Binary Shuffling: Defeating Memory Disclosure Attacks through Re-Randomization

by

David Williams-King

B.Sc. Honours Computer Science, University of Calgary, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate and Postdoctoral Studies
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

July 2014

© David Williams-King, 2014
Abstract

Software that is in use and under development today still contains as many bugs as ever. These bugs are often exploitable by attackers using advanced techniques such as Return-Oriented Programming (ROP), where pieces of legitimate code are stitched together to form a malicious exploit. One class of defenses against these attacks is Address-Space Layout Randomization (ASLR), which randomly selects the base addresses of legitimate code. However, it has recently been shown that this randomization can be unravelled with memory disclosure attacks, which divulge the contents of memory at a given address. In this work, we strengthen code randomization against memory disclosure attacks, in order to make it a viable defense in the face of Return-Oriented Programming. We propose a technique called binary shuffling, which dynamically re-randomizes the position of code blocks at runtime. While a memory disclosure may reveal the contents of a memory address (thus unravelling the randomization), this information is only valid for a very short time. Our system, called Shuffler, operates on program binaries without access to source code, and can re-randomize the position of all code in a program in as little as ten milliseconds. We show that this is fast enough to defeat any attempt at Return-Oriented Programming, even when armed with a memory disclosure attack. Shuffler adds only 10 to 21% overhead on average, making it a viable defense against these types of attack.
Preface

This thesis evolved out of a class project in Bill Aiello’s course CPSC 538W, “Online Privacy”, and from the author’s prior experience with malware and compiler toolchains. The class project was an early single-address-space relocation-based prototype that nevertheless could dynamically migrate functions (in the absence of function pointers), and was conceived and written in entirety by the author.

After this, the project proceeded with the author writing several prototypes using different technical mechanisms. The author has written all of the code. Bill Aiello and Mihir Nanavati advised the project and helped keep it going in a useful direction, with Patrick Colp and Kent Williams-King providing valuable feedback as well.

As for publications, Bill Aiello and Patrick Colp helped prepare a submission to Usenix Security 2014, which was rejected, and also worked towards a submission to ACM Computer and Communications Security (CCS) 2014, which we ultimately decided not to submit. We aim to resubmit this work to another conference in the future.
# Table of Contents

Abstract ................................................................. ii
Preface ................................................................. iii
Table of Contents .................................................... iv
List of Tables ........................................................ vii
List of Figures ........................................................ viii
Acknowledgments ..................................................... ix

1 Introduction ......................................................... 1

2 Background .......................................................... 4
  2.1 Executable Space Protection ................................. 4
  2.2 Return-Oriented Programming ............................... 5
  2.3 Threat Model ..................................................... 7

3 Design ................................................................. 8
  3.1 Architecture ..................................................... 8
    3.1.1 Location of Shuffling Mechanism: In a Separate Process . 9
    3.1.2 Burning a Core ............................................. 9
    3.1.3 What Shuffler Can Shuffle .............................. 10
    3.1.4 Heartbeat Operation ...................................... 11
4 Implementation ......................................................... 12
    4.1 The Userspace Shuffler Process ................................. 13
        4.1.1 Modifying the Target’s Memory ........................ 13
        4.1.2 When Do Writes Appear? Or, Cross-Modifying Code ...... 15
        4.1.3 Bootstrapping Shuffler (and Attach/Detach) ............. 18
    4.2 Code Discovery and Relocation ................................. 19
        4.2.1 How the Build Chain Uses Relocations ................... 19
        4.2.2 Code Discovery and Disassembly ......................... 20
        4.2.3 Relocating Code at Runtime ............................. 21
    4.3 Trampolines ..................................................... 22
        4.3.1 How to Update the Stack ................................. 22
        4.3.2 Trampoline Implementation ............................... 24
        4.3.3 Memory Protection ........................................ 25
    4.4 Completeness .................................................... 26
        4.4.1 Handling Stale Addresses ................................. 26
        4.4.2 Handling Shared Libraries ................................. 28
        4.4.3 Handling Threads and Forks ............................... 28
    4.5 Overall Operation ............................................... 28
5 Performance Evaluation ............................................. 30
    5.1 Shuffler Microbenchmarks ...................................... 30
        5.1.1 Memory Access Overhead ................................ 30
        5.1.2 Context Switch Overhead [on Ivy Bridge] ............... 31
        5.1.3 Code Injection Overhead [on Ivy Bridge] ............... 31
        5.1.4 Cache Trampoline Optimization [on Ivy Bridge] ........ 32
        5.1.5 Jump Table Optimization [on Ivy Bridge] ............... 32
    5.2 Macrobenchmarks on SPEC CPU2006 ............................. 33
    5.3 Running Nginx [on Ivy Bridge] ................................. 35
6 Security Evaluation .................................................. 36
    6.1 Memory Disclosure Attacks ..................................... 36
        6.1.1 Analysis of a Memory Disclosure Attack .................. 36
        6.1.2 Why the Attacker Has Insufficient Time .................. 37
6.2 Other Types of Attacks ............................................ 38

