SECURING BINARY PROGRAMS WITHOUT PERFECT DISASSEMBLY

by

Erick Bauman

APPROVED BY SUPERVISORY COMMITTEE:

Zhiqiang Lin, Chair

Kevin W. Hamlen, Co-Chair

Murat Kantarcioglu

Bhavani Thuraisingham
To my parents
and my girlfriend,
who have been ready for me to
finish this for a while now.
SECURING BINARY PROGRAMS WITHOUT PERFECT DISASSEMBLY

by

ERICK BAUMAN, BA, 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 2019
ACKNOWLEDGMENTS

First I would like to thank my advisor Dr. Zhiqiang Lin, who first took me in at UT Dallas, introduced me to the exciting field of computer security, and taught me how research works, how to do security research, and how to write papers.

I would also like to thank my co-advisor Dr. Kevin Hamlen, who has been helpful and supportive in my research, always ready with useful suggestions when I have gotten stuck on a difficult problem, and willing to chat about all sorts of technical and interesting topics.

I also would like to thank fellow students Huibo Wang and Raul Quinonez, who helped and supported me in my research and provided friendship and advice. I also thank all the members of Dr. Lin’s and Dr. Hamlen’s research labs. I enjoyed working with all of you.

Many, many thanks to Rhonda Walls, who has helped countless times with scheduling, hotels, conferences, and general advice. She is truly invaluable and I appreciate all her hard work behind the scenes.

I want to thank my aunts, uncles, cousins, and grandparents, Sandy and Marilyn Bauman, who have always been loving and supportive. Paw, I wish you could have been here for this.

I want to thank the best parents I could ask for, Tom Bauman and Kathy Tourtellotte. Thank you for your love, support and for encouraging me to question and investigate things.

Last but not least, I want to thank my girlfriend, Audrey Marquis, who has remained ever supportive by my side through the ups and downs of this degree. I love you.

This research was supported in part by AFOSR awards FA9550-14-1-0119 and FA9550-14-1-0173; NSF awards 1453011, 1513704, 1564112, and 1629951; and ONR awards N00014-14-1-0030 and N00014-17-1-2995. Any opinions, recommendations, or conclusions expressed are those of the author and not necessarily of the AFOSR, NSF, or ONR.

November 2019
This dissertation introduces several approaches to improving the security, performance, and
generality of binary software without source code. These source-free approaches have the ad-
vantage of being applicable to closed-source software products, which constitute the majority
of software today. Many such products suffer from security vulnerabilities, performance is-
sues in certain deployment contexts, or insufficient generality to address specialized consumer
needs. Since such software is typically distributed in purely binary form, it is difficult to im-
prove or repurpose its code via alternative approaches that analyze and modify source code.
Unfortunately, binary code is much more difficult to analyze and modify than source code, due in part to the general undecidability of perfectly disassembling raw bytes to instruction
sequences. Implementing critical fixes and improvements to mission-critical legacy software,
or software from vendors unwilling or unable to implement functionality or security patches,
therefore demands new methodologies for modifying binary code without the assistance of
sources.

Three systems are presented that address several of these challenges. The first system, SGX-
Elide, increases the confidentiality of code in SGX binaries by encrypting SGX enclave
contents without knowing the contents of the encrypted functions. SGXElide leverages a
whitelist of all essential functions to encrypt all the bytes in an enclave other than those in
essential functions for runtime enclave decryption. The second system, MULTIVERSE, introduces the concept of superset disassembly, a disassembly algorithm that elegantly sidesteps the problem of obtaining accurate disassembly by obtaining a superset of all possible instruction sequences. This allows for the construction of a static binary rewriting framework that does not rely on any heuristics for binary rewriting. The third system, MINIVERSE, builds on the ideas of MULTIVERSE to provide binary rewriting for dynamically generated code using superset disassembly. Finally, this dissertation provides an argument that superset disassembly is unsuitable for runtime re-randomization, and why its apparent benefits are actually counterproductive for re-randomization.
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** .......................................................... v  
**ABSTRACT** ................................................................. vi  
**LIST OF FIGURES** ........................................................... xi  
**LIST OF TABLES** ............................................................. xii  
**CHAPTER 1  INTRODUCTION** ................................................. 1  
1.1 Self-Decrypting SGX Enclaves with **SGXELIDE** .................. 2  
1.2 Heuristic-free Binary Rewriting with **MULTIVERSE** .......... 3  
1.3 Hardening Dynamically Generated Code with **MINIVERSE** .... 3  
1.4 Challenges with Dynamic Re-Randomization ...................... 4  
**CHAPTER 2  SGXELIDE** ..................................................... 6  
2.1 Background and Motivation ............................................. 9  
2.1.1 Intel SGX ............................................................ 9  
2.1.2 The Default Protection Model of SGX ........................... 11  
2.2 Overview ................................................................. 12  
2.2.1 Challenges ......................................................... 12  
2.2.2 Key Insights and Solutions ...................................... 14  
2.2.3 **SGXELIDE** Overview ........................................ 16  
2.2.4 How to Develop Secret Enclave Code ......................... 17  
2.3 Detailed Design .......................................................... 18  
2.3.1 Whitelist Generation ............................................ 18  
2.3.2 Sanitized Enclave Generation ................................... 19  
2.3.3 Runtime Secret Restoration .................................... 19  
2.4 Implementation .......................................................... 22  
2.5 Evaluation ............................................................... 24  
2.5.1 Experiment Setup ............................................... 24  
2.5.2 Experimental Result ............................................. 27  
2.6 Discussions .............................................................. 28  
2.7 Conclusion ............................................................... 31
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Overview of Our SGXELIDE Framework</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Runtime operation of SGXELIDE, showing how the enclave communicates with a server</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Overhead of running the SGXELIDE protected benchmark with remote data</td>
<td>31</td>
</tr>
<tr>
<td>2.4</td>
<td>Overhead of running the SGXELIDE protected benchmark with local data</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>A contrived working example that covers major challenges in x86 COTS binary rewriting</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>An Overview of MULTIVERSE</td>
<td>43</td>
</tr>
<tr>
<td>3.3</td>
<td>An illustration of our disassembly strategy</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>A mapping lookup example for a rewritten binary dynamically linked with our rewritten libc</td>
<td>48</td>
</tr>
<tr>
<td>3.5</td>
<td>Percent runtime overhead for each of the reported benchmarks</td>
<td>57</td>
</tr>
<tr>
<td>3.6</td>
<td>Runtime overhead (times) for the benchmarks with instruction counting instrumentation</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>Percent runtime overhead for the benchmarks instrumented with a shadow stack</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>An overview of MINIVERSE</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Function signatures for mmap and mprotect</td>
<td>76</td>
</tr>
<tr>
<td>4.3</td>
<td>Overhead of MINIVERSE on Lua benchmarks</td>
<td>86</td>
</tr>
</tbody>
</table>
LIST OF TABLES

2.1 Ported benchmarks divided into Untrusted (UC) and Trusted (TC) Components 25
2.2 Sanitization/restoration execution time (ms) with remote/local data . . . . . . . 28
3.1 Statistics of MULTIVERSE rewritten binaries and libraries . . . . . . . . . . . . . . . 54
4.1 Runtime performance of LuaJIT with and without MINIVERSE . . . . . . . . . . . . . . 85
7.1 Comparison w/ existing x86 binary rewriters . . . . . . . . . . . . . . . . . . . . . . . . 101
CHAPTER 1
INTRODUCTION

The use of closed-source software in industry is widespread, preventing users of such software from improving it. While this may be acceptable during the time the software is supported by the developers, software that is no longer supported for any reason is essentially frozen in time. Any new attacks against it will not be patched, and any performance issues will remain forever. Obviously the ideal response to this is to switch to different software that is supported or open source, but the reality is that legacy software in large organizations is often so deeply integrated that the cost of replacing it is too high. In this case, users of closed-source software need a way to modify binary code.

The field of binary rewriting has been around for a long time, and progressively more sophisticated techniques have been used to analyze and modify binary code. However, one particular problem starts at the very start of the process: disassembly of the bytes in a binary into assembly instructions. This at first would appear straightforward, as each byte sequence uniquely encodes a specific instruction, but some instruction set architectures, including the wildly popular x86 and x86-64 architectures, introduce several issues. Instructions are variable-length and may start at any byte offset. This means that starting disassembly from different offsets results in different instruction sequences, and the presence of indirect jump instructions means that instructions may potentially jump to any offset. Unfortunately, obtaining accurate disassembly is undecidable; it is impossible to determine an accurate disassembly for every possible program.

Thesis Statement. This dissertation argues the need for effective binary rewriting and presents a framework for understanding the challenges that are faced with analyzing and rewriting x86 binaries. It provides several techniques to deal with each of these challenges, including superset disassembly, which addresses the significant challenge of x86 disassembly.
It then explores a use-case of these approaches by hardening runtime-generated code, and discusses their limits for runtime re-randomization.

1.1 Self-Decrypting SGX Enclaves with SgxElide

Chapter 2 of this dissertation introduces SgxElide, which performs a simple form of binary rewriting without the need to disassemble any code. Instead, its intent is to statically encrypt the contents of SGX enclaves and redact the original code bytes by overwriting them with constant bytes. SGX enclaves are secure execution environments isolated from the OS and other user code, and therefore an attacker can learn nothing about the contents of an enclave (barring vulnerabilities). One exception to this is the binary code of an enclave, which is exposed prior to initializing the enclave. Therefore, SgxElide encrypts all non-essential functions statically. Later, at runtime, the encrypted code is disassembled and restored over the redacted bytes. Ultimately, SgxElide improves the security of SGX enclaves by providing code confidentiality. It avoids the need to perform function boundary identification (another hard problem in binary rewriting) with the assistance of debug info, by using function symbol name information within binaries.

SgxElide manages to avoid many of the challenges in binary rewriting because it does not actually alter the original code; when the enclave contents are encrypted, the original code bytes are sanitized by zeroing them out, but the alignment of the non-sanitized code (the code needed for decryption) is at the same position as it was before encryption. At runtime, when the code bytes are decrypted, the code bytes are restored to exactly the position they were prior to encryption. Therefore, by not altering the original bytes, SgxElide can decrypt the original code without having to address challenges such as correct disassembly.
1.2 Heuristic-free Binary Rewriting with Multiverse

While perfect disassembly is impossible, many have used heuristics to obtain “very good” disassembly. However, edge cases are where these approaches fail. Chapter 3 introduces the concept of superset disassembly, a disassembly algorithm that sidesteps the issue of obtaining perfect disassembly by computing a superset of all possible assembly instructions from a byte sequence. Essentially, instead of attempting to determine which interpretation of the bytes is correct, we assume every interpretation of the bytes may be correct.

Another important concept in binary rewriting is preserving the original control flow of a binary. We successfully address this by mediating indirect control flow transfers (iCFTs) to take target addresses to the original code and translate them to their rewritten equivalents. This avoids many common challenges in binary rewriting, such as pointer identification. While the idea of rewriting iCFTs alone is not novel, our formalization of the concepts and our conscious decision to avoid heuristics in the process results in a new approach to this problem.

Using these ideas as a foundation, we then built a general-purpose binary rewriting framework called Multiverse, which uses superset disassembly alongside several other techniques to provide the first binary rewriter framework that does not use heuristics in its rewriting step. The result of this is that the behavior of arbitrary code in the original binary is preserved after rewriting.

1.3 Hardening Dynamically Generated Code with Miniverse

After Multiverse, we wished to apply the concepts we learned to another use case, specifically to a security application. Current approaches to harden and defend dynamically generated code are somewhat limited, and many security frameworks explicitly disclaim that the framework does not work on dynamically generated code. Considering the flexibility and
robustness of superset disassembly and the other techniques from MULTIVERSE, we decided to tackle this challenge.

In chapter 4, this dissertation describes MINIVERSE, a specialized binary rewriting framework for dynamically generated code. Utilizing superset disassembly similar to MULTIVERSE, it rewrites code that has been generated at runtime to enforce a security policy, prohibiting any runtime generated code from being exploited. One useful example use case of MINIVERSE is in JIT compilers, which generate assembly at runtime in order to speed up execution of a scripting language. As JIT compilers are used for JavaScript in all modern browsers, insidious attacks have been discovered to exploit the dynamic nature of JIT compilers. MINIVERSE can be integrated into a JIT compiler without any need to know how the code is being generated, and it automatically intercepts attempts to generate code and rewrites it.

1.4 Challenges with Dynamic Re-Randomization

MINIVERSE’s defenses for dynamically generated code equate to a form of control-flow integrity (CFI) or Software Fault Isolation (SFI) by constraining control flow in the program. A related but very different approach to securing software is in randomization, a very common implementation of which is Address Space Layout Randomization (ASLR). This makes attacks more difficult by hiding the locations of code that may be exploited. However, ASLR only randomizes addresses when code is first loaded, allowing a clever attacker to find out where code has been loaded at runtime. Runtime re-randomization, which means that code is repeatedly randomized as a program runs, has already been a topic of research for some time. However, the results for randomization of binaries still has room for improvement, which appears to be an opportunity for another exciting application of superset disassembly.

Unfortunately, as this dissertation discusses in Chapter 5, we encountered an interesting limitation to superset disassembly’s usefulness. We originally planned to insert runtime re-randomization into binaries, repeatedly moving the code bytes to new locations during
runtime. This theoretically is a good security mechanism, as it obscures the location of
code bytes that an attacker may attempt to use in code reuse attacks. Ultimately, superset
disassembly’s concept of obtaining all possible instruction sequences is itself a weakness when
using it to translate original code addresses to randomized rewritten addresses, as an attacker
can use the translation mechanism in order to bypass the randomization. This shows that,
while a powerful tool, superset disassembly must be applied with consideration.
CHAPTER 2

SGXELIDE

Most software today is delivered in the form of binary executables, which are normally executed on a platform out of the control of software developers. Secrets in the form of binary code are not directly visible to most users. However, curious or malicious attackers can still reverse engineer the executable to uncover secrets (e.g., particular algorithms, data structures, or values of variables) inside the software, since they control the entire computing stack, including the operating system, libraries, runtime environment (e.g., debuggers), and the application code of the software, and they can easily disassemble the binary, or monitor (e.g., trace, or debug) the execution of the binary. Keeping code and data secret in a compiled executable has long been a challenge. Many interesting and important applications require such a capability.

One area in need of the ability to hide code secrets is in defending against cheating in computer games (Bauman and Lin, 2016). The popularity of multiplayer games, including highly competitive games that offer substantial monetary rewards in tournaments, or games with economies that allow for the trading of virtual items for real money, gives players incentive to cheat the system for personal gain. Many techniques have been developed to cheat in games (Hoglund and McGraw, 2007). For instance, attackers can reverse-engineer the game code to find exploits, search game memory for certain game values that can be changed to benefit the attacker, or use automated tools that can perform better than a human (such as improved reflexes).

In addition, vendors of software, including games, have strong incentives to prevent users from running unlicensed copies of their software. Over the past few decades, vendors have

---

1This chapter contains material previously published as: Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. SGXElide: Enabling Enclave Code Secrecy via Self-Modification. In Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO’18). The lead author, Erick Bauman, conducted the majority of the research for the paper, including implementation, part of the evaluation, and most of the writing.
developed a wide range of Digital Rights Management (DRM) (Rosenblatt et al., 2001) techniques to protect the unauthorized redistribution of their software. Historically, there has been no way to trust a remote machine, and therefore the only way to enable DRM has been through various obfuscation strategies (Collberg and Thomborson, 2002; Majumdar et al., 2006), including control flow flatting, signal-based control flow hiding (e.g., Popov et al., 2007), code encryption and packing, etc. Therefore, to prevent illegal copying DRM must remain confidential, because once the DRM is reverse-engineered, attackers can circumvent the copy protection and upload executables with the DRM removed onto the Internet, or rebuild new software using the stolen algorithm. This results in an escalating arms race between application crackers and developers, with each side trying to outwit the other.

In the past several years, we have also witnessed the growth of cloud computing, in which cloud providers host a platform for selling computing power to third parties. But unfortunately, the customer must trust the cloud provider with their confidential code or data. This prevents certain industries, such as hospitals, from being able to use the (public) cloud, due to strict privacy legislation. In addition, companies with software that may contain valuable trade secrets cannot use such platforms lest a malicious cloud provider extract and sell their secrets. This is also a concern for software that runs on an end user’s machine, as secrets can also be extracted from more traditional software.

Therefore, there is a need to provide true confidentiality guarantees. This issue has only recently begun to be solved, and one important step forward is Intel’s Software Guard eXtensions (SGX) (Hoekstra et al., 2013), which allow code to execute in a secure “enclave” opaque to any other code. This allows applications to hide their secrets from all other software on a machine, including privileged code such as the OS or hypervisor. As was shown by Haven (Baumann et al., 2015), this can be extended to protect arbitrary legacy code from being spied on by a cloud provider. SGX is already being deployed in consumer-grade hardware starting with Intel’s Skylake processor. This means that eventually a high percentage
of users may own SGX-enabled hardware, allowing software developers to take advantage of its features. This may enable much stronger copy protection and trade secrets by hiding crucial code and data in SGX enclaves.

However, despite the substantial benefits provided by SGX, there remains an unfortunate omission in what SGX currently protects, as the SGX SDK Guide explicitly states: “The enclave file can be disassembled, so the algorithms used by the enclave developer will not remain secret” [Intel Corporation (2016a)]. Therefore, while SGX does protect code and data integrity, it only protects the confidentiality of data, while code confidentiality is not assured. SGX does not protect code and data until after enclave initialization. This, combined with the fact that enclave code must be signed and cannot be modified before being loaded, means that there is no straightforward way to hide the actual enclave code from would-be attackers.

To address this shortcoming of SGX’s programming model, we introduce SgxElide, a framework that leverages SGX to provide complete confidentiality and integrity for both code and data. The key idea is to treat program code as data, and dynamically decrypt the secret code or retrieve the secret code from the trusted remote parties after an enclave is initialized. SgxElide can be almost transparently integrated into any enclave code, providing a mechanism to securely decrypt or deliver the secret code with the assistance from a trusted remote party controlled by the developer. We have implemented SgxElide atop Intel’s Linux SGX SDK [Intel Corporation (2016b)], and our evaluation with a number of programs shows that SgxElide can be used to protect code secrecy for applications of interest without introducing any additional runtime overhead for the enclave code.

In short, we make the following contributions:

- We present a systematic protection model for software that uses SGX, and show that the default SGX programming model lacks an oblivious mechanism to provide code confidentiality.
• We design a framework for providing *code confidentiality* to SGX applications, thereby providing both code and data integrity and confidentiality.

• We have implemented our framework on the Linux platform and demonstrate that it is easy to add to existing SGX projects and effective at protecting the confidentiality of code, without introducing any runtime overhead to the applications.

2.1 Background and Motivation

2.1.1 Intel SGX

At a high level, SGX allows an application or part of an application to run inside a secure *enclave*, which is an isolated execution environment. SGX hardware, as a part of the CPU, protects enclaves against malicious software, including the operating system, hypervisor, or even low-level firmware code (e.g., SMM) from compromising its confidentiality and integrity. At a low level, Intel SGX is an extension to the x86 instruction set architecture (ISA).

**SGX Enclaves.** The first step in creating an enclave is to call the instruction `ECREATE`. This allocates memory inside the Enclave Page Cache (EPC) to hold enclave code and data (Hoekstra et al., 2013). EPC memory is encrypted by the Memory Encryption Engine (MEE) and decrypted when accessed by enclave code. Enclave contents are added with the `EADD` instruction, which copies a 4KB page from ordinary memory into an EPC page (McKeen et al., 2013).

However, SGX must also calculate the enclave’s *measurement*, a cryptographic hash that is used for remote attestation. This is done with the `EEXTEND` instruction. Every time `EEXTEND` is executed, it measures 256 bytes, and therefore it must be executed 16 times to cover a full page (McKeen et al., 2013).

The enclave cannot be entered until it has been initialized with the `EINIT` instruction. However, unless the enclave’s measurement matches the original measurement calculated by
the enclave’s creator, the hardware will not initialize it. The creator of the enclave provides
the measurement inside the SIGSTRUCT data structure, which the creator signs with their
private key and provides along with the enclave.

**Attestation.** Once an enclave has been created, a signed report can be obtained from the
processor by using the new EREPORT instruction. This report provides a guarantee to enclaves
on the same machine as the enclave. This allows enclaves to perform local attestation and
to establish secure channels between each other.

