that reliably distinguish malware from non-malware on victim systems. In particular, signatures that include feature-whitelisting are less effective against our framework than against more conventional forms of obfuscation.

For future work, we intend to implement a more comprehensive system and experiments to verify and extend our preliminary results. If successful, our gadget-stitching approach will constitute a powerful tool for active defense (e.g., offensive cyber-operations), and will highlight the need for stronger, purely semantics-based defenses that place less reliance on syntactic feature detection for early warning.

PART II

DEFENSIVE SOFTWARE TECHNOLOGIES

CHAPTER 5

VERIFIED SYSTEM CALL SAFETY ENFORCEMENT[1]

In this chapter we present the first, purely static, CISC native code rewriting and in-lining system (REINS) that requires no cooperation from code-producers (i.e., is compiler-agnostic). Unlike past work, REINS can automatically rewrite large-scale, COTS, Windows applications yielded by arbitrary compilers, even with no access to source-level information (e.g., PDB files or debug symbol stores). It transparently supports a large category of production-level applications unsupportable by past efforts, including those that include event-driven OS-callbacks, dynamic linking, exceptions, multithreading, computed jumps, and mixtures of trusted and untrusted modules.

Realizing REINS for COTS x86 binary in Windows platform raises many challenges, including semantic preservation of dynamically computed jumps, code interleaved with data, function callbacks, and imperfect disassembly. We address these challenges through the design and implementation of a suite of novel techniques, including conservative disassembly and indirect jump target identification. Central to our approach is a binary transformation strategy that expects and tolerates many forms of disassembly errors by conservatively treating every byte in target code sections as both code and static data. This obviates the need for perfect disassemblies, which are seldom realizable in practice without source code.

To tame and secure unsafe logic inside the binary code, REINS automatically transforms binaries to redirect system API calls through a trusted *policy-enforcement library*. The

[1]This chapter contains material previously published as: Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 299–308, December 2012.

library thereby mediates all security-relevant API calls and their arguments before (and after) they are serviced, and uses this information to enforce safety policies over histories of these security-relevant events. Indirect control flow transfers (e.g., `call/jmp/ret`) are protected by in-lined guard code that ensures that they target safe code addresses when executed. In addition, a small, trusted verifier shifts the significant complexity of the rewriting system out of the trusted computing base (TCB) by independently certifying that rewritten binaries cannot circumvent the in-lined monitor. Thus, binaries that pass verification are guaranteed to be safe to execute. While reflective code can change its behavior in response to rewriting, verification ensures that such changes cannot effect policy-violations.

In summary, REINS makes the following contributions:

- We present the first *compiler-agnostic, machine-certifying, x86 rewriting* algorithm that supports real-world COTS binaries without any appeal to source code or debug symbols. To the best of our knowledge, all past static binary rewriting techniques require source-level or debugging information to support many COTS binary features.

- We design a set of novel techniques to support binary families for which fully correct automated disassembly is provably undecidable, including those that contain computed jumps, dynamic linking, static data interleaved with code, and untrusted callback functions invoked by the OS.

- We have implemented REINS as a proof-of-concept prototype, and tested it on a number of binaries including malware code. Our empirical evaluation shows that our system successfully preserves the behavior of non-malicious, real-world Windows applications, introducing runtime overheads of about 2.4%.

5.1 Background

5.1.1 Assumptions

The goal of our system is to tame and secure malicious code in untrusted binaries through static binary rewriting. Since a majority of malware threats currently target Windows x86 platforms, we assume the binary code is running in Microsoft Windows OS with x86 architecture. Linux binary code can be protected similarly, but is not covered by the prototype implementation covered in this chapter. (In fact, rewriting Windows binary code is much more challenging than for Linux due to the much greater diversity of Windows-targeting compilers.)

Our goal is to design a compiler-agnostic static binary rewriting technique, so we do not impose any constraints on the code-producer; it could be any Windows platform compiler, or even hand-written machine code. Debug information (e.g., PDB) is assumed to be unavailable. Like all past native code IRM systems, our fully static approach rejects attempts at self-modification; untrusted code may only implement runtime-generated code through standard system API calls, such as dynamic link library (DLL) loading. Code-injection attacks are therefore thwarted because the monitor ensures that any injected code is unreachable.

In addition, our goal is not to protect untrusted code from harming itself. Rather, we prevent modules that may have been compromised (e.g., by a buffer overflow) from abusing the system API to damage the file system or network, and from corrupting trusted modules (e.g., system libraries) that may share the untrusted module's address space. This confines any damage to the untrusted module.

5.1.2 Threat model

Attackers in our model submit arbitrary x86 binary code for execution on victim systems. Neither attackers nor defenders are assumed to have kernel-level (ring 0) privileges. Attacker-

supplied code runs with user-level privileges, and must therefore leverage kernel-supplied services to perform malicious actions, such as corrupting the file system or accessing the network to divulge confidential data. The defender's ability to thwart these attacks stems from his ability to modify attacker-supplied code before it is executed. His goal is therefore to reliably monitor and restrict access to security-relevant kernel services without the aid of kernel modifications or application source code, and without impairing the functionality of non-malicious code.

5.1.3 Attacks

The central challenge for any protection mechanism that constrains untrusted native code is the problem of taming computed jumps, which dynamically compute control-flow destinations at runtime and execute them. Attackers who manage to corrupt these computations or the data underlying them can hijack the control-flow, potentially executing arbitrary code.

While computed jumps may seem rare to those accustomed to source-level programming, they actually pervade almost all binary programs compiled from all source languages. Computed jumps typically include returns (whose destinations are drawn from stack data), method calls (which use method dispatch tables), library calls (which use import address tables), multi-way branches (e.g., switch-case), and optimizations that cache code addresses to registers.

Deciding whether any of these jumps might target an unsafe location at runtime requires statically inferring the program register and memory state at arbitrary code points, which is a well known undecidable problem. Moreover, since x86 instructions are unaligned (i.e., any byte can be the start of an instruction), computed jumps make it impossible to reliably identify all instructions in untrusted binary code; disassemblers must heuristically guess the addresses of many instruction sequences to generate a complete disassembly. Untrusted binaries (e.g., malicious code) are often specifically crafted to defeat these heuristics, thereby concealing malicious instruction sequences from analysis tools.

5.2 System Overview

Given an untrusted binary, REINS automatically transforms it so that (1) all access to system (and library) APIs are mediated by our policy enforcement library, and (2) all inter-module control-flow transfers are restricted to published entry points of known libraries, preventing execution of attacker-injected or misaligned code.

REINS' rewriter first generates a conservative disassembly of the untrusted binary that identifies all safe, non-branching flows (some of which might not actually be reachable) but not unsafe ones. The resulting disassembly encodes a control-flow policy: instructions not appearing in the disassembly are prohibited as computed jump targets. Generating even this conservative disassembly of arbitrary x86 COTS binaries is challenging because COTS code is typically aggressively interleaved with data, and contains significant portions that are only reachable via computed jumps.

To help overcome some of these challenges, our rewriter is implemented as an IDAPython (Erdélyi, 2008) program that leverages the considerable analysis power of the Hex-rays IDA Pro commercial disassembler (Hex-Rays, 2012) to identify function entrypoints and distinguish code from data in complex x86 binaries. While IDA Pro is powerful, it is not perfect; it suffers numerous significant disassembly errors for almost all production-level Windows binaries.

Thus, our rewriting algorithm's tolerance of disassembly errors is critical for success.

Our system architecture is illustrated in Figure 5.1. Untrusted binaries are first analyzed and transformed into safe binaries by a binary rewriter, which enforces control-flow safety and mediates all API calls.

A separate verifier certifies that the rewritten binaries are policy-adherent. Malicious binaries that defeat the rewriter's analysis might result in rewritten binaries that fail verification or that fail to execute properly, but never in policy violations.

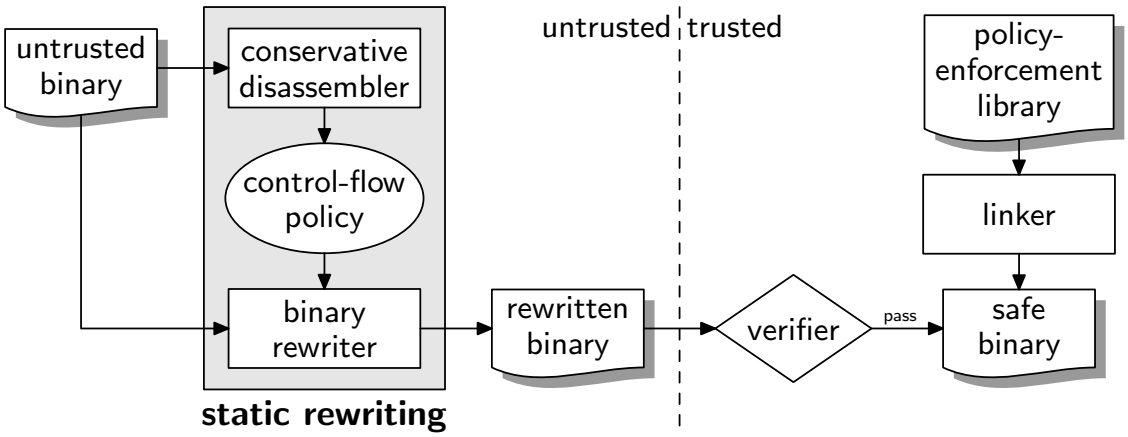

Figure 5.1. REINS architecture

5.3 Detailed Design

Rewriting Control-flow Transfers

Control Flow Safety: Our binary rewriting algorithm uses SFI (Wahbe et al., 1993) to constrain control-flows of untrusted code. It is based on an SFI approach pioneered by PittSFIeld (McCamant and Morrisett, 2006), which partitions instruction sequences into c-byte *chunks*. Chunk-spanning instructions and targets of jumps are moved to chunk boundaries by padding the instruction stream with `nop` (no-operation) instructions. This serves three purposes:

- When c is a power of 2, computed jumps can be efficiently confined to chunk boundaries by guarding them with an instruction that dynamically clears the low-order bits of the jump target.
- Co-locating guards and the instructions they guard within the same chunk prevents circumvention of the guard by a computed jump. A chunk size of $c = 16$ suffices to contain each guarded sequence in our system.
- Aligning all code to c-byte boundaries allows a simple, fall-through disassembler to reliably discover all reachable instructions in rewritten programs, and verify that all computed jumps are suitably guarded.

To allow trusted, unrewritten system libraries to safely coexist in the same address space as chunk-aligned, rewritten binaries, we logically divide the virtual address space of each untrusted process into *low memory* and *high memory*. Low memory addresses range from 0 to $d-1$ and may contain rewritten code and non-executable data. Higher memory addresses may contain code sections of trusted libraries and arbitrary data sections (but not untrusted code).

Partition point d is chosen to be a power of 2 so that a single guard instruction suffices to confine untrusted computed jumps and other indirect control flow transfers to chunk boundaries in low memory. For example, a jump that targets the address currently stored in the `eax` register can be guarded by:

```
and eax, (d − c)
jmp eax
```

This clears both the high-order and low-order bits of the target address before jumping, preventing an untrusted module from jumping directly to a system accessor function or to a non-chunk boundary in its own code, respectively. The partitioning of virtual addresses into low and high memory is feasible because rewritten code sections are generated by the rewriter and can therefore be positioned in low memory, while trusted libraries are relocatable through *rebasing* and can therefore be moved to high memory when necessary.

Preserving Good Flows: The above suffices to enforce control-flow safety, but it does not preserve the behavior of most code containing computed jumps. This is a major deficiency of many early SFI works, most of which can only be successfully applied to relatively small, `gcc`-compiled programs that do not contain such jumps. More recent SFI works have only been able to overcome this problem with the aid of source-level debug information.

Our source-free solution capitalizes on the fact that although disassemblers cannot generally identify all jumps in arbitrary binary code, modern commercial disassemblers can

heuristically identify a *superset* of all the indirect jump *targets* (though not the jumps that target them) in most binary code. This is enough information to implement a light-weight, binary lookup table that the IRM can consult at runtime to dynamically detect and correct computed jump targets before they are used. Our lookup table overwrites each old target with a tagged pointer to its new location in the rewritten code. This solves the computed jump preservation problem without the aid of source code.

Since the disassembler identifies a superset of targets and not an exact set, the lookup table implementation must be carefully designed to tolerate false positives. Misidentification of a code point as a jump target is therefore relatively harmless to Reins; each such misidentification merely increases the size of rewritten code by a few bytes due to alignment. A false negative (i.e., failure to identify one or more targets) is more serious and may lead to rewritten code that does not execute properly, but the verifier ensures that it cannot lead to a policy violation. Thus, both forms of error are tolerated.

Another major design issue is the need to arrange the lookup table so that IRM code that uses it remains exceptionally small and efficient. This is critical for achieving low overhead, since computed jumps are extremely common in real-world binaries. Our solution implements most lookups with just two non-branching instructions (a compare instruction and a conditional move), shown atop the first row of Table 5.1. This efficient implementation is achieved by tagging each lookup table entry with a leading byte that never appears as the first byte of valid code. We use a tag byte of `0xF4`, which encodes an x86 `hlt` instruction that is illegal in protected mode. The compare instruction uses this byte to quickly distinguish stale pointers that point into the lookup table from those that already point to code. The conditional move then corrects the stale ones. This succinct realization of semantics-preserving computed jump guards is the key to Reins' exceptionally low overhead.

Retaining the old code section as a data section has the additional advantage of retaining any static data that may be interleaved in the code. This data can therefore be read by

Table 5.1. Summary of x86 code transformations

Description	**Original code**	**Rewritten code**

Computed jumps with register operands	`call/jmp` r	`cmp byte ptr [`r`], 0xF4`
		`cmovz` r`, [`r`+1]`
		`and` r`,` $(d-c)$
		`call/jmp` r

Computed jumps with memory operands	`call/jmp [`m`]`	`mov eax, [`m`]`
		`cmp byte ptr [eax], 0xF4`
		`cmovz eax, [eax+1]`
		`and eax,` $(d-c)$
		`call/jmp eax`

Returns	`ret` (n)	`and [esp],` $(d-c)$ `ret` (n)
IAT loads	`mov` rm`, [IAT:`n`]`	`mov` rm`, offset tramp_`n

Tail-calls to high memory	`jmp [IAT:`n`]`	`tramp_`n`:`
		`and [esp],` $(d-c)$
		`jmp [IAT:`n`]`

the rewritten executable at its original addresses, avoiding many difficult data preservation problems that hamper other SFI systems. The tradeoff is an increased size of rewritten programs, which tend to be around twice the size of the original. However, this does not necessarily lead to an equivalent increase in runtime process sizes. Our experiences with real x86 executables indicates that dynamic data sizes tend to eclipse static code sizes in memory-intensive processes. Thus, in most cases rewritten process sizes incur only a fraction of the size increase experienced by the disk images whence they were loaded.

When the original computed jump employs a memory operand instead of a register, as shown in row 2 of Table 5.1, the rewritten code requires a scratch register. Table 5.1 uses

`eax`, which is caller-save by convention and is not used to pass arguments by any calling convention supported by any mainstream x86 compiler (Fog, 2009).[2]

A particularly common form of computed jump deserves special note. Return instructions (`ret`) jump to the address stored atop the stack (and optionally pop n additional bytes from the stack afterward). These are guarded by the instruction given in row 3 of Table 5.1, which masks the return address atop the stack to a low memory chunk boundary. Call instructions are moved to the ends of chunks so that the return addresses they push onto the stack are aligned to the start of the following chunk. Thus, the return guards have no effect upon return addresses pushed by properly rewritten call instructions, but they block jumps to corrupted return addresses that point to illegal destinations, such as the stack. This makes all attacker-injected code unreachable.

Preserving API Calls: To allow untrusted code to safely access trusted library functions in high memory, the rewriter permits one form of computed jump to remain unguarded: Computed jumps whose operands directly reference the *import address table* (IAT) are retained. Such jumps usually have the following form:

`call [IAT:`n`]`

where `IAT` is the section of the executable reserved for the IAT and n is an offset that identifies the IAT entry. These jumps are safe since the entrypoint to the APIs is hooked by Reins to ensure that they always target policy-compliant addresses at runtime.

Not all uses of the IAT have this simple form, however. Most x86-targeting compilers also generate optimized code that caches IAT entries to registers, and uses the registers as jump targets. To safely accommodate such calls, the rewriter identifies and modifies all

[2]To support binaries that depend on preserving `eax` across computed jumps, the table's sequence can be extended with two instructions that save and restore `eax`. We did not encounter any programs that require this, so our experiments use the table's shorter sequence.

Original:

```
.text:00499345  8B 35 FC B5 4D 00     mov esi, [4DB5FCh] ;IAT:MBTWC
...
.text:00499366  FF D6                 call esi
```

Rewritten:

```
.tnew:0059DBF0  BE 90 12 5D 00        mov esi, offset loc_5D1290
...
.tnew:0059DC15  80 3E F4              cmp byte ptr [esi], F4h
.tnew:0059DC18  0F 44 76 01           cmovz esi, [esi+1]
.tnew:0059DC1C  90 90 90 90           nop  (×4)
.tnew:0059DC20  81 E6 F0 FF FF 0F     and esi, 0FFFFFF0h
.tnew:0059DC26  90  (×8)              nop  (×8)
.tnew:0059DC2E  FF D6                 call esi
...
.tnew:005D1290  81 24 24 F0 FF FF 0F  and dword ptr [esp], 0FFFFFF0h
.tnew:005D1297  FF 25 FC B5 4D 00     jmp [4DB5FCh] ;IAT:MBTWC
```

Figure 5.2. Rewriting a register-indirect system call

instructions that use IAT entries as data. An example of such an instruction is given in row 4 of Table 5.1. For each such instruction, the rewriter replaces the IAT memory operand with the address of a callee-specific *trampoline chunk* (in row 5) introduced to the rewritten code section (if it doesn't already exist). The trampoline chunk safely jumps to the trusted callee using a direct IAT reference. Thus, any use of the replacement pointer as a jump target results in a jump to the trampoline, which invokes the desired function.

Dynamic linking and callbacks are both supported via a similar form of trampolining detailed in the technical report (Hamlen et al., 2010).

Examples

To illustrate the rewriting algorithm, Figs. 5.2 and 5.3 demonstrate the transformation process for two representative assembly codes.

Figure 5.2 implements a register-indirect call to a system API function (MBTWC). The first instruction of the original code loads an IAT entry into the `esi` register, which is later used as the target of the call. REINS replaces this address with the address of the in-

Original:

```
.text:00408495  FF 24 85 CC 8A 40 00  jmp ds:off_408ACC[eax*4]
...
.text:00408881  3D 8C 8A 4D 00 00     cmp byte_4D8A8C, 0
.text:00408888  74 13                 jz short loc_40889D
.text:0040888A  84 C9                 test cl, cl
.text:0040888C  74 0F                 jz short loc_40889D
...
.text:00408ACC  81 88 40 00           dd offset loc_408881
.text:00408AD0  ...                   (other code pointers)
```

Rewritten:

```
.text:00408881  F4 60 3A 4F 00        db F4, loc_4F3A60
                    - - - - - - - - - - - - - - - - -
.tnew:004F33B4  8B 04 85 CC 8A 40 00  mov eax, ds:dword_408ACC[eax*4]
.tnew:004F33BB  80 38 F4              cmp byte ptr [eax], F4h
.tnew:004F33BE  90 90                 nop  (×2)
.tnew:004F33C0  0F 44 40 01           cmovz eax, [eax+1]
.tnew:004F33C4  25 F0 FF FF 0F        and eax, 0FFFFFFF0h
.tnew:004F33C9  FF E0                 jmp eax
...
.tnew:004F3A60  3D 8C 8A 4D 00        cmp byte_4D8A8C, 0
.tnew:004F3A67  74 27                 jz short loc_4F3A90
.tnew:004F3A69  84 C9                 test cl, cl
.tnew:004F3A6B  74 22                 jz short loc_4F3A90
```

Figure 5.3. Rewriting code that uses a jump table

lined trampoline code at the bottom of the figure, which performs a safe jump to the same destination. The call instruction is replaced with the guarded call sequence shown in lines 2–7 of the rewritten binary. The compare (`cmp`) and conditional move (`cmovz`) implement the table-lookup, and the masking instruction (`and`) aligns the destination to a chunk boundary. This makes the ensuing call provably safe to execute.

Figure 5.3 shows a computed jump with a memory operand that indexes a jump table. The rewritten code first loads the destination address into a scratch register (`eax`) in accordance with row 2 of Table 5.1. It then implements the same lookup and masking guards as in Figure 5.2. This time the lookup has a significant effect—it discovers at runtime that the address drawn from the lookup table must be repointed to a new address. This preserves

the behavior of the binary after rewriting despite the failure of the disassembler to discover and identify the jump table at rewrite-time.

Memory Safety

To prevent untrusted binaries from dynamically modifying code sections or executing data sections as code, untrusted processes are executed with DEP enabled. DEP-supporting operating systems allow memory pages to be marked non-executable (NX). Attempts to execute code in NX pages result in runtime access violations. The binary rewriter sets the NX bit on the pages of all low memory sections other than rewritten code sections to prevent them from being executed as code. Thus, attacker-injected shell code in the stack or other data memory regions cannot be executed.

User processes on Windows systems can set or unset the NX bit on memory pages within their own address spaces, but this can only be accomplished via a small collection of system API functions—e.g., `VirtualProtect` and `VirtualAlloc`. The rewriter replaces the IAT entries of these functions with trusted wrapper functions that silently set the NX bit on all pages in low memory other than rewritten code pages. The wrappers do not require any elevated privileges; they simply access the real system API directly with modified arguments.

The real system functions are accessible to trusted libraries (but not untrusted libraries) because they have separate IATs that are not subjected to our IAT hooking. Trusted libraries can therefore use them to protect their local heap and stack pages from untrusted code that executes in the same address space. Our API hooks prevent rewritten code from directly accessing the page protection bits to reverse these effects. This prevents the rewritten code from gaining unauthorized access to trusted memory.

Our memory safety enforcement strategy conservatively rejects untrusted, self-modifying code. Such code is a mainstay of certain application domains, such as JIT-compilers. For these domains we consider alternative technologies, such as certifying compilers and certified,

bytecode-level IRMs, to be a more appropriate means of protection. Self-modifying code is increasingly rare in other domains, such as application installers, because it is incompatible with DEP, incurs a high performance penalty, and tends to trigger conservative rejection by antivirus products. No SFI system to our knowledge supports arbitrary self-modifying code.

Verification

The disassembler, rewriter, and lookup table logic all remain completely untrusted by our architecture. Instead, a small, independent verifier certifies that rewritten programs cannot circumvent the IAT and are therefore policy-adherent. The verifier does not prove that the rewriting process is behavior-preserving. This reduced obligation greatly simplifies the verifier relative to the rewriter, resulting in a small TCB.

The verification algorithm performs a simple fall-through disassembly of each executable section in the untrusted binary and checks the following purely syntactic properties:

- All executable sections reside in low memory.
- All exported symbols (including the program entrypoint) target low memory chunk boundaries.
- No disassembled instruction spans a chunk boundary.
- Static branches target low memory chunk boundaries.
- All computed jump instructions that do not reference the IAT are immediately preceded by the appropriate `and`-masking instruction from Table 5.1 in the same chunk.
- Computed jumps that read the IAT access a properly aligned IAT entry, and are preceded by an `and`-mask of the return address. (Call instructions must *end* on a chunk boundary rather than requiring a mask, since they push their own return addresses.)
- There are no trap instructions (e.g., `int` or `syscall`).

These properties ensure that any unaligned instruction sequences concealed within untrusted, executable sections are not reachable at runtime. This allows the verifier to limit its attention to a fall-through disassembly of executable sections, avoiding any reliance upon the incomplete code-discovery heuristics needed to produce full disassemblies of arbitrary (non-chunk-aligned) binaries.

5.4 Implementation

We have developed an implementation of Reins for the 32-bit version of Microsoft Windows XP/Vista/7/8. The implementation consists of four components: (1) a rewriter, (2) a verifier, (3) an API hooking utility, and (4) an intermediary library that handles dynamic linking and callbacks. Rather than using a single, static API hooking utility, we implemented an automated *monitor synthesizer* that generates API hooks and wrappers from a declarative policy specification. This is discussed in Section 5.5.3. None of the components require elevated privileges. While the implementation is Windows-specific, we believe the general approach is applicable to any modern OS that supports DEP technology.

The rewriter transforms Windows Portable Executable (PE) files in accordance with the algorithm in Section 5.3. Its implementation consists of about 1,300 lines of IDA Python scripting code that executes atop the Hex-rays IDA Pro 6.1 disassembler. One of IDA Pro's primary uses is as a malware reverse engineering and de-obfuscating tool, and it boasts many powerful code analyses that heuristically recover program structural information without assistance from a code-producer. These analyses are leveraged by our system to automatically distinguish code from data and identify function entrypoints to facilitate rewriting.

In contrast to the significant complexity of the rewriting infrastructure, the verifier's implementation consists of 1,500 lines of 80-column OCaml code that uses no external libraries or utilities (other than the built-in OCaml standard libraries). Of these 1,500 lines,

approximately 1,000 are devoted to x86 instruction decoding, 300 to PE binary parsing, and 200 to the actual verification algorithm in Section 5.3. The decoder handles the entire x86 instruction set, including floating point, MMX, and all SSE extensions documented in the Intel and AMD manuals. This is necessary for practical testing since production-level binaries frequently contain at least some exotic instructions. No code is shared between the verifier and rewriter.

The intermediary library consists of approximately 500 lines of C and hand-written, inlined assembly code that facilitates callbacks and dynamic linking. An additional 150-line configuration file itemizes all trusted callback registration functions exported by Windows libraries used by the test programs. We supported all callback registration functions exported by `comdlg32`, `gdi32`, `kernel32`, `msvcrt`, and `user32`. Information about exports from these libraries was obtained by examining the C header files for each library and identifying function pointer types in exported function prototypes.

Our API hooking utility replaces the IAT entries of all monitored system functions imported by rewritten PE files with the addresses of trusted monitor functions. It also adds the intermediary library to the PE's list of imported modules. To avoid expanding the size of the PE header (which could shift the positions of the binary sections that follow it), our utility simply changes the library name `kernel32.dll` in the import section to the name of our intermediary library. This causes the system loader to draw all IAT entries previously imported from `kernel32.dll` from the intermediary library instead. The intermediary library exports all `kernel32` symbols as forwards to the real `kernel32`, except for security-relevant functions, which it exports as local replacements. Our intermediary library thus doubles as the policy enforcement library.

Table 5.2. Experimental results: SPEC benchmarks

Binary Program	Size Increase			Rewriting Time (s)	Verification Time (ms)
	File (%)	Code (%)	Process (%)		
`gzip`	103	31	0	12.5	142
`vpr`	94	26	22	14.4	168
`mcf`	108	32	2	10.5	84
`parser`	108	34	1	17.4	94
`gap`	118	42	0	31.2	245
`bzip2`	102	29	0	10.8	91
`twolf`	99	24	27	25.3	245
`mesa`	104	20	6	42.4	554
`art`	108	33	14	12.4	145
`equake`	103	27	1	12.3	165
median	+103.5%	+30.0%	+1.5%	13.45s	155ms

5.5 Evaluation

5.5.1 Rewriting Effectiveness

We tested Reins with a set of binary programs listed in Tables 5.2 and 5.3. Table 5.2 lists results for some of the benchmarks from the SPEC 2000 benchmark suite. Table 5.3 lists results for some other applications, including GUI programs that include event- and callback-driven code, and malware samples that require enforcement of higher-level security policies to prevent malicious behavior. In both tables, columns 2–3 report the percentage increase of the file size, code segment, and process size, respectively; and columns 5–6 report the time taken for rewriting and verification, respectively. All experiments were performed on a 3.4GHz quad-processor AMD Phenom II X4 965 with 4GB of memory running Windows XP Professional and MinGW 5.1.6.