7 Related Work .................................................. 42
    7.1 Return-Oriented Programming ............................ 42
    7.2 Static Randomization ...................................... 42
    7.3 Just-In-Time Code Reuse Attacks ......................... 43
    7.4 Dynamic Randomization ................................. 44
    7.5 Control-Flow Integrity .................................. 44

8 Future Work and Discussion ................................. 46
    8.1 Shuffler Completeness .................................. 46
    8.2 Code Disassembly ...................................... 47
    8.3 JIT Code .................................................. 47
    8.4 Hot Patching ............................................. 48
    8.5 Shuffling an Entire System ............................ 48
    8.6 Shuffling Policy ......................................... 48

9 Conclusion ...................................................... 50

Bibliography ...................................................... 51

A Injected Code .................................................. 55
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Performance of reading 10GB of memory between processes.</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Function pointer overhead on a million calls.</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance improvement for jump table optimizations</td>
<td>33</td>
</tr>
<tr>
<td>5.4</td>
<td>SPEC CPU2006 average overhead</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1 Example ROP attack (executes a shell on 64-bit Linux). . . . . 6
Figure 3.1 Shuffler running on a separate CPU core. . . . . . . . . . . . 10
Figure 4.1 High-level architecture of Shuffler. . . . . . . . . . . . . . . 12
Figure 4.2 Remapping private memory as shared memory . . . . . . . . 14
Figure 4.3 Intel’s specification of cross-modifying code. . . . . . . . . . 16
Figure 4.4 Idiom for retrieving EIP (32-bit x86 code). . . . . . . . . . . 21
Figure 4.5 Transforming a call instruction with a trampoline. . . . . . . 23
Figure 4.6 Adding a shadow stack to form bookkeeping trampolines. . . 24
Figure 4.7 Trampoline assembly code (AT&T syntax). . . . . . . . . . . 24
Figure 4.8 Overall architecture of Shuffler. . . . . . . . . . . . . . . . . 29
Figure 5.1 SPEC CPU2006 . . . . . . . . . . . . . . . . . . . . . . . . . 34
Figure A.1 Trampoline code with bookkeeping. . . . . . . . . . . . . . . 56
Acknowledgments

My supervisors, Bill Aiello and Andy Warfield, helped me throughout my master’s program to learn to see and think about research problems at a higher level, and for this I am very grateful. Special thanks are due to Bill, who advised me on the Shuffler project right from its beginnings as a small class project, who put up with its many iterations, and who always had useful advice and kind words and time for lunch meetings. My labmate and closest lab-friend, Mihir Nanavati, helped advise the project and suggested useful directions, and was always willing to talk at length about anything I wanted; but, when Mihir masterfully disappeared to another continent right before each of our paper submission deadlines, Patrick Colp stepped in and helped out with the writing. Finally, my brother Kent, NSS-member-to-be, provided a great deal of advice in our many discussions about the project, informed by his encyclopaedic knowledge of Intel systems. This project would not have been possible without Bill, Mihir, Patrick, and Kent, and everyone else in NSS who made sure I had a good time and kept myself sane. On that front, thanks to my parents and all my friends at Green College as well, for making my time in Vancouver the best it could possibly be.

Thank you, all!
Chapter 1

Introduction

No one has yet figured out how to write code without bugs. In fact, today’s production software might contain as many as 50 defects per 1000 lines of code [21]. Many of these errors are exploitable by an attacker, particularly memory errors such as buffer overflows, use-after-free, and memory corruption. Even programs written in languages with managed memory or isolated in virtual machines are not immune, since bugs may be present in the language runtime [22] or virtual machine implementation [9].