A special platform enclave called the quoting enclave is used to support remote attest-
ation—authenticating an enclave to a remote server. This enclave signs reports with a
device-specific key, producing a structure called a quote. The device-specific key is only
visible to the quoting enclave, and therefore a remote server can trust quotes signed with
this key. The actual key itself is embedded in the processor by Intel and therefore Intel is
the root of trust ([Intel Corporation 2014; Hoekstra et al. 2013]).

After remote attestation is complete, a server is assured that the enclave it is talking to
matches its declared measurement, and a secure channel has been established between them,
allowing the server to provide secrets to the enclave.

**Sealing.** In order to save an enclave’s state, SGX provides a sealing mechanism to save and
restore data from disk. Using hardware-derived keys unique to every processor and enclave,
secrets can be decrypted and restored ([Intel Corporation 2014; Hoekstra et al. 2013]).

**Bridge Functions.** At the assembly level, enclaves have only a single entry point. In
addition, data must be copied to and from an enclave since enclave memory cannot be read
from outside the enclave. Therefore, Intel provides a mechanism to create bridge functions
in their SGX SDK. At the enclave entry point there is a function that dispatches calls into
the enclave (ecalls) to the correct enclave functions, which allows an enclave to enforce a set
of functions (specified by the developer) that can be called from untrusted code and vice
versa (i.e., ocalls, untrusted functions that can be called from trusted code). Functions are
identified by indexing into a table of function pointers. The bridge functions (both ecalls and ocalls) automatically handle copying the contents of buffers across the enclave boundary and allow for the illusion that enclave functions are being directly called from outside the enclave (Intel Corporation 2014; Hoekstra et al. 2013).

2.1.2 The Default Protection Model of SGX

While SGX has been presented as an environment guaranteeing both confidentiality and integrity within enclaves, it is possible to analyze and inspect all enclave code; the code loaded into an enclave can be disassembled prior to enclave initialization (McKeen et al. 2013). Therefore, despite the claims of SGX, there is no framework for code confidentiality by default.

- **Data Integrity.** Since the code in an enclave can be attested, it is possible to ensure all code that modifies data is trusted. Data in an enclave can be encrypted when it is sent to a remote server. This, combined with a message authentication code (MAC), ensures that an attacker can never modify data without it being detected. This is important in preventing software tampering, because it prevents an attacker from directly changing any data.

- **Code Integrity.** Similarly to data integrity, the integrity of the code in an enclave can be assured via remote attestation. The original vendor of software can ensure that the code that the client is running is identical to the code provided by the vendor. This also defends against any tampering of program code.

- **Data Confidentiality.** While an enclave is running, all data inside it is invisible to all other software running on the machine, including the OS or hypervisor. Since data sent to and from enclaves can be encrypted, an enclave can communicate with a trusted server without any chance of eavesdropping. This helps enable DRM and protection of secrets by making it impossible to perform runtime analysis on an enclave.
• **Code Confidentiality.** In contrast to runtime data, the code (and static data) of an enclave is in plain text and can be disassembled and statically analyzed. There are no current defenses in place to prevent this. This means that it is possible to reverse-engineer algorithms or DRM mechanisms by analyzing the enclave binary. Our framework aims to provide a default solution to cover this crucial aspect of enclave security.

2.2 Overview

In this section, we provide an overview of our SgxElide framework. We first describe the challenges we will be facing in §2.2.1, then present our key insights and solutions in §2.2.2. Next, we describe how SgxElide works in §2.2.3, and finally present how a developer would use our framework in §2.2.4.

2.2.1 Challenges

Our objective is to ensure the confidentiality of code inside an enclave. We focus on compiled C/C++ code (instead of scripting languages) running inside enclaves, meaning enclaves must be self-modifying in order to run code that was not present when it was first initialized. Therefore, any enclave contents that we want to keep confidential must not be visible in the initial enclave binary, and said contents must be decrypted and restored at runtime.

While adding code encryption and decryption to enclaves seems obvious and straightforward in theory, there are subtle complications and challenges in practice, particularly from the constraints of enclave code execution.

- **Enclaves must be signed and cannot be modified until after they are initialized.** We therefore must have self-modifying enclaves. In addition, since enclaves cannot be modified until after loading, we must set the code pages as writable before signing or after loading. However, it turns out that dynamically setting page permissions for an enclave at runtime is not permitted by the hardware.
• **The entire enclave cannot be encrypted.** An intuitive and simple solution may be to simply encrypt the entire enclave, leaving only a tiny bootstrap function to decrypt the rest of the code after the enclave is started. Unfortunately, calls in and out of the enclave must pass through bridge functions created during compilation by the SGX SDK. If we blindly encrypt these bridge functions, control will never reach our decryption function, as the application will crash when trying to run the encrypted entry point. In order to make our solution work with the official SDK, we must be careful not to disrupt any of the bridge functions.

• **The decryption key or plaintext secret code must not be in the enclave.** There is no way for an enclave to securely decrypt itself; any key contained in the binary can be extracted by an attacker. While the SGX platform could be used to generate a key, each piece of SGX hardware has its own hardware key. This is perfect for an enclave on a specific machine to seal its own secrets, but insufficient for deploying a general application. Since there is no way to hide the key in the enclave, the enclave must retrieve the key or secret code from a trusted remote source.

    An important result of this is that a remote enclave on an untrusted machine is inherently vulnerable to denial-of-service attacks, because it requires a trusted remote server. If an attacker prevents the remote server from communicating with the enclave, it will not function. Since this is unavoidable, we can only seek to minimize the amount of communication required with the trusted server.

• **The toolchain should have minimal changes.** In order to make our solution usable, it must be compatible with arbitrary SGX applications, requiring no major restructuring of an application in order to work.
2.2.2 Key Insights and Solutions

Our key insights. One straightforward approach might be to encrypt the entire enclave and unpack it at runtime, similar to a binary unpacker (Markus F.X.J. Oberhumer and Reiser., 2018). However, this leads to more challenges: it is not possible to create and initialize one enclave from inside another (i.e. pages added before initialization should not be in the EPC), and each enclave is designed with bridge functions designed to enter and exit the enclave, complicating unpacking.

Therefore, instead of taking inspiration from packers, we instead try a different approach: redacting confidential enclave functions and then restoring them later. This can be thought of as “sanitizing” the enclave. Next, we have to consider the best way to specify the functions to be redacted, since we are now focused on redacting certain functions in an enclave. There are two ways to approach this: with a blacklist or a whitelist.

• Blacklist. A blacklist-based approach involves specifying which functions should be redacted. One way this could be implemented is for the developer to annotate each function that they want to keep confidential. While this can keep the number of functions that are redacted to a minimum, it puts the burden of securing the enclave on the developer. It requires the developer to decide which functions are secret and mark them accordingly, leading to potential mistakes.
• **Whitelist.** A more general approach is using a whitelist. Instead of having to specify which functions to redact, we instead specify which functions we should *not* redact. This information is applicable to any enclave, and does not need to be specified by a developer because it consists of only the functions required to restore redacted functions, meaning we can redact all other functions.

**Initial Approach.** We explored several solutions before our final design. Since the objective is to provide confidentiality for code considered sensitive, we at first considered it important to encrypt only secret functions, since there may be many functions in an enclave that are safe being public and need no encryption. Therefore, we originally used a blacklist approach, requiring secret functions to be annotated. Those functions were then placed in a special text section separate from the public functions, and that section was then encrypted. At runtime, the secret functions were restored within this special memory region. This meant that the only self-modifying code was the secret text section (a smaller attack surface) at the expense of requiring developers to determine which functions were confidential. We eventually decided that greater transparency was more important, and therefore we decided not to use this approach.

**Our Solution.** We therefore use a whitelist. Since we would like our framework to be as transparent as possible to a developer, it is tempting to simply place the code for restoring the encrypted functions in the enclave’s initialization code, thus requiring no changes in how an enclave is initialized. However, there are a number of things that could go wrong when trying to restore the encrypted data. Therefore, we designed **SgxElide** so that a developer must call a single function to restore all encrypted functions. This allows them to handle errors in a way unique to their application. We used the following strategies to design **SgxElide**.
• **Sign a dummy enclave and restore all secrets after initializing.** We can create and sign a dummy enclave with most functions redacted. All functions for restoring redacted code are untouched. The dummy enclave restores the redacted code at runtime.

• **Encrypt all nonessential functions.** The enclave requires the functions required to retrieve and decrypt secret code from a server. By whitelisting all necessary functions and leaving them intact in the dummy enclave, the code will be able to initialize successfully. All other functions can be redacted.

• **Use remote attestation.** An SGX enclave can guarantee to a server it was unmodified before it was initialized via remote attestation. This means that the decryption key or code can be held remotely and is only ever provided over a secure channel to an enclave that has attested it is running the developer’s trusted dummy enclave.

• **Use both local and remote storage.** If stored locally, the secret code must be stored encrypted either in the dummy enclave or as an encrypted file on disk. However, there is no reason that the code must be stored locally. It could also be sent from a trusted server over an encrypted connection, and thus the code itself can simply be loaded after it is obtained from the server. However, this introduces a tradeoff between size overhead on the client and network overhead. If the secret code is included encrypted with the enclave, then the enclave will only need to retrieve the decryption key from the server, whereas if the code is stored remotely, then the encrypted code must be sent to the client before the enclave can be used.

### 2.2.3 SGXElide Overview

An overview of SGXElide is provided in Figure 2.1. From a developer’s perspective, there are just two components: a Sanitizer and a Runtime Restorer. To use SGXElide, an enclave
programmer will develop the (secret) enclave code as usual, integrate the regular enclave code with our library, and compile it to produce secret.so.

Next, our Sanitizer takes secret.so and redacts all user defined functions in this shared library. To know precisely which function is user defined, our Sanitizer uses a whitelist our framework provides. All functions not on the whitelist are considered user functions and will be sanitized. Our Sanitizer produces sanitized.so with all secret functions redacted. We place metadata in a enclave.secret.meta file and the original raw bytes in enclave.secret.data.

At runtime, sanitized.so is initialized as usual. Then elide_restore is invoked by developer code to perform remote attestation and then retrieves the secrets from a trusted party through a secure channel. Then our Runtime Restorer restores the secret enclave code. Note that the implementation of the Runtime Restorer is embedded in sanitized.so.

2.2.4 How to Develop Secret Enclave Code

As stated earlier, we aim to require as few changes as possible to an SGX application, and our solution was to sanitize all developer functions. This requires no input from the developer, as the whitelist is identical for all SGX applications. However, we do require a developer to call elide_restore in order to restore enclave functions. We require this instead of automatically restoring the enclave because the enclave is not necessarily entered when created. One solution would be to insert a call to elide_restore at the top of all ecalls before the original functions are restored, meaning the first ecall to be called would restore the enclave before continuing. However, this would result in unpredictable latency from the first ecall invocation. In addition, by explicitly having developers call elide_restore, they can handle various errors the enclave might encounter (e.g., a network error).

Therefore, the only changes a developer must make to the enclave application are adding the library and a single call to elide_restore. However, the library also requires an authenti-
cation server to give an attested enclave the data it needs to restore its functions. Our framework contains a very small number of public API functions: only one ecall (elide_restore) and two ocalls (elide_server_request and elide_read_file). The ocalls are automatically called by our library, so the required developer effort is minimal.

Finally, in our framework, the server stands alone and requires no developer input, but in many applications it may be desirable for the developer to add custom functionality between enclave and server. However, for the task of simply attesting that an enclave is running on real SGX hardware, the process can be automatic. Thus our framework only requires a server with access to the secret data and metadata that the enclave requires.

2.3 Detailed Design

As outlined in Figure 2.1, our system’s operation is divided into three main stages: whitelist generation (§2.3.1), enclave sanitization (§2.3.2), and runtime code restoration (§2.3.3).

2.3.1 Whitelist Generation

The first stage involves building a dummy enclave (dummy.so) containing only SGXELiDE helper functions and required SGX libraries. This stage is required to generate the whitelist of functions not to be sanitized. Building the dummy enclave is quite straightforward; we write an enclave containing the helper APIs introduced by our framework and link with the libraries they require (e.g., SGX crypto libraries).

This base enclave is analyzed by our sanitizer to extract the whitelist of functions that do not need to be redacted. Normal users of our framework will never touch dummy.so. Also, note that our sanitizer does not need to analyze dummy.so to sanitize an enclave, as the extracted whitelist can be reused across all developer enclaves.
2.3.2 Sanitized Enclave Generation

Our sanitizer component takes an unsigned enclave as input and outputs an unsigned enclave with the contents of all functions not on the whitelist removed. In order for this to work, the developer must build the enclave with the SGXELIDE library and their own functions. Then, before signing the enclave, they pass it to the sanitizer, which strips out the content of all their functions based on the whitelist functions from dummy.so, and outputs a sanitized enclave and two files pertaining to the secrets that were stripped out.

The first file produced is enclave.secret.meta, which contains information about the data itself such as its size and whether it is encrypted. If the data is encrypted, the metadata also includes the decryption key. This file must never be distributed with the enclave and only reside on the authentication server. The second file contains the confidential data (i.e., enclave.secret.data), and must be encrypted (if delivered with the enclave) or kept only with the authentication server so that its secrets cannot be leaked. An extra flag tells the sanitizer whether to encrypt the data.

2.3.3 Runtime Secret Restoration

The restoration component must run before any of the encrypted functions can be executed. These runtime libraries are compiled into the enclave and application when the developer adds SGXELIDE to their project. As shown in Figure 2.2, the library can operate in two ways depending on whether the data is stored locally or remotely. However, both approaches share common steps.

- **Step 1**: The application calls elide_restore. This is the single enclave call that the library requires to perform the entire process.

- **Step 2**: The enclave calls elide_server_request with REQUEST_META. This function connects to the server and requests the metadata about the stored secrets that the the enclave requires.
Figure 2.2. Runtime operation of SGXELIDE, showing how the enclave communicates with a server. Step ➊ and Step ➋ demonstrate communication with a server holding the secret data, while Step ➌ and Step ➍ demonstrate communication with a server holding the decryption key, with the encrypted secrets stored locally.

- **Step ➊**: The server responds with the metadata. Since this data is sent over a secure connection, it may contain the decryption key for the secret data.

After this point, the behavior diverges based on the metadata contents. If the metadata states that the secret data is encrypted, then the data is stored encrypted locally. Otherwise, the server has the data. Note that the enclave does not need to contact the server every time. SGX’s sealing mechanism provides the ability for the enclave to seal data to disk using an enclave-specific key derived from the SGX hardware key and unseal the data later, therefore allowing all accesses to the secret code after the first to require no network communications at all.

**Remote Data.** If the secret data is stored on the authentication server, it can simply be sent over the secure connection directly to the enclave, where it will be stored in its secure memory. This approach corresponds to the continued communications with the server in Figure 2.2.

- **Step ➋**: The enclave calls elide_server_request with REQUEST_DATA.
• **Step 6**: The server responds with the data. As with the other approach, this secret data is sent over a secure channel.

**Local Data.** If the data is stored locally, then it must be encrypted, and the metadata will contain the decryption key for the encrypted data. This allows the enclave to finish without needing to contact the server again. The details for this approach are shown in Figure 2.2.

• **Step 9**: The enclave calls `elide.read_file` to load the encrypted secret data file into its memory.

• **Step 10**: The enclave calls the SGX library’s standard decryption functions to decrypt the secret data.

The final two steps are identical for both approaches in Figure 2.2.

• **Step 11**: The enclave copies the original bytes over the sanitized ones. After this point, all previously encrypted functions can now be called by other enclave functions, or they can be called from outside the enclave via a corresponding bridge function.

• **Step 12**: Before shutting down, the enclave seals the secret data with its sealing key so that it will not need to contact the server in the future.

These two approaches represent a tradeoff between local storage and data transmitted over the network. Storing the enclave data locally requires less data to be sent from the server, but takes up more initial disk space, while storing the enclave on the server requires the server to send the redacted enclave content to the enclave.
2.4 Implementation

We have implemented SGXElide, which is made publicly available, in C/C++ and python on Linux. Code inside the enclave is C/C++, but the sanitizer and server components can be implemented in any language. Therefore, we wrote the sanitizer and server in python, while the enclave helper functions are written in C/C++. For enclave encryption and decryption we used the SGX SDK crypto libraries, while we used the cryptography package for the python server. Below we provide some implementation details of how we implement SGXElide.

Sanitizer. We provide a list of our library and default SGX functions from the dummy enclave to the sanitizer. Every time an enclave (e.g., enclave.so) is compiled, it is passed to the sanitizer. The sanitizer parses the ELF section headers and enumerates through each function in the shared object. Any function not on the whitelist is sanitized by overwriting its contents with zeroes. Once we finish sanitizing the original shared object, we save the original contents of the text section to an enclave.secret.data file. If the sanitizer is told to encrypt the data, it encrypts the original text section with a new encryption key and writes metadata to an enclave.secret.meta file.

Server Protocol. Communication between our client and server is simple. The client sends a single byte request representing what resource it requires (i.e., REQUEST_META in Step 2, and REQUEST_DATA in Step 1, respectively), and the server responds with the data. The client and server communicate using AES GCM encryption, and if the secret data is encrypted on disk it also uses AES GCM. The metadata provided by the server consists of the data length, offset, whether it is encrypted, and (if encrypted) its encryption key, initialization vector (IV), and MAC. The offset value is the offset of the elide_restore function from the start of the text section.

Enclave Self-Modification. There is in fact some challenge involved to enable a self-modifying enclave. An intuitive approach to this is to change the permissions of all pages
in the text section as writable right before writing to those pages. In Linux this can be
done with the \texttt{mprotect} system call, although the call itself would have to be performed
outside the enclave. However, this approach does not work, as the SGX hardware enforces
the original permissions.

Our solution to this is to set the permissions of the text section’s pages statically when we
sanitize the enclave. In ELF files, the executable file format for Linux, segments specify which
parts of the file are to be loaded at certain addresses. In the program headers table in an
ELF file, there is an entry for each segment, and each segment has a \texttt{p:\text{flags}} field specifying
the permissions for the pages in that segment. We therefore modify the \texttt{p:\text{flags}} field for the
program header entry for the text section in the enclave \texttt{.so} file; we \texttt{or} the existing field’s
value with \texttt{PF\_W} (the flag specifying a segment is writable), making the section writable.
Note that this makes the section writable through the enclave’s lifetime. We discuss ways
to mitigate this in \S 3.7.

When it is time to restore the enclave’s contents, we load the secret data retrieved
either from the server or from disk and containing the exact contents of the original text
section, which we then copy over the sanitized version of the text section in memory. We
use position-independent code to do this by taking the offset value from the metadata (offset
of \texttt{elide\_restore} from the text section’s start) and subtracting it from the address of
\texttt{elide\_restore}. This gives the starting address of the text section, so we can copy the
original text section contents directly over the sanitized contents.

This approach could be made more space efficient by keeping track of the ranges of each
sanitized function and storing only that data in \texttt{enclave.secret.data}, but this would pro-
duce a more complicated implementation not necessary for a proof-of-concept. We therefore
use the simple approach of saving and restoring the entire original text section instead of
the individual sanitized functions.
2.5 Evaluation

In this section, we present how we evaluate SGXELIDE. We describe how we created the benchmarks in §2.5.1 and how SGXELIDE performs over the benchmarks in §2.5.2.

2.5.1 Experiment Setup

Since SGX is a new platform and requires development effort to create new SGX software, we had no benchmarks available to evaluate SGXELIDE. While it is possible to use a library OS (e.g., Haven (Baumann et al., 2014)) to directly run legacy applications atop SGX, it does not offer the full benefits of enclave protection. Therefore, we must first develop applications that use SGX. It turns out this is actually non-trivial, because we have to select the appropriate programs in which to hide secrets and port them into SGX.

Benchmark Creation. We selected seven open source programs and ported them to SGX. We then inserted our framework into each ported application. As shown in Table 2.1, we selected four cryptographic algorithms, two games, and one reverse engineering challenge program. Our choice in selecting games is obvious, since games are frequent targets of reverse engineering. We use the cryptographic functions for illustrative purposes since their implementations are public and have no need to be hidden.

At a high level, an SGX application needs to be divided into trusted and untrusted components. The trusted component, containing all application secrets, will be executed inside the enclave. The rest of the application, including all runtime libraries, belongs to the untrusted component. According to the SGX developer manual (Intel Corporation 2014), a developer should make the trusted component as small as possible because larger enclaves could have more vulnerabilities.