File sizes double on average after rewriting for benign applications, while malware shows a smaller increase of about 40%. Code segment sizes increase by a bit less than half for benign applications, and a bit more than half for malware. Process sizes typically increase

Table 5.3. Experimental results: Applications and malware

Binary Program	Size Increase			Rewriting Time (s)	Verification Time (ms)
	File (%)	**Code** (%)	**Process** (%)		
`notepad`	60	31	20	1.5	18
`Eureka`	32	53	15	17.9	225
`DOSBox`	112	38	0	137.1	2394
`PhotoView`	87	57	4	3.5	49
`BezRender`	128	55	3	4.1	55
`gcc`	100	37	15	3.0	36
`g++`	100	41	16	3.0	37
`jar`	101	34	12	2.4	27
`objcopy`	122	49	23	26.9	354
`size`	103	50	116	16.3	20
`strings`	122	50	42	21.5	283
`as`	99	49	2	30.4	397
`ar`	121	50	4	21.8	285
`whetstone`	88	21	54	0.6	6
`linpack`	57	19)	31	0.6	6
`pi_ccs5`	125	28	1	5.8	66
`md5`	25	48	149	0.6	5
median	100%	41%	15%	4.1s	49ms
`Virut.a`		(rejected)		–	–
`Hidrag.a`		(rejected)		–	–
`Vesic.a`	75	34	108	0.3	194
`Sinn.1396`	37	115	93	0.2	75
`Spreder.a`	14	66	17	3.0	72
median	37%	66%	93%	0.3s	75ms

by about 15% for benign applications, but almost 90% for malware. The rewriting speed is about 32s per megabyte of code, while verification is much faster—taking only about 0.4s per megabyte of code on average.

5.5.2 Performance Overhead

We also measured the performance of the non-interactive programs in Tables 5.2 and 5.3. The runtime overheads of the rewritten programs as a percentage of the runtimes of the

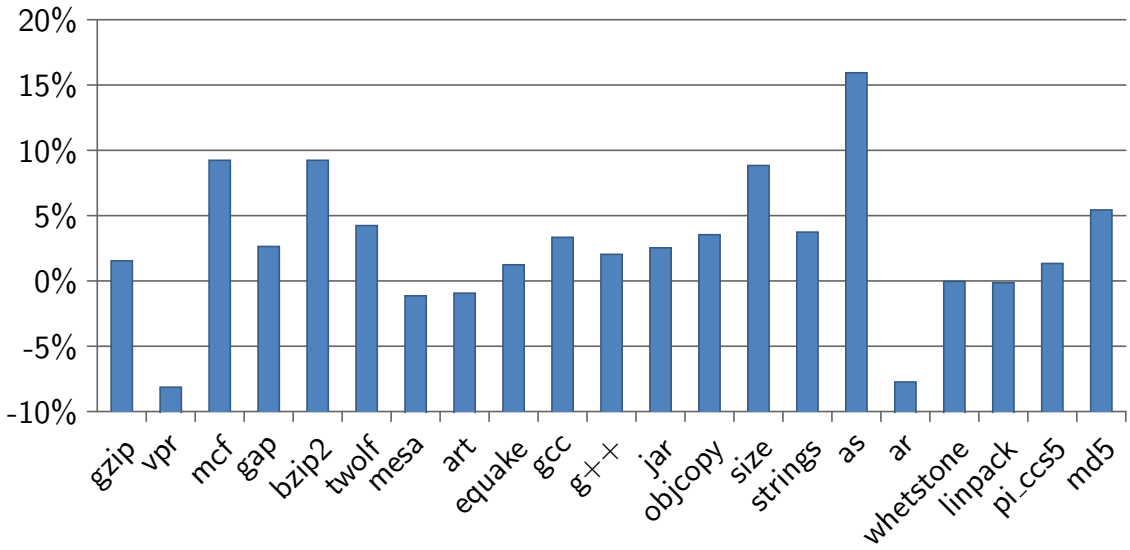

Figure 5.4. Runtime overhead due to rewriting

originals is presented in Figure 5.4. The median overhead is 2.4%, and the maximum is approximately 15%. As with other similar works (Abadi et al., 2009; Ford and Cox, 2008), the runtimes of a few programs decrease after rewriting. This effect is primarily due to improved instruction alignment introduced by the rewriting algorithm, which improves the effectiveness of instruction look-ahead and decoding pipelining optimizations implemented by modern processors. While the net effect is marginal, it is enough to offset the overhead introduced by the rest of the protection system in these cases, resulting in safe binaries whose runtimes are as fast as or faster than the originals.

The experiments reported in Tables 5.2 and 5.3 enforced only the core access control policies required to prevent control-flow and memory safety violations. Case studies that showcase the framework's capacity to enforce more useful policies are described in Section 5.5.4.

5.5.3 Policy Enforcement Library Synthesis

To quickly and easily demonstrate the framework's effectiveness for enforcing a wide class of safety policies, we developed a monitor synthesizer that automatically synthesizes the policy

```
1 function  conn = ws2_32:: connect(
2   SOCKET, struct sockaddr_in *, int) -> int;
3 function  cfile = kernel32:: CreateFileW(
4   LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
5   DWORD, DWORD, HANDLE) -> HANDLE WINAPI;

7 event  e1 = conn(_, {sin_port=25}, _) -> 0;
8 event  e2 = cfile("*.exe", _, _, _, _, _, _) -> _;

10 policy  = e1* + e2*;
```

Figure 5.5. A policy that prohibits applications from both sending emails and creating `.exe` files

enforcement portion of the intermediary library from a declarative policy specification. Policy specifications consist of: (1) the module names and signatures of all security-relevant API functions to be monitored, (2) a description of the runtime argument values that, when passed to these API functions, constitute a security-relevant *event*, and (3) a regular expression over this alphabet of events whose prefix-closure is the language of permissible *traces* (i.e., event sequences).

To illustrate, Figure 5.5 shows a sample policy. Lines 1–5 are signatures of two API functions exported by Windows system libraries: one for connecting to the network and one for creating files. Lines 7–8 identify network-connects as security-relevant when the outgoing port number is 25 (i.e., an SMTP email connection) and the return value is 0 (i.e., the operation was successful), and file-creations as security-relevant when the filename's extension is `.exe`. Underscores denote arguments whose values are not security-relevant. Finally, line 10 defines traces that include at most one kind of event (but not both) as permissible. Here, `*` denotes finite or infinite repetition and `+` denotes regular alternation.

Currently our synthesizer implementation supports dynamic value tests that include string wildcard matching, integer equality and inequality tests, and conjunctions of these tests on fields within a structure. From this specification, the monitor synthesizer generates the C source code of a policy enforcement library that uses IAT hooking to reroute calls to

```
1 function  cfile = kernel32:: CreateFileW(
2   LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
3   DWORD, DWORD, HANDLE) -> HANDLE WINAPI;
4 function  exec = kernel32:: WinExec(LPCSTR, UINT)
5   -> UINT WINAPI;

7 event  e1 = cfile("*.exe", _, _, _, _, _, _) -> _;
8 event  e2 = cfile("*.msi", _, _, _, _, _, _) -> _;
9 event  e3 = cfile("*.bat", _, _, _, _, _, _) -> _;
10 event  e4 = exec("explorer", _) -> _;

12 policy  = ;
```

Figure 5.6. Eureka email policy

`connect` and `CreateFileW` through trusted guard functions. The guard functions implement the desired policy as a determinized *security automaton* (Schneider, 2000)—a finite state automaton that accepts the prefix-closure of the policy language in line 10. If the untrusted code attempts to exhibit a prohibited trace, the monitor rejects by halting the process.

5.5.4 Case Studies

An Email Client: As a more in-depth case-study, we used the rewriting system and monitor synthesizer to enforce two policies on the *Eureka* 2.2q email client. Eureka is a fully featured, commercial POP client for 32-bit Windows that features a graphical user interface, email filtering, and support for launching executable attachments as separate processes. It is 1.61MB in size and includes all of the binary features discussed in earlier sections, including Windows event callbacks and dynamic linking. It statically links to eight trusted system libraries.

Without manual assistance, IDA automatically recovers enough structural information from the Eureka binary to facilitate the full binary rewriting algorithm presented in Section 5.3. Rewriting requires 18s and automated verification of the rewritten binary requires 0.2s.

After rewriting, we synthesized an intermediary library that enforces the access control policy given in Figure 5.6, which prohibits creation of files whose filename extensions are `.exe`, `.msi`, or `.bat`, and which prevents the application from launching Windows Explorer as an external process. (The empty policy expression in line 12 prohibits all events defined in the specification.) We also enforced the policy in Figure 5.5, but with a policy expression that limits clients to at most 100 outgoing SMTP connections per run. Such a policy might be used to protect against malware infections that hijack email applications for propagation and spamming.

After rewriting, we systematically tested all program features and could not detect any performance degradation or changes to any policy-permitted behaviors. All program features unrelated to the policy remain functional. However, saving or launching an email attachment with any of the policy-prohibited filename extensions causes immediate termination of the program by the monitor. Likewise, using any program operation that attempts to open an attachment using Windows Explorer, or sending more than 100 email messages, terminates the process. The rewritten binary therefore correctly enforces the desired policy without impairing any of the application's other features.

An Emulator: DOSBox is a large DOS emulator with over 16 million downloads on sourceforge. Though its source code is available, it was not used during the experiment. The precompiled binary is 3.6MB, and like Eureka, includes all the difficult binary features discussed earlier.

We enforced several policies that prohibit access to portions of the file system based on filename string and access mode. We then used the rewritten emulator to install and use several DOS applications, including the games Street Fighter 2 and Capture the Flag. Installation of these applications requires considerable processing time, and is the basis for the timing statistics reported in Table 5.3. As in the previous experiment, no performance

degradation or behavioral changes are observable in the rewritten application, except that policy-violating behaviors are correctly prohibited.

Malware: To analyze the framework's treatment of real-world malware, we tested REINS on five malware samples obtained from a public malware research repository: *Virut.a*, *Hidrag.a*, *Vesic.a*, *Sinn.1396*, and *Spreder.a*. While these malware variants are well-known and therefore preventable by conventional signature-matching antivirus defenses, the results indicate how our system reacts to binaries intentionally crafted to defeat disassembly tools and other static analyses. Each is statically or dynamically rejected by the protection system at various different stages, detailed below.

Virut and *Hidrag* are both rejected at rewriting time when the rewriter encounters misaligned static branches that target the interior of another instruction. While supporting instruction aliasing due to misaligned *computed* jumps is useful for tolerating disassembly errors, misaligned *static* jumps only appear in obfuscated malware to our knowledge, and are therefore conservatively rejected.

Vesic and *Sinn* are Win32 viruses that propagate by appending themselves to executable files on the C: volume. They do not use packing or obfuscation, making them good candidates for testing our framework's ability to detect malicious behavior rather than just suspicious binary syntax. With a fully permissive policy, our framework successfully rewrites and verifies both malware binaries; running the rewritten binaries preserves their original (malicious) behaviors. However, enforcing the policy in Figure 5.6 results in premature termination of infected processes when they attempt to propagate by writing to executable files. We also successfully enforced a second policy that prohibits the creation of system registry keys, which *Vesic* uses to insert itself into the boot process of the system. These effectively protect the infected system before any damage results.

Spreder has a slightly different propagation strategy that searches for executable files in the shared directory of the Kazaa file-sharing peer-to-peer client. We successfully enforced a

policy that prohibits use of the `FindFirstFileA` system API function to search for executable files in this location. This results in immediate termination of infected processes.

5.6 Discussion

In this section we first discuss the security benefits REINS provides, and then discuss the binary code conventions that are prerequisites for behavior-preservation under our binary rewriting scheme, as well as the reliability of our disassembly. The limitations of our approach are highlighted during the course of the discussion.

5.6.1 Control-flow Policies

As we have demonstrated, REINS can rewrite many complex legacy binaries, enforcing coarse-grained control-flow safety and preserving safe computed jumps without source code. However, REINS does not enforce the finer-grained control-flow integrity properties of CFI (Abadi et al., 2009). CFI uses source code or PDB files to build a control-flow graph that serves as the integrity policy to enforce. This connection to source code is foundational to CFI because any fine-grained definition of "good" control-flows invariably depends on the semantics of the source code that the untrusted binary code is intended to reflect. Without source code, there is no sensible definition of control-flow integrity for REINS to enforce.

As such, REINS and CFI have fundamentally different goals. CFI's goal is to micro-manage behavior within an untrusted binary to prevent attackers from corrupting its internal flows. In contrast, REINS' goal is to protect the environment outside the untrusted binary, not its internals. This includes external resources like the file system and network, and the trusted libraries that access them (e.g., OS/kernel libraries). The only flows that affect such resources are those that exit the untrusted code. For these, there are sensible, well-defined (but coarser-grained) control-flow policies apart from source code. For example, flows to the

stack or data are disallowed (to block code-injection attacks), and flows to trusted libraries must obey the library's interface (e.g., its export address table).

Reins prevents these forms of malicious behavior based on the security policy. It is possible for an attacker to craft ROP (Shacham, 2007) or Q (Schwartz et al., 2011) shell code to overwrite the stack pointer and break the internal control flows, but the attacker must ultimately manipulate the arguments of system calls to effect damage outside the confines of the untrusted module. These malicious system calls are detected and prevented by Reins.

5.6.2 Code Conventions

Our rewriting algorithm in Section 5.3 preserves the behavior of code that adheres to standard, compiler-agnostic x86 code generation conventions. Code that violates these conventions can yield rewritten code that fails verification or fails to execute properly, but never verified code that circumvents the monitor. Nevertheless, the practicality of the approach depends on its ability to preserve the behavior of a large class of non-malicious code. Compatibility limitations of this sort have been a major obstacle to widespread adoption of much past SFI research.

Code pointers: Reins expects each code pointer used as a jump target by untrusted code to originate from one of five sources:

- a low-memory address drawn from the program counter (e.g., a return address pushed by a `call`),
- data that points to a basic block boundary,
- a code address stored in the IAT,
- a return address pushed by a trusted caller during a callback, or
- a return value yielded by the system's dynamic linking API.

As demonstrated by our experiments, these cover a large spectrum of real-world binary code. Nevertheless, there are some unusual cases that Reins still rejects. For example, a program that computes external library entrypoints instead of requesting them from the system's linker is incompatible with Reins, and will typically crash when executed. Addressing such limitations is future work.

Reliable Disassembly: Binaries generated by most mainstream compilers mix code and static data within the `.text` section of the executable. Reins relies upon a classification algorithm that heuristically distinguishes code from data (Wartell et al., 2011). If code is misclassified as data, that code is incorrectly omitted from the rewritten binary's code section. If data is misclassified as code that looks like a possible computed jump target, the rewriter might overwrite some of the data with tagged pointers as it constructs the lookup table (see Section 5.3). This can result in corruption of the static data. However, data misclassified as code without such targets just contributes harmless, dead code to the rewritten binary's code section. Heuristics that conservatively classify most bytes as code with few computed jump targets therefore tend to work well for our system.

Function entrypoints are readily identifiable in most binaries by the characteristic function prologues and epilogues that begin and conclude most function bodies. The few remaining computed jump targets are gleaned through the disassembler's code reachability analysis and a few pattern-matching heuristics that identify instruction sequences compiled from common source language structures (e.g., switch-case statements) that often compile to computed jumps.

In practice we found that for most non-malicious programs, IDA Pro's automatic binary analysis works well, accurately identifying all code (with some data harmlessly misidentified as code) and identifying all computed jump targets (with some code harmlessly misidentified as a computed jump target). Any missed targets are easy to identify and correct

manually, since their omission causes the rewritten binary to crash at precisely the site of the misclassified address in the (now non-executable) old code segment.

Dense Computed Jump Targets: A more subtle assumption of the algorithm is that all computed jump targets in the original binary are at least $w + 1$ bytes away from the next computed jump target or following data, where w is the system word size. This is necessary to ensure sufficient space for the rewriter to write a tagged pointer at that address without overwriting any adjacent pointers or data. Entrypoints packed closer than this are rare, since most computed jump targets are 16-byte aligned for performance reasons, and since all binaries compatible with *hotpatching* have at least $w + 1$ bytes of padding between consecutive function entrypoints (Microsoft Corporation, 2005).

In the rare case that two targets are within w bytes in the original code, the rewriter strategically chooses the address of the rewritten code section so that the encodings of tagged pointers into it can occasionally overlap. For example, with tag byte $t =$ `0xF4`, the sequence `F4 00 F4 00 04 00 04` encodes two overlapping, little-endian tagged pointers to addresses `0x0400F400` and `0x04000400`. By positioning the rewritten versions of these two functions at those addresses, the rewriter can encode overlapping pointers to them in the lookup table. With chunk size $c = 16$ and memory division $d = 2^{28}$, a rewritten code base address of $2^{24}(t \,\&\, 0\text{xF}) + 2^{16}t$ supports at least 15 two-pointer collisions and 1 three-pointer collision per rewritten code page—far more than we saw in any binary we studied.

5.6.3 Other Future Work

The experiments reported in Section 5.5 focus on testing the soundness, transparency, and feasibility of our static binary rewriting algorithm on a real-world OS, and on demonstrating the enforcement of some simple but useful security policies. Past work (Hamlen et al., 2006; Ligatti et al., 2009; Rajagopalan et al., 2006) has shown that IRM systems are capable of enforcing more sophisticated temporal properties when equipped with more powerful event

languages and responses to impending policy violations that go beyond mere program termination. Developing policy-enforcement libraries that implement such policies is therefore a logical next step toward applying our framework to interesting, practical security problems for these real-world systems.

5.7 Conclusion

We have presented the design, implementation, and evaluation of a new SFI/IRM system, Reins, that monitors and restricts Windows API calls of untrusted native x86 binaries for which source code and debugging information are unavailable. The binary rewriting algorithm supports many difficult binary features, including computed jumps, dynamic linking, interleaved code and data, and OS callbacks, all without any explicit cooperation from code-producers, and it is behavior-preserving for a large class of COTS binaries. To the best of our knowledge, no past binary rewriting-based SFI work has achieved this. The enforcement mechanism requires no kernel extensions or privileges, making it applicable to shared and closed computing environments, and separate, light-weight machine-verification keeps the TCB small. Experiments on a number of COTS and malware programs show the effectiveness of Reins, and demonstrate that although rewriting doubles file sizes on average, runtimes increase by only about 2.4% and the median process size increases by only about 15%.

CHAPTER 6

SELF-TRANSFORMING INSTRUCTION RELOCATION[1]

The defensive technique exposited in this chapter leverages source-free binary rewriting to secure binaries from ROP attacks (see Section 2.2).

Recent attempts to protect against ROP have employed both static and dynamic techniques. In-place-randomization (IPR) (Pappas et al., 2012) statically smashes unwanted gadgets by changing their semantics or reordering their constituent instructions without perturbing the rest of the binary. Alternatively, ILR (Hiser et al., 2012) dynamically eliminates gadgets by randomizing all instruction addresses and using a fall-through map to dynamically guide execution through the reordered instructions. While these two approaches are valuable first steps, IPR suffers from deployment issues (since millions of separately shipped, randomized copies are required to obtain a sufficiently diverse field of application instances), and ILR suffers from high performance overhead (because of its highly dynamic, VM-based approach).

Our technique—*Self-Transforming Instruction Relocation (STIR)*—transforms legacy application binary code into *self-randomizing* code that statically re-randomizes itself each time it is loaded. The capacity to re-randomize legacy code (i.e., code without debug symbols or relocation information) at load-time greatly eases deployment, and its static code transformation approach yields significantly reduced performance overheads. Moreover, randomizing at basic block granularity achieves higher entropy than ASLR, which only

[1]This chapter contains material previously published as: Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pp. 157–168, October 2012.

randomizes section base addresses, and can therefore be susceptible to derandomization attacks (Shacham et al., 2004; Roglia et al., 2009).

STIR is a fully automatic, binary-centric solution that does not require any source code or symbolic information for the target binary program. STIR-enabled code randomly reorders the basic blocks in each binary code section each time it is launched, frustrating attempts to predict the locations of gadgets. It is therefore fully transparent, and there is no modification to the OS or compiler. This makes it easily deployable; software vendors or end users need only apply STIR to their binaries to generate one self-randomizing copy, and can thereafter distribute the binary code normally.

Randomizing legacy CISC code for real-world OS's (Microsoft Windows and Linux) without compiler support raises many challenges, including semantic preservation of dynamically computed jumps, code interleaved with data, function callbacks, and imperfect disassembly information. These challenges are detailed further in Section 6.1. In this section, we develop a suite of novel techniques, including conservative disassembly, jump table recovery, and dynamic dispatch, to address these challenges. Central to our approach is a binary transformation strategy that expects and tolerates many forms of disassembly errors by conservatively treating every byte in target code sections as both a potential instruction starting point and static data. This obviates the need for perfect disassemblies, which are seldom realizable in practice without source code.

We have implemented STIR and evaluated it on both Windows and Linux platforms with a large number of legacy binary programs. Our experimental results show that STIR can successfully transform application binaries with self-randomized instruction addresses, and that doing so introduces about 1% overhead (significantly better than ILR's 16%) on average at runtime to the applications.

In summary, this section makes the following contributions.

- We present a mostly static, instruction address randomization technique to counter ROP shell-code that reuses gadgets in legacy binaries. Randomization is achieved via a novel static binary rewriting that first transforms the binary into a self-randomizable representation, followed by a load-time phase that *stirs* the binary at program start, yielding a different ordering of the instruction addresses each time the program is launched.

- To enable static rewriting of legacy code, we have developed a number of new techniques in support of native x86 code for computed jumps, code-data interleaving, OS call-backs, dynamic linking, and disassembly error tolerance. We believe these techniques constitute a significant advancement in static binary writing.

- We have implemented our entire system on two mainstream OS architectures (Windows and Linux) and tested it with a large number of application binaries. The implementation consists of a conservative, static *disassembler* and a load-time *reassembler*. Our empirical evaluation shows that STIR is a promising approach for defending real-world legacy binaries against ROP shell code.

Section 6.1 begins with an overview of our approach. Section 6.2 elaborates by presenting a detailed design of STIR, including its static disassembler and load-time reassembler. Section 6.3 reports the results of our evaluation of STIR on over 100 Windows and Linux binaries. Section 6.4 discusses known limitations of the system. Finally, Section 6.5 concludes.

6.1 System Overview

Users of our system submit legacy x86 COTS binaries (PE files for Windows, or ELF files for Linux) to a rewriter. The rewriter disassembles, transforms, and reassembles the target code into a new, self-randomizing binary that reorders its own basic blocks each time it is executed. Instruction address randomization is achieved by randomizing the basic blocks.

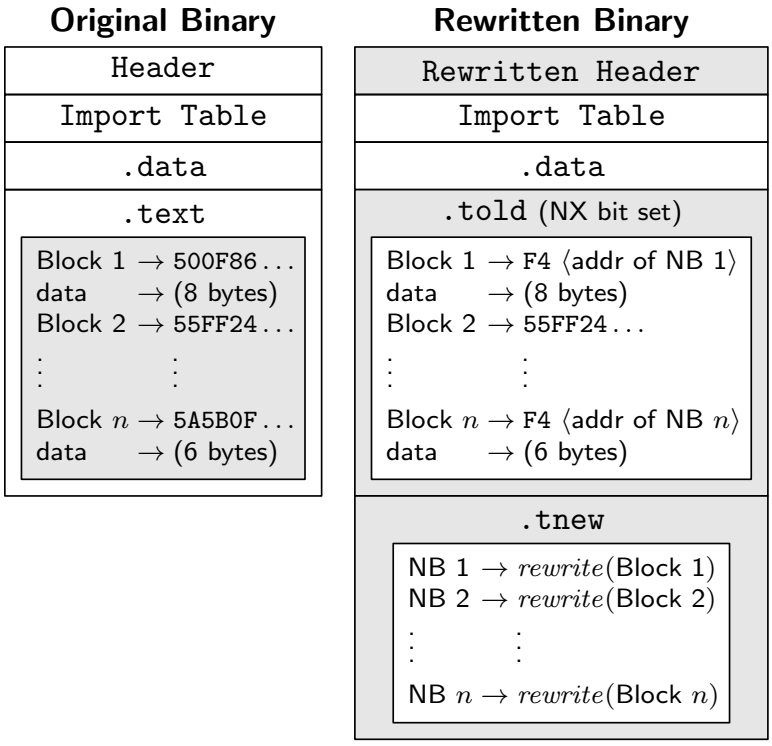

Figure 6.1. Static binary rewriting phase

No source code, relocation information, debug sections, symbol stores (e.g., PDB files), or hints are required. Expert users may guide the disassembly process through the use of an interactive disassembler like IDA Pro (Hex-Rays, 2012), but our system is designed to make such guidance unnecessary in the vast majority of cases.

6.1.1 Approach Overview

Our system architecture addresses these challenges in two phases: (1) a static phase, depicted by Figure 6.1, that transforms the binary into a more easily randomizable form, and (2) a load-time phase that stirs the binary by randomly reordering its instructions each time it starts. Code/data interleaving, imperfect disassembly, and position independent instructions are addressed by the static phase. Computed jumps and callbacks are further assisted by the load-time phase.

To solve the code/data interleaving and uncomputable disassembly problems, our static phase adopts a novel approach that treats all bytes as both data *and* code. To do so, it doubles each code segment into two separate segments—one in which all bytes are treated as data, and another in which all bytes are treated as code. This approach of keeping a copy of original binary code is partially inspired by DynInst (Hollingsworth et al., 1994) (although DynInst keeps only one copy, whereas STIR keeps both).

In the data-only copy (`.told`), all bytes are preserved at their original addresses, but the section is set non-executable (NX), safely deactivating any original, unrandomized code (including any gadgets) that it may contain. This safely preserves all the static data at its original addresses without the need to statically identify which bytes are data and which are code.

In the code-only copy (`.tnew`), all bytes are disassembled into code blocks that can be randomly stirred into a new layout each time the program starts. Any data bytes simply become harmless, unreachable code in the new binary's code section. Thus, there is no need to statically predict which bytes are part of reachable control-flows and which are not.

Random stirring of the code-only section is performed during the second stage by a trusted library statically linked into the new binary. The library randomly reorders all basic blocks in the new code section each time the program starts. The load order of the system guarantees that this library initializer code always runs before the target code it stirs.

Afterwards, some code pointers (e.g., immediate operands and those pushed onto the stack by `call` instructions) have been re-pointed to correct addresses, but others (e.g., those in method dispatch tables) continue to point into the data-only segment. Recall that the data-only segment is non-executable, so attempting to use one of these *stale pointers* as a computed jump target results in an exception (usually a crash). To solve this computed jump problem, our static phase additionally translates all computed jump instructions from the original code into a short alternative sequence in the new code that dynamically detects

Original:

```
.text:0040CC9B  FF D0        call eax
```

Rewritten:

```
.tnew:00436EDA  80 38 F4     cmp byte ptr [eax], F4h
.tnew:00436EDD  0F 44 40 01  cmovz eax, [eax+1]
.tnew:00436EE1  FF D0        call eax
```

Figure 6.2. Semantic preservation of computed jumps

and re-points old pointers to new addresses at runtime. Computed jumps are extremely common, so keeping the replacement sequence small and efficient is crucial for maintaining good performance.