Memory errors may allow an attacker to inject new data and code, examine the state of the running program, even hijack its control flow. There are many possible types of attacks, but we are concerned in particular with code reuse attacks, which leverage existing code to form an exploit. The most famous type of code reuse attack is Return-Oriented Programming [30], where control flow is directed through small gadgets which together are Turing-complete and can form any exploit.

One type of defense against memory errors is Address Space Layout Randomization (ASLR), which randomizes the base addresses of stack, heap, and libraries in a program. This prevents an attacker from hard-coding addresses and having a reproducible exploit. Unfortunately, one memory access vulnerability (which reports an address to the attacker) can allow the rest of a program’s layout to be deduced, since the entire code (or stack) segment is shifted uniformly. More recent techniques which perform fine-grained ASLR have been proposed, permuting and randomly placing functions, basic blocks, or even individual in-
Another class of defenses against memory error attacks and against Return-Oriented Programming (ROP) in particular is Control-Flow Integrity (CFI) [1,39]. CFI computes a control flow graph which contains all valid jumps and calls, and aims to prevent execution from straying outside the graph. Done properly, CFI should prevent any ROP attack, because the jumps used by ROP are not possible in a normal program execution.

Unfortunately, recent works have shown that both types of defense are quite vulnerable. Fine-grained randomization is performed only once when a program is loaded, and the memory layout remains static thereafter. So it is possible to mount a memory disclosure attack and gradually unravel the randomization until enough code is found to mount an attack [31]. This attack works in theory against any fine-grained randomization technique. Conversely, while the original CFI still remains a strong defense, it was prohibitively expensive; recent CFI techniques take shortcuts that approximate the control-flow graph in order to achieve better performance. For example, many CFI systems allow a return statement to jump to anywhere immediately following an indirect call statement. It has been shown that almost any approximation which does not enforce true CFI still provides enough gadgets for an exploit [10].

In this thesis, we resurrect fine-grained ASLR as a viable defense against code reuse attacks. We harden ASLR against memory disclosure attacks by observing that any memory disclosure can be mitigated through re-randomization, which may invalidate any previously discovered features of the address space. Our system, Shuffler, dynamically re-randomizes (or shuffles) code layout at runtime. The shuffling is performed at high frequency, as often as once per 10 milliseconds, which is fast enough to provide a strong defense against repeated memory disclosure attacks such as Snow et al’s attack [31].

Shuffler operates on commodity off-the-shelf binaries, without requiring source code or debugging information. Shuffler does not require any modifications to the operating system, but instead operates entirely in userspace. It may be embedded into the system loader, or track a process and all its children; Shuffler can even attach onto running processes and then detach later (providing, perhaps, stop-gap defense against a zero-day vulnerability). The shuffling rate can be adjusted.
dynamically according to observed system activity such as network overhead or suspicious process crashes.

We measured the performance of Shuffler on SPEC CPU 2006, and found an overhead of 10-21% on the applicable test cases. This seems appropriately small enough for our technique to be viable. Shuffler has also been tested on the web server nginx, which is perhaps a more realistic example.
Chapter 2

Background

When a running program is attacked, the attacker’s aim is usually to cause the program to run code it was not designed to run. Early stack-smashing attacks would simply exploit buffer overruns to write new code provided by the attacker directly onto the stack, and then overwrite a return address to point at the new code. Modern defenses such as executable space protection prevent new code from being written into memory, which leads to more sophisticated code reuse attacks like Return-Oriented Programming (ROP). We describe some background in detail below, leading to a description of ROP.

2.1 Executable Space Protection

Executable space protection refers to marking pages in a process’s memory region as non-executable, to prevent an attacker from running code in those memory regions. Historically, regions like the stack would have RWX permissions, allowing code injected through a buffer overrun to be executed immediately. This largely stems from x86 paging table permissions, which originally did not allow pages to be made readable without also being executable, until the introduction of the No-eXecute (NX) bit.[1]

[1]Read and write permissions have always been manipulable: bit 1 of each page table entry controls write protection, and bit 2 can be used to make a page completely inaccessible [18]. But the execute-disable (NX) bit is bit 63 and was only introduced in the first machines to use more than 32 bits for physical memory addressing (Pentium Pro and newer, anything with Physical Address Extension or PAE).
Now, most modern operating systems use the hardware NX bit, if present, to enforce executable space protection\(^{2}\) Some Linux-based systems even emulate the NX bit in software \([24, 27]\). This effectively prevents code-injection attacks from succeeding, although executables may still explicitly request RWX stacks or memory regions despite the dangers of doing so, e.g., for self-modifying code or through misconfiguration of the build toolchain \([14, 19]\).