Therefore, to port an application to SGX, we must first find the secret functions we aim to protect, put them into the trusted component, and leave the rest in the untrusted component. Take the first benchmark, AES, as an example: we protected its 4 encryption
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LOC</th>
<th>UC</th>
<th>TC</th>
<th>LOC</th>
<th>UC</th>
<th>TC</th>
<th>LOC</th>
<th>UC</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES (Kokke, 2019)</td>
<td>302</td>
<td>982</td>
<td>150</td>
<td>1,527</td>
<td>472</td>
<td>227</td>
<td>502</td>
<td>150</td>
<td>38</td>
</tr>
<tr>
<td>BinHex (Wu et al., 2014)</td>
<td>110</td>
<td>120</td>
<td>130</td>
<td>120</td>
<td>130</td>
<td>140</td>
<td>140</td>
<td>130</td>
<td>140</td>
</tr>
<tr>
<td>Crackme (0xe7, 0x1e, 2014)</td>
<td>120</td>
<td>130</td>
<td>140</td>
<td>130</td>
<td>140</td>
<td>150</td>
<td>150</td>
<td>140</td>
<td>150</td>
</tr>
<tr>
<td>DES (Hossain, 2010)</td>
<td>110</td>
<td>120</td>
<td>130</td>
<td>120</td>
<td>130</td>
<td>140</td>
<td>140</td>
<td>130</td>
<td>140</td>
</tr>
<tr>
<td>SHA-1 (Eastlake and Jones, 2011)</td>
<td>110</td>
<td>120</td>
<td>130</td>
<td>120</td>
<td>130</td>
<td>140</td>
<td>140</td>
<td>130</td>
<td>140</td>
</tr>
<tr>
<td>SHA-2 (Eastlake and Hansen, 2011)</td>
<td>110</td>
<td>120</td>
<td>130</td>
<td>120</td>
<td>130</td>
<td>140</td>
<td>140</td>
<td>130</td>
<td>140</td>
</tr>
<tr>
<td>SHA-3 (Keccak, 2019)</td>
<td>110</td>
<td>120</td>
<td>130</td>
<td>120</td>
<td>130</td>
<td>140</td>
<td>140</td>
<td>130</td>
<td>140</td>
</tr>
</tbody>
</table>

Table 2.1: Ported benchmarks divided into Untrusted (UC) and Trusted (TC) Components.
and decryption related functions. However, we also needed to port another 11 functions into the enclave as they are transitively called by the first 4 functions. Partitioning a process into untrusted and trusted components is often the most difficult step because most applications are not designed to be partitioned in this way.

When porting these functions inside the enclave, we have to declare bridge functions: ecalls for the untrusted component to call enclave code (e.g., cryptographic functions), and ocalls for enclave code to call untrusted functions (e.g., system calls). Note that this is often tedious due to the strong dependency between trusted and untrusted code. We may end up with many ecalls/ocalls, depending on the secret we aim to protect. The sizes of the ported benchmarks are shown in Table 2. Details are elided for brevity.

The secrets we aim to protect are application specific. For the cryptographic functions, the secrets are the corresponding algorithms. crackme is similar. The secrets for the games are code that loads/decrypts the assets from disk to defeat reverse engineering.

Having ported the regular programs to SGX, we next add the protection of SgxElide. For each ported program, we simply recompile them with our framework code with no enclave code modifications. We manually insert an explicit elide_restore call into the untrusted component. Therefore, as illustrated in the 5th and 6th columns in Table 2.1, the final untrusted code size is always 50 LOC more (the call to elide_restore and our library’s ocalls), and the trusted component is always 113 LOC more (our library’s ecalls and additional library helper functions).

Our above description demonstrates that it is indeed very convenient to use our framework; we simply add one function call in the untrusted component of any SGX program, and then recompile both the trusted and untrusted component with our library to get a new SGX program protected by SgxElide. The most tedious work lies in creating the SGX programs themselves. This explains why all of our original benchmark programs are mostly small to middle sized programs (from 412 LOC to 3523 LOC).
Environment Configuration. Our experiment ran on an Ubuntu 14.04.4 LTS machine with 64GB RAM and a 3.40Ghz Intel i7-6700 Skylake CPU. We compiled our benchmarks with gcc/g++ 4.8.4 and the Linux SGX SDK (Intel Corporation 2016b).

2.5.2 Experimental Result

Sanitizer. After we have compiled and linked each enclave’s code, our sanitizer takes the enclave .so as input and produces a sanitized enclave. As reported in Table 2.1, for each enclave binary, we sanitize any function not on the required function whitelist. Our sanitizer sanitizes all functions except the 170 on the whitelist. Note we have 170 unsanitized functions due to many statically linked library functions (e.g., sgx_rijndael128GCM_decrypt) in our dummy enclave, in addition to our framework’s functions. These whitelist functions consist of all the functions within a minimal enclave containing only our restoration code and standard SGX libraries, and provide the functionality for the enclave to restore the developer’s functions.

We also measured how long it takes to sanitize an enclave, as shown in Table 2.2. We ran the sanitizer 10 times per benchmark, then took the average and standard deviation. The sanitization time for each enclave is around 0.09 ms for remote data and 0.15 ms for local data as they are all of similar size. The process takes less time for remote data because in that case the sanitizer does not encrypt the secret data until runtime, when the server sends it to the enclave. Sanitization will take longer for larger enclaves, but we emphasize that this occurs offline, without any impact on runtime execution.

Runtime Restorer. When called, our runtime restorer will contact the server to retrieve and restore the sanitized functions. We measured the overhead for restoration, performing restoration 10 times for every benchmark, with the result also presented in Table 2.2. We ran the enclave application and server on the same machine connecting over network sockets, so there was very little network latency. In our testing environment the restoration process took less than 5 ms. The overhead of using remote data was very similar to local data,
Table 2.2. Sanitization/restoration execution time (ms) with remote/local data.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Remote Data</th>
<th>Local Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>0.09</td>
<td>0.01</td>
</tr>
<tr>
<td>DES</td>
<td>0.09</td>
<td>0.01</td>
</tr>
<tr>
<td>Sha1</td>
<td>0.09</td>
<td>0.01</td>
</tr>
<tr>
<td>Shas</td>
<td>0.09</td>
<td>0.00</td>
</tr>
<tr>
<td>2048</td>
<td>0.09</td>
<td>0.01</td>
</tr>
<tr>
<td>Biniax</td>
<td>0.09</td>
<td>0.00</td>
</tr>
<tr>
<td>Crackme</td>
<td>0.09</td>
<td>0.01</td>
</tr>
</tbody>
</table>

with only slightly more benchmarks taking longer with remote than local; the difference is minimal. Also, note that such overhead only occurs once (when the enclave is first created), and is fixed for each specific enclave.

**Overall Performance Overhead.** Finally, we also measured the performance overhead of SgxElide over the SGX-only versions. Since the games run forever, we did not measure their overhead and instead measured the four cryptographic programs and Crackme. We used the built-in test suites for the cryptographic programs for testing and directly execute Crackme since it does not require input. We ran the selected benchmarks 10 times each. The normalized average performance overhead for this measurement is presented in Figure 2.3 and Figure 2.4. The runtime overhead increase is tiny (all < 3% over the SGX version for both remote and local data). This is expected because all SgxElide applications have fixed startup overhead from restoring the enclave functions, determined by the amount of code restored; after that point the code is identical to the plain SGX version, and the runtime is dominated by identical enclave computations.

### 2.6 Discussions

**Security Implications.** The goal of SgxElide is to offer developers an almost transparent approach (requiring only one line of code to call `elide.restore`) to provide code secrecy. No
prior works, nor Intel, offer such a capability. While the high level concept may appear trivial, many seemingly obvious approaches are unfeasible. We had to examine various alternatives and experiment with what is and is not allowed in enclaves. The result is a combination of existing concepts such as self-modifying code.

By making enclaves self-modifying, SGXElide introduces new security challenges and changes assumptions about how enclave code can be vetted—code screening as in Apple’s iPhone app store will not work since SGXElide enclaves are self-modifying. This also introduces the security issue of whether a platform owner (e.g., a cloud provider) should trust enclave code. For instance, an enclave can pass an initial scan for malicious code, but later unpack a malicious payload. Therefore, there is a need for new research to search for solutions in defending against malicious enclaves. However, enclaves are isolated and depend on the OS and host application to interact with the outside world, so a security policy could restrict an enclave’s capabilities. Also, developers must sign enclaves before distributing them, so there is a degree of attribution that may make it possible to blacklist or identify malicious developers.

However, security concerns go beyond intentionally malicious enclaves. Since SGXElide makes the enclave text section writable, certain vulnerabilities in an enclave could result in an attacker inserting arbitrary malicious code into the enclave. We added an mprotect call revoking PROT_WRITE for the enclave text section immediately after restoring the enclave code. However, mprotect must be called outside the enclave, so this would not defend against a malicious OS or host application. Note that this still requires a vulnerability in the enclave to allow an attacker to actually modify enclave code, and this can be protected against by using software-based DEP, in which the enclave code is compiled with memory write instructions that can never write to the text section (Seo et al. 2017). Only the restoration instructions would be allowed to overwrite code. Also, while there is no way to securely change runtime permissions in the currently available SGX-v1, SGX-v2 will provide this ability (Fan 2016).
The recent discovery of the powerful controlled-channel attacks against SGX showed that enclave code could potentially leak extensive amounts of data to a malicious OS (Xu et al., 2015). However, our solution is in fact an excellent defense against such attacks, because controlled-channel attacks require knowledge of the code in order to extract secrets. If the code itself is hidden, an attacker will not have this information.

**Limitations and Future Work.** We have demonstrated that we can use SgxElide to protect enclave code secrecy, but there are several ways to improve our work. One is to make our framework totally transparent if a user does not mind unpredictable runtime latency imposed during restoration. As discussed in §2.2.4, we decided to have developers explicitly call `elide_restore`; we could remove this explicit call by having restoration occur the first time an ecall is made.

Second, we only ported a handful of benchmarks to evaluate our framework. Our immediate task is to investigate using SgxElide to protect large scale and more practical software. Another valuable research direction lies in developing automatic techniques to partition code (Lind et al., 2017), which could significantly boost the speed of enclave code development.

Third, we currently only focus on code secrecy against existing reverse engineering techniques. SgxElide does not protect against data leakage vulnerabilities. We will look into how to secure the enclave against other attacks.

Finally, we did not implement a few details of our framework, such as the final sealing step or performing full attestation. These would be important for an actual production system, but our implemented framework is sufficient to demonstrate the effectiveness of our approach.
2.7 Conclusion

In this paper, we presented SGXELIDE, a framework to ensure enclave code confidentiality. By treating code as data, we dynamically restore code at runtime by writing the decrypted code over the sanitized functions. SGXELIDE can easily be integrated with any SGX project to provide code secrecy, with secrets delivered by a developer-controlled trusted party. We have implemented SGXELIDE atop the Linux SGX SDK, and our evaluation with SGX programs shows that SGXELIDE can be used to protect the code secrecy of practical applications without any runtime overhead after the enclave is initialized.
CHAPTER 3
MULTIVERSE

In many systems and security applications, there is a need to statically transform COTS binaries. Software fault isolation (SFI) (Wahbe et al., 1993), including Control Flow Integrity (CFI) (Abadi et al., 2009), constrains the program execution to only legal code by rewriting both data accesses and control flow transfer (CFT) instructions. Binary code hardening (e.g., STIR (Wartell et al., 2012b)) rewrites and relocates instructions, randomizing their addresses to mitigate control flow hijacks. By lifting binary code to an intermediate representation (e.g., LLVM IR), various compiler-missed platform-specific optimizations can also be performed (Anand et al., 2013).

Given so many applications centered around binary code transformation, significant efforts have been made over the past few decades to develop various binary rewriters, particularly for Intel x86/x64 architectures due to their dominance in modern computing. Early approaches for transforming these binaries require special support from compilers or make compiler-specific assumptions. For instance, SASI (Erlingsson and Schneider, 1999) and PittSFIeld (McCamant and Morrisett, 2006) only recognize gcc-produced assembly code—not in-lined assembly from gcc. CFI (Abadi et al., 2009) and XFI (Erlingsson et al., 2006) rely upon compiler-supplied debugging symbols to rewrite binaries. Google’s Native Client (NaCl) (Yee et al., 2009) requires a special compiler to compile the target program, and also limit API usage to NaCl’s trusted libraries. These restrictions have blocked binary rewriting from being applied to the vast majority of COTS binaries or to more general software products.

1This chapter contains material previously published as: Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In Network and Distributed Systems Security (NDSS) Symposium 2018. The lead author, Erick Bauman, conducted the majority of the research for the paper, including implementation, evaluation, and most of the writing.
More recent approaches have relaxed the assumption of compiler cooperation. Stir (Wartell et al., 2012b) and Reins (Wartell et al., 2012a) rewrite binaries using a reassembling approach without compiler support; however, they still rely upon imperfect disassembly heuristics to handle several practical challenges, especially for position-independent code (PIC) and callbacks. Object Flow Integrity (Wang et al., 2017) has more recently improved callback support, but only by relying on higher-level information, such as interface description language files. Ccfir (Zhang et al., 2013) transforms binaries using relocation metadata, which is available in many Windows binaries. SecondWrite (O’Sullivan et al., 2011) rewrites binaries without debugging symbols or relocation metadata by lifting the binary code into LLVM bytecode and then performing the rewriting at that level. However, it still assumes knowledge of well-known APIs to handle callbacks, and uses heuristics to handle PIC. Lifting to LLVM bytecode can also yield large overheads for binaries not easily representable in that form, such as complex binaries generated by dissimilar compilers. BinCFI (Zhang and Sekar, 2013) presents a set of analyses to compute the possible indirect control flow (ICF) targets and limit ICF transfers to only legal targets. However, BinCFI can still fail when code and data are intermixed. Recently, Uroboros (Wang et al., 2015) presented a set of heuristics to recognize static memory addresses and relocate and reassemble them for binary code reuse, but experimental results still show it has false positives on the SPEC2006 benchmarks.

Thus, nearly all static Intel CISC binary rewriters in the literature to date rely upon various strong assumptions about target binaries in order to successfully transform them. While each is suitable for particular applications, they each lack generality. End users cannot be confident of the correctness of the rewritten code, since many of the algorithms’ underlying assumptions can be violated in real-world binaries. To advance the state-of-the-art, we present MULTIVERSE, an open source, next generation binary rewriter that is able to statically rewrite x86 binaries without heuristics; binaries rewritten without heuristics have the same semantics as the original.
To this end, we address two fundamental challenges in COTS binary rewriting: (1) how to disassemble the binary code and cover all legal instructions, and (2) how to reassemble the rewritten instructions and preserve the original program semantics. To solve the first challenge, we propose a superset disassembling technique, through which each offset of the binary code is disassembled. Such disassembling creates a (usually strict) superset of all reachable instructions in the binary. The intended reachable instructions are guaranteed to be within the superset, thereby achieving complete recovery of the legal intended instructions (i.e., completeness).

To address the second challenge, we borrow an instruction reassembling technique from dynamic binary instrumentation (DBI) [Luk et al., 2005], which mediates all the indirect \textit{CFT} (iCFT) instructions and redirects their target addresses to the rewritten new addresses by consulting a mapping table from old addresses to new rewritten addresses created during the rewriting. Since all iCFTs are instrumented in a very similar way to how dynamic binary instrumentation rewrites the binary at runtime, the original program semantics are all preserved, achieving soundness (i.e., all program semantics, including control flow destinations, are identical to the original binary). Therefore, our approach is sound and complete with respect to the original static binary’s intended instructions and execution semantics.

In summary, our main contributions are as follows:

- We present \textsc{Multiverse}, the first static binary rewriter built on a foundation of both soundness and completeness, raising assurance in the correct execution of rewritten binaries.
- We design a superset disassembling technique, which does not make any assumptions on where a legal instruction should start and instead disassembles and reassembles each offset, achieving complete recovery of original instructions.
- We also develop a static instruction reassembling technique, which translates all indirect control flow transfer instructions (including those in the library) and redirects
their target addresses to correct ones, achieving the soundness of original program execution.

- We have implemented these techniques in our prototype, and evaluated it with the SPECint 2006 benchmark suite. Experimental results show that MULTIVERSE correctly rewrites all the test binaries. A comparison with dynamic instrumentation also shows that the static instrumentation enabled by MULTIVERSE has better average performance.
- We have also demonstrated one security application of using MULTIVERSE to implement a shadow stack. In doing so we provide a sample of the possibilities of the security applications of MULTIVERSE.

3.1 Background and Overview

3.1.1 Scope and Assumptions

The goal of this paper is to develop a new binary transformation algorithm that improves the practicality and generality of existing code transformation applications, such as binary code hardening. Our approach generalizes to arbitrary OSes and Intel-based CISC architectures, but for expository simplicity we here focus on 32-bit x86 binaries running atop Linux (ELF-32) generated by mainstream compilers such as gcc or llvm. We assume no restriction of original program source code, which can even be hand-written assembly. Although we focus on mainstream compilers for our presentation, our approach accommodates most statically obfuscated binaries (e.g., instruction aliasing, code and data interleaving, etc.). We do not automatically support code that loads shared libraries dynamically, such as with dllopen. However, such binaries can still be rewritten after manual recovery of dynamically loaded libraries. In addition, like all existing static binary rewriters, we do not handle any self-modifying or packed code (such as self-extracting compressed software) or JIT-compiled
code. Support for such code requires dynamic rewriting since such code is not visible or does not exist in a static binary.

We focus on x86 instead of x64 because legacy x86 applications are less likely to have source available, and many code transformations target older legacy code. In addition, while the differences between x86 and x64 for the purposes of binary rewriting are not too significant, there are some engineering differences that distract from the discussion of binary rewriting. Therefore, we focus on x86 for the purposes of this paper.

3.1.2 Challenges

There are enormous challenges in designing a general binary rewriter. To illustrate these challenges clearly, Figure 3.1 presents a contrived working example. This simple program sorts an array of strings in ascending or descending order (depending on the least significant bit of the program’s pid) using libc’s qsort API. When printing out the mode (ascending or descending sort) or printing each array element, the program uses the function get_fstring, defined in fstring.asm, to determine the format string it should use. This function is written in assembly to show a simple example of interleaved code and data. With this working example, we can organize the challenges into the following categories:

C1: Recognizing and relocating static memory addresses. Compiled binary code often refers to fixed addresses, especially for global variables. Code transformations that move these targets must update any references to them. However, it is very challenging to recognize these address constants within disassembled code and data sections, since there is no syntactic distinction between an address and an arbitrary integer value.

In our working example, the modes variable (line 17 of sort.c) is an array of function pointers stored in the .data section at address 0x0804a03c. Were we to move .data, we would need to identify and change all references to this array to point to its new location. In
Figure 3.1. A contrived working example that covers major challenges in x86 COTS binary rewriting.

more complicated applications, it is difficult to reliably differentiate between a pointer-like integer and a pointer—a major challenge in static binary rewriting.

C2: Handling dynamically computed memory addresses. In addition to static memory addresses, there are also dynamically computed memory addresses. A particular challenge concerns iCFTs whose target addresses are computed at runtime. For instance, an indirect jump target can be computed from a base address plus an offset, and a function pointer can be initialized to a function address also computed at runtime. These pointers can even undergo arbitrary binary arithmetic, be encoded (e.g., using a hash table), or be...
dereferenced in a number of layers (e.g., double pointers or triple pointers), before they are used. Unlike direct CFTs whose targets are explicit, iCFT targets often cannot be predicted statically. Remapping iCFT targets reliably is therefore a central challenge for binary rewriting.

When our working example calls one of the function pointers in the modes array, it is difficult to reliably predict which function will be called until runtime. As shown in the assembly, the mov at 0x804867d sets eax to the value of stack variable p, which determines the mode (0 or 1). The mov instruction at 0x8048681 then assigns eax the address held in the index of array modes determined by the mode (the address held at 0x0804a03c+0 or 0x0804a03c+4). Finally, it calls the address in eax. Statically predicting which addresses to update, while possible in this simple example, can quickly become intractable (e.g., if the array were dynamically allocated with unknown length).

**C3: Differentiating code from data.** In x86, there is no syntactic distinction between code and data within binaries (Caballero and Lin, 2016). More specifically, code and data can be interleaved. This is typical in hand-written assembly, and in modern compilers that aggressively interleave static data within code sections for performance reasons. Also, code bytes are unaligned—they can start at any offset within executable segments.