We discovered a 2-instruction sequence, shown in Figure 6.2, that reliably re-points such code pointers with low overhead. Conceptually, our solution repurposes part of the data-only segment as a lookup table that maps old code addresses to the new one. Instructions at likely computed jump targets in the old code are overwritten during the static and load-time phases with pointers to their corresponding locations in the new code segment. Even though static disassemblers cannot reliably distinguish code from data or anticipate the destinations of specific computed jump instructions, they can identify likely computed jump targets, such as function prologues, with high accuracy. Such heuristics suffice to identify a superset of all computed jump targets, although which computed jumps go where remains unknown.

The instruction sequence in Figure 6.2 leverages this information by conditionally replacing each code pointer with the correct pointer stored at the location to which it points, efficiently patching the computed control-flows at runtime. To easily distinguish old code pointers from new, our lookup table prefixes each pointer entry with tag byte `0xF4`, which encodes an illegal x86 instruction. Thus, pointers whose destinations start with the tag are stale and need to be updated, while those that do not are already correct.

Callbacks are facilitated by a similar process that involves rerouting control-flows that cross the user-code/OS boundary through the trusted library, which re-points callback

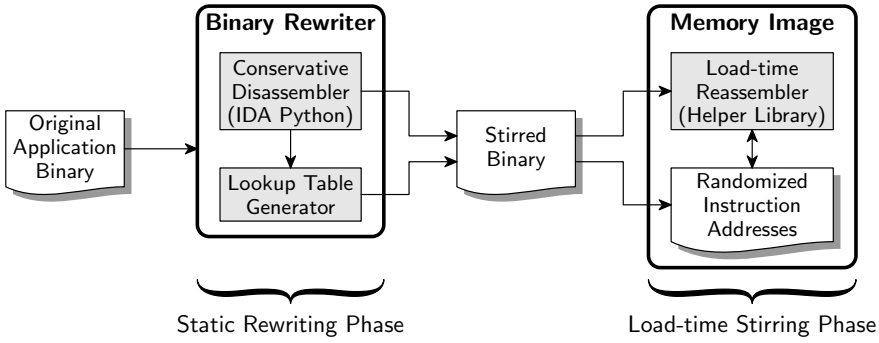

Figure 6.3. System architecture

pointers. The details of this process and the preservation of position-dependent instructions are discussed in greater detail the in the next section.

6.2 Detailed Design

The architecture of STIR is shown in Figure 6.3. It includes three main components: (1) a conservative disassembler, (2) a *lookup table generator*, and (3) a load-time reassembler. At a high level, our disassembler takes a target binary and transforms it to a randomizable representation. An address map of the randomizable representation is encoded into the new binary by the lookup table generator. This is used by the load-time reassembler to efficiently randomize the new binary's code section each time it is launched.

This section presents a detailed design of each component. We first outline the static phase of our algorithm (our conservative disassembler and lookup table generator) in Section 6.2.1, followed by the load-time phase (our reassembler) in Section 6.2.2. Section 6.2.3 walks through an example. Finally, Section 6.2.4 addresses practical compatibility issues.

6.2.1 Static Rewriting Phase

Target binaries are first disassembled to assembly code. We use the IDA Pro disassembler from Hex-rays for this purpose (Hex-Rays, 2012), though any disassembler capable of accurately identifying likely computed jump targets could be substituted.

Algorithm 2 *Trans*(α, c): Translate one instruction

Input: address mapping $\alpha : \mathbb{Z} \rightharpoonup \mathbb{Z}$ and instruction c
Output: translated instruction(s)

```
if IsComputedJump(c) then
  op ← Operand(c)
  if IsRegister(op) then
    return [ cmp op, F4h;
             cmovz op, [op+1]; c ]
  else if IsMemory(op) then
    Operand(c) ← eax
    return [ mov eax, op;
             cmp [eax], F4h;
             cmovz eax, [eax+1]; c ]
  end if
else if IsDirectJump(c) then
  t ← OffsetOperand(c)
  return c with operand changed to α(t)
else
  return c
end if
```

The resulting disassembly may contain harmless errors that misidentify data bytes as code, or that misidentify some code addresses as possible computed jump targets; but errors that omit code or misidentify data as computed jump targets can lead to non-functional rewritten code. We therefore use settings that encourage the disassembler to interpret all bytes that constitute valid instruction encodings as code, and that identify all instructions that implement prologues for known calling conventions as possible computed jump targets. These settings suffice to avoid all harmful disassembly errors in our experiments (see Section 6.3).

The assembly code is next partitioned into basic blocks, where a basic block can be any contiguous sequence of instructions with a single entry point. Each block must also end with an unconditional jump, but STIR can meet this requirement by inserting `jmp 0` instructions (a semantic no-op) to partition the code into arbitrarily small blocks during

Algorithm 3 Translate all instructions

Input: instruction list C
Output: rewritten block list B

```
B ← [ ]
α ← ∅
t ← base address of .told section
t′ ← base address of .tnew section
for all c ∈ C do
  if IsCode(c) then
    α ← α ∪ {(t, t′)}
    t′ ← t′ + |Trans(ι, c)|
  end if
  t ← t + |c|
end for
for all c ∈ C do
  if IsCode(c) then
    append Trans(α, c) to B
  end if
end for
return B
```

rewriting. The resulting blocks are copied and translated into a new binary section according to Algorithms 2–3, which we implemented as an IDAPython script.

Algorithm 2 translates a single instruction into its replacement in the new code section. Most instructions are left unchanged, but computed jumps are replaced with the lookup table code described in Section 6.1.1, and direct branches are re-pointed according to address mapping α.

Algorithm 3 calls Algorithm 2 as a subroutine to translate all the instructions. Its initial pass first computes mapping α by using identity function ι as the address mapping.[2] The second pass uses the resulting α to generate the final new code section with direct branches re-targeted.

[2]Some x86 instructions' lengths can change when ι is replaced by α. Our rewriter conservatively translates these to their longest encodings during the first pass to avoid such changes, but a more optimal rewriter could use multiple passes to generate smaller code.

Once the new code section has been generated, the lookup table generator overwrites all potential computed jump targets t in the original code section with a tag byte `0xF4` followed by 4-byte pointer $\alpha(t)$. This implements the lookup table described in Section 6.1.1.

It may seem more natural to implement range checks to identify stale pointers rather than using tag bytes. However, in general a stirred binary may consist of many separate modules, each of which has undergone separate stirring, and which freely exchange stale code pointers at runtime. Since each module loads into a contiguous virtual address space, it is not possible to place all the old code sections within a single virtual address range. Thus, implementing pointer range checks properly would require many nested conditionals, impairing performance. Our use of tag bytes reduces this to a single conditional move instruction and no conditional branches.

The resulting binary is finalized by editing its binary header to import the library that performs binary stirring at program start. PE and ELF headers cannot be safely lengthened without potentially moving the sections that follow, introducing a host of data relocation problems. To avoid this, we simply substitute the import table entry for a standard system library (`kernel32.dll` on Windows) with an equal-length entry for our library. Our library exports all symbols of the system library as forwards to the real system library, allowing it to be transparently used as its replacement. This keeps the header length invariant while importing all the new code necessary for stirring.

6.2.2 Load-time Stirring Phase

When the rewritten program is launched, the STIR library's initializer code runs to completion before any code in STIR-enabled modules that link to it. On Windows this is achieved by the system load order, which guarantees that statically linked libraries initialize before modules that link to them. On Linux, the library is implemented as a shared object (SO) that is injected into the address space of STIR-enabled processes using the LD_PRELOAD

environment variable. When this variable is set to the path of a shared object, the system loader ensures that the shared object is loaded first, before any of the other libraries that a binary may need.

The library initializer performs two main tasks at program start:

1. All basic blocks in the linking module's `.tnew` section are randomly reordered. During this stirring, direct branch operands are repointed according to address mapping α, computed during the static phase.

2. The lookup table in the linking module's `.told` section is updated according to α to point to the new basic block locations.

Once the initialization is complete, the `.tnew` section is assigned the same access permissions as the original program's `.text` section. This preserves non-writability of code employing W$\oplus$X protections.

To further minimize the attack surface, the library is designed to have as few return instructions as possible. The majority of the library that implements stirring is loaded dynamically into the address space at library initialization and then unloaded before the stirred binary runs. Thus, it is completely unavailable to attackers. The remainder of the library that stays resident performs small bookkeeping operations, such as callback support (see Section 6.2.4). It contains less than 5 return instructions total.

6.2.3 An Example

To illustrate our technique, Figure 6.4 shows the disassembly of a part of the original binary's `.text` section and its counterparts in the rewritten binary's `.told` and `.tnew` sections after passing through the static and load-time phases described above.

The disassembly of the `.text` section shows two potential computed jump targets, at addresses `0x404B00` and `0x404B18`, that each correspond to a basic block entry point. In the

rewritten `.told` section, the underlined values show how each is overwritten with the tag byte `0xF4` followed by the 4-byte pointer $\alpha(t)$ that represents its new location in the `.tnew` section.[3] All remaining bytes from the original code section are left unchanged (though the section is set non-executable) to ensure that any data misclassified as code is still accessible to instructions that may refer to it.

The `.tnew` section contains the duplicated code after stirring. Basic blocks `0x404B00`, `0x404B10` and `0x404B18`, which were previously adjacent, are relocated to randomly chosen positions `0x51234C`, `0x53AF21` and `0x525B12` (respectively) within the new code section. Non-branch instructions are duplicated as is, but static branches are re-targeted to the new locations of their destinations. Additionally, as address `0x525B16` shows, branch instructions are conservatively translated to their longest encodings to accommodate their more distant new targets.

6.2.4 Special Cases

Real-world x86 COTS binaries generated by arbitrary compilers have some obscure features, some of which required us to implement special extensions to our framework to support them. In this section we describe the major ones and our solutions.

Callbacks:

Real-world OS's—especially Windows—make copious use of callbacks for event-driven programming. User code solicits callbacks by passing code pointers to a *callback registration function* exported by the system. The supplied pointers are later invoked by the OS in response to events of interest, such as mouse clicks or timer interrupts. Our approach of dynamically re-pointing stale pointers at the sites of dynamic calls does not work when the call site is located within an unstirred binary, such as an OS kernel module.

[3] This value changes during each load-time stirring.

Original .text:

```
.text:00404AF0  00 4B 40 00           .dword 00404B00h
.text:00404AF4  18 4B 40 00           .dword 00404B18h
.text:00404AF8  CC (×8)               .align 16
.text:00404B00  8B 04 85 F0 4A 40 00  mov eax,[eax*4+404AF0h]
.text:00404B07  FF E1                 jmp eax
.text:00404B09  CC CC CC CC CC CC CC  .align 16
.text:00404B10  55                    push ebp
.text:00404B11  8B E5                 mov esp, ebp
.text:00404B13  C3                    retn
.text:00404B14  CC CC CC CC           .align 8
.text:00404B18  55                    push ebp
.text:00404B19  83 F8 01              cmp eax, 1
.text:00404B1C  7D 02                 jge 404B20h
.text:00404B1E  33 C0                 xor eax, eax
.text:00404B20  8B C1                 mov eax, ecx
.text:00404B22  E8 D9 FF FF FF        call 404B00h
```

STIRred .told (Jump Table):

```
.told:00404AF0  00 4B 40 00 18 4B 40 00
.told:00404AF8  CC CC CC CC CC CC CC CC
.told:00404B00  F4 4C 23 51 00 40 00 FF
.told:00404B08  E1 CC CC CC CC CC CC CC
.told:00404B10  55 8B E5 C3 CC CC CC CC
.told:00404B18  F4 12 5B 52 00 02 33 C0
.told:00404B20  8B C1 E8 D9 FF FF FF
```

STIRred .tnew:

```
.tnew:0051234C  8B 04 85 F0 4A 40 00  mov eax,[eax*4+404AF0h]
.tnew:00512353  80 38 F4              cmp F4h, [eax]
.tnew:00512356  0F 44 40 01           cmov eax, [eax+1]
.tnew:0051235A  FF E1                 jmp eax
                  ... (other basic blocks) ...
.tnew:00525B12  55                    push ebp
.tnew:00525B13  83 F8 01              cmp eax, 1
.tnew:00525B16  0F 8D 00 00 00 02     jge 525B1Eh
.tnew:00525B1C  33 C0                 xor eax, eax
.tnew:00525B1E  8B C1                 mov eax, ecx
.tnew:00525B20  E8 27 C8 FE FF        call 51234C
                  ... (other basic blocks) ...
.tnew:0053AF21  55                    push ebp
.tnew:0053AF22  8B E5                 mov esp, ebp
.tnew:0053AF24  C3                    retn
```

Figure 6.4. A stirring example

To compensate, our helper library hooks (Hoglund and Butler, 2006) all import address table entries of known callback registration functions exported by unstirred modules. The hooks re-point all calls to these functions to a helper library that first identifies and corrects any stale pointer arguments before passing control on to the system function. This interposition ensures that the OS receives correct pointer arguments that do not point into the old code section.

Position Independent Code:

PIC instructions compute their own address at runtime and perform pointer arithmetic to locate other instructions and data tables within the section. An underlying assumption behind this implementation is that even though the absolute positions of these instructions in the virtual address space may change, their position relative to one another does not. This assumption is violated by stirring, necessitating a specialized solution.

All PIC that we encountered in our experiments had the form shown in the first half of Figure 6.5. The call instruction has the effect of pushing the address of the following instruction onto the stack and falling through to it. The following instruction pops this address into a register, thereby computing its own address. Later this address flows into a computation that uses it to find the base address of a global offset table at the end of the section. In the example, constant `56A4h` is the compile-time distance from the beginning of the pop instruction to the start of the global offset table.

To support PIC, our rewriter identifies call instructions with operands of 0 and performs a simple data-flow analysis to identify instructions that use the pushed address in an arithmetic computation. It then replaces the computation with an instruction sequence of the same length that loads the desired address from the STIR system tables. This allows the STIR system to maintain position independence of the code across stirring. In Figure 6.5, the `nop` instruction is added to ensure that the length matches that of the replaced computation.

Original:

```
.text:0804894B  E8 00 00 00 00    call 08048950h
.text:08048950  5B                pop ebx
.text:08048951  81 C3 A4 56 00 00 add ebx, 56A4h
.text:08048957  8B 93 F8 FF FF FF mov edx, [ebx-8]
```

Rewritten:

```
.tnew:0804F007  E8 00 00 00 00    call 0804F00Ch
.tnew:0804F00C  5B                pop ebx
.tnew:0804F00D  BB F4 DF 04 08    mov ebx, 0804DFF4h
.tnew:0804F012  90                nop
.tnew:0804F013  8B 93 F8 FF FF FF mov edx, [ebx-8]
```

Figure 6.5. Position-independent code

Our analysis is not guaranteed to find all possible forms of PIC. For example, PIC that uses some other instruction to compute its address, or that allows the resulting address to flow through the heap before use, would defeat our analysis, causing the rewritten binary to crash at runtime. However, our analysis sufficed to support all PIC instances that we encountered, and compiler documentation of PIC standards indicates that there is only a very limited range of PIC implementations that needs to be supported (Oracle Corporation, 2010).

Statically Computed Returns:

Although returns are technically computed jumps (because they draw their destinations from the stack), our rewriting algorithm does not guard them with checks for stale pointers. This is a performance optimization that assumes that all return addresses are pushed onto the stack by calls; thus, no return addresses are stale.

This assumption was met by all binaries we studied except for a certain pattern of initializer code generated by GNU Compilers. The code sequence in question pushes three immediate operands onto the stack, which later flow to returns. We supported this by treating those three instructions as a special case, augmenting them with stale pointer checks that correct them at the time they are pushed instead of at the time they are used. A more

	Original .text:	Jump table .told:
2 bytes lost	func_1: .text:40EAA9 33 C0 xor eax, eax .text:40EAAB C3 retn func_2: .text:40EAAC 50 push eax	func_1: .text:40EAA9 F4 2E .text:40EAAB 04 func_2: .text:40EAAC F4
0 bytes lost	func_1: .text:40EAA9 33 C0 xor eax, eax .text:40EAAB 5B pop ebx .text:40EAAC 5E pop esi .text:40EAAD C3 retn func_2: .text:40EAAE 50 push eax	func_1: .text:40EAA9 F4 2E .text:40EAAB 25 .text:40EAAC 42 .text:40EAAD 00 func_2: .text:40EAAE F4

Figure 6.6. Overlapping function pointers

general solution could rewrite all return instructions with stale pointer guards, probably at the cost of performance.

Short Functions:

Our jump table implementation overwrites each computed jump target with a 5-byte tagged pointer. This design assumes that nearby computed jump targets are at least 5 bytes apart; otherwise the two pointers must overlap. An example of this type of jump table collision is shown in Figure 6.6, where the first row has two jump table destinations overlapping two bytes of each other, and the second row does not overlap at all. Such closely packed destinations are rare, since most computed jump destinations are already 16-byte aligned for performance reasons, and since all binaries compatible with *hot-patching* technology have at least 5 bytes of padding between consecutive function entry points (enough to encode a long jump instruction) (Microsoft Corporation, 2005).

In the rare case that two computed jump targets are closer, the rewriter strategically chooses stirred block addresses within the new code section whose pointer representations can safely overlap. For example, if the `.tnew` section is based at address `0x04000000`, the byte sequence `F4 00 F4 00 04 00 04` encodes two overlapping, little-endian, tagged pointers

to basic block addresses `0x0400F400` and `0x04000400`, respectively. This strategy suffices to support at least 135 two-pointer collisions and 9 three-pointer collisions per rewritten code page—far more than we saw in any binary we studied.

6.3 Empirical Evaluation

6.3.1 Effectiveness

Rewriting Time and Space Overheads: To evaluate the effectiveness of our system, we tested both the Windows and Linux versions of STIR with a variety of COTS and benchmark binaries. Both Windows and Linux tests were carried out on Windows 7 and Ubuntu 12 running on an Intel Core i5 dual core, 2.67GHz laptop with 4GB of physical RAM.

On Windows, we tested STIR against the SPEC CPU 2000 benchmark suite as well as popular applications like Notepad++ and DosBox. For the Linux version, we evaluated our system against the 99 binaries in the coreutils toolchain (v7.0) for the Linux version. Due to space limitations, figures only present Windows binaries and a selection of 10 Linux binaries. In all of our tests, stirred binaries exhibited the same behavior and output as their original counterparts. Average overheads only cover binaries that run for more than 500ms.

Figure 6.7 shows how the rewriting phase affects the file size and code section sizes of each binary, which increase on average by 73% and 3% respectively. However, runtime process sizes increase by only 37% on average, with the majority of the increase due to the additional library that is loaded into memory. Our current helper library implementation makes no attempt to conserve its virtual memory allocations, so we believe that process sizes can be further reduced in future development. Occasionally our disassembler is able to safely exclude large sections of static data from rewritten code sections, leading to decreased code sizes. For example, `mesa`'s code section decreases by more than 15%. On average, static rewriting of Windows binaries requires 45 seconds per megabyte of code sections, whereas

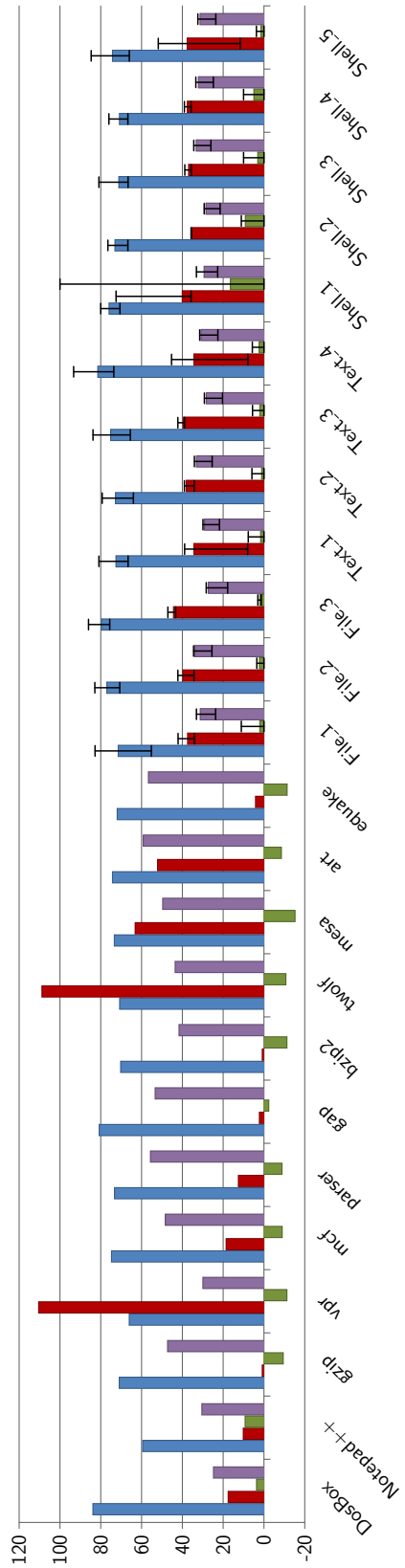

Figure 6.7. Static rewriting times and size increases

Table 6.1. Linux test programs grouped by type and size

Group	**Sizes** (KB)	**Programs**
File_1	17–37	dircolors, ln, mkdir, mkfifo, mknod, mktemp, rmdir, sync
File_2	41–45	chgrp, chmod, chown, dd, rm, shred, touch, truncate
File_3	49–97	chcon, cp, df, dir, install, ls, mv, vdir
Text_1	21–25	base64, cksum, comm, expand, fmt, fold, paste, unexpand
Text_2	25–29	cut, join, md5sum, nl, sha1sum, shuf, tac, tsort
Text_3	29–37	cat, csplit, head, sha224sum, sum, tr, uniq, wc
Text_4	37–89	od, pr, ptx, sha256sum, sha384sum, sha512sum, sort, split, tail
Shell_1	5–17	basename, dirname, env, false, hostid, link, logname, uptime
Shell_2	17–21	arch, echo, printenv, true, tty, unlink, whoami, yes
Shell_3	21	group, id, nice, noshup, pathchk, pwd, runcon, sleep
Shell_4	21–29	chroot, expr, factor, pinky, readlink, tee, test, uname, users
Shell_5	30–85	date, du, printf, seq, stat, stty, su, timeout, who

Linux binaries require 31 seconds per megabyte. Linux filenames in Figure 6.7 are grouped by type (File, Text, and Shell) and by program size due to the large number of programs. Table 6.1 lists the programs in each group.

Figure 6.8 displays all file info for every program STIR was tested on; code size increase, file size increase, runtime process size increase, and static rewriting time. No file names are displayed due to the number of programs, but they are ordered from left to right as the following: 2 GUI Windows Programs, 10 SPEC2000 benchmark programs compiled with Visual Studio 2010, 99 alphabetical coreutils Linux Programs. Figure 6.9 displays the execution time increase and execution overhead seen in all Linux binaries whose execution time was less than 5ms.

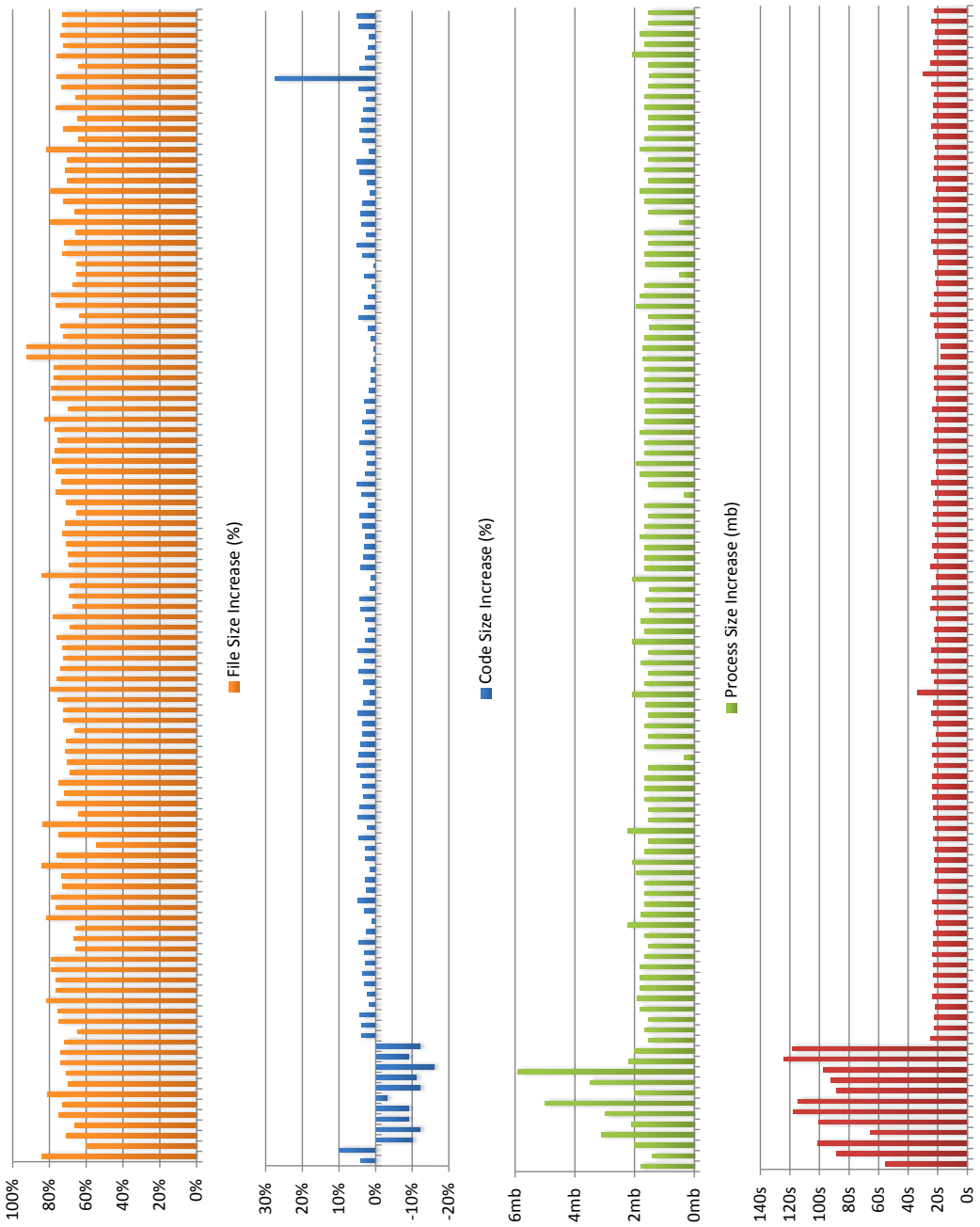

Figure 6.8. Tested File Information

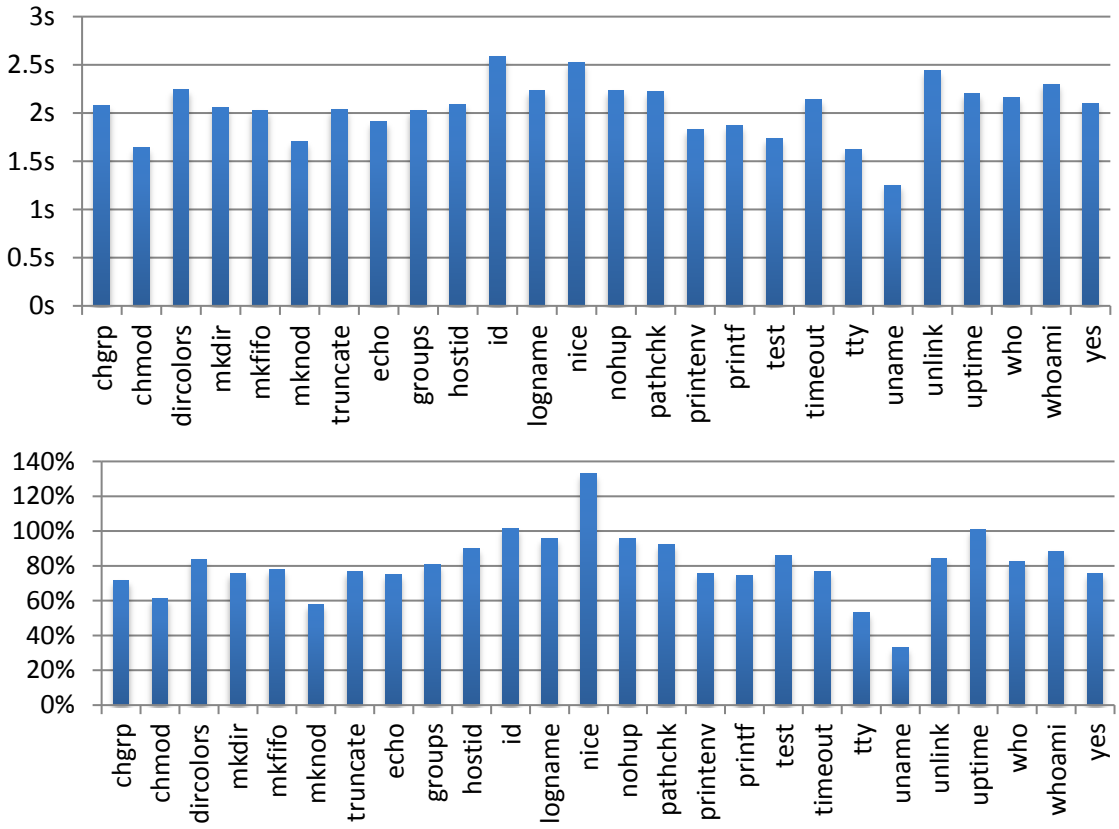

Figure 6.9. Execution Time Increase and Overhead for Fast Running Programs (<5ms)

Gadget Elimination: One means of evaluating ROP attack protection is to count the number of gadgets that remain after securing each binary. There are several tools available for such evaluation, including Mona (Corelan Team, 2012) on Windows and RoPGadget (Salwan, 2012) on Linux. We used Mona to evaluate the stirred Windows SPEC2000 benchmark programs. Mona reports the number of gadgets the binary contains after the load-time phase is complete. We define a gadget as *unusable* if it is no longer at the same virtual address after basic block randomization. Figure 6.10 shows that on average STIR causes 99.99% of gadgets to become unusable. The only gadgets that remain after randomization of the test programs consist of a `pop` and a `retn` instruction that happened to fall onto the same address. Most malware payloads are not expressible with such primitive gadgets to our knowledge.