### 2.2 Return-Oriented Programming

Return-Oriented Programming (ROP) \([30]\) is a class of attack that works even in the presence of executable space protection. The basic idea is to reuse fragments of code already present within a legitimate executable’s code segment, combining them into a malicious exploit (without writing any new code). The fragments of code (called gadgets) are normally very short and end in return statements (hence the name Return-Oriented Programming).

An attacker deploying a ROP attack begins by exploiting a memory error vulnerability, e.g., a buffer overrun that allows the current return address on the stack to be overwritten, and several words of memory to be written beyond that. When the current function goes to return, it instead jumps to an address of the attacker’s choosing, which is the first gadget to be executed. Each gadget runs a small amount of code and then executes a return statement; each return reads its target address and then pops it off the stack. In this way, the attacker can direct execution through a whole series of gadgets simply by writing their return addresses in sequence onto the stack.

The attacker can also place data onto the stack, and load it into registers by executing a gadget which pops data off the stack—for example, \texttt{pop \%rax ; ret}\texttt{,} which is a very common instruction sequence in function epilogues. After finding a number of gadgets in a program or library’s code, the attacker can characterize them by what side effects they have, what registers are modified, and what functions or traps are called. It is relatively straightforward to find a Turing-complete set of such gadgets, so that any desired exploit can be compiled down into a se-

\(^{2}\)Windows calls this Data Execution Prevention (DEP), OpenBSD calls it Write-XOR-Execute (W\(^{X}\)), and other systems like Linux just call it NX bit support.
quence of gadgets (using a constraint or SAT solver) \cite{12,28}. Even basic libraries that are always present, like libc.so or kernel32.dll, can be shown to contain a Turing-complete set of gadgets \cite{10,33}.

As an example, see Figure 2.1. Here the stack has been set up with the addresses of five gadgets and some data. The first two gadgets simply clear a register, then use a return statement to pop the next return address off the stack. Gadgets $g_3$ and $g_4$ load data values from the stack, or pointers into the stack. And gadget $g_5$ initiates a system call. The attack shown will execute system call 0x3b (execve) with the three arguments /bin/sh, 0, and 0 — in other words, it opens a shell, the holy grail for an attacker. A real attack would likely not have these gadgets available and may have to construct equivalent behaviour from many more gadgets, but this complete compromise of the target program would likely still be possible.

![Figure 2.1: Example ROP attack (executes a shell on 64-bit Linux).](image)

Finally, note that x86 and x86_64 employ dense, variable-length instruction sets. In other words, common or older instructions might have short one- or two-
byte encodings, while more recent and more complex instructions might be much longer (up to a maximum of 15 bytes [23]). And furthermore, the instruction set is dense, meaning that most sequences of bytes represent valid instruction sequences. Thus, an attacker can search for code beginning at any address, even in the middle of the original instructions, further increasing the chances of finding useful gadgets.

Thus, Return-Oriented Programming (and variations like Jump-Oriented Programming [3]) is a very powerful type of code reuse attack which can be carried out against modern systems. The defense most commonly proposed for ROP is Control-Flow Integrity, though recent CFI systems are too lax with their policies [10]. We claim to show that (dynamic) randomization-based techniques also show promise as ROP defenses.

### 2.3 Threat Model

In this work, we assume a user who is running a modern operating system with execution space protection, to handle traditional code-injection attacks. The operating system and system administrator are trusted. Shuffler provides protection to one or more userspace programs run by the user. We assume these target programs are well formed (e.g., compiler-generated code, which is organized into blocks that never issue call instructions into the middle of other blocks), and that Shuffler is able to discover the code in the target (e.g., through the use of the symbol table, discussed in Section 4.2.2).