Lines 12 and 14 of fstring.asm exhibit data bytes amid code bytes. Linear sweep-based disassemblers often misinterpret these as code bytes, resulting in disassembly errors that yield garbage instructions and omit subsequent reachable instructions (e.g., the last mov instruction on line 16). Such garbage instructions can be seen in the disassembly starting at address 0x80485df. While a disassembler using recursive traversal can follow the control flow from the jz instruction to avoid some of these errors, a more complicated program with indirect control flow to the after label would make it difficult to statically determine which offsets are valid.
C4: Handling function pointer arguments (e.g., callbacks). Functions that expect function pointers as arguments can fail after binary transformation if the referant code is moved but the referring argument is not updated accordingly. Function pointer arguments are usually used in callbacks, where a code pointer is passed from the program as a computed jump destination. Unlike typical dynamically computed memory addresses (C2), which are visible to the rewritten binary, callback pointers are often used in library spaces. As mentioned in C1, it is already challenging to recognize static memory addresses, and it is even more challenging to recognize arguments with function pointer types at the binary level.

Our working example includes a call to the libc function qsort, which expects a callback function as its last argument, at lines 11 and 14 of sort.c. It uses this function to compare each element pair when it sorts the array, and the user must provide a comparison function that is meaningful for the array argument. In the example, the function supplied depends on the mode (ascending/descending). The assembly for the call for mode 1 is shown starting at address 0x80485fa. The mov instruction at that address moves the address of the gt function (0x80485a0) on the stack as the argument for qsort. If we move the location of gt but do not modify this address, qsort will call the wrong code.

C5: Handling PIC. While mainstream compilers generate mainly position dependent code by default, they can also generate PIC, which can be loaded at arbitrary addresses. PIC is typically achieved via instructions that dynamically compute their own addresses and expect to find other instructions or variables at known relative offsets. These instructions can break the program if a rewriter fails to identify them.

We compile cmp.c in the working example with the gcc flag -fPIC, which ensures that all functions in that file are compiled as PIC. The results are shown in the disassembly of our working example. Since PIC uses its own address to compute offsets, it uses a call instruction to compute its own position in the form of a special function that retrieves the instruction pointer. This function, _i686.get.pc_thunk.bx, is shown at 0x80485cc, and
consists only of an instruction that saves the return address into ebx. The print_array function uses this address to compute the address of array. Relocating this code without any modifications causes an incorrect address to be computed, usually resulting in a crash.

### 3.1.3 Key Insights

Binary rewriting is not a new problem. Over the past few decades, a tremendous amount of effort has been devoted to developing various binary rewriters for different purposes under different constraints. Drawing from these existing efforts, including related works using dynamic binary instrumentation (e.g., the widely used PIN (Luk et al., 2005)), we have derived and systematized the following key insights to address each of the above challenges.

**S1: Keeping original data space intact.** We can strategically avoid the need to recognize static memory addresses of data if we retain and preserve all bytes that the program might read as data. Since code sections might contain data bytes, we can preserve such data by retaining an old copy of each code section at its original location. For security-focused applications, we can set the original code section non-executable. This approach is used by several existing rewriters (e.g., SECONDWRITE (O’Sullivan et al., 2011), BINCFI (Zhang and Sekar, 2013), STIR (Wartell et al., 2012b), and REINS (Wartell et al., 2012a)).

**S2: Creating a mapping from the old code space to the new rewritten code space.** As discussed in C2, there are various forms of dynamically computed memory addresses. Heuristic approaches that attempt to statically identify base addresses and then update each associated offset accordingly are unreliable, since x86 address spaces are typically flat, allowing any base address to potentially index any higher address. Fortunately, we have another unique key observation: *Instead of identifying the base addresses and rewriting them to point to the new location, we can just focus on the final target addresses and ignore how it is computed.*
More specifically, even though a target address can be encoded or computed through many layers of pointers, its final runtime value must eventually flow to the iCFT as its argument. (Note that direct CFTs are not a problem since their target addresses are explicit.) Therefore, if we can map each possible destination address in the old address space to an address in our new, rewritten code, and if we make the mapping available at runtime, then we can rewrite each iCFT to look up the new address immediately after the old destination address has been computed. This allows us to automatically solve C2 without relying on any heuristics.

**S3: Brute force disassembling of all possible code.** Disassembling is a perennial problem for static binary analyses. Unlike many prior efforts, we observe that while it is challenging to correctly disassemble arbitrary code, we can instead find a superset of the disassembled code (by brute force disassembling every executable byte offset), and the result will contain the correct disassembly somewhere in the set (which is why we call our approach superset disassembly). While disassembling from each offset has been explored in malware analysis (Kruegel et al., 2004) (Wartell et al., 2014) (Ladakis et al., 2015), the resulting disassemblies are intended for reverse-engineering obfuscated code, finding function entry/exit points, and other analysis purposes; no attempt has been made to link the superset code and make it runnable. A new challenge for us therefore concerns linking the instructions in the superset. Fortunately, if we translate all iCFT instructions (as we do in S2), then we can link them together by using our old address to new address mapping table.

**S4: Rewriting all user level code including libraries.** One possible solution to C4 is to identify each function that uses function pointer arguments in external libraries and patch the function pointer address so that callbacks correctly reference our new text section. Many prior rewriters use this approach, including STIR (Wartell et al., 2012b), REINS (Wartell et al., 2012a), and SECONDWRITE (O’Sullivan et al., 2011). However, if our transformation algorithm is sufficiently general, we can instead expand our rewriting to include all program
code including libraries. All callbacks will then be executed correctly (by S2) without having to identify callback arguments.

This solution also has the considerable benefit of accommodating C++ exceptions, wherein the .eh_frame holds information about exception handler addresses, which may be called from a different module than the caller, effectively acting like a callback. By rewriting the instruction that jumps to the exception handler, C++ exceptions are transparently handled.

**S5: Rewriting all call instructions in order to handle PIC.** It is challenging to identify PIC in the binary code, because there are a great diversity of instructions that derive code or data offsets from PIC-computed self-addresses. However, after careful examination of x86 instruction semantics, we find that only the `call` instruction, which pushes the instruction pointer onto the stack, can be feasibly used to compute the base address used in subsequent PIC offset calculations. Therefore, we translate each `call` instruction in the original code into an explicit `push` of the *old (unmodified)* return address followed by a `jmp` to a new, rewritten address (computed from the old target address by querying the mapping table). This transparently preserves PIC because any subsequent address arithmetic will compute a correct old code address, which will be correctly remapped to a correct new address when it finally flows to an iCFT (by S2). If PIC is used to access data, then the correct data is accessed because the pushed address does not flow to an iCFT, and is therefore not remapped.

### 3.1.4 Overview

From the insights above, we have created MULTIVERSE² a binary rewriter that accepts an ELF-32 binary or shared library and transforms it to produce a new rewritten binary. As

---

²We call our system MULTIVERSE, since it conservatively assumes that any instruction path that “can happen, does happen” [Cox and Forshaw 2012].
illustrated in Figure 3.2, our system consists of two separate phases: the mapping phase, and the rewriting phase.

In the mapping phase, we use our superset disassembler to disassemble the binary at every byte offset starting from the lowest code address. This produces many copies of the same code, since instruction sequences at most offsets eventually align with an instruction disassembled from a previous offset. We avoid this unnecessary duplication by ceasing disassembly at the first redundant offset of each sequence, and later inserting an unconditional jump to the code previously disassembled for that offset. At a high level, we generate our mapping by disassembling each instruction, determining the length the rewritten instruction will have in the final code, and then using this information to create a final mapping from old addresses to new addresses that will be used in the next phase and placed into .localmapping.

In the rewriting phase, our instruction rewriter again iterates through each disassembled instruction and generates the new bytes that will be placed in the .newtext section. We must make this second pass because we cannot generate the final code without already knowing the complete mapping. For instance, if an instruction in the old code refers to a specific offset at a higher address that we have not yet disassembled, we will not know what new offset to use when rewriting the instruction.
Once the new text section is created, we pass it to our ELF writer (which is not shown in Figure 3.2), which takes the new entry address, mapping, and rewritten text and creates the final rewritten binary. The ELF writer modifies the ELF header and PHDRs in order to create a new segment to hold the new text. The original .text section is left in its original location as non-executable data to support reads from the .text section (e.g., for jump tables). Next, we present the detailed design of MULTIVERSE. We first describe our mapping in §3.2 and then explain our rewriting in §3.3.

3.2 Mapping

3.2.1 Superset Disassembler

“When in doubt, use brute force.” – Ken Thompson

Our superset disassembler disassembles instruction sequences starting from every byte offset in the binary’s text section. This approach can also be considered a form of brute-force disassembly, i.e., we are finding the intended sequences of instructions by brute-force disassembling every possible offset. However, without any further refinement, this would produce a huge number of duplicate subsequences. Therefore, we keep a list of all offsets we have already disassembled, and we stop disassembly from an offset if we reach an instruction we have already encountered. We can do this because we can connect a sequence of instructions to another in our rewriting phase via insertion of an unconditional jump.

A high-level illustration of how the brute-force disassembly process works is shown in Figure 3.3 and Algorithm 1. We start disassembling from offset zero and disassemble instructions until we reach an illegal instruction. Although we could stop disassembling at jmp or ret instructions (and disassemble the bytes after that instruction in a later pass), we simply try to disassemble as many as possible in each pass, partly for simplicity and partly for code locality reasons. By disassembling as many in a sequence as possible and not breaking up
Algorithm 1: Superset Disassembly

<table>
<thead>
<tr>
<th>input</th>
<th>empty two-dimensional list instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>string of raw bytes of text section bytes</td>
</tr>
<tr>
<td>output</td>
<td>all disassembled instructions are in instructions</td>
</tr>
</tbody>
</table>

1. for start_offset ← 0 to length(bytes) do
2. offset ← start_offset;
3. while legal(offset) and offset ∉ instructions and offset < length(bytes) do
4.   instruction ← disassemble(offset);
5.   instructions[start_offset][offset] ← instruction;
6.   offset ← offset + length(instruction);
7.   if offset ∈ instructions then
8.     instructions[start_offset][offset] ← “jmp offset”;

our rewritten code into distant chunks, we are able to benefit from locality in some cases when a program is using short unconditional jumps without having to do advanced analyses. That said, while we currently try to keep long contiguous instruction sequences, we have no restriction on how we organize the new instructions. For example, we could easily break longer sequences into blocks of any size by inserting jmp instructions in our rewritten code. Since every instruction in the old code is mapped, we can then move each block to any location in the new code space. This gives us flexibility depending on what the use case is for our rewritten binaries (i.e., we have the capability to freely shuffle the program instructions, which would be useful for software diversity).
Once we are done disassembling from offset zero, either because we eventually encountered an illegal instruction or $\text{offset} \geq \text{length(bytes)}$, we start disassembling from offset one. As illustrated in Figure 3.3, we show a case that the instruction sequence starting from offset one shortly encounters an offset that we already encountered in our previous pass starting from offset zero (condition $\text{offset} \notin \text{instructions}$ is false for the while loop at line 3 in algorithm 1), so when we rewrite our code we simply insert a jump at the end of that instruction sequence to go to the corresponding instruction from our disassembly from offset zero (line 8). The same thing happens from offsets two and five in Figure 3.3. However, offsets three and six instead encounter invalid byte sequences that do not correspond to any valid instruction encoding, so we simply stop the disassembly from that offset (condition $\text{legal(offset)}$ for the while loop at line 3 is falsified).

We can potentially eliminate all code at the end of an instruction sequence ending in illegal code starting from the last CFT instruction (e.g., jmp or ret). We conservatively include conditional CFT instructions as stopping points, since obfuscated code could include garbage bytes after an always-taken conditional jump. This ensures that the removed bytes are safe to omit, since that subsequence will never be executed unless the original program had a fatal error; if those instructions were executed in the original binary, it would crash when it reached the illegal sequence. This process continues until we reach the end of the text section and have disassembled every possible instruction offset (at which condition $\text{offset} < \text{length(bytes)}$ for the while loop is false, and $\text{start_offset} = \text{length(bytes)}$).

### 3.2.2 Mapping Generation

In order to generate the mapping, we retrieve each disassembled instruction from our brute-force disassembler and determine the length of the rewritten instruction that will be in our final binary. It is important to note that since instructions may refer to addresses not yet in our mapping, there is no way to generate the final rewritten bytes in this phase. Therefore,
we instead calculate how long each rewritten instruction will be. For most instructions, we make no modifications, and the length is the same.

Specifically, we rewrite all call, jmp, jcc, and ret instructions. All the jcc instructions involve simply changing the offset for the instruction. However, since we are inserting code, a jump short instruction may not have space for a larger offset; if a jcc instruction was originally written with a short encoding, we expand it to the longer jump near encoding instead, which allows for larger offsets before we know the actual offsets. The other instructions involve adding multiple instructions, so it is important to know how many bytes this adds when we build the final mapping. In practice, we run our rewriter on the instructions with placeholder addresses to substitute for the addresses we do not know, and then retrieve the length of the rewritten instructions in bytes. This strategy of keeping track of instruction length also makes conversion of MULTIVERSE to perform instrumentation quite straightforward; we can simply add the length of inserted instructions to the length of the rewritten instruction.

While we are building our mapping, we maintain a mapping from each old address to the size of the new bytes. When we build the final mapping, we convert the size to the corresponding offset in the new text section. By deferring this to after we disassemble all bytes, we obtain the flexibility to place blocks of new bytes in any order in our mapping as long as we end each block with a jmp instruction to the new instruction address corresponding to the next instruction in the old binary, or as long as we split instructions into basic blocks, in which case we would simply need to change direct control flow destinations.

### 3.2.3 Mapping Lookups

For static memory addresses, we modify instructions statically using our mapping offline. However, dynamically computed addresses (C2) require our mapping to be present in the binary at runtime for dynamic lookup and we must use an efficient data structure for reducing
Figure 3.4. A mapping lookup example for a rewritten binary dynamically linked with our rewritten libc.

runtime overhead. To this end, we generate a flat table of four-byte offsets large enough to have an entry for every byte in the old text section. This allows us to directly index into the table by computing the offset of the old address from the base address of the old text section. For offsets that did not disassemble to a valid instruction, we simply set the entry to 0xffffffff. For performing lookups in the table, we insert a small assembly function into the binary to look up an address from the old text section and return the corresponding new text section address.

We use the eax register as input to pass the old address that we want to look up to the function. Then we use our own PIC (getting the instruction pointer with a call) to obtain the offset to the mapping and look up the entry of the old address. If the entry is 0xffffffff, then the original program had an error and is attempting to jump to an illegal instruction. In such a case, we immediately trigger a segfault by jumping to a hlt instruction. If the entry is a valid address, then we return the address in eax. If the address is outside the range of the mapping, then the program may be attempting to call a library function, so we pass the address to be resolved by the global lookup function, which we discuss in §3.2.4.

Figure 3.4 shows a mapping lookup example for a rewritten binary, showing both its new text and data sections and the text and data sections of a modified libc. When a
rewritten instruction in .newtext requires that we dynamically look up an address, we first call local_lookup (1), which we have placed at the start of the .newtext section. This function knows its offset from .localmapping, so it can perform a lookup of the destination address (2). If the address is in the range of the old text section, then it simply returns the new address, our rewritten instruction jumps to that address, and the process is complete. If the address to look up is outside the old text section, we must refer to the global mapping.

3.2.4 Global Mapping

Since we are rewriting libraries as well as the original binary, each library has its own local mapping for its new text section. Since the libraries may be loaded dynamically, we must maintain a global mapping between the multiple new text sections that we have generated. As such, we have created a global mapping table and a global lookup function (global_lookup) that determines which local lookup function to call to resolve an address.

Function global_lookup operates at a page-level granularity. When a library is loaded, its old text section is mapped to one or more pages. The functionality of global_lookup is, therefore, to return the address of each library’s local lookup function for every page in the library’s original text section. In particular, as shown in Figure 3.4, if a local_lookup call does not have an address in its .localmapping, then it must call global_lookup.

As in the previous example, when an instruction in the .newtext section needs to perform a dynamic lookup, it first calls local_lookup (1). If the requested address is that of the libc function qsort, then it is outside the application’s .text section (2). Therefore, it calls global_lookup (3), which finds the entry for libc’s local_lookup in .globalmapping (4). It calls libc’s local_lookup (5), which is then able to find the updated address in its .localmapping (6). Once the new address is found, libc’s local_lookup returns the address to global_lookup, which returns the address to the main binary’s local_lookup, which finally returns the address to the rewritten instruction.
3.3 Rewriting

In our rewriting phase, we use our mapping to rewrite all the call/jmp/ret/jcc instructions from the old binary in order to preserve the original program’s CFTs. When we are rewriting each instruction, we go through the same instructions that we processed during the mapping phase. When we encounter a byte offset that has already been disassembled and rewritten, we insert a jmp instruction to the new address of the already rewritten instruction at the end of the current sequence. This allows us to only rewrite the instruction at each offset once.

**Rewriting direct CFT instructions (jcc/jmp/call).** All jcc instructions are direct CFTs, so we statically rewrite each jcc instruction by changing its offset. In addition, call and jmp instructions with an immediate operand can also be statically rewritten by changing the offset. However, jmp short and jcc short instructions only hold a 1-byte displacement, which may not be large enough when we expand our new text section; an instruction’s destination may become too distant in our new binary. Therefore, we expand these instructions to their longer encoding (jmp near and jcc near), allowing for a 4-byte displacement.

**Rewriting indirect jmp/call instructions.** Our static rewriting of indirect control flow instructions implements a dynamic lookup. We must perform a lookup at runtime of the destination address, since the runtime-computed address will point to the old text section. The transformations for jmp and call instructions are slightly different.

- **jmp:** If the instruction is jmp [target], we rewrite it to the following six instructions:

  ```
  mov [esp-32], eax
  mov eax, target
  call lookup
  mov [esp-4], eax
  mov eax, [esp-32]
  jmp [esp-4]
  ```

  We save eax to an area outside the stack, move the original target to eax, and call lookup, which will perform local_lookup first and then global_lookup if necessary
(detailed in §3.2.3). We then save the result (which was returned in eax) outside the stack, restore eax, and jump to the new destination. Also note that we are storing eax at esp-32. This is because the lookup function may push up to seven 4-byte values on the stack, and we must store the value outside the reach of the growing stack in order to avoid overwriting the value.

- **call**: If the instruction is a call [target] instruction, we also push the old return address (the address of the instruction after the old call, in order to transparently handle PIC), resulting in these seven instructions:

  mov [esp-32], eax
  mov eax, target
  push old_return_address
  call lookup
  mov [esp-4], eax
  mov eax, [esp-28]
  jmp [esp-4]

  Note that we restore eax from esp-28 because we pushed the 4-byte return address on the stack, decrementing esp by 4.

**Rewriting ret instructions.** Since we rewrite call instructions to push the old return address on the stack, we must perform a runtime lookup of the new return address. In addition, a ret instruction may also specify an immediate value specifying a number of additional bytes to pop off the stack, so we may also need to increment esp an additional amount. For example, for ret 8 we need to add 8 to esp. This results in six or seven instructions:

  mov [esp-28], eax
  pop eax
  call lookup
  add esp, pop_amount; Only add if immediate
  mov [esp-4], eax
  mov eax, [esp-(32+pop_amount)]
  jmp [esp-4]

  The location where we saved the value of eax is relative to esp, so we must calculate the offset. However, the calculation 32 + pop_amount is performed statically. Also note that the
preceding examples are for rewriting the original binary. We must insert slightly different
code for shared objects because we do not know the runtime base address for a shared object.
This means we insert PIC (similar to what we do in the local_lookup) in order to push
the correct old return value on the stack for each call instruction; we must obtain the base
address of the old text section of the shared object, which is loaded to a random address. This
slightly increases the overhead for shared libraries when compared to the original binary.

3.4 Implementation

We have implemented Multiverse atop a number of open source binary analysis and
rewriting projects. In particular, we used the python bindings for Capstone (Quynh and
Capstone Team, 2019) as our underlying disassembler engine, and we used pyelftools (Ben-
dersky, 2019) to parse the ELF data structures. We used pwntools (Gallopsled et al., 2019)
to reassemble the instructions. Additionally, we developed over 3,000 lines of our own python
code to implement our algorithm and maintain our data structures, and over 150 lines of
assembly, some of which is embedded as string templates in our python code. We also de-
veloped over 200 lines of C for the global mapping population function that is run when a
rewritten executable starts. Other than the global mapping population function, all of the
code that we rewrite or insert into the binary is written in assembly. Note that our system
can easily support any disassembler that can perform linear disassembly.