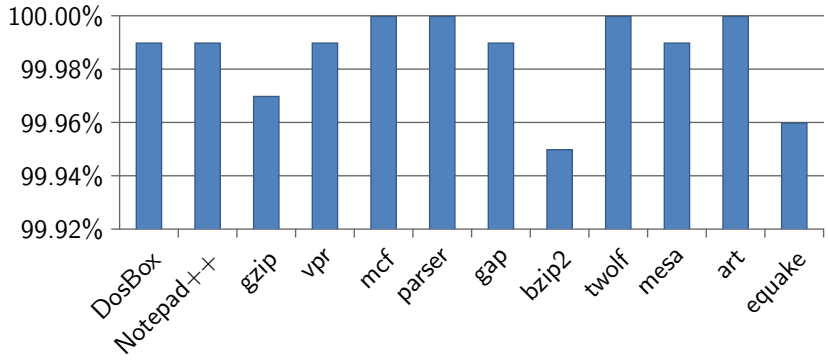

Figure 6.10. Gadget reduction for Windows binaries

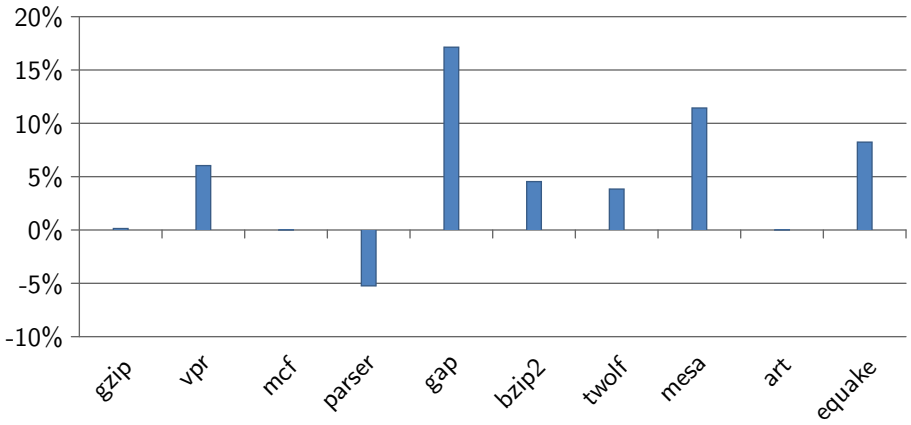

Figure 6.11. Runtime overheads for Windows binaries

We also applied the Q exploit hardening system (Schwartz et al., 2011) to evaluate the effectiveness of our system. Since Q is a purely static gadget detection and attack payload generation tool, running Q dynamically after a binary has been stirred is not possible. Instead, we ran Q on a number of Linux binaries (*viz.*, rsync, opendchub, gv, and proftpd) to generate a payload, and then ran a script that began execution of the stirred binary, testing each of the gadgets Q selected for its payload after randomization. Attacks whose gadgets all remained usable after stirring were deemed successful; otherwise, Q's payload fails. In our experiments, no payload generated by Q was able to succeed against STIR.

6.3.2 Performance Overhead

Runtime performance statistics for Windows and Linux binaries are shown in Figs. 6.11 and 6.12, respectively, with each bar reflecting the application's median overhead over 20 trials. The median overhead is 4.6% for Windows applications, 0.3% for Linux applications, and 1% overall.

To isolate the effects of caching, Figure 6.12 additionally reports the runtime overhead (discounting startup and initialization time) of *unstirred* Linux binaries, in which the load-time stirring phase was replaced by a loop that touches each original code byte without rewriting it, and then runs this unmodified, original code. This potentially has the effect of pre-fetching some or all of the code into the cache, decreasing some runtimes (although, as the figure shows, in practice the results are not consistent). Stirred binaries exhibit a median overhead of 1.2% over unstirred ones.

Amongst the Windows binaries, the `gap` SPEC2000 benchmark program consistently returns the worst overhead of 35%. This may be due to excessive numbers of callback functions or computed jumps. In contrast, the `parser` benchmark actually increases in speed by 5%. We speculate that this is due to improved locality resulting from separation of static data from the code (at the expense of increased process size). On average, the SPEC2000 benchmarks exhibit an overhead increase of 6.6%. We do not present any runtime information for DosBox and Notepad++, since both are user-interactive. We did, however, manually confirm that all program features remain functional after transformation, and no performance degradation is observable.

To separate the load-time overhead of the stirring phase from the rest of the runtime overhead, Figure 6.13 plots the stirring time against the code size. As expected, the graph shows that the increase in load-times is roughly linear with respect to code sizes, requiring 1.37ms of load-time stirring per KB of code on average.

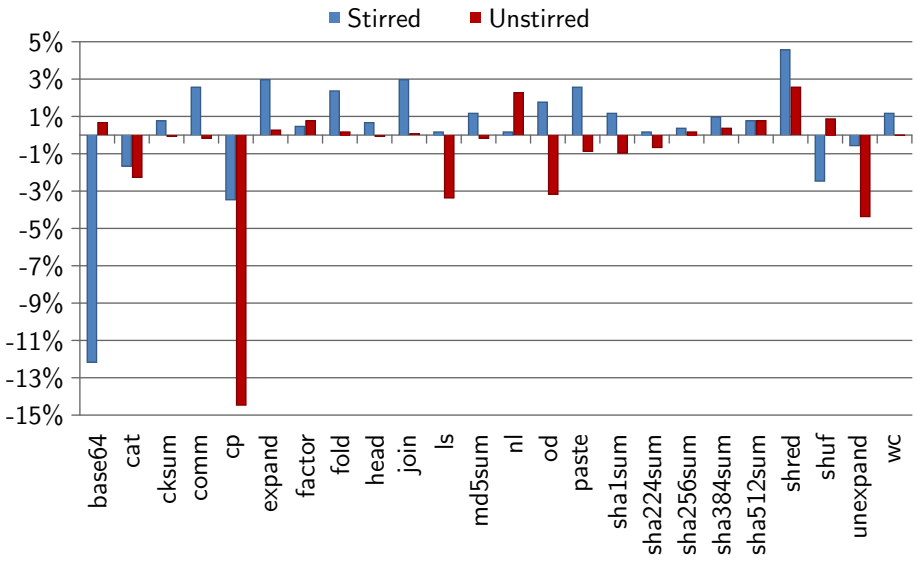

Figure 6.12. Runtime overheads for Linux binaries

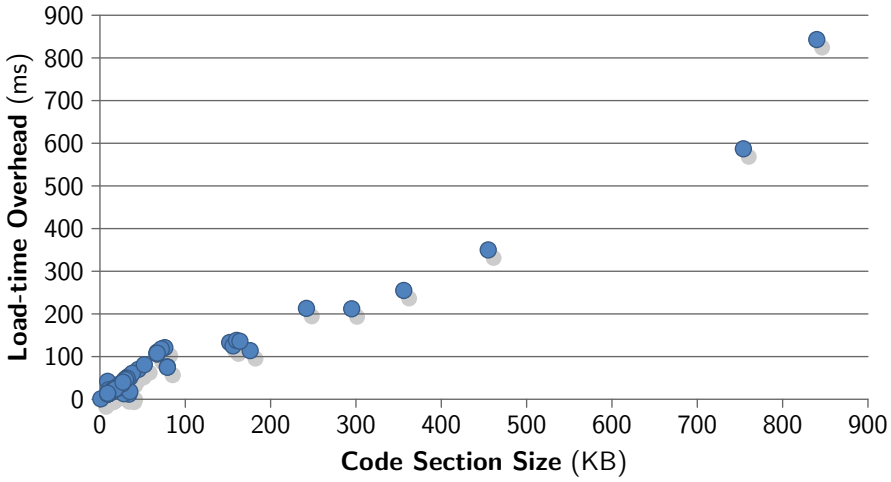

Figure 6.13. Load-time overhead vs. code size

For the most part, none of our tests require manual intervention by the user; all are fully automatic. The only exception to this is that IDA Pro's disassembly of each SPEC2000 benchmark program contained exactly two identical errors due to a known bug in its control-flow analysis heuristic. We manually corrected these two errors in each case before proceeding with static rewriting.

6.4 Limitation and Future Work

In this section we first examine the entropy of our randomization, and then discuss limitations of STIR and propose future work.

6.4.1 Randomization Entropy

Our threat model assumes that attackers do not have unrestricted read access to process address spaces. (Such powerful access renders address randomization defenses ineffective.) Under this assumption, the search space that an attacker must explore to defeat STIR is much larger than for ASLR—including 64-bit ASLR and PIE (Drepper, 2005)—because even the relative locations of gadgets within sections are randomized by STIR.

For example, past work concludes that a brute force attack against an exploitable Apache web server equipped with standard (fixed) PaX ASLR requires an expected $2^n/2 = 2^{n-1}$ probes for success, where n is the number of bits of randomness in the address space (Shacham et al., 2004). Re-randomizing ASLR doubles this to 2^n expected probes. However, STIR increases this to an expected $(2^n)!/(2(2^n - g)!)$ probes even without re-randomization, where g is the number of gadgets in the payload, since each probe must independently guess the positions of all g gadgets. On a 64-bit architecture with 14-bit aligned pages and 1 bit reserved for the kernel (i.e., $n = 50$), the expected number of probes for a g=3-gadget attack is therefore over 7.92×10^{28} ($\approx 2^{49}2^{48}/2$) times greater with STIR than with re-randomizing ASLR.

Brute force attacks that defeat ASLR by randomly guessing gadget locations over many trials (Shacham et al., 2004) or using the location of one binary feature to infer the locations of others (Roglia et al., 2009) therefore fail when applied to STIR. Runtime re-stirring could enlarge the search space even further; this is an avenue of future work (see Section 6.4.2).

The entropy of our randomization procedure is a function of the average basic block size of each binary relative to the size of the code section in which it resides. STIR also chooses

random, independent base addresses for the new code section(s) and lookup table(s). This level of entropy is sufficient to render almost all gadgets unusable in our tests, but more cautious users may wish to increase the entropy further. To do so, STIR could be extended to pad rewritten code sections with random, unreachable instruction sequences that do not contain returns or jumps. This extra information would increase the number of basic blocks and decrease their relative size, thus increasing the entropy of the system.

6.4.2 Limitations and Future Work

STIR currently only randomizes code available at load time; code generated at runtime is not stirred. Thus, obfuscated code that unpacks itself at runtime derives little benefit from STIR, since the unpacked code is never randomized. Similarly, JIT-compiled code that is generated at runtime remains unrandomized (although the static JIT compiler is randomized by STIR). Periodically re-stirring stir-enabled binaries at runtime could help protect runtime-generated code, but is difficult to realize in practice because it introduces new, more complex varieties of stale pointers for each round of stirring. Implementing a runtime re-stirring system is therefore reserved as a subject of future work.

Our present work focuses on randomizing application main modules rather than libraries because library randomization is a less pressing concern (since libraries already benefit from ASLR randomization) and is significantly easier to realize (since almost all libraries support rebasing, and therefore contain relocation information). Our current implementation therefore only randomizes application main modules, not Windows DLLs or Linux shared objects (SOs). However, library stirring is not difficult to add. It merely requires the implementation of a custom loader that adjusts export address tables of stirred libraries after stirring but before dynamic linking. This is a conceptually easy extension that we intend to undertake in the future.

The experiments reported in Section 6.3 divide basic blocks at unconditional jump instructions already present in the original code. However, STIR can divide basic blocks at

any instruction boundary by inserting jumps that explicitly fall through (i.e., `jmp 0`). This increases the entropy by partitioning the code into smaller, more numerous basic blocks. It also has the benefit of breaking (rather than merely relocating) ROP gadgets that rely on aliased instruction sequences whose encodings span the encodings of adjacent instructions in the application programming. Relocating the instructions so that they are non-adjacent breaks such a gadget. The tradeoff is increased code size (due to the new instructions) and increased runtime overhead (due to the larger number of jumps). Whether this tradeoff is worthwhile is an open question that should be investigated by future work.

Although our randomization strategy defeats typical ROP attacks that chain gadgets, it does not protect against control-flow hijacking attacks that simply call a legitimate computed jump target (e.g., an original method) with corrupted arguments. STIR's address translation logic permits such jumps because it detects and re-points stale pointers to legitimate computed jump targets at runtime. At the binary level, there is little that distinguishes a non-corrupted but stale pointer to such a method (e.g., one drawn from a method dispatch table) from one created by an attacker. Blocking these attacks requires a more refined control-flow integrity policy that dictates exactly which computed jumps may target which methods, like by Abadi et al. (2009). Reliably extracting such information from legacy binaries is a difficult open problem.

As mentioned in Section 6.2.4, our system requires a list of all callback registration functions in unstirred libraries. While the callback registration functions exported by system libraries are theoretically well documented parts of the public system API, in practice we have found that some are less than well documented. For example, some compilers generate calls to internal Windows `libc` functions for which we could find no documentation in any reference manual. To determine the signatures of these callees we were forced to disassemble and reverse-engineer the system modules that contain them. Maintaining a complete list of callback registration functions for a large OS can therefore be challenging when the system API documentation is incorrect or incomplete.

6.5 Conclusion

We have presented STIR, a system that imbues legacy x86 binaries with self-randomizing instruction addresses. The system supports COTS binaries for both Windows and Linux, including those with dynamically computed jumps, code-data interleaving, OS callbacks, dynamic linking, and imperfect disassemblies. The system is compiler-agnostic and requires no form of code-producer cooperation. We have devised and implemented an array of novel techniques to address these challenges. Our evaluation shows that STIR can randomize a large body of large-scale, legacy x86 code, introducing about 1% runtime overhead on average to randomized applications.

CHAPTER 7

OPAQUE CONTROL-FLOW INTEGRITY[1]

Code-reuse attacks, like those described by Bletsch et al. (2011) have become a mainstay of software exploitation over the past several years, due to the rise of data execution protections that nullify traditional code-injection attacks. Rather than injecting malicious payload code directly onto the stack or heap, where modern data execution protections block it from being executed, attackers now ingeniously inject addresses of existing in-memory code fragments (gadgets) onto victim stacks, causing the victim process to execute its own binary code in an unanticipated order (Shacham, 2007). With a sufficiently large victim code section, the pool of exploitable gadgets becomes Turing-complete (Homescu et al., 2012), facilitating the construction of arbitrary attack payloads without the need for code-injection. Such payload construction has even been automated (Schwartz et al., 2011). As a result, code-reuse has largely replaced code-injection as one of the top software security threats.

This has motivated copious work on defenses that can address code-reuse threats. Prior defenses can generally be categorized into two major paradigms: CFI (Abadi et al., 2009) and artificial software diversity (Cohen, 1993).

CFI restricts all of a program's runtime control-flows to a graph of whitelisted control-flow edges. Usually the graph is derived from the semantics of the program source code or a conservative disassembly of its binary code. As a result, CFI-protected programs reject control-flow hijacks that attempt to traverse edges not supported by the original program's semantics. *Fine-grained CFI* monitors indirect control-flows *precisely*; for example, function

[1]This chapter contains material accepted for publication as: Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, February 2015.

callees must return to their exact callers. Although such precision provides the highest security, it also tends to incur high performance overheads, for example about 21% for Abadi et al. (2009). Because this overhead is often too high for industry adoption, researchers have proposed many optimized, coarser-grained variants of CFI. *Coarse-grained CFI* trades some security for better performance by reducing the precision of the checks. For example, functions must return to valid call *sites* (but not necessarily to the particular site that invoked the callee). Unfortunately, such relaxations have proved dangerous—a number of recent proof-of-concept exploits have shown how even minor relaxations of the control-flow policy can be exploited to effect attacks (Davi et al., 2014; Göktaş et al., 2014; Carlini and Wagner, 2014). Table 7.1 summarizes the impact of several of these recent exploits.

Artificial software diversity offers a different but complementary approach that randomizes programs in such a way that attacks succeeding against one program instance have a very low probability of success against other (independently randomized) instances of the same program. Probabilistic defenses rely on memory secrecy—i.e., the effects of randomization must remain hidden from attackers. One of the simplest and most widely adopted forms of artificial diversity is Address Space Layout Randomization (ASLR), which randomizes the base addresses of program segments at load-time. Unfortunately, merely randomizing the base addresses does not yield sufficient entropy to preserve memory secrecy in many cases; there are numerous successful derandomization attacks against ASLR (Shacham et al., 2004; Strackx et al., 2009; Serna, 2012b,a; Joly, 2013; Evans, 2013). Finer-grained diversity techniques obtain exponentially higher entropy by randomizing the relative distances between all code points. For example, binary-level Self-Transforming Instruction Relocation (STIR) (Wartell et al., 2012a) (Chapter 6) and compilers with randomized code-generation (Homescu et al., 2013) have both realized fine-grained artificial diversity for production-level software at very low overheads.

Recently, a new wave of *implementation disclosure attacks* (Snow et al., 2013; Bittau et al., 2014; Seibert et al., 2014) have threatened to undermine fine-grained artificial diversity

Table 7.1. Overview of control-flow integrity bypasses

		CFI (Abadi et al., 2009)	bin-CFI (Zhang and Sekar, 2013)	CCFIR (Zhang et al., 2013)	kBouncer (Pappas et al., 2013a)	ROPecker (Ding et al., 2014)	ROPGuard (Fratric, 2012)	EMET (Microsoft, 2014)
DeMott (2014)	Feb 2014							☹
Göktaş et al. (2014)	May 2014	☹	☹	☹				
Davi et al. (2014)	Aug 2014		☹		☹	☹	☹	☹
Göktaş et al. (2014)	Aug 2014				☹	☹		
Carlini and Wagner (2014)	Aug 2014				☹	☹		

defenses. Implementation disclosure attacks exploit information leak vulnerabilities to read memory pages of victim processes at the discretion of the attacker. By reading the in-memory code sections, attackers thwart the memory secrecy assumptions of artificial diversity, rendering their defenses ineffective. Since finding and closing all information leaks is well known to be prohibitively difficult and often intractable for many large software products, these attacks constitute a very dangerous development in the cyber-threat landscape; there is currently no well-established, practical defense.

This chapter presents *Opaque CFI* (O-CFI): a new approach to coarse-grained CFI that strengthens fine-grained artificial diversity to withstand implementation disclosure attacks. The heart of O-CFI is a new form of control-flow check that conceals the graph of abusable control-flow edges even from attackers who have complete read-access to the randomized binary code, the stack, and the heap of victim processes. Such access only affords attackers knowledge of the intended (and therefore non-abusable) edges of the control-flow graph, not the edges left unprotected by the coarse-grained CFI implementation. Artificial diversification is employed to vary the set of unprotected edges between program instances, maintaining the probabilistic guarantees of fine-grained diversity.

Experiments show that O-CFI enjoys performance overheads comparable to standard fine-grained diversity and non-opaque, coarse-grained CFI. Moreover, O-CFI's control-flow checking logic is implemented using Intel x86/x64 memory-protection extensions (MPX) that are expected to be hardware-accelerated in commodity CPUs from 2015 onwards. We therefore expect even better performance for O-CFI in the near future.

Our contributions are as follows:

- We introduce O-CFI, the first low-overhead code-reuse defense that tolerates implementation disclosures.
- We describe our implementation of a fully functional prototype that protects stripped, x86 legacy binaries without source code.

- Analysis shows that O-CFI provides quantifiable security against state-of-the-art exploits, including JIT-ROP (Snow et al., 2013) and Blind-ROP (Bittau et al., 2014).
- Performance evaluation yields competitive overheads of just 4.92% for computation-intensive programs.

7.1 Threat Model

Our work is motivated by the emergence of attacks against fine-grained diversity and coarse-grained control-flow integrity. We therefore introduce these attacks and distill them into a single, unified threat model.

7.1.1 Bypassing Coarse-Grained CFI

Ideally, CFI permits only programmer-intended control-flow transfers during a program's execution. The typical approach is to assign a unique ID to each permissible indirect control-flow target, and check the IDs at runtime. Unfortunately, this introduces performance overhead proportional to the degree of the graph—the more overlaps between valid target sets of indirect branch instructions, the more IDs must be stored and checked at each branch. Moreover, perfect CFI cannot be realized with a purely static control-flow graph; for example, the permissible destinations of function returns depend on the calling context, which is only known at runtime. Fine-grained CFI therefore implements a dynamically computed shadow stack, incurring high overheads (Abadi et al., 2009).

To avoid this, coarse-grained CFI implementations resort to a reduced-degree, static approximation of the control-flow graph, and merge identifiers at the cost of reduced security. For example, bin-CFI (Zhang et al., 2013) and CCFIR (Zhang and Sekar, 2013) use at most three IDs per branch, and omit shadow stacks.

Recent work has demonstrated that these optimizations open exploitable security holes. By choosing ROP gadgets that start at a function entry point or are call-preceded, it is

possible to build ROP chains that bypass CFI (Göktaş et al., 2014), including subverting CCFIR and bin-CFI. Related works (Davi et al., 2014; Carlini and Wagner, 2014) have similarly shown that call-preceded gadgets can bypass bin-CFI as well as other low-overhead approaches that only check control-flow transfers before potentially dangerous function calls (Fratric, 2012; Pappas et al., 2013a; Ding et al., 2014; Microsoft, 2014). Table 7.1 maps coarse-grained CFI approaches to the corresponding proof-of-concept bypasses. Note that the bypass of the original CFI approach assumes that returns are not tracked precisely using a shadow stack.

Just-In-Time Code Reuse: Until recently, most threat models for CFI and artificial diversity defenses assumed that the memory contents of protected processes were hidden from attackers. The advent of Just-In-Time ROP (JIT-ROP) (Snow et al., 2013) demonstrated that this assumption might be unrealistic in practice due to the existence of implementation disclosure vulnerabilities. Using heap feng shui (Sotirov, 2007), JIT-ROP places a buffer next to a string and a button object. By overflowing the buffer, the string length is set arbitrarily high, allowing the attacker to read any byte in the virtual address space. Parsing the button object through the overflowed string yields a reference to a mapped code page.

Typically, attackers need more than a single 4K page worth of code to find enough gadgets to mount a code-reuse attack. To discourage brute-force searches for more code pages, artificial diversity defenses routinely mine the address space with unmapped pages that abort the process if accessed (Backes and Nürnberger, 2014). JIT-ROP evades these mines by disassembling the initial code page and carefully traversing only direct references to other code pages to recursively discover enough gadgets to mount a ROP attack. Since gadget locations are no longer unknown to the attacker, reliable construction of custom ROP chains becomes possible despite the fine-grained randomization defense.

Blind ROP: While JIT-ROP targets scripting-enabled clients, Blind Return Oriented Programming (BROP) (Bittau et al., 2014) targets vulnerable Internet-facing services, such as

web-servers, that restart after a crash. It capitalizes on the observation that child processes created with the `fork` system call on Linux must be randomized in the same way as their parent in order to continue executing. The attack uses a buffer overflow to overwrite the stack byte-by-byte. Byte values are chosen so that correct guesses cause the server to continue responding as intended, while incorrect guesses solicit a crash and restart. By distinguishing these two outcomes, attackers can remotely infer secret stack cookie values to bypass stack guards and discover gadget locations. Once the `write` system function is located (typically in less than 4000 guesses) the entire code section can be exfiltrated to an attacker-controlled server, after which a traditional ROP attack can be launched against the vulnerable system. Like JIT-ROP, the attack defeats ASLR, DEP, stack canaries and fine-grained code randomization on 64-bit systems.

Side Channel Disclosures: Recent work has even shown that under certain circumstances, gadget locations can be leaked through side channels, such as timing channels (Seibert et al., 2014). This underscores the difficulty of fully protecting software against all implementation disclosure vulnerabilities. Complete protection entails mitigation of all side channel information leaks, which is widely recognized as prohibitively difficult for most non-trivial software products.

7.1.2 Assumptions

Given these sobering realities, we adopt a conservative threat model that assumes that attackers will eventually find and disassemble all code pages in victim processes. Our threat model therefore assumes that the adversary knows the complete in-memory code layout—including the locations of any gadgets required to launch a ROP attack. We also assume that the attacker can read and write the full contents of the heap and stack, as well as any data structures used by the dynamic loader. In keeping with common practice, we assume that

data execution protection is activated, so that code pages are either writable or executable but not both.

However, we assume that attackers cannot safely perform a comprehensive, linear scan of virtual memory, since defenders may place unmapped guard pages at random locations. Instead, attackers must follow references from one disclosed memory page to another (Snow et al., 2013) or resort to guessing (Bittau et al., 2014) in order to avoid inadvertently touching one of these mined pages and alerting defenders (e.g., triggering re-randomization). Successful attacks against our system are therefore those that reliably traverse control-flow edges not intended by the original program semantics without triggering an invalid access violation.