We assume the attacker does not have direct access to the target system, but rather is situated remotely on the network, or perhaps may craft a self-contained exploit (such as a malicious Word document or PDF file). As in Snow et al [31], we assume an adversary with advance knowledge of a) a memory access vulnerability that can be invoked repeatedly, b) one valid code address, and c) a vulnerability which allows control-flow hijacking (in order to initiate execution of an exploit). Under these conditions, Snow et al have shown that any static randomization technique can be defeated; but their analysis does not extend to dynamic randomization, our proposed method of defense.
Chapter 3

Design

Our aim in this work is to resurrect fine-grained ASLR so that it becomes a useful defense against Return-Oriented Programming. Ideally, we could completely isolate each block of code within a target program, preventing an exploit from transferring control-flow between gadgets. This would defeat any code-reuse attack that needs code from multiple blocks. In reality, however, all code blocks within a process exist in the same address space and can always reach each other—assuming that the attacker knows where they are.

Thus, our approach is to dynamically move (or shuffle) code blocks from one address to another. This approximates the isolation ideal, and prevents an attacker from reliably jumping between code blocks. With continuous, high-frequency shuffling, discovering the location of a code block is of limited use to an attacker, because the block will not live at that address for long. Constructing a successful ROP attack in this environment is difficult.

3.1 Architecture

The goals of our system, Shuffler, are presented below.

1. **Security**: We should provide strong randomization guarantees without introducing additional attack vectors to the system.

2. **Deployability**: Shuffler should require minimal system modifications, and should provide protection for existing unmodified application binaries.
3. **Performance**: Programs run under Shuffler should approach native performance (in terms of wall clock time).

The architecture of our system, Shuffler, is presented below.

### 3.1.1 Location of Shuffling Mechanism: In a Separate Process

Our aim is to move the code around inside a target process while it is running. But it is dangerous to embed the shuffling mechanism inside the same process, where it may be vulnerable to memory disclosure attacks on the target. Since Shuffler must keep track of the shuffled locations of every code block at all times, its data structures are a prime target for memory disclosure attacks (because they provide an easy way to unravel the randomization).

This suggests two other designs: either store the shuffling data structures inside the kernel, or push the data structures into a separate user space process. Moving the shuffling into the kernel does provide the easiest access to page tables for easy page-granularity shuffling, direct page read/writes, and other mechanisms that would make shuffling easy to implement (and as a precedent, existing ASLR mechanisms are kernel-level). However, a userspace solution has the advantage of easy deployability to existing systems running stock kernels, and limits the potential damage an attacker can cause by attempting to attack Shuffler itself. We chose to design Shuffler as a userspace process. It runs with no additional special privileges, executing in userspace under the same user ID as the target process.

### 3.1.2 Burning a Core

The Shuffler process runs on a dedicated CPU core, “burning” the core to provide the security afforded by shuffling with minimal impact on other cores’ performance. As shown in Figure 3.1, the shuffling core runs in parallel to the real application, so that a single-threaded shuffled application now needs two cores.\(^1\)

Since burning a core allows the target application to run at nearly native performance, this seems a reasonable trade-off for security; modern CPUs have many

---

\(^{1}\) Actually, only a small portion of the shuffling CPU’s time is used, perhaps one fifth in our tests, so that one shuffling core would suffice to handle multiple application cores.
cores, and forthcoming models have even more, though it is uncertain whether applications can scale up to that many cores. Note that dedicating cores to specialized tasks to ensure good performance is not without precedent: for example, on mainframes, IBM provides extra cores solely for offloading Java Virtual Machines [17] so that Java performance will be closer to native performance.

Unfortunately, from the hardware’s perspective, since shuffling is happening from a different core, one processor is modifying the code of another processor. This is called \textit{cross-modifying code} and it is very slow: as is the case for self-modifying code, whenever code is modified at runtime, the processor must flush its pipelines and clear instruction caches. However, any case where code blocks are permuted must involve code modification, because embedded jumps are normally relative and must be updated. So this high cost cannot be avoided entirely. We mitigate it by generating new code asynchronously, a technique which is made possible by having Shuffler run on a separate core.

\subsection*{3.1.3 What Shuffler Can Shuffle}

Choosing to use a separate userspace process for Shuffler is excellent for deployability. Shuffler may be installed by a non-root user without needing a custom kernel, or even a reboot. We further extend this deployability by allowing Shuffler to begin shuffling in two ways:

1. Using a custom loader, which may replace the system loader to catch all
newly run processes.

2. Attaching onto any running process (and optionally detaching later).

In principle, we want Shuffler to be able to handle anything that a userspace process might do: load shared libraries, create threads, call fork/exec, etc. Supporting all of these features is a only matter of implementation; no major design changes would need to be made. We implemented as many as possible within our available time frame.