There are several Linux-specific issues that had to be addressed. First of all, the Linux
kernel loads a special shared library, the Virtual Dynamic Shared Object (VDSO), which
has no actual corresponding .so file in the filesystem. One use for this is to run the correct
code depending on whether the kernel supports the sysenter instruction for syscalls rather
than int 80. If it does, then control is redirected to the VDSO for each system call. Since
our solution requires no changes to the OS, we cannot change the VDSO, so we must in-
stead rewrite the return address before every call into the VDSO. Metadata regarding this
is passed to every process in the Auxiliary Vector, which is stored on the stack after the environment variables before application start. We insert code to parse this data structure at the new entry point of the rewritten binary, and we save the address of the VDSO syscall code. Later, whenever the application is about to jump to an address that the local lookup function does not recognize, our global lookup function performs a special check for whether the destination is the VDSO syscall code. If so, we rewrite the return address on the stack from the old address to the new address.

In addition, the dynamic linker is loaded before any other .so file, and libc calls various functions within it. In order to avoid rewriting the dynamic linker, we instead marked its address range in the global mapping as a special case, allowing us to rewrite return addresses whenever it is called. However, for dynamic function resolution, when an address is first resolved control first goes to the dynamic linker, and then the dynamic linker redirects control directly to the destination, not allowing any of our rewritten code to translate the old address to a new address. We resolve this by setting the environment variable LD_BIND_NOW to 1, which forces the loader to resolve all symbols and place their addresses in the GOT before the program starts. This prevents the loader from directly rerouting control to old text sections. This may increase the startup time of a rewritten binary, and symbols may be resolved that are never used, but this does not affect the correctness or safety of the rewritten binary. In fact, disabling lazy loading is part a defense designed to increase the security of the loader mechanism (Federico et al., 2015).

Finally, we must populate the global mapping when the application starts, as we will not know the actual addresses of each library until they are loaded into memory. Therefore, we insert the global mapping population function, which runs before start in the rewritten binary, to find the address ranges of each library and write them to the global mapping. Later, during execution, the global lookup function uses these mappings to resolve the locations of local lookup functions.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Calls</th>
<th>Jumps</th>
<th>I. Calls</th>
<th>I. Jumps</th>
<th>C. Jumps</th>
<th>Rets</th>
<th>.text (KB)</th>
<th>.newtext (KB)</th>
<th>(×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>30888</td>
<td>24778</td>
<td>3896</td>
<td>4442</td>
<td>126876</td>
<td>22306</td>
<td>1047</td>
<td>5146</td>
<td>12.88</td>
</tr>
<tr>
<td>bzip2</td>
<td>1100</td>
<td>1050</td>
<td>170</td>
<td>152</td>
<td>7342</td>
<td>874</td>
<td>55</td>
<td>268</td>
<td>70.71</td>
</tr>
<tr>
<td>gcc</td>
<td>110122</td>
<td>64532</td>
<td>8916</td>
<td>15680</td>
<td>380920</td>
<td>45410</td>
<td>3225</td>
<td>15290</td>
<td>10.32</td>
</tr>
<tr>
<td>mcf</td>
<td>276</td>
<td>216</td>
<td>44</td>
<td>78</td>
<td>1300</td>
<td>250</td>
<td>12</td>
<td>57</td>
<td>202.98</td>
</tr>
<tr>
<td>gobmk</td>
<td>23548</td>
<td>14946</td>
<td>3550</td>
<td>3480</td>
<td>117378</td>
<td>20918</td>
<td>1488</td>
<td>6520</td>
<td>5.39</td>
</tr>
<tr>
<td>hmmer</td>
<td>8020</td>
<td>4942</td>
<td>556</td>
<td>666</td>
<td>28924</td>
<td>4106</td>
<td>277</td>
<td>1279</td>
<td>22.56</td>
</tr>
<tr>
<td>sjeng</td>
<td>2566</td>
<td>2338</td>
<td>256</td>
<td>658</td>
<td>12236</td>
<td>1570</td>
<td>132</td>
<td>604</td>
<td>36.17</td>
</tr>
<tr>
<td>libquantum</td>
<td>1094</td>
<td>758</td>
<td>94</td>
<td>146</td>
<td>3376</td>
<td>812</td>
<td>40</td>
<td>184</td>
<td>93.73</td>
</tr>
<tr>
<td>h264ref</td>
<td>7124</td>
<td>6518</td>
<td>1782</td>
<td>2000</td>
<td>47850</td>
<td>6318</td>
<td>520</td>
<td>2441</td>
<td>16.23</td>
</tr>
<tr>
<td>omnetpp</td>
<td>33578</td>
<td>10032</td>
<td>3830</td>
<td>1782</td>
<td>51642</td>
<td>14326</td>
<td>635</td>
<td>3225</td>
<td>13.49</td>
</tr>
<tr>
<td>astar</td>
<td>912</td>
<td>552</td>
<td>162</td>
<td>160</td>
<td>3314</td>
<td>750</td>
<td>39</td>
<td>184</td>
<td>92.52</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>115154</td>
<td>58678</td>
<td>39392</td>
<td>14630</td>
<td>307122</td>
<td>58678</td>
<td>39392</td>
<td>14630</td>
<td>7.60</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>32798</td>
<td>33370</td>
<td>9816</td>
<td>9012</td>
<td>189384</td>
<td>32458</td>
<td>1735</td>
<td>8435</td>
<td>9.77</td>
</tr>
<tr>
<td>libgcc.s.so.1</td>
<td>2158</td>
<td>2514</td>
<td>374</td>
<td>484</td>
<td>12862</td>
<td>1740</td>
<td>112</td>
<td>538</td>
<td>9.70</td>
</tr>
<tr>
<td>libm.so.6</td>
<td>5450</td>
<td>8870</td>
<td>874</td>
<td>892</td>
<td>21760</td>
<td>7406</td>
<td>277</td>
<td>1268</td>
<td>9.51</td>
</tr>
<tr>
<td>libstdc++.so.6</td>
<td>22456</td>
<td>10418</td>
<td>43008</td>
<td>43000</td>
<td>10418</td>
<td>43000</td>
<td>10418</td>
<td>43000</td>
<td>9.53</td>
</tr>
</tbody>
</table>

Table 3.1: Statistics of Multiverse rewritten binaries and libraries.
Note that the global mapping only needs to be inserted for executables (i.e., only executables have `.globalmapping`, while all executables and shared libraries have their own `.localmapping`). Specifically, we define the global lookup and mapping to start at the constant address `0x7000000`, which places it below all the sections of most binaries. For unusual binaries with a different layout, we can place it at a different constant address if necessary. Shared libraries will need to call the global lookup function, but since we place it at a fixed address, it does not need to appear in the shared libraries; all the dynamic libraries will call the same global lookup function address because they know it will be mapped there at runtime. This is not restrictive, as we can simply rewrite all the libraries again if we need to change the global mapping to a different address.

### 3.5 Evaluation

In this section, we report our evaluation results. We first report how we evaluate effectiveness in §3.5.1 and then report the MULTIVERSE performance overhead in §3.5.2. For our benchmarks, we used all 12 SPECint 2006 benchmark programs. We also had to rewrite the shared libraries used by the benchmarks. We did not test with the SPECfp benchmarks because we did not focus on rewriting Fortran programs, of which there are several in SPECfp. However, in theory, our rewriter should work on Fortran programs as well. Our test machine runs Ubuntu 14.04.1 LTS, and has an Intel i7-2600 CPU running at 3.40GHz, with 4GiB of RAM.

#### 3.5.1 Effectiveness

We first demonstrate the effectiveness of MULTIVERSE’s implementation by comparing the output of the original and rewritten binaries. By showing that all rewritten binaries produce identical output to the original, we can be confident of the correctness of the implementation of our design. To this end, we executed both the rewritten version and the original version
of the corresponding benchmark, and compared their output. All the rewritten binaries run correctly, producing the same output as the original program. We did not attempt to exhaustively run all the branches of the two versions and simply used the same configuration to run them.

Table 3.1 summarizes the rewriting statistics, including the binaries and libraries we had to rewrite for the SPEC benchmarks. One interesting detail is the similarity in size overhead for each of the text sections; they all increase in size between 4–5 times. In most x86 binaries, instructions are on average a little over 3 bytes (Blem et al., 2013), so we speculate that may explain this consistent size increase. For every instruction on average, it may only take an offset of 3-4 bytes to encounter an offset that was already assembled in a previous starting offset (i.e., every 4–5 bytes). We will investigate the implications of this in future work.

The .newtext sections shown in the table do not include the local mapping, which is always 4 times larger than the original due to the fact that we must store 4-byte entries for every byte offset in the text section. In addition, every binary also contains the slightly more than 4MB global mapping. It would be possible for us to allocate the global mapping to .bss in future work and eliminate this static file overhead, but for now we fill the space with 0xffffffff bytes. The effect of this can be seen in the last column of Table 3.1. The size overhead for 429.mcf looks remarkably high because the original binary is very small, and the fixed overhead of the global mapping dominates the rest of the code. Therefore, for large applications, this increase will be less noticeable and the percent increase in size will be much less. This means that as the size of the initial binary increases, size overhead will approach the increase in size of the new text section plus the local mapping, which averages to only around 9 times (4-5 times for .newtext, plus 4 times for the local mapping). Also notice that this pattern is demonstrated in the shared libraries we rewrote; since we do not need to include a global mapping in an .so, the overhead is lower and more consistent.

Real-World Binaries. We also tested Multiverse with other real-world software to demonstrate its effectiveness. First, we rewrote all the binaries in the GNU Core Utilities,
which contain the implementations of utilities found on all Unix-like systems. This provides a diverse set of utility applications. We also tested Multiverse by rewriting a number of other applications, including a graphical browser, web server, and graphical game, among others. In total we rewrote 126 binaries and 77 libraries (the applications shared many common libraries) comprising a total of 54MB. All worked as expected.

3.5.2 Performance

We also measured the runtime overhead of our rewritten binaries. We ran the SPECint benchmarks 10 times each, both on the original binaries and our rewritten binaries. We took the averages from the benchmark results.

**Rewriting Performance.** We first rewrote both the benchmarks and all their required shared libraries. As shown in the first bar in Figure 3.5, a few benchmarks had very high overhead, especially 471.omnetpp and 483.xalancbmk. This is because of our very generic handling of control flow during the rewriting. Since we make very few assumptions (discussed in §3.1.1) about the instructions in a binary, this sometimes results in surprisingly high overhead. Both 471.omnetpp and 483.xalancbmk are C++ applications, and therefore we suspect the high overhead results from the use of C++ features that become very expensive.
after being rewritten. In addition, frequent calls to library functions require a more expensive call to the global lookup, so this may be a factor. However, the other benchmarks all have less than 100% overhead, and most are below 50%. The average runtime overhead when rewriting the main binary and libraries was 60.42%.

We then investigated the impact of some optimizations. In particular, since the contents of libraries are often known, it is acceptable in some use cases to rewrite only the main binary. Therefore, we decided to implement the other approach we discussed for solving C4: only rewriting the main binary (and leaving the libraries unmodified), and treating callbacks as a special case. This requires a list of library functions that take callbacks, since all callback parameters must be rewritten to point to their corresponding addresses in .newtext. Since it is difficult to compute this list automatically, we populated it manually. With this approach, we were able to reduce overhead, in some cases significantly (e.g., 456.hmmer and 471.omnetpp). The results are shown in the second bar in Figure 3.5, and average overhead was 34.17%. This also allows the global lookup to be omitted, shrinking the size of the rewritten binary in addition to improving overhead. Rewriting the main binary makes sense in many use cases; sometimes the libraries do not need to be instrumented, or are more trustworthy than the main binary.

In most binaries, PIC uses the get pc thunk function to get the code address. We found that if we added the extra assumption that code would never attempt to get its own address without using get pc thunk (a reasonable assumption for well-behaved x86 binaries), we addressed C5 far more efficiently. This is a significant optimization because we no longer need to rewrite all call and return instructions to push and translate old addresses. The effect of this change is clearly shown in the third bar in Figure 3.5. Average overhead with this optimization was 8.29%. This demonstrates how a few well-chosen assumptions can result in vastly improved performance. Therefore, in the future we can add settings for other common patterns in binaries to improve practical performance when certain properties of a binary are known, while keeping the core rewriter generic enough to handle almost any binary.
It is important to emphasize that writing only the main binary and removing generic PIC rely upon significant assumptions. Specifically, rewriting only the main binary assumes knowledge of all callback arguments for functions, and removing generic PIC assumes that the only PIC in the binary is the thunk. Thus, these two optimizations will not work for certain binaries. However, we demonstrated these optimizations to show the performance improvements they would provide, and to show the potential for adding assumptions in cases in which it is safe to do so. Our core rewriter does not make these assumptions.

**Instrumentation Performance.** Making no changes when rewriting a binary is of limited utility. Binary instrumentation is a much more interesting application that Multiverse facilitates, since we can insert arbitrary code around any existing instruction. We implemented a straightforward instrumentation API to add assembly before any instruction.

Since many tools already perform binary instrumentation, we decided to compare the performance of our instrumented binaries with PIN ([Luk et al. 2005](#)), a widely-used framework for dynamic binary instrumentation from Intel that does not use any heuristics to dynamically disassemble and instrument a binary. We chose PIN due to its popularity and its avoidance of heuristics when rewriting. The goal of this evaluation is to show that the cost of our mapping lookups compares favorably with the state-of-the-art, and thus that instrumenting with our tool is practical.
We decided to compare the overhead of a simple instruction counting instrumentation. Pintool has example tools for instruction counting, and for our evaluation we selected the example Pintool that inserts a call to increment a counter for every instruction. Pin has more efficient strategies for instruction counting, such as inserting instrumentation at the basic block level and incrementing by the number of instructions in each basic block, but our tool does not currently use a basic block abstraction. Therefore we had to compare instrumentation of individual instructions, resulting in higher overhead than would normally be expected from an instruction counting Pintool. However, this demonstrates a case in which a static approach is preferable: When each individual instruction requires greater analysis, there is greater advantage in performing the analysis offline.

We wrote our own MULTIVERSE script to insert almost identical assembly to that inserted by Pin for each instruction. (Our insertions are not quite identical to Pin’s due to optimizations Pin implements when inserting code, making our inserted assembly slightly less efficient.) We rewrote the SPECint benchmarks with our tool for the main binary with all libraries, as well as with our two optimizations (rewriting only the main binary and rewriting the main binary without generic PIC). Similarly, we used the instruction counting Pintool that instrumented all instructions, and we also created a modified version to only instrument the main binary. We compared the performance by taking the average of running each benchmark 10 times and computing the overhead relative to the results from the unmodified benchmarks.

The results are shown in Figure 3.6. MULTIVERSE outperformed Pin in many of the benchmarks, and in four cases (400.perlbench, 403.gcc, 462.libquantum, and 483.xalancbmk), Pin’s performance was substantially worse than MULTIVERSE. This is likely because Pin uses dynamic instead of static instrumentation; any new code encountered by Pin must be analyzed and instrumented on the fly. For benchmarks such as 400.perlbench or 403.gcc, there are likely many new paths that are encountered through-
out execution, whereas some of the benchmarks with similar performance between Multiverse and PIN may have more loops that PIN can instrument once, leaving only the analysis code to run every subsequent loop. Thus, the additional overhead of Pintool’s runtime instrumentation code is probably the cause of such high runtime overhead.

Pintool does run slightly faster for a few benchmarks (e.g. 401.bzip2 and 464.h264ref). These are likely the result of PIN’s superior optimization of inserted code. For the inserted Multiverse assembly, we saved and restored flags before and after every inserted set of instructions. PIN, on the other hand, was able to analyze instructions as it encountered them, inserting code to save and restore flags only when necessary based on analysis of the instrumented code. Therefore, once analysis code is actually inserted, the resulting assembly produced by Pin should be faster than that produced by Multiverse. (Manual analysis of instrumented assembly showed various optimizations performed by PIN.) Since we did no analysis to optimize our inserted code, we have the opportunity to perform static optimizations in the future and increase our performance. Regardless, Multiverse’s performance is already promising.

Another interesting result is that the performance improvements from our heuristic optimizations were less significant in most benchmarks when compared to the overhead introduced by the instrumentation. This shows that despite the higher overhead of Multiverse without heuristics, it can be practical in certain instrumentation contexts and can be used for instrumenting binaries that do not obey common assumptions without introducing unacceptable overhead when compared to existing production tools.

3.6 Security Applications

There is potential for many interesting security applications with Multiverse. Binary rewriting is a foundational technique for increasing security in programs without source available, and Multiverse makes this more practical for arbitrary binaries by making
Figure 3.7. Percent runtime overhead for the benchmarks instrumented with a shadow stack. rewriting without heuristics possible. Because of this, and to show the potential of the framework, we used Multiverse to implement a shadow stack.

Shadow stacks are used to protect return addresses on the stack by allocating a separate memory region for the shadow stack, and inserting code that saves return addresses to the shadow stack whenever a function is called. When a function returns, the inserted code either checks whether the return address in the stack and shadow stack match, or simply overwrites the address in the stack with the one stored in the shadow stack. This ensures that an attacker cannot overwrite return addresses in the stack during ROP attacks. Shadow stacks can therefore be considered a form of backward-edge CFI (Gu et al., 2017). We implemented an overwriting, no-zeroing, parallel shadow stack (Dang et al., 2015).

Implementing a shadow stack with our framework was quite straightforward, requiring only the insertion of instructions for every `call` and `ret` instruction and allocating shadow stack memory. Since we rewrite `call` instructions in Multiverse to a `push/jmp` pair that pushes the original return address, we must insert our code after the `push` but before the `jmp`. We insert the following two instructions for each `call`, which writes the return address on the top of the stack into the parallel shadow stack:

```
    pop [esp + (shadow_stack_offset - 4)]
    sub esp, 4
```
Multiverse rewrites \texttt{ret} instructions into \texttt{pop/jmp} instructions, in which the return address at the top of the stack is popped and passed to the lookup function to determine the rewritten jump target. Therefore, we simply insert two instructions directly before the rewritten \texttt{ret} code to overwrite the return address on the stack with the corresponding shadow stack contents (hence an \textit{overwriting} shadow stack):

\begin{verbatim}
add esp, 4
push [esp + (shadow_stack_offset - 4)]
\end{verbatim}

We rewrote the SPECInt benchmarks and their libraries without either of our optimizations, because the shadow stack does not work when only rewriting the main binary. Some functions in \texttt{libc} call code in the main binary, and since \texttt{libc} is not rewritten when we use our optimizations, it does not push the return address on the shadow stack. Therefore, when the main binary returns to \texttt{libc}, there is no entry in the shadow stack and the program crashes. This is not a limitation of our framework, but rather an implementation challenge for shadow stacks if one does not intend to instrument all calls. Therefore, for our shadow stack proof-of-concept, we focus only on our general approach of rewriting everything.

\textbf{Shadow Stack Performance.} We ran the SPECInt benchmarks 10 times and computed the overhead relative to the results of the unmodified benchmarks. The results are shown in the second column of Figure 3.7. Performance results are similar to our framework with no instrumentation, with an average increase of 11.64\% when compared to the overhead in the first column (the overhead of Multiverse with no instrumentation, which is the same as the first column in Figure 3.5). The benchmark with the highest increase over no instrumentation was 483.xalancbmk, with an increase of 44.49\%, which was also the benchmark with the highest overhead for all shadow stack implementations in (Dang et al., 2015). Several other benchmarks have a very low increase in overhead, such as 429.mcf (0.19\%) and 473.astar (2.44\%); and one benchmark, 456.hmmer, was in fact faster (-4.02\%). Speed improvements are reported for several benchmarks in the reference implementation (Dang et al., 2015) as well, so this is not too surprising.
The overhead difference over our baseline rewriter for SPECint is more significant than the overhead shown in [Dang et al. 2015]; however, the data presented there is for SPECint benchmarks compiled with -O3, which the paper claims is marginally faster than the default -O2 optimization level we used. In addition, the reference implementation ([Dang et al. 2015]) assumes knowledge of the correct assembly, and inserts the instrumentation before the object files are assembled, yielding the lowest instrumentation cost possible. Finally, that implementation instruments function prologues instead of call instructions; we instrument call instructions because we do not have reliable information about function entry points, which may result in more instrumentation (and less code locality). These differences likely contribute to the higher increase in overhead for MULTIVERSE when compared to a near-optimal shadow stack implementation.

We also decided to implement the same type of shadow stack in PIN and compare the performance. We attempted to implement the shadow stack as close as possible in behavior to our implementation in MULTIVERSE. However, the inserted code was written in C++ instead of assembly, and we used the PIN API to access the value of esp since it has a different value when our inserted analysis code accesses it directly. The results are shown as the third column in Figure 3.7 and the performance is significantly slower than MULTIVERSE’s implementation. While we do believe that the PIN implementation can be optimized, the PIN API appears to make it difficult to directly access register values in an original instruction’s context as efficiently as assembly code directly inserted before that instruction. This may be one reason for the significant overhead.