7.2 O-CFI Overview

O-CFI combines insights from CFI and automated software diversity. It extends CFI with a new, coarse-grained CFI enforcement strategy inspired by bounds-checking, that validates control-flow transfers without divulging the bounds against which their destinations are checked. Bounds-checking is fast, the bounds are easier to conceal than arbitrary gadget locations, and the bounds are randomizable. This imbues CFI and fine-grained software diversity with an additional layer of protection against code-reuse attacks aided by implementation disclosures. As a result, O-CFI enjoys performance similar to coarse-grained CFI, with probabilistic security guarantees similar to those of fine-grained artificial diversity in the absence of implementation disclosures.

Following traditional CFI, an O-CFI policy assigns to each indirect branch site in the program a *destination set* that captures its set of permissible destination addresses. Such a graph can be derived from program source code or (with lesser precision) a conservative disassembly of the program's object code. We next reformulate this policy as a bounds-checking problem by reducing each destination set to only its minimal and maximal members.

This approximation of the control-flow policy can be efficiently enforced by confining each branch to the memory-aligned addresses within its destination set range. All intended destination addresses are aligned within these bounds, so the enforcement conservatively preserves intended control-flows. Code layout is optimized to tighten the bounds, so that the set of unintended, aligned destinations within the bounds remains minimal. These few remaining unintended but reachable destinations are protected by the artificial diversity half of our approach.

The artificial diversity half of our approach probabilistically protects the aligned, in-bounds, but policy-violating control-flows by applying fine-grained randomization to the binary code at load-time. While the overall strategy for implementing this randomization step is based on prior works (Wartell et al., 2012a), its purpose in O-CFI is fundamentally different. Randomizing the code layout *does not conceal the new layout from attackers*, since our threat model assumes attackers can read all the randomized code. Rather, its purpose is to randomize the *bounds* to which each branch is constrained. The bounds imposed upon each branch are not disclosed by the binary code since bounds values are stored in protected registers, not expressed as immediate instruction arguments. Thus, attackers who can read the binary code must nevertheless guess which control-flow hijacks trigger an out-of-bounds branch violation and which do not.

Reformulating CFI in this way forces attackers to change their plan of attack. The recent attacks against coarse-grained CFI succeed by finding exploitable code that is reachable due to policy-relaxations needed for acceptable performance. These relaxations admit an alarming array of false-positives: instead of identifying the actual caller, all call-preceded instructions are incorrectly identified as permitted branch destinations. Such instructions saturate a typical address space, giving attackers too much wiggle room to build attacks. O-CFI counters this by changing the approximation approach: each branch destination is restricted to a relatively short span of aligned addresses, with all the bounds chosen pseudo-randomly at load-time. This greatly narrows the field of possible hijacks, and it removes the

opportunity for attackers to analyze programs ahead of time for viable ROP gadget chains. In O-CFI, no two program instances admit the same set of ROP payloads, since the bounds are all randomized every time the program is loaded.

Since the security of coarse-grained CFI depends in part on the precision of its policy approximation, it is important to note that O-CFI policy approximations can be made more precise by tightening the bounds imposed upon each branch. This effectively reduces the space of attacker guesses that might succeed in hijacking any given branch. To reduce this space as much as possible, we introduce a novel binary code optimization, called *portals*, that minimizes the distance covered by the lowest and greatest element of each indirect branch's destination set.

Our fine-grained artificial diversity implementation is an adaptation and extension of binary stirring (Wartell et al., 2012a) (Chapter 6). Binary stirring randomizes the ordering of basic blocks within code sections each time a program binary is loaded into memory. The stirring has the effect of randomizing bounds to defeat attackers armed with implementation knowledge, and affords even higher probabilistic protections against attackers who lack such knowledge. Thus, O-CFI offers security that strictly subsumes and exceeds traditional fine-grained code randomization.

In order to protect against information leaks that might disclose bounds information, our implementation is carefully designed to keep all bounds opaque to external threats. They are randomly chosen at load-time (as a side-effect of binary stirring) and stored in a *bounds lookup table* (BLT) located at a randomly chosen base address. Since the table size is very small relative to the virtual address space, and since attackers cannot safely perform brute-force scans of the full address space (see Section 7.1.2), guessing the location of the BLT is probabilistically infeasible for attackers. No code or data sections contain any pointer references to BLT addresses; all references are computed dynamically at load-time and stored henceforth exclusively in protected registers.

7.2.1 Bounding the Control Flow

For each indirect branch site with (non-empty) destination set D, O-CFI guards the branch instruction with a bounds-check that continues execution only if the impending target t satisfies $t \in [\min D, \max D]$. Indirect branch instructions include all control-flow transfer instructions that target computed destinations, including return instructions. Failure of the bounds-check solicits immediate process termination with an error code (for easier debugging). Termination could be replaced with a different intervention if desired, such as an automated attack analysis or alarm, followed by restart and re-randomization.

The bounds-check implementation first loads the pair $(\min D, \max D)$ from the BLT into registers via an indirect, indexed memory reference. The load instruction's arguments and syntax are independent of the BLT's location, concealing its address from attackers who can read the checking code. The impending branch target t is then checked against the loaded bounds. If the check succeeds, execution continues; otherwise the process immediately terminates with a bounds range (#BR) exception. The #BR exception helps distinguish between crashes and guessing attacks. To resist guessing attacks (e.g., BROP), web servers and other services should use this exception to trigger re-randomization as they restart.

Table 7.2 contains pseudocode for the guards. The BLT securely stores $(\min D, \max D)$ pairs for all branches, and is indexed using unique branch IDs (`branch_id` in Table 7.2).

Following the approaches of PittSFIeld (McCamant and Morrisett, 2006), Google Native Client (Yee et al., 2009), and Reins (Wartell et al., 2012b) (Chapter 5), O-CFI also aligns all policy-permitted indirect branch destinations to power-of-two addresses, and masks the low-order bits of all indirect branch arguments to force their targets to aligned addresses. This prevents attackers from diverting control to misaligned instructions that are not intended to be reachable by any legitimate flow of the original program. This is important since any gadgets formed from misaligned instructions do not receive bounds checks.

Table 7.2. Pseudo-code to constrain branch bounds.

Description	**Original code**	**Rewritten code**

Indirect branches	`call/jmp` $\langle dest \rangle$	t `:=` $\langle dest \rangle$
		t `:=` t & *align_mask*
		(b_{min}, b_{max}) `:= blt[branch_id]`
		`if not` $b_{min} \leq t \leq b_{max}$`:`
		`  abort(#BR);`
		`call/jmp` t

Returns	`ret` $\langle n \rangle$	`[esp] :=` `[esp]` & *align_mask*
		(b_{min}, b_{max}) `:= blt[branch_id]`
		`if not` $b_{min} \leq$ `[esp]` $\leq b_{max}$`:`
		`  abort(#BR);`
		`ret` $\langle n \rangle$

To bypass these checks, an attacker must craft a payload whose every gadget is properly aligned and falls within the bounds of the preceding gadget's conclusory indirect branch. The odds of guessing a reachable series of such gadgets decrease exponentially with the number of gadgets in the desired payload.

7.2.2 Opacifying Control-flow Bounds

Diversifying bounds: The bounds introduced by O-CFI constitute a coarse-grained CFI policy. Section 7.1 warns that such coarse granularity can lead to vulnerabilities. However, to exploit such vulnerabilities, attackers must discover which control-flows adhere to the CFI policy and which do not. To make the impermissible flows opaque to attackers, we use diversity. Our prototype uses a modified version of the technique outlined by Wartell et al. (Wartell et al., 2012a), which shuffles the basic block order at program load-time. The general approach could alternatively be implemented as a compiler-based defense for software whose source codes are available.

Performing fine-grain code randomization at load-time indirectly randomizes the ranges used to bound the control-flow. In contrast to other CFI techniques, attackers therefore do not have *a priori* knowledge of the control-flow bounds.

Preventing Information Leaks: Attackers bypass fine grained diversity using information leaks, such as those described in Section 7.1.1. Were O-CFI's control-flow bounds expressed as constants in the instruction stream, attackers could bypass our defense via information leaks. To avoid this, we instead confine this sensitive information to an isolated data page, the BLT. The BLT is initialized at a random virtual memory address at load-time, and there are no pointer references (obfuscated or otherwise) to any BLT address in any code or data page in the process. This keeps its location hidden from attackers.

Furthermore, we take additional steps to prevent accidental BLT disclosure via pointer leaks. Our prototype stores BLT base addresses in segment selectors—a legacy feature of all x86 processors. In particular, each load from the BLT uses the `gs` segment selector and a unique index to read the correct bounds. We only use the `gs` selector for instructions that implement bounds checks, so there are no other instructions that adversaries can reuse to learn its value. Attackers are also prevented from executing instructions that reveal the contents of the segment registers, since such instructions are privileged.

To succeed, attackers must therefore (i) guess branch ranges, or (ii) guess the base address of the BLT. The odds of correctly guessing the location of the BLT are low enough to provide probabilistic protection. On 32-bit Windows Systems, for instance, the chances of guessing the base address are

$$\frac{1}{\frac{2^{31}}{2^{12}}} = \frac{1}{524,288}$$

and on 64-bit Windows, the chances are

$$\frac{1}{\frac{2^{43}}{2^{12}}} = \frac{1}{2,147,483,648}$$

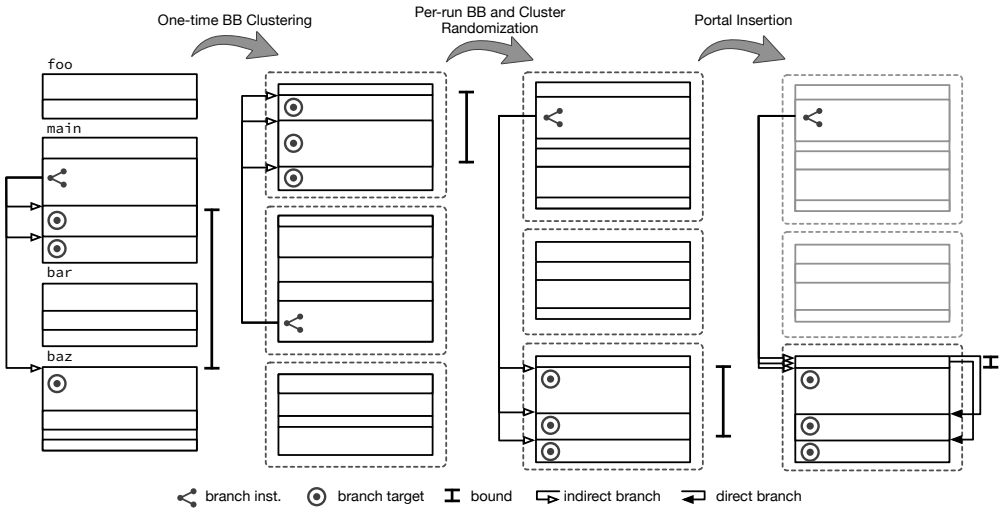

Figure 7.1. O-CFI code layout transformation. Clustering occurs once, before the program executes (2nd column). Basic block and cluster randomization (3rd column), and portal insertion (4th column) occurs at load-time.

or less than one in two billion. Incorrect guesses alert defenders and trigger re-randomization with high probability (by accessing an unallocated memory page).

The likelihood of successfully guessing a reachable gadget chain is a function of the length of the chain and the span of the bounds. The next section therefore focuses on reducing the average bounds span.

7.2.3 Tightening Control-flow Check Bounds

The distance between the lowest and highest intended destinations of any given indirect control-flow transfer instruction depends on the code layout. Placing indirect branches close to their targets both reduces bounds and improves locality, elevating both security and efficiency. Therefore we organize the code segment into *clusters*—one per indirect branch—each containing the basic blocks targeted by a particular branch.

To accommodate blocks that are destinations of multiple distinct branch instructions, we consider three options: (i) put the block in one cluster and expand the bounds of other branches to include its address, (ii) create duplicate copies of the block in multiple clusters,

or (iii) add a *portal* block to each cluster, which unconditionally jumps to the block. Each solution incurs a trade-off: expanding bounds reduces security, creating duplicates increases code size, and portals introduce runtime overhead. The options are not mutually exclusive, affording optimizers a range of strategies. Our experiments indicate that portals are often the best choice (see below).

Figure 7.1 depicts the clustering, randomization, and portal creation phases, and their effects on the layout of the code segment—particularly with respect to bounds sizes. To prevent the load-time, fine-grained randomization phase from increasing the resulting bounds ranges (e.g., by placing two targets at opposite ends of the address space) and consequently nullifying the advantages of clustering, the load-time initializer proceeds in two stages, both depicted in the third column of Figure 7.1. First, it randomizes the locations of blocks within each cluster. This effectively changes instruction addresses without affecting the bounds of any of the indirect branches. Next, it randomizes the cluster order, further increasing entropy.

Minimizing Branch Ranges with Portals: A portal is a direct jump to a block. Jumping to a portal is therefore semantically equivalent to jumping directly to the block it targets. Placing a portal to a stray target within the cluster that contains all of the other targets of a branch avoids code duplication, and only widens the bounds by the portal size. Even when all n targets of a branch fall within a single cluster, bounds ranges can be further minimized by creating n portals to its n blocks. We call a collection of portals within a common cluster a *nexus*.

The *capacity* of the portal system limits the number of portals per nexus. Varying nexus capacity allows O-CFI to be tuned to different requirements. Setting it to zero prevents the creation of any portals, forcing the optimizer to choose alternative options. At the other extreme, setting no upper limit allows a portal to be created for *every* target, reducing all bounds ranges to wt, where w is the alignment width (usually 16 or 32 bytes; see Section 7.4.1) and t is the number of targets of the branch. At this setting, all indirect

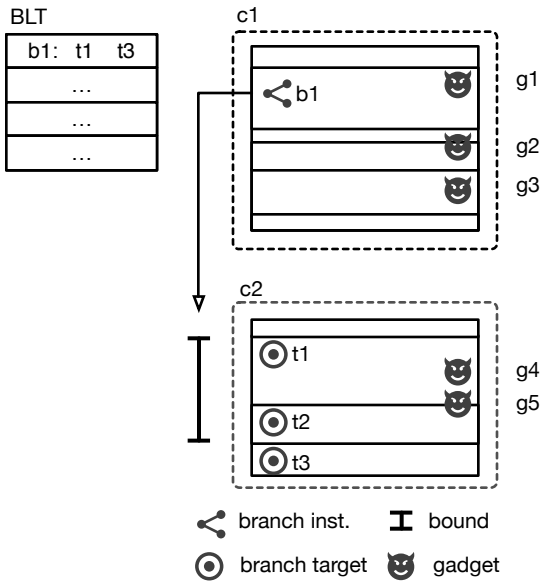

Figure 7.2. Chaining gadgets in O-CFI

branches can only branch into a nexus, and through them, only to exactly those addresses that have been statically identified as targets. Thus, O-CFI with unbounded nexus capacity enforces fine-grained, static CFI.

The extra layer of indirection imposed by a portal has a minor impact on runtime; there is thus a trade-off between security and performance. Users may opt for full CFI enforcement with O-CFI for security-critical components, and lower the nexus capacity to a desired performance level for less critical software. In our experiments, we found that a nexus capacity of 12 results in a significant reduction in bounds sizes with imperceptible performance effects. All of our experiments in Section 7.4 use this nexus capacity. Section 7.4.4 details how different nexus capacities affect bound ranges.

7.2.4 Example Defense against JIT-ROP

The following example illustrates how O-CFI secures binaries against disclosure attacks. Consider a binary whose code segment contains five useful gadgets $g_1, \ldots, g_5$. Each gadget terminates in an indirect branch protected by a bounds check. Figure 7.2 shows such a

binary after basic block clustering and randomization. Cluster c_2 contains the statically identified targets for the terminating indirect branch b_1 of gadget g_1. O-CFI constrains b_1 so it allows branches to a subset of the instructions contained in cluster c_2 (with similar constraints placed on other branches).

Under appropriate conditions, a disclosure attack such as JIT-ROP is able to recover a large portion of the runtime layout of the binary (Snow et al., 2013). Our system does not prevent any of the six JIT-ROP steps. We allow code pages to be harvested, gadgets identified, and a payload to be dynamically crafted. We may even allow a few initial gadgets in the payload to execute. However, O-CFI hides the valid ranges for each gadgets from attackers even in the presence of implementation disclosure vulnerabilities. Recall that for this guessed payload to successfully execute, every gadget in the payload chain must fall within the valid range of the preceding gadget.

In our example, if g_1 is selected to be part of the payload, it can only be chained with gadget g_4 or g_5. Attempting to jump from g_1 to any other gadget triggers a bounds violation that stops the attack. Similarly, an attack that hijacks a control-flow to c_1 can only redirect it to gadgets g_1, g_2, or g_3; all other gadgets are outside cluster c_1 and are therefore detected as impermissible destinations of the hijacked branch. Broadly speaking, all links in a payload's chain must traverse edges in the Cartesian product of the (aligned) gadget sets within the corresponding clusters.

A successful attack must therefore limit itself to an extremely sparse graph of available edges. Our experiments (see Section 7.4.3) indicate that in practice the probability of successfully chaining gadgets in such a sparse graph is very low—just 0.01% for a four-gadget payload. The entropy of our procedure is further analyzed in Section 7.5.1.

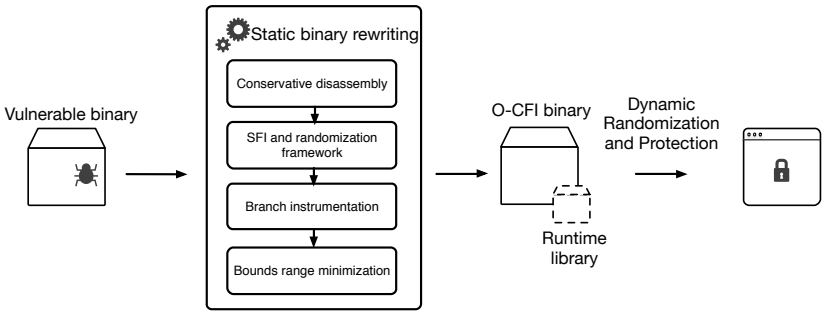

Figure 7.3. O-CFI architecture. A vulnerable COTS x86 binary is analyzed and instrumented statically to create the corresponding O-CFI binary. At load-time, a runtime library added to the O-CFI binary randomizes the code layout and bounds the targets of every indirect branch.

7.3 O-CFI Implementation

We have implemented a fully functional prototype of O-CFI for the Intel x86 architecture. Our implementation uses a binary rewriting framework that secures COTS x86 binaries without source, debug, or relocation information. Like traditional CFI, however, we emphasize that O-CFI is equally suitable for inclusion in a compiler.

Our rewriter generates a transformed version of the binary that leverages

1. a coarse-grained CFI policy that bounds control-flows,

2. fine-grained randomization to protect against traditional ROP attacks and to diversify control-flow bounds so they become unknown and unreliable for attackers,

3. x86 segmentation registers to prevent accidental leakage of the bounds lookup table (BLT), and

4. an SFI framework similar to PittSFIeld (McCamant and Morrisett, 2006) to enforce instruction alignment. This prevents attackers from accessing misaligned instructions to bypass bounds checks.

The architecture of O-CFI is shown in Figure 7.3, and consists of two major phases. The first is a one-time static binary rewriting phase that minimizes bounds via clustering

and inserts the runtime library that implements the dynamic phase. The second is a load-time, dynamic randomization and protection phase in which the runtime library randomizes the code layout, and populates the BLT to reflect the new locations of indirect branch targets. The dynamic phase also performs additional optimization, reducing bounds ranges to improve security.

7.3.1 Static Binary Rewriting

Conservative Disassembly

We begin by disassembling the code section using a conservative disassembler. Similar to the approach outlined by Wartell et al. (Wartell et al., 2012a), the code section is duplicated, with the old copy (renamed to `.told`) serving as a read-only data segment and the new copy (called `.tnew`) containing the rewritten executable code. The `.told` section is set non-executable, and all code blocks identified as possible targets of indirect jumps are overwritten with a five-byte tagged pointer. The tagged pointer consists of a tag byte (`0xF4`) followed by the four-byte address of the new location of that block in the `.tnew` section. The tag byte allows for an efficient runtime mechanism to redirect stale pointers to their correct targets, and is explained below.

Since the `.told` section preserves all static data at its original addresses, data pointers in the rewritten code section continue to behave correctly. This makes the rewriting system robust to disassembly errors that misclassify data as code.

Disassembly errors that misclassify code as data could omit such code from the `.tnew` section, resulting in a crash at runtime. To avoid this, we use settings that encourage the disassembler to interpret all bytes with valid instruction encodings as code. In our experiments, these settings suffice to avoid all disassembly errors that affect proper code translation.

SFI and Randomization Framework

To prevent attacks from jumping over the guards that constrain branch ranges, the new code segment is split into power-of-two sized basic-blocks called *chunks* (McCamant and Morrisett, 2006). Guard instructions and the branches they guard are always co-located within a common chunk, and branch targets are confined to chunk boundaries, with padding inserted where necessary to preserve this property. Confining branches to chunk boundaries is efficiently realized as a single `and` instruction per branch, which clears the lower i bits of the target address (where the size of a chunk is 2^{i+1} bytes).

Thus, control-flow within a chunk is linear, passing over every instruction from start to end. This chunking and masking regimen ensures that control-flow guards execute before every indirect branch. Additionally, it prevents jumps to misaligned (and hence unguarded) instruction sequences, reducing the attack surface to the set of statically disassembled (and hence protected) gadgets.

Direct branches are statically rewritten to reference their new target addresses. Indirect branches require extra effort, since their exact targets only become known at runtime. At runtime, there are two common cases to consider: (a) the impending target is already within the `.tnew` section (e.g., it was pushed by a `call` instruction), or (b) the impending target is a *stale pointer* that points into the `.told` section (e.g., it was loaded from a method dispatch table stored as heap data, which the rewriter does not rewrite). The first case requires no special treatment; the second solicits an efficient dynamic lookup and redirection of the stale pointer to its new location (Wartell et al., 2012a). Specifically, we implement a two-instruction sequence that tests for the presence of the tag byte at the target address, and if present, substitutes the current target with the address stored after the tag, which points to the block's new location in the `.tnew` section. The tag byte is chosen to be an illegal instruction encoding, so that no non-stale code pointer ever points to such a byte.

The stale pointer redirection mechanism is not relied upon for security. Like all indirect branch targets, redirected pointers undergo a mask and bounds-check before becoming control-flow destinations. Thus, corrupting or defeating the redirection mechanism does not circumvent the security policy.

The ability to redirect code pointers lays the foundation for load-time randomization. Once the new randomized locations for basic blocks have been finalized, updating the values in the `.told` section allows our redirection mechanism to correctly redirect all indirect branches to the new, randomized block locations. Direct branches are simply modified in-place.

Branch Instrumentation

The techniques described above provide a framework that can rewrite COTS binaries to provide SFI and fine-grained randomization. This suffices to protect against traditional ROP attacks, but not against implementation disclosure vulnerabilities, which require the additional hardening implemented by O-CFI's bounds-checking. Bounds-checking is applied after stale pointer redirection alongside masking, to further limit the set of accessible gadgets. Due to the enforced instruction alignment, attack payloads cannot circumvent these bounds checks.

Furthermore, due to randomization, the bounds remain unknown to implementation-aware attackers, and vary from program instance to program instance. Attacks cannot statically pre-compute bounds ranges because the runtime randomization phase changes bounds values on each execution. They also cannot dynamically leak the bounds, all of which are stored securely in the BLT and never leaked to the stack or heap. Attackers must therefore hazard guesses as to which gadget chains are safely accessible for any given program instance.

The bounds-checking implementation logic is detailed in Table 7.2. In our implementation for MPX-enabled, 32-bit systems, each BLT entry consists of two 32-bit pointers.

Accurate Target Identification

To ensure that we identify all intended targets of indirect branches, we employ disassembly heuristics that identify a superset of potential targets. As an example, we follow the following sequence of steps to identify the set of potential targets for a return instruction:

1. Identify all code references to the function that contains the return. This includes direct and indirect branches to the function entry point, as well as to any basic block within the function.

2. For each identified branch that is not a call, find all code references that flow into it.

3. Recursively traverse all non-call references until a fixed point is reached (i.e., a set with only calls).

4. The instruction immediately after each call forms the target set for that return.

Our heuristics are tuned to prefer false positives (non-targets treated as possibly valid destinations), since such errors do not significantly affect the operation of our system. In particular, each such error only marginally weakens the system's security (by admitting an unnecessary control-flow link that remains guarded by randomization) and slightly increases generated code size. A compiler-side solution could be more precise, at the cost of requiring source code and recompilation of programs.

Bounds Range Minimization

As discussed in Section 7.2.3, we use a combination of clustering and portals to reduce bounds ranges. While the portals themselves are created only at binary load-time, it is in the static phase that branch targets are clustered together and empty nexuses created. Algorithm 4 gives a high level overview of our clustering algorithm. Each cluster created

Algorithm 4 *CreateClusters*(S): Cluster basic blocks to place the targets of indirect branches as close together as possible.

Input: S {the set of the basic blocks in the code segment}
Output: C {a set of clusters, one per indirect branch. Each $c \in C$ is a set of basic blocks that contain all the targets for a specific branch, as well as an empty nexus where portals can be inserted at runtime.}

```
C ← ∅
for all b ∈ Branches(S) do
  c ← ∅
  for all t ∈ Targets(b) do
    b' ← GetBasicBlock(t)
    if b' ∉ C then
      c ← c ∪ {b'}
    end if
  end for
  {The nexus is an empty basic block to hold portals.}
  c ← c ∪ CreateNexus()
  C ← C ∪ {c}
end for
{Add unclaimed basic-blocks into a single final cluster.}
C ← C ∪ {(S − ⋃C)}
```

in this step gets an empty nexus. In our implementation, all nexuses are homogeneous in size, but more sophisticated implementations could tailor nexus sizes to individual branches based on the size of their statically determined target set.

Organizing the code into power-of-two sized chunks (for SFI enforcement) impacts portals. In the absence of chunking, the size of each portal is the five bytes required for a direct jump; but chunking rounds this up to the nearest multiple of the chunk-size. In our implementation, this makes each portal 16 or 32 bytes long. Though this slightly increases both file and code sizes, it only marginally affects the average bounds size, and does not noticeably impact performance. Section 7.4.4 contains a detailed breakdown of how bounds sizes vary with the number of portals per cluster.

7.3.2 Accelerated Bounds Checks

To optimize performance, we leverage the Intel memory-protection extensions (MPX) for x86/64 architectures to store and check bounds (Intel, 2013). MPX instructions will be supported in Intel processors from 2015 onwards, so our approach will benefit from hardware acceleration in the near future. MPX instructions execute as NOPs on non-MPX processors.

MPX provides hardware accelerated protection against buffer overflow or underflow attacks, and introduces new registers and instructions to this end. The eight new bounds registers each hold two pointers, and can be used to store both the lower and upper bounds associated with a pointer value. New MPX instructions allow for quick loading and querying of these bounds registers. We use the three MPX instructions shown in Table 7.3. Instruction `bndmov` loads bounds from the BLT into bounds register `bnd`, and `bndcl` and `bndcu` verify that the target address is within the loaded bounds.

To secure binaries intended for use on non-MPX systems, O-CFI also has a legacy mode that uses the `cmp` and `cmovcc` instructions to guard branches. Although the lack of dedicated range checking instructions makes these guards less efficient than their MPX-enabled counterparts, binaries rewritten in this mode receive the same level of protection. Section 7.3 describes our guard implementations in-depth, and we evaluate their performance characteristics in Section 7.4.