3.1.4 Heartbeat Operation

Shuffler operates in a loop, shuffling every function in turn of a target program. For security, Shuffler attempts to guarantee that each function will be moved within a given time period (e.g., 100 milliseconds). The loop runs periodically at this rate, and enters an error condition if at any point Shuffler cannot move all the functions within the time allotted.

The time period may be adjusted by the user, because higher shuffling rates provide greater security (in the sense that any code address remains valid for a shorter period of time), while lower shuffling rates provide better performance. This rate could even be adjusted as Shuffler is running, in response to any increases in suspicious network traffic or program crashes.
Chapter 4

Implementation

Figure 4.1: High-level architecture of Shuffler.

Figure 4.1 describes Shuffler’s operation at a high level. In brief, Shuffler runs as a separate userspace process, and attaches onto the target process with the ptrace API. Shuffler is able to inject code into the target and modify its memory directly (described in Section 4.1.1). Special trampolines are introduced for function calls (Section 4.3) to make shuffling easier. The blocks of code that make up
trampolines and the target’s functions are tracked within Shuffler, and randomly shuffled in the target inside sandbox regions of memory (see Section 4.2). Shuffler is implemented for 64-bit Linux on the x86_64 architecture.

### 4.1 The Userspace Shuffler Process

As mentioned in Section 3.1.1, we chose to run Shuffler in a separate userspace process. However, it requires a fair amount of engineering effort to achieve all the tasks that Shuffler needs to do from userspace. The main challenges are 1) getting read-write access to the target’s memory, and 2) ensuring that the target runs new code after it is updated (instead of stale cached code). The following sections describe how this is possible to implement from userspace.

#### 4.1.1 Modifying the Target’s Memory

The simplest way to manipulate another process’s state is through the ptrace process introspection API (designed for debuggers). As a first step, the Shuffler process attaches onto the target process with ptrace. This grants Shuffler the ability to read and modify saved registers, change the instruction pointer, intercept signals, and modify the target’s memory.\(^1\)

However, using ptrace to access the target’s memory is very slow, since the API requires that a separate system call be made to access each word of memory.\(^2\) One of the main tasks that Shuffler performs is to copy functions in order to shuffle them, so we need a faster mechanism.

Our main technique is to use shared memory maps. These allow portions of a process’s address space to be mapped to the same physical pages as another process’s address space (possibly at different virtual addresses). Once set up, this allows Shuffler to modify the target’s memory directly with ordinary writes. There are two instances where Shuffler uses shared memory maps:

1. **Sandboxes** are large empty regions within which code can be shuffled randomly. To create them, Shuffler injects code into the target which opens

---

\(^1\) Shuffler can only attach onto executables which are not setuid/setgid [34]. This is a security limitation of the ptrace API.

\(^2\) There have been attempts to improve this: the PTRACE_MULTI patch [11] to Linux provides one solution, but it is not in the mainline kernel.
Figure 4.2: How private mmap’ed regions are remapped as shared memory.

a temporary backing file, resizes it to the desired size, and mmaps it into the target’s memory. Shuffler can then map the temporary file into its own address space.

2. Shuffler also needs to map the existing code segments as shared memory maps, in order to shuffle (and erase) the code within. However, such regions have already been mapped as private memory maps (by the loader and shared library runtime). So, as shown in Figure 4.2, Shuffler a) injects code into the target which b) creates a temporary mapping and copies the data to this mapping, c) unmaps the original private region and maps it again as a shared region, then d) copies the data back in and unmaps the temporary region; finally, Shuffler can e) map the shared region into its own address space. The remapping operation is performed while the target is stopped to prevent any race conditions.

Shared memory maps allow Shuffler to access remote addresses with reads and writes, the same way as local addresses. This leads to a clean internal architecture.