3.7 Limitations and Future Work

Our current implementation of MULTIVERSE is merely a proof of concept. While our general approach can support x64 applications and other operating systems, our prototype currently
supports only x86 Linux ELF executables and .so files. In this section, we discuss some of the limitations of our system and outline how we plan address them in future work.

Supporting x64 applications mainly entails changing the assembly language of the rewritten instructions and lookup functions, but there are a few more significant changes needed. For example, in x64, instructions can directly access the contents of the rip register, so PIC does not require thunks. This changes the way PIC must be handled by the rewriter, since all references to rip must be modified to accommodate the position of the new code. Our progress on x64 support is almost complete.

Our system has several aspects that can be optimized. Since our priority was generality, we do not address special situations in which we could optimize away some of our more expensive runtime computations, especially when rewriting both .so files and the main binary. Meanwhile, in optimizing our mapping data structures for speed, we have made the tradeoff of additional space overhead. As shown by the optimizations we have already performed, we expect that future refinements will yield gains in both speed and size.

In addition, while we have developed a fundamental building block for rewriting binaries, we have demonstrated only one concrete application. However, as discussed at the end of §3.5.2, our prototype instrumentation framework using MULTIVERSE facilitates instrumentation, such as counting the number of instructions executed. In addition, we have demonstrated a practical example of instrumentation with a shadow stack. We will expand on this instrumentation ability and its use cases in future work.

Other applications of MULTIVERSE, such as binary hardening and other transformations that alter some of the original instructions in a binary, may require minor changes to be compatible with our mapping and lookup strategy, but MULTIVERSE should not impose significant challenges to implementing known techniques such as SFI. In fact, the mapping could potentially be used to assist enforcement for some security policies. We are working on applications such as these for future work.
3.8 Conclusion

We have presented MULTIVERSE, the first static binary rewriting tool that can correctly rewrite an x86 COTS binary without using heuristics. It consists of two fundamental techniques: superset disassembly that completely disassembles the binary code into a superset of instructions that contain all legal instructions, and instruction rewriting that is able to relocate instructions to any other location by interposing all the iCFTs and redirecting them to the correct new addresses. We have implemented MULTIVERSE atop a number of binary analysis and rewriting tools (e.g., CAPSTONE, pyelftools and pwntools), and tested with SPECint 2006. Our experimental results show that MULTIVERSE is able to rewrite all of the testing binaries and the runtime overhead for the new rewritten binaries is still reasonable. Comparison with another solution without heuristics (dynamic instrumentation) also shows that the static instrumentation enabled by MULTIVERSE can achieve better average performance, and our shadow stack implementation shows that MULTIVERSE can be used for actual security applications.
Dynamically generated code, while undesirable from a security perspective, is valuable for solving certain problems. For example, every modern browser contains dynamically generated code within its JavaScript engine, as just-in-time (JIT) compiled code is crucial for browser performance on modern websites. Another useful application of dynamically generated code is binary packers like UPX (Markus F.X.J. Oberhummer and Reiser, 2018), which self-unpack compressed executables at runtime, allowing for a small compressed executable that only expands in memory.

Unfortunately, the ability to exploit dynamically generated code is sufficiently powerful that it raises serious threats that attackers might inject their own code through these code generation mechanisms. In addition, the risks exposed by dynamically generated code introduce entire new classes of attacks, such as JIT spraying (Blazakis, 2010) against JIT compilers.

Control-Flow Integrity (CFI) (Abadi et al., 2005) has been found to be an effective defense against many forms of control-flow hijacking attacks, theoretically including the types of attacks effective against runtime generated code. There has been extensive research into providing CFI protections at both the source and binary level, but unfortunately current state-of-the-art CFI defenses are actually not effective for runtime-generated code (Xu et al., 2019).

Solutions have been developed to harden JIT compilers, such as with RockJIT (Niu and Tan, 2014), which is designed specifically for use with JIT compilers, and not arbitrary dynamically generated code. In addition, it was evaluated on a single JavaScript engine, and requires changes to that engine’s code and some understanding of its internal workings.

Designing a rewriting framework for a specific type of generated code, such as JIT compilers for a specific language, leads to assumptions about the nature of the generated code
and restricts general applicability. It ignores the many ways runtime-generated code is used in software today. JIT compilers extend beyond just JavaScript and the JVM, and there are now JIT-compiled engines for almost any popular language that isn’t natively compiled. Binary packers are not just used for installers or malware, as some software uses dynamically decompressed modules to save space. Even more seriously, some applications are self-patching, hooking calls in their address space to change function behaviour at runtime (Xu et al., 2019). Therefore, there is a need to investigate what abstractions and assumptions are required to support generic dynamically generated code, and where the line must be drawn for performance and security.

In order to harden arbitrary dynamically generated code, we introduce Miniverse, a framework for intercepting, rewriting, and hardening most types of dynamically generated code regardless of the source.

4.1 Background and Overview

Fundamentally, any approach attempting to harden software that produces dynamically generated code must pair some form of binary rewriter with the original program, as it is impossible to statically determine what dynamic code an arbitrary program may generate. Therefore, at runtime the generated code must be intercepted and hardened. However, the choice of rewriter requires care.

A binary rewriter can be static, in which the rewriting is done before execution, or dynamic, in which rewriting is done at runtime. Static rewriting has the benefit of performing heavier analysis of the code prior to execution, resulting in lower runtime overhead than dynamic rewriting solutions, which intercept and alter code as it is executed. The weakness of static rewriting is its imprecision, as it is impossible to determine certain code properties prior to execution. As dynamically generated code is obviously produced at runtime, it may seem obvious to attempt to rewrite the generated code with a dynamic rewriter.
However, recent advances in static rewriting have made “static” rewriting of dynamically generated code possible. By embedding a static binary rewriter into the program, we can rewrite any dynamically generated code once, which then can be run multiple times; the initial rewriting overhead only happens the first time it is generated. If the dynamically generated code is further changed, it can be re-rewritten, and the previous generation of rewritten code can be freed. This cleanly separates the generated code into insecure freshly generated code that needs rewriting and rewritten secure code that can be safely executed. MINIVERSE explores the challenges of producing such a framework, and the requirements for correctly intercepting and rewriting dynamic code.

4.1.1 Challenges

Intercepting Dynamically Generated Code

Handling arbitrary dynamically generated code poses some challenges. While code must obviously be generated before it is executed, knowing when a program is finished with code generation is impossible until it actually attempts to execute the code. In addition, self-modifying code may theoretically change its own bytes as it executes, blurring the line between generating and executing the code. This is the worst-case scenario for protecting dynamically generated code, as every memory write could potentially invalidate the next instruction to be executed.

To more efficiently rewrite dynamically generated code, we choose to support only programs whose generator code is not in the same memory page as the dynamic code it generates. Programs that violate this constraint might crash. This supports most non-malicious forms of self-modification, but not tricky self-modifying code that changes bytes out from under itself (e.g., for self-obfuscation). This admits most benign uses of dynamic code generation, and allows us to focus on splitting code generation into a generating phase and an executing phase.
A related challenge is dealing with memory regions that are both writable and executable. This is strongly discouraged by security professionals, as it allows for the possibility of code injection attacks, but unfortunately it is still used in commercial software. However, this can be dealt with by the observation that there is no need to allow a program to set a memory region as both writable and executable even if it wants to; if we intercept any attempt to set memory permissions, we can force any memory region allocated as writable and executable to be only writable. This allows us to choose whether a dynamically generated code region is currently *generating* and is writable, or is *executing* and is read-only. If a code region is in the executing phase, we can redirect any attempts to execute it to our rewritten, hardened version of the code region. The first time a program attempts to execute a writable code region, we rewrite it immediately and then jump to the corresponding location in our hardened code. If the program later attempts to write to a read-only code region, we then set it back to writable, remove our now obsolete hardened code, and jump back to the original code attempting to modify that region. This cycle can continue for as many code generations as the original program wants.

The approach of intercepting attempts to write to read-only code or execute writable but not executable code requires a mechanism to actually be able to intercept such attempts. We address this by implementing our own signal handler that is triggered whenever a segfault occurs. This covers both of these cases and allows us to catch any attempt of the program to modify or execute its dynamically generated code.

### Rewriting Dynamically Generated Code

In the case of JIT compilers, improved performance is the primary motivation for dynamically generating code. As JIT compilers are finely tuned to produce fast assembly code, we must be careful not to severely impact performance; JIT compilation must still benefit performance. However, we must also provide solid security primitives to provide for effective SFI and CFI. Therefore, we must keep performance in mind when solving rewriting challenges.
One significant challenge in rewriting dynamically generated instructions is handling indirect control flow instructions; these instructions jump to an address determined at runtime. Traditional static rewriting approaches must determine where these original addresses are found in the binary and change them to point to their new values, but this is imprecise. We instead can rewrite the instruction into a sequence of instructions that do a runtime lookup of the old address to the new address using a lookup table containing valid jump targets. However, performing even a simple lookup can be expensive when done on every indirect control flow instruction.

At first, we followed the approach used by STIR [Wartell et al., 2012b] and REINS [Wartell et al., 2012a] for translating addresses at runtime. This approach uses the original code as a lookup table, overwriting the original bytes at a target with the new address of the rewritten target. This used 5 bytes for each target address by tagging the address with an 0xf4 byte, which encodes the hlt instruction and is illegal in userspace. However, this has issues with target addresses closer than 5 bytes apart, and may overwrite inline data in the original code bytes.

It may seem unlikely for there to be inline data in runtime generated code, and that there should not be a risk of overwriting any inline data, but we in fact encountered that exact situation in a real-world JIT compiler. Non-dynamic JIT compiler code sometimes reads the direct contents of an immediate stored in a mov instruction within the dynamically generated code, as part of code that deals with a form of data relocation. This means that overwriting the original bytes is dangerous, and we cannot directly use that approach.

We therefore decided to store the lookup entries at a fixed offset from all dynamically generated code addresses instead of directly on top of them, which has a similar overhead but does not interfere with the original code. This also means that if we rewrite everything with the same lookup mechanism, we do not need to tag addresses at all, as all valid lookups will be of old addresses, and any invalid addresses (such as from an attack) will either cause
a segfault on attempting to perform the lookup, which we will catch in the segfault handler, or can be caught by the CFI protections. For the purposes of rewriting only dynamically generated code, however, we do tag whether an instruction should be looked up or not using the lower bits of addresses (since permissible targets are word-aligned in our model, and therefore the lowest-order bits are unused) in order to handle jumps back to un-rewritten static code. We also currently tag overlapping targets to avoid looking them up, since there is no space to store more than a single byte in the worst case. In this case we fall back to a slow but more robust lookup method.

However, storing the lookup entries at a fixed offset does present the new challenge of managing to reliably allocate memory at a single fixed offset from every dynamic code region. Fortunately, the dynamic code generation we have examined mostly allocates a few large regions early on and does not allocate more and more over time.

Another challenge is encountered when dealing with two specific cases of indirect control flow: call and return instructions. In x86 assembly, the only way to obtain the value of the instruction pointer (eip) is with a call instruction, which pushes the address of the following instruction on the stack before jumping to the target of the call. This is used in position-independent code (PIC) to find data at a fixed offset from the code, and is unfortunately also found in some real-world dynamically generated code. This unfortunately means that simply changing the target of a rewritten call is insufficient; the original code will use the return address pushed by the call in an address calculation, and if the address points to rewritten code, the resulting address will be incorrect and cause a crash.

We address this challenge by rewriting all call instructions to push the original return address that the original call would have pushed, and rewriting return instructions to look up the new address from the old return address like other indirect control flow instructions, as was done in Multiverse (Bauman et al., 2018). This has a performance penalty, as it impacts the hardware’s branch predictor for function calls, but is faster than in Multiverse due to our use of a more efficient lookup mechanism.
4.1.2 Scope and Assumptions

MINIVERSE is intended to provide code hardening primitives to any application that generates code at runtime, without knowledge of the code generation mechanisms. Therefore, MINIVERSE does not need access to source code to rewrite generated code. Our proof-of-concept implementation currently uses source code information to insert the hooks needed to trigger the rewriter in the first place, but this is merely an implementation convenience and can be avoided in the future by performing traditional static binary rewriting prior to the dynamic rewriting phase.

While MINIVERSE can handle arbitrary generated code, our prototype only currently supports applications that do not register their own signal handlers. Since MINIVERSE registers its own segfault handler, this can interfere with other signal handlers. Future implementations can support signal-handling applications by intercepting any attempts to register a signal handler and inserting MINIVERSE’s handler before the application-provided handlers in the handler chain.

In order to provide proper security policy enforcement, we assume that if deployed, MINIVERSE will be paired with a compatible CFI mechanism for all non-dynamic code. Without this, the original non-dynamic code could be exploited. However, we only rewrite dynamic code in our evaluation of MINIVERSE, in order to avoid conflating its overheads with those of any other defenses with which it is paired.

For our current implementation, we focus on the Linux platform. We also assume that the dynamic code generation is non-malicious and uses the conventional Linux syscalls and their corresponding POSIX functions for memory allocation (mmap) and setting memory protections (mprotect). Our approach can handle direct system calls, but this requires instrumenting all int 80 and syscall instructions to check for the corresponding system calls, which our current prototype does not yet implement.
4.1.3 Overview

As shown in Figure 4.1, **MINIVERSE** is comprised of two major components, both contained within the target application’s address space: a dynamic code interposition layer and a binary rewriter. The interposition layer contains memory protection hooks, a dynamic code region mapper, and a signal handler, while the rewriter contains a superset disassembler, code transformation component, and security policy component.

Before program execution, the interposition layer must be initialized. First, functions that change memory protections are hooked with wrapper functions. Crucially, this prevents unsafe code from ever executing, as we can intercept all attempts to set a memory region as executable. By never allowing dynamically generated code to be set as executable, we can enforce that the only dynamic code that is run is the code we generate.

Our dynamic code region mapper tracks all the dynamically generated code regions and maps them to our rewritten versions. This allows us to track arbitrary, independent code regions of different sizes, and map addresses from those regions directly to instructions in our rewritten code regions.
We register our signal handler to be called whenever a segfault occurs. Since we prevent generated code from being set as executable, this will catch any attempt to run generated code, which we then redirect to our rewritten, safe version using our region mapping.

The rewriter’s foundation is the superset disassembler, which disassembles from every offset in a generated code region, extracting all possible instruction sequences from the region’s bytes while eliminating duplicate sequences and invalid instructions. The results from the disassembler are passed to the code transformation component.

The code transformation component converts original instructions to their rewritten versions while following the security policy provided by the security policy component. The code transformation component is mainly concerned with redirecting control flow, both to enforce security policies and to ensure all rewritten jumps point to their corresponding rewritten targets.

The security policy component defines legal jump targets for indirect control flow instructions. This policy may be obtained by analysis of the generated code or a static policy. The particular implementation of this component are out of scope for this work, as Miniverse is simply intended to provide the primitives required to enforce security policies.

4.2 Detailed Design

4.2.1 Dynamic Code Interposition

In order to securely redirect control from and rewrite dynamically generated code, we provide an interposition layer to generically control attempts to generate and execute code. Whenever the program attempts to change memory permissions, we detect it with our memory protection hooks and add dynamic code regions to our region maps. Then, when the application attempts to execute its generated code, we catch the resulting segfault and redirect control to our rewritten version using our region maps.
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);

int mprotect(void *addr, size_t len, int prot);

Figure 4.2. Function signatures for mmap and mprotect.

Memory Protection Hooks

Before code is initially generated, an application must first allocate the memory it will write to. If it allocates the memory with the W+X permissions set, then it will be obvious ahead of time that the program may attempt to execute that memory in the future. However, it is considered dangerous to allocate memory that is both writable and executable (which W⊕X/DEP prevents unless a process specifically requests otherwise), so well-behaved and non-malicious dynamic code generators (such as JIT compilers) are more likely to toggle a region’s permissions between writable and executable as it modifies the region. Regardless of the approach used, we ultimately need to detect when a program is attempting to create an executable region or set a region to be executable, and prevent this.

There are a limited number of ways that a program may allocate memory, and fewer that allow the memory to be executable. Regardless of any higher level library functions that are used, eventually the application must perform a system call to request memory to be allocated by the kernel. We therefore wrap the mmap and mprotect functions with our own custom functions that mask off the PROT_EXEC permission (a flag that may be set in the prot argument of the functions as shown in Figure 4.2), preventing any memory allocations performed by the application from being executable. We do not interfere with the dynamic loader’s memory allocations, so dynamic libraries work as expected. If we wished to, we could also intercept those allocations, which would allow us to rewrite dynamically loaded libraries on the fly. However, here we are focusing on application code.
The `mmap` function maps an address range into a process's address space. This memory may be backed by the contents of a file, or may be anonymous memory with no backing file. The calling code may specify a desired base address for the mapped memory, or may allow the kernel to choose a random base address. This is essentially the core mechanism used by Address Space Layout Randomization (ASLR) to load shared libraries at random addresses (PaX Team, 2003). However, for our purposes we simply mask off the `PROT_EXEC` permission if the `prot` argument has both `PROT_WRITE` and `PROT_EXEC` set. Therefore, we do not currently prevent an application from mapping a file as executable directly. The `mprotect` function changes the permissions of existing memory regions, which allows a program to set the code it has just generated in a writable region to be executable.

For memory regions set to be both writable and executable, either via an initial `mmap` or later by an `mprotect` call, then it is impossible to know ahead of time when the program is finished generating code and will attempt to execute it. While we do prevent the program from setting any memory regions as executable, we will not know when to rewrite the code until the program attempts to execute it, and the program may continue to make changes to the code after it initially executes. It is even possible that the code may be self-modifying, which is especially pernicious for any system attempting to protect the generated code. However, it is possible to override the memory permissions to keep memory regions as writable until they are executed, rewrite the regions, and switch the original region to not writable or executable. Then, if code ever attempts to modify a rewritten region, we would catch the resulting segfault and mark the region as needing to be rewritten again. Code that performs this way would suffer a significant performance penalty under this approach, due to segfaults on both attempts to execute and write to the code, but would be handled correctly.

However, for well-behaved dynamic code generation that uses `mprotect` to switch its generated code between being writable or executable (which is done by the JIT compilers in our experiments), we can actually rewrite a full region as soon as the program attempts to
set it as executable. Then, when it tries to jump to the generated code later, we can simply redirect it to the already rewritten version. Therefore, in our mprotect wrapper, we call the rewriter as soon as mprotect is called with the PROT_EXEC permission. Then, if the program needs to change the generated code and calls mprotect for that memory region again, this time with PROT_WRITE, we can detect this in our wrapper, mark the region in our region mapping as not yet rewritten, and free the old rewritten code.

**Dynamic Code Region Mapper**

A program that dynamically generates code may separate the generated code into multiple non-contiguous regions in memory. We must therefore keep track of each generated code region in order to rewrite them and redirect control flows to the rewritten version of the corresponding region.

From our tests, the number of generated code regions in the programs we have evaluated is small (less than a dozen). Therefore, we maintain an array of region structs containing the address and size of the original and rewritten regions, as well as a pointer to a mapping between the two. This could be converted to a more efficient data structure for a hypothetical program that would spawn many dynamic code regions if such a program were to be found.

The mapping component is called by either our mmap/mprotect hooks (when a region is set to executable), or by our signal handler (when the program attempts to execute a generated code region that is not yet rewritten). It in turn calls our binary rewriter, which generates both the rewritten version of the code and the mapping between the original and rewritten code. The mapping itself is simply an array of pointers (4 byte values for 32-bit addresses) corresponding to every byte in the original code region. The mapping is therefore exactly 4 times the size of the original region. This mapping, however, is only used for the initial lookup when the signal handler is called. All indirect control flow lookups within the rewritten code are performed quite efficiently, as discussed in 4.2.2.
Signal Handler

For the purposes of evaluating MINIVERSE, we do not rewrite the non-dynamic code in a binary. This presents us with the problem of transferring control between original non-dynamic code and rewritten dynamic code. The original code is unaware of our rewritten code. One possible solution is to rewrite function pointers at the source, but that raises issues with position-independent code, as well as the fact that identifying and tracking pointers is a challenging problem in its own right.

We therefore take the approach of using hardware signals to detect when the program attempts to execute a dynamically generated code address. We register as the segfault (SIGSEGV) handler, which is called whenever any code in the process triggers a segfault. This occurs when code attempts to jump to instructions in the dynamically generated regions that we have forced to be non-executable, but it also may occur in the case of program error, such as a null pointer dereference.