The final consolidated sequence of instructions needed to enforce bounds, prevent execution of unintended instructions, and allow fine-grained randomization is shown in Table 7.4. Column three shows the instructions used when targeting MPX-compatible platforms, while the instructions used to enforce bounds on non-MPX, legacy processors are shown in column four. In each listing, instructions appearing before the first chunk boundary are appended to the preceding chunk, or wherever they best fit. Subsequent instructions are confined to dedicated chunks in order to maintain security.

Table 7.3. MPX instructions used in O-CFI

Syntax	Description
`bndmov bnd, m64`	Move upper and lower bound from `m64` to bound register `bnd`.
`bndcl bnd, r/m32`	Generate a #BR if `r/m32` is less than the lower bound in `bnd`.
`bndcu bnd, r/m32`	Generate a #BR is `r/m32` is higher than the upper bound in `bnd`.

MPX Mode: In MPX mode, we use 16-byte chunks. For indirect branches, lines 1 and 11 preserve the `eax` register, which is used as a scratch space. Lines 3 and 4 implement the dynamic lookup and redirection mechanism for stale code pointers. Lines 5, 6, and 10 load the bounds associated with this branch into bounds register `bnd1` and then compare it against the target address. If the target address is outside the bounds, a #BR exception is raised, and the program halted. Line 9 masks the target address, forcing it to a chunk boundary.

Line 8 foils hijackers who attempt to abuse the final chunk as a gadget. The earlier chunk boundary (above line 5) needs no such protection because all logic above it is strictly for preserving program functionality, not for enforcing security. Thus, jumping to that boundary during a code-reuse attack does not help the attacker—the resulting gadget implements a fully guarded jump.

The process is shorter for returns, since returns do not require stale pointer correction. The full return guard code therefore fits within two 16-byte chunks.

Legacy Mode: To protect binaries executing on processors without MPX support, O-CFI emits legacy mode guards. This mode uses 32-byte chunks and stores BLT entries consisting of three 32-bit pointers: the address of an abort function, the lower bound, and the upper bound, respectively. Comparison (`cmp`) instructions compare the target address against

Table 7.4. Summary of code transformations

Description	Original Code	Rewritten Code (MPX-mode)	Rewritten Code (Legacy-mode)

Indirect Branches	`call/jmp` *r*/[*m*]	`1: mov [esp-4], eax`
		`2: mov eax,` *r*/[*m*]
		`3: cmp byte ptr [eax], 0xF4`
		`4: cmovz eax, [eax+1]`
		— 16-byte chunk boundary —
		`5: bndmov bnd1, gs:[branch_id]`
		`6: bndcu bnd1, eax`
		`7: jmp 9`
		— 16-byte chunk boundary —
		`8: xor eax, eax`
		`9: and al,` *align_mask*
		`10: bndcl bnd1, eax`
		`11: xchg eax, [esp-4]`
		`12: call/jmp [esp-4]`
			`1: mov [esp-8], eax`
			`2: push ecx`
			`3: mov eax,` *r*/[*m*]
			`4: cmp byte ptr [eax], 0xF4`
			`5: cmovz eax, [eax+1]`
			— 32-byte chunk boundary —
			`6: and al,` *align_mask*
			`7: mov ecx, branch_id`
			`8: cmp eax, gs:[ecx+4]`
			`9: cmovb eax, gs:[ecx]`
			`10: cmp eax, gs:[ecx+8]`
			`11: cmovae eax, gs:[ecx]`
			`12: pop ecx`
			`13: xchg eax, [esp-8]`
			`14: call/jmp [esp-8]`

Returns	`ret`	— 16-byte chunk boundary —
		`1: xchg eax, [esp]`
		`2: and al,` *align_mask*
		`3: bndmov bnd1, gs:[branch_id]`
		`4: jmp 6`
		— 16-byte chunk boundary —
		`5: xor eax, eax`
		`6: bndcu bnd1, eax`
		`7: bndcl bnd1, eax`
		`8: xchg eax, [esp]`
		`9: ret`
			`1: xchg eax, [esp]`
			`2: push ecx`
			— 32-byte chunk boundary —
			`3: and al,` *align_mask*
			`4: mov ecx, branch_id`
			`5: cmp eax, gs:[ecx+4]`
			`6: cmovb eax, gs:[ecx]`
			`7: cmp eax, gs:[ecx+8]`
			`8: cmovbe eax, gs:[ecx]`
			`9: pop ecx`
			`10: xchg eax, [esp]`
			`11: ret`

both bounds. Conditional move instructions `cmovb` and `cmovae` replace the intended target address with the abort function if either test fails. The abort function halts (and optionally re-randomizes and restarts) the program.

Legacy mode's larger chunk size and inclusion of abort function pointers in the BLT are design choices intended to balance speed and space overheads given the lengthier instruction sequences required to implement the guards without MPX support. The lengthier sequence fits (just barely) within one 32-byte chunk without protection of an internal chunk boundary, making that chunk size the best choice. (As before, the first chunk boundary requires no protection, since the instructions that follow it implement a fully guarded jump.)

7.3.3 Dynamic Randomization and Protection

The dynamic phase of our system proceeds at load-time, before the binary executes. In this phase the basic blocks are randomized for diversity, bounds on indirect branches

Algorithm 5 *RuntimeSetup*(C, BLT): Perform runtime randomization, and bounds range setup and optimization.

Input: C {clustered code segment}, BLT {bounds table}
Output: C {randomized, bounds optimized code segment}
  RandomizeCode(C)
  CreateAllPortals(C)
  UpdateDirectBranches(C)
  UpdateJumpTable(C)
  UpdateBoundsTable(BLT)
  SetupSegmentedAccess(BLT) {Move the bounds table to a random page and set up segmented memory access to it via the `gs` register.}

Algorithm 6 *RandomizeCode*$(C, Shuffle)$: Randomize basic blocks in a cluster-aware manner.

Input: C {clustered code segment}, *Shuffle* {a method that takes a set as input and outputs a random ordering}
Output: a randomized code segment
  {Shuffle basic-blocks within the cluster.}
  $R \leftarrow \emptyset$
  for all $c \in C$ **do**
    $R \leftarrow R \cup \{Shuffle(c)\}$
  end for
  {Shuffle the order of clusters within the code segment.}
  return *Shuffle*(R)

Algorithm 7 *CreatePortal*(N, t): Create a portal to t from nexus N.

Input: N {a nexus}, t {target address}
Output: Add a direct jump to t in the first available slot in N, and returns its address.
  for $i = 1$ **to** *Capacity*(N) **do**
    if $N[i]$ is an empty slot **then**
      $N[i] \leftarrow CreateDirectJump(t)$
      return *Address*$(N[i])$
    end if
  end for
  return *null* {Return null if insufficient capacity.}

are fixed, and bounds ranges are further minimized. O-CFI uses a runtime library for this purpose, which it injects into the Import Address Table (IAT) of the rewritten binary during the static phase.

Algorithm 8 *CreateAllPortals*(C): Fill each nexus with portals to targets until (a) there are no more targets for that branch, or (b) the nexus capacity is reached.

Input: C {the ordered set of clusters, post randomization}
Output: Fill nexuses with portals.

```
for all c ∈ C do
  for i = 1 to Capacity(c.nexus) do
    b ← GetParentBranch(c)
    f ← FarthestTarget(c, b)
    q ← CreatePortal(c.nexus, f)
    if q ≠ null then
      UpdateTargets(b, f, q)
    end if
  end for
end for
```

The Windows load order guarantees that all statically linked libraries initialize before the modules that link to them. The dynamic phase is thus carried out by the intermediate library in its initialization code. Algorithm 5 describes the order of steps taken by the initialization code.

First, the two-step process detailed in Algorithm 6 is used to randomize the code segment without affecting bounds ranges. Next, each nexus is populated using a greedy algorithm that creates portals to the farthest targets of its parent branch step-by-step, until its capacity is exhausted. Algorithm 8 shows the pseudo-code for this process while Algorithm 7 details how individual portals are created.

Once the targets of all branches have been finalized, direct branch operands are relocated to their correct locations and all jump-table entries in the `.told` section are updated with their new addresses. The BLT is also updated to reflect the new ranges for each branch, after which it is then moved to a random page of memory. Finally, the `gs` segment register is updated to point to the new base address of the BLT.

7.3.4 Platform Support and Infrastructure

We have implemented O-CFI for 32-bit versions of Windows (XP/Vista/7/8). Our O-CFI approach however, is not OS specific and is also applicable to 64-bit versions of Windows as well as Linux and OS X.

The implementation consists of three parts: (i) a static rewriter, (ii) an intermediate library, and (iii) an API hooking utility. The rewriter is implemented as a 2600 line IDA Python program. IDA Python programs are parsed and interpreted by the Hex-rays IDA Pro 6.1 disassembler. One of IDA Pro's many uses is as a malware reverse engineering and de-obfuscating tool, and it features many powerful code analyses that heuristically recover program structural information without assistance from source code or debug information. Our system leverages these analyses to automatically distinguish code from data, and to identify indirect branches and possible targets of such branches.

The intermediary library consists of approximately 500 lines of C and hand-written, in-lined assembly code that facilitates run-time fine-grained randomization and subsequent portal creation. It also provides support for callback functions and dynamic linking. An additional 150-line configuration file itemizes all trusted callback registration functions exported by Windows libraries used by the benchmarks.

The API hooking utility adds the intermediary library to the PE's list of imported modules. To avoid expanding the size of the PE header (which could shift the positions of the binary sections that follow it), our utility simply changes the library name `kernel32.dll` in the import section of the name of our intermediary library instead. The intermediary library exports all `kernel32` symbols as forwards to the real `kernel32`, except for security relevant functions, which it exports as local replacements.

7.4 Evaluation

We tested O-CFI with binaries from the SPEC2000 benchmark suite listed in Table 7.5. All results detailed below were obtained on an Asus G53SW machine with 4GB of RAM and the Intel Core i7 2630QM processor.

7.4.1 Rewriting and Space Overheads

Table 7.5 reports the percentage increase in file size and code size in the rewritten binaries, as well as the time taken by O-CFI to rewrite each binary. Our prototype rewrites about 60KB of code per second on average. In legacy mode, binaries increase by an average of 194%, while the code segment size increases by about 145%. For binaries secured with MPX mode guards, these measures are reduced to 137% for the file size increase, and roughly 71% for the increase in code segment size. This difference reflects the relative cost of 32-byte chunks in legacy mode versus 16-byte chunks in MPX mode.

Although O-CFI's size increase may appear substantial, we believe that in many cases trading increased memory usage for substantially better security at high runtime performance is a worthwhile exchange. Our design is therefore calibrated to favor security and speed over space overheads in common cases.

7.4.2 Performance Overheads

Figure 7.4 shows the performance impact of O-CFI. We report overheads for both MPX and legacy mode guards for each binary. In both cases, they are measured as the mean percentage increase in runtimes between the rewritten and original binary over 20 trials. The average overhead across all benchmarks is 4.92% when using the legacy-mode guards shown rightmost in Table 7.4.

We cannot provide precise numbers on MPX-enabled hardware since such hardware is not yet available. However, since MPX instructions are treated as NOPs by the processor

Table 7.5. Space and rewriting overheads

Binary Program	Size Increase (Legacy) File (%)	Size Increase (Legacy) Code (%)	Size Increase (MPX) File (%)	Size Increase (MPX) Code (%)	Rewriting Time (s)
`gzip`	194	148	134	72	4.07
`vpr`	167	126	118	59	6.71
`mcf`	206	153	142	75	3.48
`parser`	207	158	141	76	5.96
`gap`	216	162	152	85	1.32
`bzip2`	194	150	133	72	3.50
`twolf`	165	109	120	53	7.84
`art`	204	152	140	74	3.62
`equake`	193	144	134	70	3.62
mean	+194%	+145%	+137%	+71%	5.85s

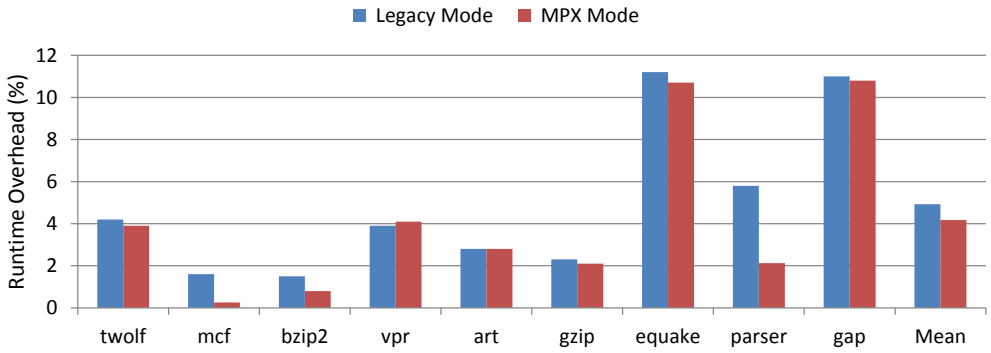

Figure 7.4. O-CFI runtime overhead

used in our experiments, and since long NOP instructions have roughly the same execution characteristics as ALU operations with register operands on Core i7 processors (Fog, 2014), we can use our current hardware to estimate the performance impact of MPX acceleration. We find that MPX instructions decrease the performance impact to 4.17% and conclude that O-CFI is likely to benefit from MPX support from 2015 and onwards.

Both types of guards show similar characteristics across the benchmarks, exhibiting negligible overhead with `mcf` and `bzip2`, while `gap` and `equake` incur the largest overheads. A few benchmarks (`art` and `gzip`) exhibit marginally better performance with the legacy mode guards.

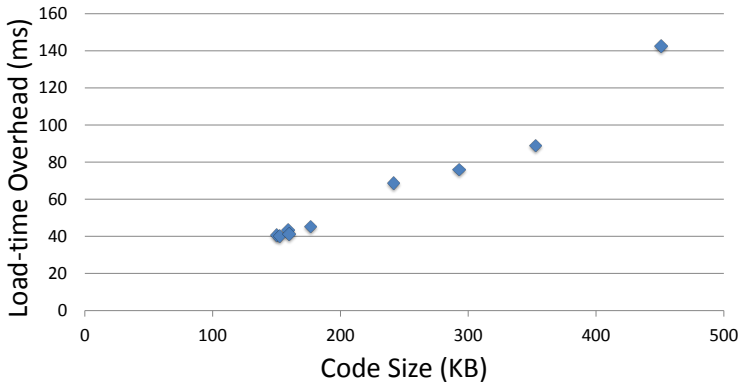

Figure 7.5. O-CFI load-time overhead

We also measured the overhead of the load-time phase, including cluster-based randomization, bounds range updating, and portal creation. These results are reported in Figure 7.5. Unsurprisingly, there is a linear relationship between the size of the code section and the overhead induced at load-time.

7.4.3 Security

To evaluate the security properties of O-CFI under our threat model, we evaluate the probability of chaining gadgets assuming full disclosure of the code segment. That is, we assume that the attacker knows the precise address of every gadget in the rewritten binary. Due to using segmented memory to access the BLT, an attacker is forced to guess the ranges of each gadget. (Section 7.2.2 examines the probability of guessing the address of the BLT itself.)

Under these assumptions, we carry out two experiments. First, we evaluate the statistical likelihood of chaining gadgets without violating branch bounds. Second, we attempt to use post-randomization gadget addresses to manually craft practical attacks against rewritten binaries, and evaluate how O-CFI resists implementation disclosure in practice.

Chaining Gadgets

We use the Mona ROP-generation tool (Corelan Team, 2012) to identify all gadgets in a secured binary after the load-time phase, simulating full disclosure of the code segment. To simulate an attacker's attempts to guess branch ranges, payload chains are created by randomly selecting gadgets from the discovered gadget set. When a selected gadget falls outside the bounds of the previous gadget in the chain, the chain is terminated and its length is considered the payload length for that run. By repeating this process over multiple runs, we estimate the probability of successfully crafting payloads of various lengths.

This procedure estimates the probability of success of disclosure attacks resembling JIT-ROP, where a failure (such as a #BR violation) results in program termination and subsequent re-randomization of code and re-selection of bounds ranges.

BROP-like attacks, on the other hand, exploit the lack of re-randomization when binaries are respawned via the `fork` system call. A crucial advantage of our system against such attacks is the fact that a bounds violation does not automatically crash the program; rather, it invokes the registered #BR exception handler. The exception handler responds to the attack by forcing re-randomization, leaving BROP attacks on the same footing as JIT-ROP attacks.

In our experiments, we ran 100,000 runs of the experimental methodology (for each benchmark), and then measured the mean chance of success at crafting payloads of increasing lengths. This evaluates the security of O-CFI against disclosure attacks. The results are reported in Table 7.6.

Disclosure attacks in our experiments are unsuccessful at chaining more than four gadgets from any of the binaries. Moreover, the chance of chaining even a four-gadget payload is about 0.01% on average—a strong indicator that O-CFI offers powerful probabilistic protection against such attacks.

Table 7.6. Gadget chain lengths across SPEC benchmarks

Gadget Chain Size	**Chance** (%)
2	2.0
3	0.8
4	0.01
5	–

Crafting Practical Attacks

Mona is capable of building practical gadget chains based on a heuristic search. In particular, it can look for the gadgets necessary for attackers to (i) gain execute permissions, (ii) disable DEP, and (iii) allocate a new page with execute permissions.

To evaluate security against a more practical disclosure attack, we leveraged this capability, and tried to build chains using gadgets from both the original and rewritten binaries. For the rewritten binaries, we filtered any chains that resulted in bounds range exceptions.

Mona found partial chains (i.e., chains that incorporate some additional gadgets from dynamically linked modules) for all original binaries, and found full chains for some. However, no complete or partial chains could be found for any of the rewritten binaries. This provides additional evidence of the effectiveness of O-CFI against implementation disclosures.

7.4.4 Portal Efficacy

We also tested the O-CFI's effectiveness at reducing bounds ranges. Table 7.7 shows the factor by which average bounds sizes reduce as the nexus capacity varies from 3 to 18, and when the capacity is left unbounded. The bounds for each case are compared against a baseline binary that has a nexus capacity of zero. Across most binaries, with the exception of `gap`, the bounds sizes decrease roughly linearly as the number of portals increases. When the capacity is left unbounded, the bounds reduce by a factor of almost 308. In other words,

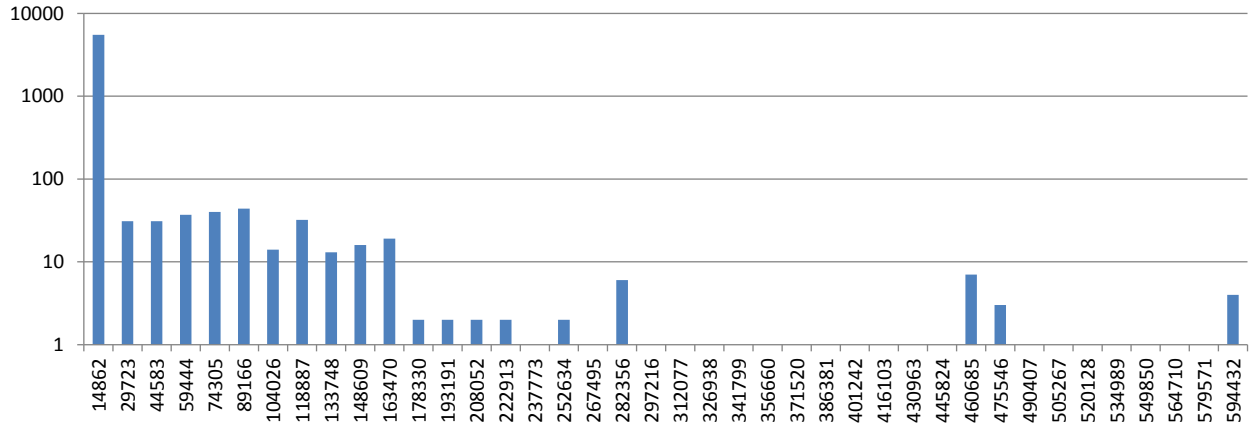

Figure 7.6. Bounds range histogram for a nexus capacity of 12. The vast majority of bounds have span under 15K.

Table 7.7. Bounds range reduction factors with portals

Binary Program	**Nexus Capacity**						
	3	**6**	**9**	**12**	**15**	**18**	∞
`gzip`	3.11	4.34	5.01	5.99	6.96	8.60	288.19
`twolf`	2.70	3.61	4.42	5.51	6.82	7.96	310.67
`vpr`	2.75	4.17	5.54	6.95	9.23	10.68	287.79
`gap`	2.11	2.57	3.05	3.49	3.94	4.37	255.46
`equake`	2.90	5.73	6.93	10.47	12.94	16.94	352.73
`art`	3.33	5.22	6.80	8.60	13.55	16.93	368.97
`mcf`	3.37	6.09	6.86	9.58	12.63	18.94	353.72
`bzip2`	2.87	4.38	6.32	7.43	8.52	13.34	277.88
`parser`	2.87	4.65	5.38	6.05	6.91	9.15	275.49
median	2.89	4.53	5.59	7.12	9.05	11.88	307.88

the average bounds range for a binary with unbounded nexus capacities is only about 0.3% of the range for a binary that does not use portals.

As mentioned earlier, our implementation uses a nexus capacity of 12, which reduces bounds by a factor of about 7, while only marginally affecting code size or runtime performance. Figure 7.6 is a histogram of bounds sizes across all the benchmarks when the capacity is set to 12, with the counts measured on a logarithmic scale. Although there is

a fairly wide variance in ranges, the overwhelming majority of bounds are less than 15,000 bytes in size.

7.4.5 Security against Theoretical Full-Knowledge Attack

In the previous section, we evaluated whether a ROP generation tool (Mona) can construct effective attacks for attackers who have not located the BLT (cf., Section refsec:assumptions). Although we protect the BLT from memory disclosure, we now consider an extraordinarily capable attacker who either (a) discovers all code pages, disassembles their contents, and infers all bounds from full knowledge of the program control flow, or (b) somehow locates and reads the BLT.

To the explore such an attacker's capabilities, we extended and adapted the Frankenstein mutation engine (Mohan and Hamlen, 2012) to search for a ROP chain that implements the `VirtualAlloc` or `VirtualProtect` payloads from Section 7.4.3 without violating any bounds. Frankenstein uses a constraint-solving algorithm to find gadget chains that realize a user-specified goal state. The goal processor state for our payloads was expressed as the stack layout needed for a protection-disabling system API call. Our tool has basic semantic understanding of a subset of x86 instruction sequences, mainly pertaining to their effect on the stack. It leverages this understanding to search for a satisfying sequence of gadgets from a given gadget pool.

When testing against a binary, the gadget pool is initialized to the set of gadgets found by Mona. Finally, for each runtime-randomized layout that the tool is provided, the BLT-permitted range for each indirect branch is added as a constraint.

We used the tool on each of the SPEC2000 binaries, across 100 randomized code layouts each. Our results mirror the practical payload tests—although we found partial chains for the original binaries, we were unable to find any ROP chains that pass the bounds checks in binaries protected by O-CFI.

7.5 Discussion

7.5.1 Branch Range Entropy

A bounds-guessing disclosure attack succeeds when every gadget falls within the range of the indirect branch in the preceding gadget. If a binary contains b indirect branches split across c clusters, and every indirect branch represents a potential gadget, each cluster contains on average b/c gadgets.

Given a randomly chosen initial gadget, the chances of choosing a second gadget that falls within the range of the first is $\frac{b/c}{b}$ or $1/c$. Thus, the chances of creating a payload with n gadgets becomes $(1/c)^{n-1}$. The average value of c across the SPEC binaries is 23. Thus, we find that the chance of creating a bounds-adherent payload falls below 0.01% when $n > 4$. This supports the experimentally obtained values for disclosure attacks in Table 7.6.

Rather than guessing branch ranges, an implementation disclosure attack could try to guess the location of the BLT in at attempt to recover all bounds, so that it may craft a payload in a more guided fashion. As discussed in Section 7.2.2, the probability of doing so falls as low as 1/2,147,483,648 on x86-64 systems. Attackers who beat the odds and manage to locate the BLT still face the daunting challenge of leveraging the leaked information to craft a gadget sequence that (i) is not bounds-violating, and (ii) expresses a meaningful payload. Section 7.4.5 demonstrates that this is quite difficult given the sparsity of control-flows allowed by O-CFI.

7.5.2 Control-flow Obfuscation

Section 7.4.5 evaluates adversaries who use implementation disclosures, binary reverse engineering, and control-flow fingerprinting to infer and recover the complete control-flow graphs of victim programs, and it concludes that such attackers still cannot reliably craft significant-length gadget chains. To further frustrate such efforts, O-CFI could be coupled with

code obfuscation and anti-piracy strategies that hamper effective reverse-engineering. For example, instruction-level diversification (e.g., register allocation and instruction schedule randomization), code duplication, opaque predicates (Majumdar and Thomborson, 2005), control-flow flattening (Wang et al., 2000), and non-readable code pages (Backes et al., 2014) are all supportable by O-CFI.

7.5.3 External Module Support

O-CFI allows secured binaries to call library functions, and fully supports rewriting of libraries. We here discuss this capability in the context of Microsoft Windows libraries, but the approach generalizes to Linux as well.

Calling external modules from secured binaries

The Import Address Table (IAT) of secured binaries is set non-writable throughout program execution, preventing attackers from corrupting it to redirect control-flow to arbitrary locations. As such, calls made to external modules through the IAT need not be constrained by O-CFI, and work correctly unchanged.

Most binaries also access external modules using dynamic linking system calls, such as `LoadLibrary` and `GetProcAddress` on Windows. We use trampolines (Yee et al., 2009) to support such calls. In particular, the intermediary library that performs load-time randomization also intercepts all calls to the OS's dynamic linking API. The interception dynamically loads the requested library, but returns a chunk-aligned pointer to a trampoline within the caller's address space instead of the address of the requested function. When the trampoline is called, it safely transfers control to the library, ensuring compatibility between dynamically linked O-CFI and non-O-CFI binaries.

Securing Libraries

Libraries can be secured almost identically to program main modules. The only difference is that the returns of exported functions need special bounds that permit cross-module control-flows. O-CFI therefore creates a return-portal for each exported function and exports its address. This way, library returns become intra-module, and their bounds can be set to the range of all locally identified targets plus the address of the exported portal.

7.5.4 Approach Limitations and Future Work

Our prototype implementation of O-CFI relies on the static-rewriting phase to identify and protect all branch ranges. As such, it is unable to secure code that is generated dynamically. Consequently, although our technique is compatible with binaries that generate JIT-code, our system cannot protect the JIT-compiled code. Homescu et al. (Homescu et al., 2013) randomize the output of JIT-compilers. We believe that their strategy could be applied to realize O-CFI for JIT code without modifying the JIT compiler.

Our O-CFI prototype implementation is presently incompatible with Windows Component Object Model (COM). COM uses binary reflection to dynamically inform loading modules of the methods a COM library supports. Once the COM library is loaded, calls to its methods are implemented as indirect calls. Because indirect calls in O-CFI binaries are masked to chunk-aligned targets for safety, and because any dynamically loaded COM module has function entry points that are not located at these boundaries, any attempt to jump to one of these functions crashes the binary. Supporting COM requires extending our implementation with a mechanism that dynamically creates (chunk-aligned) trampolines for the functions of loaded COM modules. This is reserved for future work.

A compiler-based implementation of O-CFI has access to a more precise control-flow graph of the target binary than is available to our COTS-compatible, binary-level prototype. Because completely accurate disassembly of x86 code is, in general, impossible (Horspool

and Marovac, 1980; Cohen, 1993), our binary-level prototype is prone to false-negatives when identifying branch targets. (False-positive are harmless, at worst contributing to an increased bounds range for that branch.) Such false-negatives lead to runtime #BR violations for legitimate control-flows. This is a standard limitation of source-free CFI, and is addressable through improved disassembly heuristics like those discussed by Wartell et al. (2014).

7.6 Conclusions

CFI and artificial software diversity are well-established strategies for protecting software against code-reuse attacks, including ROP attacks. Recent advances in offensive security have alarmingly demonstrated how to bypass both: The security relaxations introduced by coarse-grained CFI to achieve acceptable performance are exploitable by skillful control-flow hijacking, and implementation disclosure vulnerabilities can be leveraged to derandomize even fine-grained artificial diversity defenses.

O-CFI combines and extends both CFI and fine-grained diversity to address this dual threat of code-reuse and implementation disclosure attacks. To do so, we reformulate CFI as a bounds-checking problem, and repurpose fine-grained binary code randomization to diversify and conceal the exploitable edges of the protected program's control-flow graph. As a result, O-CFI can protect software even against attackers who have complete read-access to the randomized program code.

Our prototype implementation demonstrates that O-CFI can be effectively applied to protect legacy binaries without source code, and experimental evaluation exhibits performance overheads of just 4.92%. Performance is expected to be even higher on future-generation processors, since our bounds-checking implementation centers around Intel MPX instructions that will be hardware-accelerated on forthcoming Intel-based processors.

CHAPTER 8

RELATED WORK

8.1 Malware Detection

Both the creation and the detection of malware that self-modifies to defeat signature-based detectors are well-studied problems in the literature (Nachenberg, 1997; Ször, 2005). Self-modifying malware has existed at least since the early 1990s and has subsequently become a major obstacle for modern malware protection systems. For example, Sophos Security (2013) reports that over 75% of malware uses self-modifying mechanisms in an attempt to evade detection.

Most self-modifying malware uses encryption or packing as the primary basis for its modifications. The majority of the binary code in such polymorphic malware exists as an encrypted or packed payload, which is unencrypted or unpacked at runtime and executed. Signature-based protection systems typically detect polymorphic malware by identifying distinguishing features in the small unencrypted code stub that decrypts the payload (Kruegel et al., 2005). More recently, metamorphic malware has appeared, which randomly applies binary transformations to its code segment during propagation in order to obfuscate features in the unencrypted portion. An early example is the MetaPHOR system (Walenstein et al., 2006), which has become the basis for many other metamorphic malware propagation systems.

Reversing these obfuscations to obtain reliable feature sets for signature-based detection has been a topic of research for over a decade (Kruegel et al., 2005; Brushi et al., 2007; Walenstein et al., 2006), but case studies have shown that current antivirus detection schemes

remain vulnerable to simple obfuscation attacks until the detector's signature database is updated to respond to the threat (Christodorescu and Jha, 2004).

To our knowledge, all existing self-modifying malware mutates randomly. Our work on expoiting an anti-virus interface (see Chapter 3) therefore differs from past approaches in that it proposes an algorithm for choosing obfuscations that target and defeat specific malware defenses. These obfuscations could be inferred and applied fully automatically in the wild, thereby responding to a signature update without requiring re-propagation by the attacker. We argue that simple signature updates are therefore inadequate to defend against such an attack.

Researchers have proposed alternatives to static feature-based malware detection. Dynamic approaches monitor the execution of a binary and attempt to use behavioral features to make classification decisions. These include monitoring API call sequences (Shankarapani et al., 2011), analysing behavior across different environments for divergence (Kirat et al., 2014), or virtual-machine based methods, such as those used by Jiang et al. (2010). The main drawback of dynamic techniques is their higher overhead compared to feature-based detection (Li et al., 2006; Newsome et al., 2005; Singh et al., 2004).

Machine learning-based techniques extract different features from binaries to train a classifier that can then be used to test whether a binary is malicious. For example, Masud et al. (2008) use binary n-grams from executables, Santos et al. (2013) use opcode relevance and opcode sequence frequencies, while Rieck et al. (2011) use behavioral information and clustering.

8.2 Metamorphic Engines

Metamorphic malware changes the structure of its payload with each generation to evade discovery. Metamorphic engines typically do this using a bottom-up approach: Starting with a disassembler to recover assembly code for the payload, they perform a series of

obfuscation phases, followed by application of an assembler to generate mutated native code. Researchers have proposed criteria that could be used to better categorize and compare different metamorphic engines (Walenstein et al., 2007). Most engines use a combination of the following five phases to obfuscate their payloads (O'Kane et al., 2011): *Garbage insertion* adds unreachable code to the original code. *Code substitution* replaces opcodes with functionally equivalent but structurally different opcodes. *Code insertion* in-lines semantically ineffectual code sequences or harmless computations. *Register swapping* reallocates registers, and *control flow scrambling* adds jumps and reorders function calls.

Borello et al. (2010) developed a generic metamorphic engine that uses variations of the techniques described above, in conjunction with data obfuscation, and showed that it was able to evade detection by antivirus tools. However, researchers have shown that most metamorhic malware currently seen in the wild lack this level of sophistication and can be detected using a combination of heuristics and existing techniques (Canfora et al., 2014; Konstantinou and Wolthusen, 2008; Leder et al., 2009; Chouchane and Lakhotia, 2006). Thus, altough the potential for metamorhic malware to evade feature-based detection exists, it is not currently leveraged to its maximum capability by current-generation malware.

Frankenstein's (see Chapter 4) gadget-based obfuscation is a more principled approach to metamorphism. It both widens the pool of possible mutations for greater diversity and tailors its mutations to local defenses for more targeted attacks. For example, code substitution in a metamorphic engine is typically performed by comparing instruction opcodes against a fixed table of alternative sequences and then randomly choosing one from amongst them. This induces a degree of randomness with respect to generated code sequences, but does not ensure that the generated sequences are vastly different, nor that they do not contain features widely recognized as malicious.

Frankenstein uses a more top-down approach. By starting with a high-level representation of the payload logic and searching benign files for viable gadgets, it implicitly combines all

5 phases described above. This combination gives it the ability to create mutants with a greater diversity than standard bottom-up approaches.

8.3 Program Equivalence

Reasoning about program equivalence arises in connection with translation validators and certifying compilers. A translation validator shows that compiler optimizations are semantics-preserving by proving the semantic equivalence of the original program and its compiler-optimized counterpart. Approaches include instrumenting the compiler (Kanade et al., 2006), verifying a simulation relationship between the two programs (Necula, 2000; Sridhar et al., 2014), and constructing value-graphs of the two programs and proving their syntactic equivalence (Tristan et al., 2011; Tate et al., 2009).

Certifying compilers prove that object code respects the semantics of the higher-level source code whence it was generated. Most certifying compilers do not prove full program equivalence but instead reduce the complexity by considering only a subset of verifiable properties, such as type-safety (Necula and Lee, 1998; Chen et al., 2007). Certifying compilers output object code, type specifications, and code annotations. The annotations and type specifications can then be fed into a certifier which either outputs a proof of correctness or a counterexample that violates type safety.

Frankenstein differs from these related fields in that it does not attempt to provide any formal evidence of semantic equivalence for mutants. That is, although all mutants satisfy the abstract specification whence they were generated, the mutator is under no obligation to provide any evidence of semantic preservation or equivalence. Thus, there is neither validation nor certification. Frankenstein does, however, leverage many theoretical foundations underlying this past research, including pre- and post-conditions for semantic blueprint specification, abstract (symbolic) interpretation for gadget discovery, and abstract machine semantics for gadget analysis and arrangement.

8.4 Superoptimizing Compilers

Superoptimization was first proposed by Massalin (Massalin, 1987), and refers to the transformation of a loop-free code sequence into the most optimal set of assembly-level instructions. Optimality in this context is decided by the speed of the generated sequence, and hence superoptimizing compilers attempt to find the fastest sequence of assembly instructions that are equivalent to the input code. Such compilers use a lookup table populated with parametrized replacement rules to perform their optimizations. The lookup table can be generated manually as is the case with peephole optimizers, or generated automatically based on a training set of binaries (Bansal and Aiken, 2008a). Researchers have also shown the applicability of superoptimization for efficient multi-way branching (Sayle, 2008), or automated binary translation across architectures (Bansal and Aiken, 2008b). Recent research has also evaluated the effectiveness of superoptimization in the Java Virtual Machine (Hume and Watson, 2013) and the LLVM IR (Auler, 2011).

In an abstract sense, metamorphic obfuscation can be viewed as a superoptimization problem where the model for optimality is not the speed of the generated code, but its dissimilarity to previous versions of the code. However, since similarity of programs is a high-dimensionality metric, there is no one unique solution to the obfuscation problem. (In fact, discovering ever more solutions is a goal of most obfuscation.)

Gadget-based obfuscation can also be compared to superoptimization, but with a subtle difference. The model for optimality in this case is the generated sequence's similarity to existing benign code. Additionally our technique does not use a pre-defined set of replacement rules, forgoing them for a top-down approach to finding viable sequences from benign files.

8.5 Binary Rewriting

Static Approaches: Static binary rewriting is a core technology for many applications, such as software fault isolation (Wahbe et al., 1993), static instrumentation, in-lined reference monitoring (Schneider, 2000), and tamper-proofing (Anckaert et al., 2007). There are a large body of rewriting techniques, including PittsField (McCamant and Morrisett, 2006), Google Native Client (Yee et al., 2009), and Diablo (Sutter et al., 2005). Their rewriting techniques typically target the assembly code yielded by a specific compiler with specific compiler options (Wahbe et al., 1993; Small and Seltzer, 1996; Erlingsson and Schneider, 1999; McCamant and Morrisett, 2006; Yee et al., 2009) or type-safe byte-code languages (Chen and Roşu, 2005; Bauer et al., 2005; Aktug and Naliuka, 2007).

However, most static binary rewriters make strong assumptions about target binaries in order to successfully preserve their behavior. In contrast, Reins, STIR and O-CFI are compiler-agnostic, require no relocation or debugging information, and have no reliance on symbol stores. This is important for practical applicability since most COTS legacy binaries lack this information.

Dynamic Approaches: Dynamic binary instrumentation is another approach to binary mutation. Systems such as DynInst (Hollingsworth et al., 1994) and program shepherding (Kiriansky et al., 2002) have the potential to intercept, analyze, and modify instructions at runtime to remove or modify the locations of gadgets, if extended with some form of basic block randomization.

However, as demonstrated by DROP (Chen et al., 2009) and ROPdefender (Davi et al., 2011), these systems tend to exhibit significantly higher overheads. STIR inlines the analysis necessary for basic block randomization, whereas dynamic instrumentation defers much of this analysis to computationally expensive runtime context switches between the application and the VM. For example, DynInst and DynamoRIO (Cheng et al., 2006) exhibit 10–160× overhead and 30% overhead, respectively (Lam et al., 2011; Cheng et al., 2006).

8.6 Control-flow Integrity

Control Flow Integrity (CFI) was first proposed by Abadi et al. (Abadi et al., 2005) in 2005. In its original iteration, CFI constrains the indirect branches in a binary to flow only to one of the exact set of statically identified targets for that branch. Although CFI can secure a binary against both traditional ROP attacks as well as the new wave of implementation-disclosure based attacks, it has not met with widespread adoption in the industry. We believe the two main reasons for this to be (i) difficulty reconstructing an accurate CFG for a binary without access to source code or debug symbols, neither of which are readily available for the large majority of COTS binaries, and (ii) higher overheads compared to randomization based solutions.

Researchers have attempted to address both points in subsequent publications. A number of low-overhead solutions have been proposed which impose very loose integrity checks on program executions. These include ROPecker (Ding et al., 2014), ROPGuard (Fratric, 2012) and kBouncer (Pappas et al., 2013b).

ROPecker and kBouncer use the x86 last branch record (LBR) register set to accomplish their checks. kBouncer, for instance, performs CFI validation on the LBR during any Windows API invocation, and checks to ensure that all returns all are call-preceded—i.e., that the return address points to an instruction immediately following a call instruction. ROPecker creates an offline gadget database which is then compared at runtime with LBR entries to ensure that control-flow is not being redirected to a gadget. ROPGuard also performs CFI validation on Windows API calls. Like kBouncer, it ensures that return addresses are call-preceded. It also validates that the memory word before a return address is the start address of the API function.

CFI for COTS binaries (Zhang and Sekar, 2013) is another proposed coarse-grained CFI solution. As the name implies, this technique can be applied to binaries without access to source code or debug information. It relies on a static disassembly step where all potential

branch (and return) targets are identified, and all indirect branches are instrumented with code that jumps to a CFI validation routine. The validation routine ensures that target and return addresses are either call-preceded, or belong to the set of statically identified candidate targets. CFI for COTS binaries incurs a runtime overhead of 8.54%, lower than traditional CFI, but higher than those typically achievable by randomization-based defenses.

Similar to CFI for COTS binaries, Compact Control Flow Integrity and Randomization (CCFIR) (Zhang et al., 2013) applies coarse-grained CFI to binaries without source code or debug information (but with relocation information). In this technique, legal targets for indirect branches are collected into a separate Springboard section, and indirect branches are only allowed to flow into the Springboard. CCFIR also incorporates some elements of code-randomization—target entries are placed at random locations within the Springboard. Although this confers it an additional degree of security against traditional ROP attacks, disclosure-attacks are able to read the full contents of the Springboard, nullifying its advantages against that class of attacks.

Davi et al. (Davi et al., 2014) test these coarse-grained CFI solutions and show that they fail to adequately secure binaries against ROP attacks.

Forward CFI (Tice et al., 2014) and SafeDispatch (Jang et al., 2014) are two recent compiler-based CFI solutions. Forward CFI protects binaries by inserting validation checks for all forward-edge control flows. It is composed of three similar techniques, all of which make use of a central table to store trusted code pointers. A secured binary is halted when an indirect branch attempts to flow to a location not found in one of these tables. Forward CFI shows promisingly low overheads (between 1-8.7%), but needs binaries to be recompiled with a modified compiler before they can be secured. Furthermore, this solution is only intended to secure forward control-flows, and does not protect against attacks that rely purely on `ret` terminated gadgets.

SafeDispatch protects C++ binaries from *virtual table hijacking* by recompiling binaries with a modifed C++ compiler that instruments all virtual method call sites with runtime

checks that ensure that all method calls jump to valid implementations during execution. Like Forward CFI, this technique offers a low performance overhead (2.1%), but requires binaries to be recompiled in order to secure them. Additionally, SafeDispatch only protects virtual method calls, leaving binaries vulnerable to ROP attacks that rely on modified return addresses on the stack. In its current implementation SafeDispatch is also not compatible with virtual calls to subclass methods defined in dynamically linked libraries, and cannot protect against arbitrary code-pointers in function arguments lists, such as callback pointers.

8.7 Software Fault Isolation

REINS (Chapter 5) is related to SFI, whose works can be divided into (1) source-level approaches, which instrument untrusted code with dynamic security guards at compile-time, (2) binary-level approaches, which secure untrusted code at a purely binary level, and (3) system-level approaches, which secure the software at system call level.

Source-level Approaches: Most SFI implementations target source code and therefore insert security guard instructions at compile-time. Examples include StackGuard (Cowan et al., 1998), DFI (Castro et al., 2006), WIT (Akritidis et al., 2008), BGI (Castro et al., 2009), G-Free (Onarlioglu et al., 2010), and CFL (Bletsch et al., 2011). Source-level approaches differ significantly from the problem of securing COTS native code because a compiler typically has full control over the structure of the binary it generates, its pointer representations, and its implementation of computed jumps. In contrast, SFI systems for legacy native code cannot statically distinguish code pointers from data, recover control-flow or data-flow graphs reliably, or detect all instruction aliasing. Enforcing SFI without this information introduces many challenges.

The primary disadvantage of source-level approaches is their reliance on the support of a cooperating code-producer, who must (re)compile the untrusted or insecure code using a special compiler. Such cooperation is not a reasonable expectation for many classes of

untrusted code, which are distributed as raw native code produced by arbitrary compilers, and that target mainstream system APIs such as Microsoft Windows or Linux.

Binary-level Approaches: Binary-level approaches require less compiler cooperation. They can be further divided into those that operate dynamically and those that operate statically:

Dynamic binary approaches use dynamic binary translation—Vx32 (Ford and Cox, 2008), Strata (Scott and Davidson, 2002) or Libdetox (Payer and Gross, 2011) for example—program shepherding (Kiriansky et al., 2002; Bania, 2011), or safe loading like TRuE (Payer et al., 2012) to dynamically copy and instrument untrusted code into a sandbox at runtime. Any flows that attempt to escape the sandbox recursively re-trigger the copying process, keeping all untrusted, reachable code within the sandbox.

In contrast, static binary approaches in-line guard instructions into untrusted binary code statically before the code executes, and do not perform any code generation or translation at runtime. The only SFI systems other than Reins that target legacy, untyped, native code binaries to our knowledge are CFI (Abadi et al., 2009), XFI (Erlingsson et al., 2006), PittSFIeld (McCamant and Morrisett, 2006), NaCl (Yee et al., 2009), and SecondWrite (Smithson et al., 2010). CFI/XFI achieves reliable disassembly by consulting PDB files, which contain debugging information. The debugging information reveals important structural and typing information from the application source code without disclosing the source code text; however, PDB files are only produced by Microsoft compilers, and most code-producers do not disclose them to the public. This significantly limits the domain of binaries to which CFI/XFI is applicable. PittSFIeld and NaCl are similarly limited—PittSFIeld only supports `gcc`-produced assembly code and NaCl requires untrusted code to be (re)compiled by their tool chain.

SecondWrite tackles the problem of rewriting COTS binaries without debug or relocation metadata, but it does not support formal machine-verification, has not yet been applied to

realize complete fault isolation, and is not yet mature enough to rewrite full-scale COTS applications (O'Sullivan et al., 2011).

Dynamic vs. static approaches have historically suffered a compatibility vs. performance trade-off. That is, the dynamic approaches can currently handle a much larger class of binaries than the static ones, including large-scale COTS applications, but at the expense of significant runtime overheads, for example a 70% slowdown in Strata (Scott and Davidson, 2002). In addition, dynamic SFI systems are difficult to formally verify, and cannot be deployed on architectures that prohibit runtime code generation or that lack the hardware-level VM support that is often necessary to achieve reasonable performance.

In contrast, static approaches offer much lower overheads and formal, machine-checkable proofs of safety, but currently support only a very restricted set of binary programs that do not include most COTS applications. Therefore, Reins is the first purely static binary SFI system capable of supporting nearly arbitrary, large-scale, COTS Windows applications produced by mainstream compilers, including those that contain computed jumps, dynamic linking, and event-driven OS callbacks.

System-level Approaches: There are also many system-level approaches, such as Janus (Wagner, 1999), SysTrace (Provos, 2003), and Ostia (Garfinkel et al., 2004). These use system call interposition to enforce policies that prevent abuse of the system API.

Unlike binary rewriting approaches, system-level approaches are transparent to the binary code. However, they cannot block attacks of one module upon another within the same address space, they cannot be deployed as a service (because the full implementation must reside on the client machine), and they can introduce compatibility problems, such as incorrect replication of OS semantics (Garfinkel, 2003).

8.8 Security Through Artificial Diversity

Cohen was first to describe software diversity as a defense mechanism (Cohen, 1993). Forrest subsequently demonstrated stack-layout randomization as a defense against stack smashing (Forrest et al., 1997). The subsequent work on artificial diversity is extensive, and is explained in-depth by systematization works (e.g., (Larsen et al., 2014)).

Randomization has emerged as a popular strategy to increase the diversity of software and has been widely instantiated in existing works, such as ASLR (PaX Team, 2003; Bhatkar et al., 2003, 2005), instruction set randomization (ISR) (Barrantes et al., 2003), data randomization (Cowan et al., 2003; Cadar et al., 2008; Bhatkar and Sekar, 2008), OS interface randomization (Chew and Song, 2002), and multi-variant systems (Cox et al., 2006; Bruschi et al., 2007; Salamat et al., 2009).

Address Space Layout Randomization: ASLR is a practical technique that has been adopted by many modern OSes such as Windows and Linux. The goal of ASLR is to obscure the location of code and data objects that are resident in memory, including the addresses of the program stack, heap, and shared library code (PaX Team, 2003; Bhatkar et al., 2003; Xu et al., 2003; Bhatkar et al., 2005). ASLR is currently implemented through modifying the OS kernel (PaX Team, 2003), system loader (Xu et al., 2003), and application source or binary code (Bhatkar et al., 2003, 2005). However, all of these approaches require source code information (e.g., debug symbols or relocation data) in order to randomize the instruction addresses of most main modules. This motivates our work, which extends instruction address randomization to the majority of legacy main modules that lack such information.

In addition, existing source-agnostic ASLR approaches are limited to randomizations of relatively low granularity, leaving them vulnerable to derandomization attacks that can succeed even when the address space is large (Roglia et al., 2009). For example, ASLR can relocate and reorder some sections as wholes, but not the relative positions of the binary

features within the sections. This leaves them vulnerable to attacks that reliably infer the relative positions of vulnerable code features irrespective of the size of the address space. In contrast, STIR (see Chapter 6) randomizes the relative positions of such features, defeating such attacks.

Instruction Set Randomization: ISR is an approach to prevent code injection attacks by randomizing the underlying system instructions (Barrantes et al., 2003; Kc et al., 2003). In this approach, instructions are encrypted with a set of random keys and then decrypted before being fetched and executed by the CPU. ISR is effective for preventing code injections but cannot prevent ROP attacks. The technique is also hard to deploy in practice, requiring encryption of any supported software.

Data Randomization: As a dual to ISR, program data can also be encrypted and decrypted. PointGuard (Cowan et al., 2003) encrypts all pointers while they reside in memory and decrypts them only before they are loaded into CPU registers. Recent work has presented a new data randomization technique that provides probabilistic protection against memory exploits by XORing data with random masks (Cadar et al., 2008; Bhatkar and Sekar, 2008). DSR can help to prevent ROP attacks by decrypting attacker-injected code pointers to random addresses. However, data randomization requires recompilation of programs, which hinders its practicality.

OS Interface Randomization: System call mappings, global library entry points, and stack placement can all be randomized to mitigate buffer overflow attacks by increasing the heterogeneity of computer systems (Chew and Song, 2002). Similarly, RandSys (Jiang et al., 2007) combines ASLR and ISR to randomize the system service interface when loading a program, and de-randomizes the instrumented interface for the correct execution at runtime. Similar to ISR, OS interface randomization cannot prevent ROP attacks in which all the attack code is drawn from the existing content of the victim address space.

Multi-variant Systems: Our work is also related to N-variant systems (Cox et al., 2006; Bruschi et al., 2007; Salamat et al., 2009), which likewise leverage diversification to improve security. N-variant is an architectural framework that employs a set of automatically diversified variants to execute a common task. Any divergence among the outputs raises an alarm and can hence detect the attack. DieHard (Berger and Zorn, 2006) is a simplified multi-variant framework that uses heap object randomization to make the variants generate different outputs in case of error or attack. Exterminator (Novark et al., 2007) extends this idea to derive runtime patches and automatically fix program bugs. Multi-variant systems frustrate ROP attacks by forcing the attacker to simultaneously subvert all the running variants, but require source code information in order to successfully apply comprehensive, semantics-preserving diversification of large applications.

Randomization and implementation-aware attacks: The work most closely related to O-CFI (see Chapter 7) is the concurrently developed Oxymoron system (Backes and Nürnberger, 2014). Oxymoron uses a pure code-randomization approach to resist JIT-ROP attacks. By generating code that does not contain any direct references to other code pages, it impedes the JIT-ROP attack step that recursively infers new code page addresses by disassembling leaked code pages. Instead of direct code references, inter-page references use an indirection table similar to our BLT, whose base address is stored in an x86 segment register to protect it from accidental disclosure.

However, Oxymoron allows raw code addresses to flow into registers, into the heap, and onto the stack (e.g., as return addresses pushed by call instructions). As a result, it is vulnerable to buffer over-read attacks that disclose the stack contents (Strackx et al., 2009), which can hold a wealth of raw return addresses and function pointers, each of which potentially reveals a 4K page of gadgets for attackers to abuse. In contrast, we pessimistically assume that many or all code addresses will eventually leak to attackers (because they are

stored in too many places). Instead, O-CFI conceals the control-flow policy graph, whose details can be much more easily protected from disclosure.

Giuffrida et al. (2012) present a comprehensive compiler-based software diversification approach that allows live re-randomization. With this approach, programs are distributed as pre-linked LLVM bitcode and native machine code. To randomize a running process, the host compiles a new program variant using the bitcode which includes meta-data to allow state migration from the old to the new process. A dedicated randomization manager uses this meta-data to transfer the state from the old process to its newly created counterpart. The resulting overheads depend on the re-randomization frequency. Snow et al. (2013) report that JIT-ROP attacks can run in as little as 2.3 seconds. Re-randomizing every two seconds, however, adds an overhead of about 20%.

8.9 ROP Defenses

In addition to diversification defenses, there are other techniques that specifically target ROP attacks. DROP (Chen et al., 2009) instruments program binary code and monitors the frequency of return instructions, which tend to rise during ROP attacks that rely heavily upon stack pointers to hijack control-flows. While DROP has been shown to be effective for ROP shell code detection, it suffers up to 5× performance overhead on average.

ROPdefender (Davi et al., 2011) is another binary instrumentation-based technique, which duplicates return addresses on a shadow stack and further evaluates each return instruction during program execution to detect mismatched calls and returns. ROPdefender is quite effective, and unlike DROP it only introduces 2× performance overhead. Other defenses, such as return-less kernels (Li et al., 2010) and gadget-less binaries (Onarlioglu et al., 2010), eliminate return instructions during compilation.

There are also two compiler-based approaches to defeating ROP attacks. G-Free (Onarlioglu et al., 2010) removes gadgets from program binaries at compile time by eliminating all

unaligned free-branch instructions and protecting the remaining aligned free-branch instructions. The other compiler approach generates return-less code by removing the `ret` opcode to prevent gadget formation (Li et al., 2010).

Most recently, IPR (Pappas et al., 2012) and ILR (Hiser et al., 2012) have been proposed to alleviate the problem of ROP attacks. IPR uses in-place code randomization and instruction replacement to eliminate gadgets. Since its transformations are strictly in-place, it can only eliminate gadgets for which a semantics-preserving, size-invariant code transformation can be found. On average, 77% of gadgets meet this requirement, and can therefore be eliminated or broken by the technique. However, with large binaries that contain more than 100K gadgets, this is not enough to ensure secure execution. Also, IPR is difficult to deploy since each randomized application copy must be separately distributed.

ILR adopts a highly dynamic approach that first statically randomizes most instruction addresses and then dynamically guides control-flows through the randomized layout at runtime using a VM equipped with a fall-through map. However, like IPR, it is unable to move all instructions (mainly due to indirect branches), and therefore consistently preserves the locations of some gadgets. In addition, its dependence on a VM inevitably leads to higher performance overheads than purely static approaches.

STIR and O-CFI are orthogonal and complementary to these techniques in that they can be transparently applied to legacy code without code-producer cooperation.