We can distinguish valid code regions from invalid code regions by searching for the faulting address in our dynamic code region mapping; any address within one of our recorded regions is valid and can be translated to a rewritten region, while any other address is invalid (a program bug) and we can abort the program. It is important to note that all valid addresses for MINIVERSE will be non-executable addresses, and we know these regions to be dynamically generated code. Therefore, for this instance, the only difference that could occur between the original program and the program with MINIVERSE would be if a bug in the original program that would have originally jumped to unmapped memory instead jumped to a valid code region. However, this would still end up jumping to code that we control, with our security policy enforced.

If an address is identified as a valid dynamic code region, we check whether it is rewritten and rewrite it immediately if not. Then we look up the corresponding address in our rewritten code and set that to be the returning address from the segfault handler. When the handler
returns, control is returned to that address as if nothing had happened prior to the segfault, with all registers and the stack unchanged. The program then continues executing as if it had always jumped to the rewritten code.

4.2.2 Binary Rewriter

MINIVERSE’s binary rewriter is called by the dynamic code interposition layer to translate dynamically generated code to a safe version that is run instead of the original code. Because this must all occur at runtime as code is generated, it must be lightweight and fast. The entire binary rewriter component resides in the address space of the host application it is rewriting.

Superset Disassembler

We use the superset disassembly technique described in MULTIVERSE (Bauman et al., 2018), in which every offset in a region is disassembled and the resulting instruction sequences are then trimmed to eliminate redundancy and stitched together.

Our disassembler needs to be small and fast, and does not require all the details about instructions. In fact, detailed information is only needed for the subset of instructions that change control flow; for all other instructions we only need to know the instruction’s length.

Disassembly proceeds by starting at each offset and then performing normal linear disassembly until one of three conditions is met: disassembly reaches the end of the code region, an already-disassembled offset is encountered, or a bad instruction is encountered. Disassembling starting from each offset generates instruction sequences that eventually result in an instruction address that is identical to an address already disassembled in a previous pass, and continuing to disassemble would produce duplicate entries. Therefore, when an already-encountered offset is found, the disassembler returns a special jump instruction that
the code transformation component uses to stitch partially overlapping instruction sequences together.

Upon encountering an unknown/illegal instruction, the disassembler indicates that the current instruction sequence is invalid, which the code transformation component can use to eliminate such sequences from the generated code.

**Code Transformation**

The code transformation component is the heart of MINIVERSE. It takes the original instructions returned by the superset disassembler and transforms them into hardened instruction sequences that provide the primitives required for CFI and SFI policies.

The primary mechanism to sandbox instructions is to divide instructions into 16-byte aligned chunks, and only allowing indirect jumps to the start of a chunk. This is done by masking off indirect jump targets before the jump, thereby restricting jump destinations. This prevents an attacker from jumping into the middle of an instruction, and prevents them from escaping the sandbox; all indirect jumps are preceded by this masking operation.

Most instructions can be passed through the rewriter unmodified. The instructions that we are concerned with are control flow instructions and instructions at identified jump target addresses. Instructions identified as jump targets must be aligned to the start of a chunk, but otherwise are unchanged. Therefore, there are three major categories of control flow instructions: direct control flow instructions, indirect control flow instructions, and return instructions (which are a special case of indirect control flow).

Direct control flow instructions include conditional jump instructions (such as jne) and direct call/jumps. These are the most straightforward to rewrite because the target offset is encoded into the instruction. When performing initial instruction translation, we record relocation entries for these instructions, as they may jump forward into not-yet-rewritten code at the time we first encounter them. After initial rewriting, we then go through these
relocation entries and update their offsets to jump to the new rewritten target. If an instruction is a call, then we must convert the call into a jump and insert an extra push instruction so that we push the old return address that the original call would have pushed on the stack. This is necessary for code that uses return addresses to calculate offsets to data.

Indirect control flow instructions require the insertion of additional instructions, as it is impossible to determine where the instruction may jump to until it actually runs. Therefore, we insert instructions before the control flow to take the original address (which points to old, un-rewritten code), and look up the new rewritten target. We need an extremely fast lookup mechanism, as there are many indirect control flows in a binary. Therefore, we store the target address in a code-sized lookup table at a fixed offset (that we can calculate in advance to ensure there is free space) from the original target. Then, the instruction simply adds original address + offset and reads the bytes at that address. We encode in the lower bits whether that offset is a valid target, but not for security, as we then mask off the address before jumping to it; this ensures that even if an attacker compromised the lookup table, the control flow is forced to be chunk-aligned. We mark whether an offset is valid in order to distinguish old addresses (which need to be looked up) from rewritten or external static addresses (which must not be looked up). If any target addresses overlap, we mark the target as invalid, which means the lookup simply jumps to the old address instead of the new address. This allows our segfault handler to catch the resulting segfault and look up the address the slow but robust way.

Return instructions must be rewritten to pop an old address from the stack, look it up similarly to indirect control flow instructions, mask off the resulting address, and jump to it. The process, while similar to other indirect control flow instructions, requires some slightly different code; a call instruction is rewritten to what is essentially a push; jmp pair, while a return becomes pop; jmp. Extra work must be done for return instructions that take an immediate value to adjust the stack pointer, as we must perform the stack pointer adjustment manually.
Security Policies

MINIVERSE is intended to provide the security primitives required to enforce CFI and SFI policies. Therefore, we do not perform CFG analysis or identify function entry points. While we do have to identify jump targets for indirect control flow instructions, (as we insert lookup entries for every identified jump target), this in itself does not enforce a security policy, as lookup entries can be faked by an attacker.

Our policy enforcement comes after a lookup. We do not trust the address obtained from the lookup. In MINIVERSE, we mask off the bottom bits of all target addresses after looking them up so that all targets must fall at the start of a generated code chunk. Since no instructions are allowed to span across a chunk boundary in the generated code, masking off addresses prevents any indirect jump from jumping into an instruction sequence we do not control; the start of any chunk is also the start of a rewritten instruction, and therefore any attempt to escape our sandbox by controlling indirect control flow addresses will be prevented. Additional policies can be added by performing additional checks on addresses, and security enforcement comes from validating whether the destination address is allowed.

4.3 Implementation

MINIVERSE is written in about 2000 lines of C with some inline assembly and raw byte sequences used in instruction translation. This includes several test cases we wrote to reproduce common patterns we encountered with dynamically generated code.

Both of MINIVERSE’s major components, the dynamic code interposition layer and the binary rewriter, must first be placed into the target application that we wish to protect. This could be achieved by modifying the application binary, but for MINIVERSE we embed all our code into the target application at compile time. To do this, we statically link MINIVERSE in the host program at compile time, add the compiler flags `-Wl,-wrap=mmap`
\texttt{-Wl,-wrap=mprotect} to be passed to the linker in order to wrap the \texttt{mmap} and \texttt{mprotect} functions, and manually insert a single call in the \texttt{main} function to register the segfault handler. This is all that is required to add MINIVERSE to a program, but we are working towards making it possible to insert MINIVERSE directly into a binary without access to the source code, as there is no fundamental dependency on the source code.

In contrast to the disassembler in MULTIVERSE \cite{Bauman2018}, our disassembler is written in C and embedded in the process’s address space so that it may disassemble instructions dynamically at runtime. It uses the udis86 library \cite{Thampi2014} as its underlying linear disassembler due to its small size and lack of dynamic memory allocation.

\subsection{Evaluation}

We evaluated MINIVERSE on popular JIT compilers for two well known languages: LuaJIT \cite{Pall2019} for Lua, and Spidermonkey \cite{MozillaFoundation2019} (the JavaScript engine of Firefox) for JavaScript. Both of these JIT compilers generate native code from scripting language code at runtime. We inserted MINIVERSE into both and ran the resulting binaries on test suites and benchmarks for the JIT compilers’ respective scripting languages.

Since the dynamic code generation for JIT compilers is done for performance reasons, performance must be better with MINIVERSE than it would be without the dynamic code generation. This is easy to compare for JIT compilers, because they often have a way to run code in interpreted mode only, without using the JIT compiler at all. In order for MINIVERSE to be practical, performance must lie somewhere in between purely interpreted code and JIT-generated code. If a language’s non-JIT interpreter is faster than its JIT compiler with MINIVERSE, then it would be better to simply use the interpreter, as a pure interpreter does not have any of the dangers of runtime code generation.

We therefore evaluated the performance of the set of benchmarks used by the LuaJIT team to compare LuaJIT to the official Lua interpreter, which are 29 Lua scripts that per-
Table 4.1. Runtime performance of LuaJIT with and without Miniverse.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LuaJIT -joff (ms)</th>
<th>LuaJIT (ms)</th>
<th>LuaJIT w/ Miniverse (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>array3d</td>
<td>4820</td>
<td>250</td>
<td>320</td>
</tr>
<tr>
<td>binary-trees</td>
<td>3550</td>
<td>2170</td>
<td>57760</td>
</tr>
<tr>
<td>chameneos</td>
<td>4020</td>
<td>4060</td>
<td>3770</td>
</tr>
<tr>
<td>coroutine-ring</td>
<td>950</td>
<td>930</td>
<td>950</td>
</tr>
<tr>
<td>euler14-bit</td>
<td>13700</td>
<td>1090</td>
<td>1340</td>
</tr>
<tr>
<td>fannkuch</td>
<td>24660</td>
<td>4550</td>
<td>6310</td>
</tr>
<tr>
<td>fasta</td>
<td>17140</td>
<td>6980</td>
<td>11680</td>
</tr>
<tr>
<td>k-nucleotide</td>
<td>10000</td>
<td>4980</td>
<td>322710</td>
</tr>
<tr>
<td>life</td>
<td>730</td>
<td>490</td>
<td>670</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>20680</td>
<td>2450</td>
<td>3560</td>
</tr>
<tr>
<td>mandelbrot-bit</td>
<td>27400</td>
<td>1730</td>
<td>4520</td>
</tr>
<tr>
<td>md5</td>
<td>32280</td>
<td>1230</td>
<td>1640</td>
</tr>
<tr>
<td>nbody</td>
<td>10550</td>
<td>1370</td>
<td>2070</td>
</tr>
<tr>
<td>nsieve</td>
<td>5880</td>
<td>2060</td>
<td>2180</td>
</tr>
<tr>
<td>nsieve-bit</td>
<td>14080</td>
<td>1090</td>
<td>1300</td>
</tr>
<tr>
<td>nsieve-bit-fp</td>
<td>12280</td>
<td>2620</td>
<td>4220</td>
</tr>
<tr>
<td>partialsums</td>
<td>3220</td>
<td>1730</td>
<td>19840</td>
</tr>
<tr>
<td>pidigits-nogmp</td>
<td>12810</td>
<td>2750</td>
<td>850</td>
</tr>
<tr>
<td>ray</td>
<td>10460</td>
<td>1390</td>
<td>2030</td>
</tr>
<tr>
<td>recursive-ack</td>
<td>640</td>
<td>190</td>
<td>330</td>
</tr>
<tr>
<td>recursive-fib</td>
<td>6720</td>
<td>1020</td>
<td>1810</td>
</tr>
<tr>
<td>revcomp</td>
<td>1300</td>
<td>1030</td>
<td>47910</td>
</tr>
<tr>
<td>scimark-fft</td>
<td>40630</td>
<td>4400</td>
<td>5490</td>
</tr>
<tr>
<td>scimark-lu</td>
<td>34560</td>
<td>2400</td>
<td>3620</td>
</tr>
<tr>
<td>scimark-sor</td>
<td>23190</td>
<td>1990</td>
<td>2100</td>
</tr>
<tr>
<td>scimark-sparse</td>
<td>23270</td>
<td>3540</td>
<td>4640</td>
</tr>
<tr>
<td>series</td>
<td>2020</td>
<td>1680</td>
<td>38330</td>
</tr>
<tr>
<td>spectral-norm</td>
<td>17250</td>
<td>2220</td>
<td>2280</td>
</tr>
<tr>
<td>sum-file</td>
<td>690</td>
<td>690</td>
<td>660</td>
</tr>
</tbody>
</table>

form a variety of algorithms and common scientific computing tasks. We ran the benchmarks with the same arguments as the LuaJIT team, running the interpreter with JIT-compilation disabled (via the -joff flag), with JIT-compilation enabled, and with JIT-compilation enabled with Miniverse. All these benchmarks ran identically with and without Miniverse. We followed the precedent of the LuaJIT team, running each benchmark three times and selecting the fastest result. Our test machine runs Ubuntu 14.04.4 LTS with an Intel i7-2600 CPU running at 3.40GHz, with 8GiB of RAM.

Table 4.1 shows the raw performance numbers for LuaJIT with JIT disabled (-joff), LuaJIT, and LuaJIT with Miniverse, while Figure 4.3 shows a comparison in terms of per-
Figure 4.3. Overhead of MINIVERSE on Lua benchmarks. Values represent how many times faster each benchmark is than with LuaJIT’s JIT compiler turned off. Higher is better, and negative values mean the benchmark ran N times slower than with JIT compilation disabled.

Performance relative to LuaJIT with JIT disabled. The figure shows how many times faster (or slower, for negative values) a benchmark ran relative to only the interpreter. As we hoped, most of the benchmarks still performed better than interpreter-only mode even with MINIVERSE enabled. There were, however, a few anomalous benchmarks, notably binary-trees, k-nucleotide, partialsums, revcomp, and series, which had significantly worse performance with MINIVERSE. These benchmarks merit deeper investigation, as it is not clear why the performance impact on these is so significant. One possible reason could be an increased number of indirect control flows compared to other benchmarks. One other unusual benchmark is chameneos, which was slower than interpreter-only mode for LuaJIT, yet actually ran faster with MINIVERSE enabled. This unlikely result was consistent across multiple runs of the benchmark. However, the difference in performance is minor, and may be the result of few to no indirect control flows or better alignment.
Fortunately, the results with LuaJIT show that most of the benchmarks tested do indeed appear to run faster with JIT+MINIVERSE than with only the interpreter. This means that even with the additional overhead added by MINIVERSE, the speed gain added by converting scripting language bytecodes to native instructions is significant.

Our evaluation of the JavaScript benchmarks is still in progress, and is therefore reserved for future work. MINIVERSE already passes a majority of the JIT-specific tests in the Spidermonkey test suite, including many of the JavaScript benchmarks from benchmark suites such as Sunspider and V8, but some tests still fail. We believe this to mostly be a matter of some engineering details; improving the compatibility is an objective for our ongoing and future research.

4.5 Conclusion

In this paper we have presented MINIVERSE, a framework for hardening arbitrary runtime generated binary code. Utilizing superset disassembly to obtain all possible instructions, and then sandboxing those instructions to ensure control flow is constrained by a security policy, MINIVERSE provides a mechanism to bring SFI or CFI to runtime generated code, regardless of the source. Our evaluation of MINIVERSE shows it has decent overheads and good compatibility with popular and widely-used JIT compilers.
Address Space Layout Randomization, or ASLR (PaX Team, 2003), is a technique used to prevent attacks on software written in memory-unsafe languages. Prior to ASLR, an attack could perform code-reuse attacks such as return-oriented programming (ROP) (Shacham, 2007) attacks that utilize existing pieces of code (called gadgets) in the address space of a binary to launch an attack. By randomizing the base addresses of modules within a binary, it increases the difficulty of attacks that rely on knowing the location of that code. Exploits are made more difficult because an attacker does not know where to find the code that they wish to execute.

However, since ASLR was first introduced, new attacks were discovered, such as JIT-ROP (Snow et al., 2013) and BROP (Bittau et al., 2014), that allow an attacker to discover gadgets in a running binary even without initially knowing its memory layout. This is possible because of memory disclosure attacks, which allow an attacker to leak memory contents at runtime, and server daemons that restart crashed child processes, which share their memory layout with the parent and therefore can be crashed repeatedly in the efforts to determine memory contents. This can defeat even fine-grained randomization, since prior knowledge of the memory contents is not needed.

One response to these attacks has been an increased interest in repeated runtime re-randomization (coarse or fine-grained), which renders any previously discovered addresses useless after each randomization. While an appealing direction for runtime obfuscation of binaries against attacks, runtime re-randomization runs the risk of high overhead and inaccuracy when working without source. The high overhead, unlike the very low overhead of regular ASLR, deters adoption, while inaccuracy can lead to runtime crashes.
In this chapter, we examine why attempting to use superset disassembly and its related techniques to solve this problem is not feasible without tackling additional research problems in this area. We will discuss several other works that have already approached runtime re-randomization in various ways, what they have done to address the challenges in relocating code at runtime, the limitations of their approaches, and a fundamental limitation of address-translation-based runtime re-randomization that has not been examined in detail until now.

5.1 Background

One of the main questions that arises when considering runtime re-randomization is that of relocating the code itself. Simply moving code bytes from one place to another is possible if the code in question is compiled to be position-independent, but even position-independent code (PIC) has pitfalls. For example, PIC computes data addresses by retrieving its own address at runtime and adding a fixed offset to its data section, so data associated with a module, such as a shared library, must move with the code. Even accommodating for this, however, doesn’t address a problem central to relocating code: code pointers themselves.

At runtime, there may be code pointers in the stack (e.g., return addresses and local variables), the heap, and in global variables. When changing the position of code, all these pointers are no longer valid and will almost certainly immediately lead to a crash. Therefore, any solution attempting to relocate code must address this problem.

5.1.1 Pointer Relocation

A reasonable and obvious approach is to find and relocate all code pointers in memory whenever the code is moved. Unfortunately, perfectly distinguishing a pointer-like integer from an actual pointer in memory is not possible, and pointer tracking can become quite expensive.
One example of a work that performs pointer relocation is RuntimeASLR (Lu et al., 2016), which tracks pointers during runtime so that each pointer can be updated when code is relocated. This is achieved by using dynamic binary instrumentation, which has high overheads. RuntimeASLR avoids the impact of these higher overheads by only randomizing on fork, and removes the instrumentation on child processes. Therefore, RuntimeASLR defends a specific type of process (server daemons) from a specific type of attack (clone-probing attacks) and is not a generic randomization tool. Such an approach would result in far too high overhead.

Another work that relocates pointers, but without heavyweight pointer tracking, is that introduced by Giuffrida et al. (Giuffrida et al., 2012), which discusses runtime re-randomization as part of its OS-level randomization. It leverages compiler-level knowledge to identify certain categories of pointers, but even with this information, identifying some pointers requires cooperation from the programmer, as even at the C level, some cases may be undecidable. This approach does not extend well to binaries without debug information.

Therefore, pointer relocation can be accurate (with pointer tracking) but slow, or fast (with compiler assistance and heuristics) but inaccurate (without compiler/developer help). Therefore, without more developments in this area, we investigate a different approach.

5.1.2 Address Translation

While it seems obvious at first that instruction pointers must be relocated, there is an alternative approach. Instead of changing pointers directly, we can use a lookup table to translate old targets to point to the current location of the randomized code, and therefore changing code location will only require updating the lookup table, as the original pointers in memory can remain unchanged.

Shuffler (Williams-King et al., 2016) actually replaces all uses of pointers with entries into pointer tables, essentially converting all pointers in a process into indexes. Then, when
code locations are randomized, the contents of these pointer tables are randomized, but not the indexes in memory. However, to identify code pointers, it leverages relocation data from compiler debug symbols, meaning stripped binaries are unsupported.

ReRanz (Wang et al., 2017) attempts to directly translate original addresses to randomized addresses via trampolines, address translation entries at a fixed offset from each target basic block. Their approach needs to categorize returns and jumps into categories, and relies on correct identification of basic blocks, and therefore does still rely on correct disassembly.

5.1.3 Address Translation with Superset Disassembly

With superset disassembly, it is possible to obtain all possible instructions in an instruction sequence, allowing for the translation of any old address into a new address. This approach was used in Multiverse to avoid updating pointers to point to rewritten code. Such an approach can similarly be used for runtime re-randomized code, in which a mapping for instruction addresses from each byte offset can be used to translate old addresses into new ones. The appeal of this is that there is no reliance on disassembly heuristics to obtain correct disassembly and basic blocks; all original code addresses can be reliably translated to their randomized counterparts. This therefore avoids the uncertainties of previous translation-based approaches.

5.2 Fundamental Address Translation Limitations

Unfortunately, despite its apparent advantages, address translation suffers from a fundamental problem: since the original addresses do not change during randomization, an attacker can simply use old addresses, which will be automatically translated by the randomization framework to its randomized version of the code. Without relocating the pointers, the constant old pointer values are a vector for attack.
Previous works performing address translation are aware of this problem, but it remains unclear what benefit randomization adds beyond the (non-random) defenses such solutions typically use to prevent attackers from re-using old pointers. For example, Shuffler’s use of indexes removes raw code pointers from memory, providing a layer of abstraction. However, this implements a form of coarse-grained CFI, in regards to its use of tables of valid jump targets without restricting flow to those targets. Shuffler takes a different approach with return addresses, encrypting them by xor-ing them with a random value. However, for either kind of pointer, this first line of defense against attackers seems to be the most important, with an attack becoming possible if gadgets can be constructed from indexes, or if an attacker can break the return address encryption.

ReRanz does not use indexes, but instead translates old code pointers directly. However, it determines a subset of basic blocks that can be targets for jumps. Again, this implements coarse-grained CFI; the ReRanz paper admits that although the authors attempted to minimize the risks, code-reuse attacks are possible. Therefore, how much more effective is adding randomization on top of these defenses? Is the additional overhead worth it?

The above analysis indicates that any attack that would succeed against coarse-grained CFI should also succeed against these randomization solutions (although an attacker would first need to convert raw pointers to any encoding of valid targets a defense may have). In order for these randomization defenses to be effective, they need to block attacks that coarse-grained CFI would not be able to block in order to demonstrate they provide any additional benefit. Any address-translation-based randomization solution must demonstrate that it provides a security benefit over coarse-grained CFI (or any defense used to protect the translation mechanism), as otherwise they are simply reinventing CFI with an extra layer of complexity on top with no clear security benefit.

This discovery also demonstrates why superset disassembly is unsuitable for re-randomization; for all current works, defending the translation mechanism requires deciding on a subset
of “correct” disassembled instructions in order to determine safe targets for translation, which would defeat the purpose of obtaining every possible instruction sequence. Even if we were to use superset disassembly anyway, we argue that translation-based re-randomization is still an unsolved challenge that must be met first.

5.3 Conclusion

Runtime re-randomization is an appealing target use-case for superset disassembly. Although at first it appears quite promising, superset disassembly’s ability to translate any valid instruction address is not useful when it comes to re-randomization, as the address translation mechanism would automatically translate old non-randomized pointers, allowing an attacker to simply reuse the old pointers. Even with obfuscation, the original pointers cannot change between randomizations, so attacks are not mitigated. Attempting to restrict the possible targets for these translated pointers essentially results in a re-implementation of coarse-grained CFI. It still remains to be determined whether adding randomization on top of coarse-grained CFI provides much additional benefit, but either way, it brings back many heuristics and difficult research problems that using superset disassembly avoids. As it stands, we conclude that using superset disassembly is a counterproductive direction for addressing runtime re-randomization, as the question to address first is whether address-translation-based re-randomization actually provides any benefits.
6.1 **SgxElide**

SgxElide’s self-modifying enclaves encrypt enclaves in a content-agnostic way, but the process does require some developer involvement. Even if the process were made fully transparent by removing the requirement to manually insert a call to `elide_restore`, the enclave still would need to be freshly compiled with SgxElide’s components. For the expected use-case of protecting newly developed enclave code (as would be most likely with a newer technology like SGX), this is likely sufficient. However, it is plausible to imagine that in the future a developer might be able to reuse binary enclaves from other developers. Any alterations to the enclave would require re-signing, but if a developer wished to hide which third-party enclaves were being used, they could encrypt and re-sign a third party enclave as their own.

To enable source-free enclave encryption may require custom binary rewriting software to keep the enclave shared object working correctly for SGX. The `elide_restore` function would need to be inserted into the shared object, a call to it would need to be inserted at the enclave entry point, and it would need to be made conditional to only be called upon the first time the enclave is entered. This would be a useful future direction to allow for enclave encryption with only minimal changes to the toolchain, no changes to source, and in fact no need for source at all.

As mentioned in chapter 2, the security risks introduced by the self-modifying code can be further addressed. Other than the obvious need to add support for changing runtime permissions in SGX-v2 in order to prevent the enclave from being writable and executable, we could also harden the enclave with the same mechanism as that in Miniverse; since the self-decrypting enclave is producing runtime-generated code, SgxElide can benefit from the protections offered by Miniverse.
6.2 **Multiverse**

While **Multiverse** provides a good demonstration of being able to rewrite arbitrary binary code without using heuristics, it does suffer from some difficulties in writing some binaries. This doesn’t have to do with the rewriting component, but rather the limitations of the framework to handle arbitrary ELF files. Adding new segments to ELF files can actually be quite difficult, especially if the file is stripped and has no padding. Since some segments cannot be moved at all without changing the offset they are loaded at (which would break any code relying on data being at specific addresses), the file structure becomes very inflexible. The workarounds that were used in **Multiverse** to rewrite binaries still fail on unusual edge cases when attempting to create the final binary. One solution to this could be to use a custom loader, which can be specified in the ELF file. This was actually investigated, but was considered too complicated and outside our scope to pursue when implementing **Multiverse**. However, this may be a viable solution that is worth investigating further.

Another problem that was encountered was actually finding and rewriting all the shared objects required by some binaries. To avoid using any heuristics, all code in all libraries must be rewritten, but for dynamically loaded libraries during execution (and not just at program start), programs had to be run and a list of required libraries had to be compiled from multiple executions. This requires too much manual effort, and it could be useful to automatically attempt to extract and rewrite as many dynamic libraries as possible. However, some library paths were found to be hard-coded, meaning we must overwrite the original libraries with the rewritten ones in system paths, which requires root permissions and could interfere with normal non-rewritten processes. Rewriting these libraries could be improved by identifying and changing the paths to those libraries in the binary.

Another concern in **Multiverse** is that very large 32-bit programs with many libraries could run out of address space with the high space overhead of **Multiverse**. Since the
no-heuristic mode must rewrite all libraries, and shared libraries are loaded at random addresses, space is wasted by gaps between libraries that eventually become too small for a memory allocation, leading to such large programs being unable to run. For such complicated programs, using heuristics tends to break things, so this can make it impossible to rewrite some very large 32-bit programs. However, MULTIVERSE does now support x86-64, so this should now be addressed, but experiments in this regard have not yet been done due to the manual effort required in rewriting libraries.

Finally, MULTIVERSE only supports Linux, and it would be beneficial if it could be ported to work on other platforms as well. Windows is a notoriously difficult platform when it comes to binary rewriting, so many additional challenges would need to be overcome to be able to deal with the Windows environment.

6.3 Miniverse

While MINIVERSE already shows promising results with real-world dynamic code generation, there are still several unfinished edge-cases that must be addressed before a proper evaluation of MINIVERSE can be done. Our tests with Firefox’s JavaScript engine have resulted in a rewritten executable that passes many of the tests in its test suite, but it still fails on a number of tests. Many of the problems and solutions discussed in the design of MINIVERSE came as a result of making more of these tests pass, so there may still be some additional discoveries as we approach completion.

In order to demonstrate the general applicability of MINIVERSE, we also will need to evaluate MINIVERSE on more than just JIT compilers. We have already investigated rewriting UPX-compressed binaries, but this will require some additional engineering to handle the less-conventional ELF files produced by UPX. We may also seek out other software that dynamically generates code.
7.1 SGX

Binary Code Protection. The goal of code obfuscation is to disrupt analysis of the code and deter reverse engineering efforts. In general, there are three types of widely used binary analysis platforms: disassembler, debugger, and VM. Consequently, obfuscation techniques can be categorized into anti-disassembler, anti-debugger, and anti-VM. For each category, there exists a variety of techniques. For example in the anti-disassembler category, there exists the techniques of garbage code insertion (e.g., (Linn and Debray, 2003)), control flow obfuscation (e.g., (László and Kiss, 2009)), instruction aliasing (e.g., (Mudge et al., 1996)), or binary code compression and encryption (e.g., various packers such as UPX (Markus F.X.J. Oberhummer and Reiser, 2018)).

SGX offers a new way of protecting binary code by using the security provided by enclaves. With hardware-enforced security provided by SGX, existing disassembler, debugger, and VM-based reverse engineering techniques will no longer work.

SGX Applications. Since SGX holds great potential to solve challenging security problems (Anati et al., 2013), many efforts have started to explore the potential of SGX. Haven (Bau-mann et al., 2014), SCONE (Arnautov et al., 2016), and Graphene SGX (Tsai et al., 2017) run native applications inside SGX enclave without modification. VC3 (Schuster et al., 2015) demonstrated code and data confidentiality of MapReduce computation in the cloud. SGXRand (Chandra et al., 2017) mitigated the side channel leakage of data analytics with SGX by using data noise. Most recently, SGX-BigMatrix (Shaon et al., 2017) provided a framework to secure matrix operations for data analytics in the cloud.

Improving Security with SGX. SGX-LAPD (Fu et al., 2017) thwarts the controlled side channel attacks (Xu et al., 2015) via compiler extensions. T-SGX (Shih et al., 2017) defeats
them via hardware transactional memory (HTM) and compiler extensions, and Cloak \cite{Gruss2017} defeats cache side channel attacks against SGX with just HTM. SGX-Shield \cite{Seo2017} enables ASLR to defeat memory corruptions, and SGXBounds \cite{Kuvaiskii2017} provides memory safety for enclave programs.

7.2 Binary Rewriting

Rewriting binary code can be dated back to the late 1960s \cite{DeutschGrant1971}, when it was first used for flexible performance measurement. Later in the late 1980s and early 1990s, rewriting binaries for RISC architectures (e.g., Alpha, SPARC, and MIPS), in which code is well-aligned and instructions have a fixed length, became quite popular in applications such as instrumentation (e.g., PIXIE \cite{Chow1986}, ATOM \cite{SrivastavaEustace1994}), performance measurement and optimization (e.g., QPT \cite{LarusBall1994} and EEL \cite{LarusSchnarr1995}), and architecture translation \cite{Sites1993}. In contrast to RISC architectures, rewriting binaries for CISC such as x86 is much more challenging. We traced the earliest attempt to rewrite x86 binaries to Etch \cite{Romer1997}. In the following, we review prior efforts of x86 static binary rewriting and compare them with Multiverse. We here omit a survey of dynamic binary rewriting (e.g., PIN \cite{Luk2005}), since it encounters a different set of challenges.

Targeting instrumentation, profiling, and optimization, Etch \cite{Romer1997} made a first step towards rewriting arbitrary Win32/x86 binaries, potentially without any relocation entries or debugging symbols. However, the implementation details of Etch are not open and it is very likely that Etch used heuristics for disassembling, recognizing static memory addresses, handling callbacks and PIC. Instead of rewriting arbitrary x86 binaries, SASI \cite{ErlingssonSchneider1999} focused on rewriting only gcc produced binaries for SFI \cite{Wahbe1993}. Similarly, PittSFIELD \cite{McCamantMorrisett2006} and Native Client (NaCl) \cite{Yee2009} also require cooperation from compilers.
Unlike stripped binaries, object code contains rich information, such as where the code is located, and which data contains static memory addresses. Therefore, a number of efforts focused on rewriting x86 object code without heuristics. PLTO (Schwarz et al., 2001) and DIABLO (Put et al., 2005) are both examples that target program optimization and profiling. With debugging symbols (which is almost as informative as source code), Vulcan (Srivastava et al., 2001) (and PEBIL (Laurenzano et al., 2010)) can correctly rewrite x86 binaries without using any heuristics. Vulcan was later used in two security systems: CFI (Abadi et al., 2009) and XFI (Erlingsson et al., 2006). PEBIL was later extended and used in StackArmor (Chen et al., 2015) for stack memory protection.

BIRD (Nanda et al., 2006) is the first system that targets COTS binary rewriting. In particular, BIRD first uses compiler idioms and various assumptions to disassemble as many instructions as possible, and it further improves results with an on-demand runtime disassembler. Meanwhile, with a runtime exception mechanism, BIRD can also safely handle PIC and callbacks. In addition to regular applications such as profiling, BIRD has been used to harden a binary for foreign code detection.

SECONDWRITE (O’Sullivan et al., 2011) lifts the disassembled code into LLVM IR (Anand et al., 2013) to optimize the original binaries for better runtime performance by leveraging the strength of LLVM optimization. While SECONDWRITE can rewrite a binary without supplementary information (Smithson et al., 2013), it still uses heuristics for binary code disassembling and callback handling, and it also does not handle PIC in its current implementation (Smithson et al., 2013). It also inherits limitations of the LLVM IR, which cannot easily encode certain native code programs containing instruction sequences that LLVM compilers do not generate.

DYNINST (Bernat and Miller, 2011) is a framework that supports both static and dynamic binary instrumentation. While the published literature on DYNINST is unclear on whether it supports PIC and callbacks, our investigations showed that it uses heuristics to handle PIC
based on particular patterns. DYNINST has been improved and used in various applications, such as performance studies and binary hardening (e.g., PathArmor [van der Veen et al., 2015], TypeArmor [van der Veen et al., 2016], and CodeArmor [Chen et al., 2017]).

**Stir** [Wartell et al., 2012b] and **Reins** [Wartell et al., 2012a] preserve the original program’s functionality by tracking basic block addresses and then randomizing them or in-lining them with security logic (respectively) to mitigate attacks. Both systems rely on a *shingled disassembly* strategy that resembles our superset disassembly, but that applies imperfect machine learning heuristics to optimize rewritten code sizes [Wartell et al., 2014]. Subsequent work has leveraged these foundations to implement Opaque Control-flow Integrity [Mohan et al., 2015]. Callback support for these systems has recently been automated by the advent of Object Flow Integrity [Wang et al., 2017], but PIC support remains heuristic-based. Lookup table implementation in these systems is also less robust than **Multiverse**, since it adopts an encoding that assumes iCFT targets are at least 5 bytes apart—an assumption violated by many COTS binaries.

**Ccfir** [Zhang et al., 2013] leverages relocation information to disassemble program code. While Ccfir does not have any issues in recognizing addresses or handling PIC due to the use of relocation information, it still uses heuristics for disassembly and handling callbacks. Also assuming correct disassembly, **Bistro** [Deng et al., 2013] rewrites binaries for binary code hardening and reuse [Caballero et al., 2010; Zeng et al., 2013]. It also uses heuristics to handle PIC and callbacks.

To our knowledge, **BinCFI** [Zhang and Sekar, 2013] is the first system that can safely handle static memory addresses, PIC, and callbacks. While it also has attempts for better disassembly, BinCFI still cannot handle interleaved code and data well. **Psi** [Zhang et al., 2014] makes BinCFI more general as a framework for binary rewriting in various other applications such as profiling. **Uroboros** [Wang et al., 2015] makes binary disassembly reassemblable by using the same disassembling algorithm from BinCFI, but it still uses a
Table 7.1. Comparison w/ existing x86 binary rewriters.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Year</th>
<th>w/o Relocation</th>
<th>w/o (Debugging) Symbols</th>
<th>w/o Heuristics for PIC</th>
<th>w/o Heuristics for Callbacks</th>
<th>w/o Heuristics for Disassemble</th>
<th>Profiling</th>
<th>Optimization</th>
<th>Binary Code Hardening</th>
<th>Control Flow Integrity</th>
<th>Binary Code Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etch</td>
<td>(Romer et al., 1997) 1997</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SASI</td>
<td>Erlingsson and Schneider (1999) 1999</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PLETO</td>
<td>Schwarz et al., 2001 2001</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vulcan</td>
<td>Srivastava et al., 2001 2001</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Diablo</td>
<td>Put et al., 2003 2005</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CFI</td>
<td>Abadi et al., 2009 2009</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XFI</td>
<td>Erlingsson et al., 2006 2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PittSField</td>
<td>McCamant and Morrisett (2006) 2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bird</td>
<td>Nanda et al., 2006 2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NACL</td>
<td>Yee et al., 2009 2009</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pebil</td>
<td>Laurenzano et al., 2010 2010</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SecondWrite</td>
<td>O'Sullivan et al., 2011 2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DynInst</td>
<td>Bernal and Miller, 2011 2011</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>STIR/REINS</td>
<td>Warren et al., 2012 2012</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CoFI</td>
<td>Zhang et al., 2013 2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bistro</td>
<td>Deng et al., 2013 2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BinCFI</td>
<td>Zhang and Sokr, 2013 2013</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PST</td>
<td>Zhang et al., 2014 2014</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Uroboros</td>
<td>Wang et al., 2016 2016</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ramblr</td>
<td>Wang et al., 2017 2017</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multiverse</td>
<td>2018</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

number of heuristics to differentiate memory addresses and constant integers when relocating a binary. The recent extension of Uroboros (Wang et al., 2016) has been made to be a more general static binary instrumentation framework.

Most recently, Ramblr (Wang et al., 2017), built atop angr (Shoshitaishvili et al., 2016), has attempted to remove the static memory heuristics used by Uroboros (Wang et al., 2015). For example, Uroboros (Wang et al., 2015) assumes that code pointers in the data section are $n$-byte aligned and only point to function entry points or jump table entries. Ramblr (Wang et al., 2017) instead performs localized data flow and value-set analysis to recognize pointers and integers. Note that Ramblr and angr do not aim to solve other disassembly problems such as handling PIC and callbacks without using heuristics, which is the core focus of Multiverse.
Compared to all of the existing works, we notice that Multiverse is the first system that is founded on an approach starting with no heuristics in x86 COTS binary rewriting. It can be used in all existing applications, such as profiling, optimization, binary code hardening, CFI, and binary code reuse. A summary of our comparison is presented in Table 7.1.
CHAPTER 8

CONCLUSION

This dissertation has presented three works demonstrating approaches to modifying binary code. The first, SGXELIDE, does not bother attempting to disassemble the functions that it wishes to encrypt, but instead simply encrypts the raw bytes, avoiding the need to understand the content it modifies. The second, MULTIVERSE, introduces superset disassembly in the service of creating a general-purpose binary rewriter that avoids using heuristics during rewriting. The third work, MINIVERSE, extends the ideas from MULTIVERSE in order to bring security hardening to arbitrary dynamically generated code. Finally, this dissertation discusses the limitations of superset disassembly to show why it is not suitable for runtime re-randomization.
REFERENCES

0xe7, 0x1e (2014). An easy Linux crackme. https://exploit.ph/reverse-engineering/2014/05/11/an-easy-linux-crackme/


Intel Corporation (2016a). Intel Software Guard Extensions SDK for Linux OS. https://download.01.org/intel-sgx/linux-1.6/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.6_Open_Source.pdf

Intel Corporation (2016b). Intel(R) Software Guard Extensions for Linux OS. https://github.com/01org/linux-sgx


BIографICAL SKETCH

Ever since a young age, Erick Bauman was interested in computers. He first discovered programming with BASIC on his graphing calculator, and started taking programming classes as soon as he could. Always wanting to create something new, he found that programming was the perfect tool for creativity, and he chose to study computer science when he entered university. During his undergraduate career, Erick would create programs to help with non-programming classes, wanting to leverage the power of computing regardless of the topic. Upon graduation, Erick wanted to take his studies further and decided to pursue a PhD to expand his knowledge.
CURRICULUM VITAE

Erick Bauman
December 2019

Contact Information
Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.
Email: erick.bauman@utdallas.edu

Educational History:
B.A. in Computer Science, Southwestern University, Georgetown, TX, USA, 2013
M.S. in Computer Science, University of Texas at Dallas, Richardson, TX, USA, 2018
Ph.D. in Computer Science, University of Texas at Dallas, Richardson, TX, USA, Dec. 2019

Experience:
Research Assistant, University of Texas at Dallas, 2015 – Dec. 2019
Research Intern, Fujitsu Laboratories of America, Summer 2018
Teaching Assistant, University of Texas at Dallas, Fall 2014

Honors and Activities:
Louis Beecherl, Jr. Graduate Fellowship, University of Texas at Dallas, 2017 – 2018
Ericsson Graduate Fellowship, University of Texas at Dallas, 2016 – 2017
Excellence in Education Doctoral Fellowship, University of Texas at Dallas, 2015 – 2016
Excellence in Education Doctoral Fellowship, University of Texas at Dallas, 2014 – 2015
Excellence in Education Doctoral Fellowship, University of Texas at Dallas, 2013 – 2014
Phi Beta Kappa, Member, 2013
Association for Computing Machinery Student Chapter President, Southwestern University, 2011 – 2013
Pi Mu Epsilon, Member, 2012
Upsilon Pi Epsilon, Student Chapter President, Southwestern University, 2011 – 2013
Grogan Lord Award in Computer Science, Southwestern University, 2011
Dean’s List, Southwestern University, 2009 – 2013
Eagle Scout with Bronze Palm, 2008