CHAPTER 9

CONCLUSIONS

This dissertation presented a two-fold method to advance software defenses to better combat future threats. First, it drew attention to the inability of current defenses to scale up to next-generation malware obfuscations by presenting two obfuscation techniques in Part I that bypass today's malware detection tools. Chapter 3 programmatically exploits the Microsoft Windows antivirus interface as a side-channel to infer a machine learning model that accurately predicts defense classification decisions. This model is then used to make directed changes to a malware to ensure it is classified as benign by the tool. Frankenstein, described in Chapter 4, constructs malware entirely out of code sequences harvested from benign binaries—ensuring that a feature-based classifier cannot reliably detect it without also flagging the benign donor binaries as malicious.

Second, Part II presented three new defenses that can be used to secure binaries against both current and future threats. All defenses are capable of protecting binaries without access to source code or debug symbols, and are thus compatible with both legacy and commercial-off-the-shelf (COTS) applications. Chapter 5 introduced Reins, which rewrites binaries using an approach similar to PittSFIeld to provide Software Fault Isolation (SFI). This is leveraged to in-line guard instructions that constrain runtime behavior by monitoring API calls. Reins comes with a policy language and policy synthesis engine that allows end-users to specify custom API-based security policies for rewritten binaries. Reins also comes with a separate verifier consisting of just 1500 lines of OCaml code, allowing Reins itself to be placed outside of the trusted computing base. All this is achiveable with an overhead of only 2.5% for rewritten binaries.

Chapter 6 presented STIR, a mechanism to secure both Windows and Linux binaries against ROP attacks. STIR introduces the concept of fine-grained randomization, rewriting binaries so that the order of their basic-blocks is randomized on each execution. Binaries secured with STIR only incur an overhead of 1.4%, making it extremely efficient in practice.

Opaque CFI (O-CFI) was presented in Chapter 7. O-CFI uses a combination of fine-grained randomization and control-flow integrity to bound where indirect branches are allowed to jump. The allowed range for each branch is minimized using a combination of target clustering and nexuses, providing a high degree of probabilistic protection against implementation-aware attacks such as JIT-ROP and BROP. By implementing the bounds as ranges rather than discrete values, O-CFI is able to guard branches efficiently, inducing less than 5% overhead overall.

Computer security has never been a fixed battlefield. It evolves constantly as defenders and attackers constantly vie to outdo each other. However, even a dynamic environment such as this is marked by periods of significant change—where one side must adapt quickly or risk major losses. We believe that we are currently in the midst of such a period. For confirmation, one need only consider the number of security breaches occurring in companies worldwide on a weekly basis, or the recurrent unearthing of new activist-targeting malware, or the ongoing revelations about state-funded weaponized software.

Better fundamental techniques for safeguarding software are prerequisites for solving this crisis. Defensive tools must move away from their current reliance on structural information for their classification decisions and embrace a more semantics-aware approach that promises to be more resilient to next-generation malware threats and zero-days alike.

REFERENCES

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti (2005). Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353.

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti (2009). Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions Information and System Security 13*(1), 4:1–4:40.

Akritidis, P., C. Cadar, C. Raiciu, M. Costa, and M. Castro (2008). Preventing Memory Error Exploits with WIT. In *Proceedings of the IEEE Symposium Security and Privacy (S&P)*, pp. 263–277.

Aktug, I. and K. Naliuka (2007). ConSpec – A Formal Language for Policy Specification. In *Proceedings of the Workshop on Run Time Enforcement for Mobile and Distributed Systems*, pp. 45–58.

Anckaert, B., M. Jakubowski, R. Venkatesan, and K. D. Bosschere (2007). Run-Time Randomization to Mitigate Tampering. In *Proceedings of the International Conference on Advances in Information and Computer Security*, pp. 153–168.

Andersen, S. (2004). Part 3: Memory Protection Technologies. In V. Abella (Ed.), *Changes in Functionality in Windows XP Service Pack 2*. Microsoft TechNet. http://technet.microsoft.com/en-us/library/bb457155.aspx.

Auler, R. (2011). Superoptimization for LLVM IR. Google Summer of Code 2011.

Backes, M., T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny (2014). You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the ACM Conference Computer and Communications Security CCS*, pp. 1342–1353.

Backes, M. and S. Nürnberger (2014). Oxymoron - Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the USENIX Security Symposium*.

Bania, P. (2011). Securing The Kernel via Static Binary Rewriting and Program Shepherding. http://piotrbania.com/all/articles/pbania-securing-the-kernel2011.pdf.

Bansal, S. and A. Aiken (2008a). Binary Translation using Peephole Superoptimizers. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pp. 177–192.

Bansal, S. and A. Aiken (2008b). Binary translation using peephole superoptimizers. In *OSDI*, Volume 8, pp. 177–192.

Barrantes, E. G., D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi (2003). Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 281–289.

Bauer, L., J. Ligatti, and D. Walker (2005). Composing Security Policies with Polymer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 305–314.

Berger, E. D. and B. G. Zorn (2006). DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 158–168.

Bhatkar, S., D. C. DuVarney, and R. Sekar (2003). Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the USENIX Security Symposium*, pp. 105–120.

Bhatkar, S. and R. Sekar (2008). Data Space Randomization. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 1–22.

Bhatkar, S., R. Sekar, and D. C. DuVarney (2005). Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the USENIX Security Symposium*, pp. 255–270.

Bittau, A., A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh (2014). Hacking Blind. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, S&P '14.

Bletsch, T., X. Jiang, and V. Freeh (2011). Mitigating Code-reuse Attacks with Control-flow Locking. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 353–362.

Bletsch, T., X. Jiang, V. Freeh, and Z. Liang (2011). Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pp. 30–40.

Borello, J.-M., É. Filiol, and L. Mé (2010). From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques. *Journal in computer virology 6*(3), 277–287.

Boser, B. E., I. M. Guyon, and V. N. Vapnik (1992). A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the ACM Workshop on Computational Learning Theory*, pp. 144–152.

Bruschi, D., L. Cavallaro, and A. Lanzi (2007). Diversified Process Replicæ for Defeating Memory Error Exploits. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference*, pp. 434–441.

Brushi, D., L. Martignoni, and M. Monga (2007). Code Normalization for Self-Mutating Malware. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, Volume 5 (2), pp. 46–54.

Buchanan, E., R. Roemer, H. Shacham, and S. Savage (2008). When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 27–38.

Cadar, C., P. Akritidis, M. Costa, J.-P. Martin, and M. Castro (2008). Data Randomization. Technical Report MSR-TR-2008-120, Microsoft Research.

Cadar, C., V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler (2006). EXE: Automatically Generating Inputs of Death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 322–335.

Canfora, G., F. Mercaldo, C. A. Visaggio, and P. Di Notte (2014). Metamorphic malware detection using code metrics. *Information Security Journal: A Global Perspective 23*(3), 57–67.

Carlini, N. and D. Wagner (2014). ROP is still dangerous: Breaking modern defenses. In *Proceedings of the USENIX Security Symposium*.

Castro, M., M. Costa, and T. Harris (2006). Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 147–160.

Castro, M., M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black (2009). Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the ACM Symposium Operating Systems Principles*, pp. 45–58.

Checkoway, S., L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy (2010). Return-oriented Programming Without Returns. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 559–572.

Chen, F. and G. Roşu (2005). Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, pp. 546–550.

Chen, P., H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie (2009). DROP: Detecting Return-Oriented Programming Malicious Code. In *Proceedings of the International Conference on Information Systems Security*, pp. 163–177.

Chen, P. M. and B. D. Noble (2001). When Virtual is Better than Real. In *Proceedings of the Workshop Hot Topics in Operating Systems*, pp. 133–138.

Chen, Y., L. Ge, B. Hua, Z. Li, and C. Liu (2007). Design of a Certifying Compiler Supporting Proof of Program Safety. In *Proceedings of the Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 127–138.

Cheng, W., Q. Zhao, B. Yu, and S. Hiroshige (2006). TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications*, pp. 749–754.

Chew, M. and D. Song (2002). Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University.

Chouchane, M. R. and A. Lakhotia (2006). Using engine signature to detect metamorphic malware. In *Proceedings of the ACM workshop on Recurring Malcode*, pp. 73–78. ACM.

Christodorescu, M. and S. Jha (2004). Testing Malware Detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 34–44.

Cohen, F. (1993). Operating System Protection Through Program Evolution. *Computers and Security 12*(6), 565–584.

Corelan Team (2012). Mona. http://redmine.corelan.be/projects/mona.

Cortes, C. and V. Vapnik (1995). Support-vector networks. *Machine Learning 20*(3), 273–297.

Cowan, C., S. Beattie, J. Johansen, and P. Wagle (2003). PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pp. 91–104.

Cowan, C., C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang (1998). StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*.

Cox, B., D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser (2006). N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the USENIX Security Symposium*.

Davi, L., A.-R. Sadeghi, D. Lehmann, and F. Monrose (2014). Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the USENIX Security Symposium*.

Davi, L., A.-R. Sadeghi, and M. Winandy (2011). ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *Proceedings of the ACM Symposium on Computer and Communications Security (CCS)*, pp. 40–51.

DeMott, J. (2014). Bypassing EMET 4.1. http://labs.bromium.com/2014/02/24/bypassing-emet-4-1/.

Ding, X., R. Deng, Z. Zhou, Y. Cheng, and M. Yu (2014). ROPecker: A generic and practical approach for defending against rop attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

Drepper, U. (2005). Security Enhancements in Red Hat Enterprise Linux (beside SELinux), version 1.6, section 5: Position Independent Executables. http://www.akkadia.org/drepper/nonselsec.pdf.

Erdélyi, G. (2008). IDAPython: User Scripting for a Complex Application. Bachelor's thesis, EVTEK University of Applied Sciences.

Erlingsson, Ú., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006). XFI: Software Guards for System Address Spaces. In *Proceedings of the Symposium Operating Systems Design and Implementation*, pp. 75–88.

Erlingsson, Ú. and F. B. Schneider (1999). SASI enforcement of security policies: A retrospective. In *Proceedings of the ACM Workshop on New security paradigms (NSPW)*, pp. 87–95.

Evans, C. (2013). Exploiting 64-bit Linux like a boss. http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html.

Fog, A. (2009). *Calling Conventions for different C++ compilers and operating systems*. Copenhagen University College of Engineering.

Fog, A. (2014). Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, amd and via cpus. http://www.agner.org/optimize/instruction_tables.pdf.

Fogla, P., M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee (2006). Polymorphic Blending Attacks. In *Proceedings of the USENIX Security Symposium*, Volume 15.

Ford, B. and R. Cox (2008). Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference*, pp. 293–306.

Forrest, S., A. Somayaji, and D. Ackley (1997). Building Diverse Computer Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '97, pp. 67–72.

Fratric, I. (2012). Runtime prevention of return-oriented programming attacks. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf.

Garfinkel, T. (2003). Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Garfinkel, T., B. Pfaff, and M. Rosenblum (2004). Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the Network and Distributed Systems Security Symposium*.

Giuffrida, C., A. Kuijsten, and A. S. Tanenbaum (2012). Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the USENIX Security Symposium*, SEC '12, pp. 475–490.

Godefroid, P., M. Y. Levin, and D. Molnar (2008). Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Göktaş, E., E. Athanasopoulos, H. Bos, and G. Portokalidis (2014). Out of control: Overcoming control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

Göktaş, E., E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the USENIX Security Symposium*.

Goodrich, M. T. and R. Tamassia (2005). *Data Structures and Algorithms in Java* (4th ed.). New York: Wiley.

Grace, M., Y. Zhou, Q. Zhang, S. Zou, and X. Jiang (2010). RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 281–294.

Guilherme, V. and P. Szor (2012). The Day of the Golden Jackal - The Next Tale in the Stuxnet Files: Duqu.

Hamlen, K. W., V. Mohan, and R. Wartell (2010). Reining in Windows API Abuses with In-lined Reference Monitors. Technical Report UTDCS-18-10, U. Texas at Dallas.

Hamlen, K. W., G. Morrisett, and F. B. Schneider (2006). Computability Classes for Enforcement Mechanisms. *ACM Transactions Programming Languages and Systems (TOPLAS) 28*(1), 175–205.

Hex-Rays (2012). The IDA Pro Disassembler and Debugger. www.hex-rays.com/idapro.

Hiser, J. D., A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson (2012). ILR: Where'd My Gadgets Go? In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 571–585.

Hoglund, G. and J. Butler (2006). *Rootkits: Subverting the Windows Kernel*, Chapter 4: The Age-Old Art of Hooking, pp. 73–74. Pearson Education, Inc.

Hollingsworth, J. K., B. P. Miller, and J. Cargille (1994). Dynamic Program Instrumentation for Scalable Performance Tools. In *Proceedings of the Scalable High Performance Computing Conference*, pp. 841–850.

Homescu, A., S. Brunthaler, P. Larsen, and M. Franz (2013). librando: Transparent Code Randomization for Just-in-Time Compilers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 993–1004.

Homescu, A., M. Stewart, P. Larsen, S. Brunthaler, and M. Franz (2012). Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, WOOT '12, pp. 64–76.

Horspool, R. N. and N. Marovac (1980). An approach to the problem of detranslation of computer programs. *Comput. J. 23*(3), 223–229.

Hume, T. and D. Watson (2013). Is superoptimization viable for VM instruction sets? *Software: Practice and Experience*.

Intel (2013). Introduction to Intel Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

Jang, D., Z. Tatlock, and S. Lerner (2014). SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

Jiang, X., H. J. Wang, D. Xu, and Y.-M. Wang (2007). RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems*, pp. 209–218.

Jiang, X., X. Wang, and D. Xu (2010). Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction. *ACM Transactions on Information and System Security (TISSEC) 13*(2), 12.

Joly, N. (2013). Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php.

Kanade, A., A. Sanyal, and U. Khedker (2006). A PVS based framework for validating compiler optimizations. In *Proceedings of the IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 108–117.

Kc, G. S., A. D. Keromytis, and V. Prevelakis (2003). Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 272–280.

Kim, H.-A. and B. Karp (2004). Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the USENIX Security Symposium*, Volume 13, pp. 271–286.

Kirat, D., G. Vigna, and C. Kruegel (2014). Barecloud: bare-metal analysis-based evasive malware detection. In *Proceedings of the USENIX Security Symposium*, pp. 287–301.

Kiriansky, V., D. Bruening, and S. P. Amarasinghe (2002). Secure Execution via Program Shepherding. In *Proceedings of the USENIX Security Symposium*, pp. 191–206.

Kolter, J. Z. and M. A. Maloof (2004). Learning to Detect Malicious Executables in the Wild. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 470–478.

Konstantinou, E. and S. Wolthusen (2008). Metamorphic virus: Analysis and detection. *Royal Holloway University of London 15*.

Kreibich, C. and J. Crowcroft (2004). Honeycomb: Creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review 34*(1), 51–56.

Kruegel, C., E. Kirda, D. Mutz, W. Robertson, and G. Vigna (2005). Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID)*, pp. 207–226.

Lam, M. O., J. K. Hollingsworth, and G. Stewart (2013). Dynamic floating-point cancellation detection. *Parallel Computing 39*(3), 146–155.

Lam, M. O., J. K. Hollingsworth, and G. W. Stewart (2011). Dynamic Floating-Point Cancellation Detection. In *Proceedings of the International Workshop on High-performance Infrastructure for Scalable Tools*.

Larochelle, D. and D. Evans (2001). Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the USENIX Security Symposium*.

Larsen, P., A. Homescu, S. Brunthaler, and M. Franz (2014). SoK: Automated software diversity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

Leder, F., B. Steinbock, and P. Martini (2009). Classification and detection of metamorphic malware using value set analysis. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pp. 39–46. IEEE.

Lee, D. (2012). Flame: Massive Cyber-Attack Discovered, Researchers Say. *BBC News*.

Li, J., Z. Wang, X. Jiang, M. Grace, and S. Bahram (2010). Defeating Return-oriented Rootkits with "Return-less" Kernels. In *Proceedings of the European Conference on Computer Systems*, pp. 195–208.

Li, Z., M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez (2006). Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 32–47.

Ligatti, J., L. Bauer, and D. Walker (2009). Run-Time Enforcement of Nonsafety Policies. *ACM Transactions Information and System Security 12*(3), 19:1–19:41.

Lyda, R. and J. Hamrock (2007). Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security & Privacy 5*(2), 40–45.

Majumdar, A. and C. Thomborson (2005). Securing mobile agents control flow using opaque predicates. In *Proceedings of the International Conference on Knowledge-based Intelligent Information and Engineering Systems*, Volume 3, pp. 1065–1071.

Massalin, H. (1987). Superoptimizer: a look at the smallest program. In *ACM SIGPLAN Notices*, Volume 22, pp. 122–126. IEEE Computer Society Press.

Masud, M. M., L. Khan, and B. M. Thuraisingham (2008). A Scalable Multi-level Feature Extraction Technique to Detect Malicious Executables. *Information System Frontiers 10*(1), 33–35.

McCamant, S. and G. Morrisett (2006). Evaluating SFI for a CISC Architecture. In *Proceedings of the USENIX Security Symposium*, pp. 209–224.

Microsoft (2014). Enhanced Mitigation Experience Toolkit. https://www.microsoft.com/emet.

Microsoft Corporation (2005). Using Hotpatching Technology to Reduce Servicing Reboots. http://technet.microsoft.com/en-us/library/cc787843.aspx.

Microsoft Developer Network (MSDN) Digital Library (2009). http://msdn.microsoft.com/en-us/library/ms537369(VS.85).aspx.

Mitchell, T. M. (1997). *Machine Learning*. New York: McGraw-Hill.

Mohan, V. and K. W. Hamlen (2012). Frankenstein: Stitching Malware from Benign Binaries. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, Bellevue, Washington, pp. 77–84.

Nachenberg, C. (1997). Computer Virus-antivirus Coevolution. *Communications of the ACM 40*(1), 47–51.

Necula, G. C. (1997). Proof-carrying Code. In *Proceedings of the ACM Principles of Programming Languages*, pp. 106–119.

Necula, G. C. (2000). Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 83–94.

Necula, G. C. and P. Lee (1998). The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 333–344.

Newsome, J., B. Karp, and D. X. Song (2005). Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 226–241.

Novark, G., E. D. Berger, and B. G. Zorn (2007). Exterminator: Automatically Correcting Memory Errors with High Probability. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 1–11.

O'Kane, P., S. Sezer, and K. McLaughlin (2011). Obfuscation: The hidden malware. *IEEE Security & Privacy 9*(5), 41–47.

Onarlioglu, K., L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda (2010). G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 49–58.

Oracle Corporation (2010). Position-Independent Code. In *Linker and Libraries Guide*. http://docs.oracle.com/cd/E19082-01/819-0690/chapter4-29405/index.html.

O'Sullivan, P., K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis (2011). Retrofitting Security in COTS Software With Binary Rewriting. In *Proceedings of the International Information Security Conference*, pp. 154–172.

Pappas, V., M. Polychronakis, and A. D. Keromytis (2012). Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 601–615.

Pappas, V., M. Polychronakis, and A. D. Keromytis (2013a). Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the USENIX Security Symposium*, SEC'13, Berkeley, CA, USA, pp. 447–462. USENIX Association.

Pappas, V., M. Polychronakis, and A. D. Keromytis (2013b). Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the USENIX Security Symposium*, Washington, D.C., pp. 447–462. USENIX.

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

Payer, M. and T. R. Gross (2011). Fine-grained User-space Security through Virtualization. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference Virtual Execution Environments*, pp. 157–168.

Payer, M., T. Hartmann, and T. R. Gross (2012). Safe Loading – A Foundation for Secure Execution of Untrusted Programs. In *Proceedings of the IEEE Symposium Security and Privacy (S&P)*, pp. 18–32.

Provos, N. (2003). Improving Host Security with System Call Policies. In *Proceedings of the USENIX Security Symposium*.

Quinlan, J. R. (2003). *C4.5: Programs for Machine Learning* (5th ed.). San Francisco, CA: Morgan Kaufmann.

Rajagopalan, M., M. A. Hiltunen, T. Jim, and R. D. Schlichting (2006). System Call Monitoring Using Authenticated System Calls. *IEEE Transactions Dependable and Secure Computing 3*(3), 216–229.

Rieck, K., P. Trinius, C. Willems, and T. Holz (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security 19*(4), 639–668.

Roemer, R., E. Buchanan, H. Shacham, and S. Savage (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC) 15*(1), 2:1–2:34.

Roglia, G. F., L. Martignoni, R. Paleari, and D. Bruschi (2009). Surgically Returning to Randomized lib(c). In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 60–69.

Salamat, B., T. Jackson, A. Gal, and M. Franz (2009). Orchestra: Intrusion Detection using Parallel Execution and Monitoring of Program Variants in User-space. In *Proceedings of the ACM European Conference on Computer Systems*, pp. 33–46.

Salwan, J. (2012). ROPgadget. http://shell-storm.org/project/ROPgadget.

Sanger, D. E. (2012). Obama Order Sped Up Wave of Cyberattacks Against Iran. *The New York Times*.

Santos, I., F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas (2013). Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences 231*, 64–82.

Sayle, R. A. (2008). A superoptimizer analysis of multiway branch code generation. In *GCC Developers' Summit*, Volume 103.

Schneider, F. B. (2000). Enforceable Security Policies. *ACM Transactions Information and Systems Security 3*(1), 30–50.

Schultz, M. G., E. Eskin, E. Zadok, and S. J. Stolfo (2001). Data Mining Methods for Detection of New Malicious Executables. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 38.

Schwartz, E. J., T. Avgerinos, and D. Brumley (2011). Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*.

Scott, K. and J. Davidson (2002). Safe Virtual Execution Using Software Dynamic Translation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 209–218.

Seibert, J., H. Okhravi, and E. Söderström (2014). Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the ACM Conference on Computer and Communications Security CCS*, pp. 54–65.

Serna, F. J. (2012a). CVE-2012-0769, the case of the perfect info leak. Google Security Team. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

Serna, F. J. (2012b). The info leak era on software exploitation. In *Black Hat USA*.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 552–561.

Shacham, H., M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh (2004). On the Effectiveness of Address-Space Randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 298–307.

Shankarapani, M. K., S. Ramamoorthy, R. S. Movva, and S. Mukkamala (2011). Malware detection using assembly and api call sequences. *Journal in computer virology 7*(2), 107–119.

Singh, S., C. Estan, G. Varghese, and S. Savage (2004). Automated Worm Fingerprinting. In *Proceedings of the Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, Volume 6, pp. 45–60.

Small, C. and M. I. Seltzer (1996). A Comparison of OS Extension Technologies. In *Proceedings of the USENIX Annual Technical Conference*, pp. 41–54.

Smithson, M., K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua (2010). Binary Rewriting without Relocation Information. Technical report, U. Maryland.

Snow, K. Z., F. Monrose, L. V. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi (2013). Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, S&P '13, pp. 574–588.

Solar Designer (1997). "return-to-libc" attack. Bugtraq.

Sophos Security (2013). Security Threat Report 2013. http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf.

Sotirov, A. (2007). Heap Feng Shui in JavaScript. Accessed 14 July 2014. https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.

Sridhar, M., R. Wartell, and K. W. Hamlen (2014). Hippocratic Binary Instrumentation: First Do No Harm. *Science of Computer Programming (SCP), Special Issue on Invariant Generation 93*(B), 110–124.

Strackx, R., Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter (2009). Breaking the Memory Secrecy Assumption. In *Proceedings of the Second European Workshop on System Security*, New York, NY, USA, pp. 1–8. ACM.

Sutter, B. D., B. D. Bus, and K. D. Bosschere (2005). Link-time Binary Rewriting Techniques for Program Compaction. *ACM Transactions Programming Languages and Systems (TOPLAS) 27*(5), 882–945.

Ször, P. (2005). *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

Tate, R., M. Stepp, Z. Tatlock, and S. Lerner (2009). Equality Saturation: A New Approach to Optimization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 264–276.

Tice, C., T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike (2014). Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the USENIX Security Symposium*.

Tristan, J.-B., P. Govereau, and G. Morrisett (2011). Evaluating Value-graph Translation Validation for LLVM. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 295–305.

van de Ven, A. (2004). New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. Whitepaper WHP0006US, Red Hat. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf.

VX Heavens (2009). http://vx.netlux.org.

Wagner, D. A. (1999). Janus: An Approach for Confinement of Untrusted Applications. Master's thesis, U. California at Berkeley.

Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham (1993). Efficient Software-based Fault Isolation. In *Proceedings of the ACM Symposium Operating Systems Principles*, pp. 203–216.

Walenstein, A., R. Mathur, M. R. Chouchane, and A. Lakhotia (2006). Normalizing Metamorphic Malware Using Term Rewriting. In *Proceedings of the IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, pp. 75–84.

Walenstein, A., R. Mathur, M. R. Chouchane, and A. Lakhotia (2007). The design space of metamorphic malware. In *Proceedings of the International Conference on i-Warfare & Security (ICIW)*, pp. 241–248.

Wang, C., J. Hill, J. Knight, and J. Davidson (2000). Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia Charlottesville.

Wang, K., J. J. Parekh, and S. J. Stolfo (2006). Anagram: A content anomaly detector resistant to mimicry attack. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID)*, pp. 226–248.

Wang, K. and S. J. Stolfo (2004). Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the International Conference on Recent Advances in Intrusion Detection (RAID)*, pp. 203–222.

Wartell, R., V. Mohan, K. W. Hamlen, and Z. Lin (2012a). Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 157–168.

Wartell, R., V. Mohan, K. W. Hamlen, and Z. Lin (2012b). Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, pp. 299–308.

Wartell, R., Y. Zhou, K. W. Hamlen, and M. Kantarcioglu (2014). Shingled Graph Disassembly: Finding the Undecidable Path. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pp. 273–285.

Wartell, R., Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham (2011). Differentiating Code from Data in x86 Binaries. In *Proceedings of the European Conference Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, Volume 3, pp. 522–536.

Witten, I. H. and E. Frank (2005). *Data Mining: Practical Machine Learning Tools and Techniques* (2nd ed.). San Francisco, CA: Morgan Kaufmann.

Wu, Z., S. Gianvecchio, M. Xie, and H. Wang (2010). Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the ACM Conference on Computer and Communications Security (CSS)*, pp. 536–546.

Xu, J., Z. Kalbarczyk, and R. K. Iyer (2003). Transparent Runtime Randomization for Security. In *Proceedings of the International Symposium on Reliable Distributed Systems*, pp. 260–269.

Yee, B., D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2009). Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the IEEE Symposium Security and Privacy (S&P)*, pp. 79–93.

Zhang, C., T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou (2013). Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, SP '13, Washington, DC, USA, pp. 559–573. IEEE Computer Society.

Zhang, M. and R. Sekar (2013). Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX Security Symposium*, SEC '13, pp. 337–352.

Zhao, Z. and G.-J. Ahn (2013). Using Instruction Sequence Abstraction for Shellcode Detection and Attribution. In *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, pp. 323–331.

VITA

Vishwath Mohan was born in Bangalore, India in the November of 1984 and spent his childhood playing every computer game he could get his hands on. This transitioned into a curiosity about the process of making games themselves, which introduced him to the world of programming.

Many years and a few thousand unfinished game development projects later, he found he was far more motivated to test new ideas out than to take proven ones to completion (second only to continue playing every computer game he could find). This mindset put him on a road that led to pursuing a Masters degree in Computer Science at the University of Texas at Dallas. His interest at the time was artificial intelligence, but attending a course on Language Based Security, taught by professor Dr. Kevin Hamlen, introduced him to the complex and fascinating world of system security.

That proved to be a turning point in his studies, convincing him to pursue a Ph.D in Computer Science under Dr. Hamlen learning about the arcane and sometimes logic-defying intricacies of native code security.

After completing his Ph.D, Vishwath will begin working at Google, where he hopes to use his security expertise to make the online world a safer place.