---

3This is what POSIX shared memory objects were designed for, but they are a libc-level abstraction and difficult to use from injected code. The Linux implementation (see sysdeps/unix/sysv/linux/shm_open.c) just uses mmap and ftruncate anyway.
Other Methods of Accessing Memory

Other techniques for accessing shared memory are possible. For example, there is already a mechanism similar in spirit to our shared memory maps built into Linux: the /proc/pid/mem file, which provides direct access to a target’s memory. One might reasonably ask why we did not use this, since it is recommended by [11] and easier to use (it is the mechanism employed by GDB). However, while this file can be opened and read with syscalls like pread, it cannot be mmaped, which prevents the clean architecture allowed by shared memory maps. Using /proc/pid/mem is also significantly less efficient, as shown in our evaluation (Section 5.1.1).

One other option is to use the new pair of system calls introduced in Linux 3.2, namely process_vm_readv() and process_vm_writev(), which are especially designed to transfer data between process address spaces (and are essentially a realization of the PTRACE.MULTI proposal [11]). However, invoking these system calls requires a kernel context switch instead of simple writes; while the calls allow operations to be batched together by accepting a vector of requests, this creates additional overhead and complexity. These new system calls are also less efficient than shared memory maps (see Section 5.1.1).

Finally, shared memory maps allow a function to be moved with a single copy, while any of these other methods require two copies (because one endpoint of each transfer must be in the calling process). This is why we use shared memory maps, despite the additional complexity in setting them up.

4.1.2 When Do Writes Appear? Or, Cross-Modifying Code

Shuffler creates shared memory maps in order to write code to shared pages, and have that code appear in the target process. This process, where one processor generates code for another processor, is called cross-modifying code (a generalization of self-modifying code, where one processor generates its own code). Unfortunately, there are a number of caveats with cross-modifying code. First, it is very slow, because CPUs are not designed to have code modified without warning (caches need to be invalidated, pipelines cleared, etc). Second, there are timing issues because writes may be cached on one processor and not appear right away.

4However, we do use process_vm_readv/writev to bootstrap shared memory maps!
on another; and third, the target processor may run old versions of the code out of stale instruction caches or through speculative execution, unless it is given proper notification of the code modification.

**The Specification of Cross-Modifying Code**

Modern CPUs will reorder instructions, speculatively execute upcoming instructions and branches, prefetch memory for the instruction cache, and so on. The instruction cache and data caches are separate. This is why according to Section 8.1.3 of the Intel manuals [18] “Handling Self- and Cross-Modifying Code”, cross-modifying code should be written as shown in Figure 4.3.

```c
/* Action of Modifying Processor */
Memory_Flag = 0; /* Set Memory_Flag to value other than 1 */
Store modified code (as data) into code segment;
Memory_Flag = 1;

/* Action of Executing Processor */
WHILE (Memory_Flag != 1)
    Wait for code to update;
ELIHW;
Execute serializing instruction; /* For example, CPUID */
Begin executing modified code;
```

Figure 4.3: Intel’s specification of cross-modifying code.

In other words, the writer clears a flag before writing the modified code, and then sets the flag afterwards.\(^5\) The executing processor spins until it sees the flag set, which means that all previous writes have become visible; then it issues `cpuid` to cancel speculative execution of old code; and only then can it jump to the new code.

In the case of Shuffler, this could translate to an implementation as follows. Shuffler writes out new copies of functions, issues a store memory fence, and then increments a “generation counter” which is on a shared memory page. Then Shuf-

\(^5\)Although the manual doesn’t mention this, to be fully correct, there should probably be a store memory fence `sfence` before setting the flag, to make sure writes are not re-ordered.
Shuffler injects code to spin until the correct value of the generation counter is seen, at which point the target can continue executing and Shuffler may begin erasing old functions. However, this is not our current implementation; see below.

Making Use of RWX Pages

One possible shortcut to implementing cross-modifying code is to use memory pages that are marked write-through; thus, the write-back cache is never used, and writes are never re-ordered. If writes are “atomic” (this includes 64-bit aligned quadword writes on 64-bit processors, described below), then they will be effectively placed straight into main memory, synchronously.

Such pages could be set up from the operating system (or emulated by always flushing changes immediately after writing to a page\(^6\)). However, we noticed that pages marked RWX and mapped as shared pages into two different address spaces act as write-through pages (in Linux). Writes are propagated immediately. So previous versions of Shuffler used this method (and the current version can still fall back to it). However, this means that a huge number of pages in the target process must be marked with RWX permissions. This opens up the door to older code injection attacks; even if we are preventing code reuse attacks, it is not much use if simple buffer overrun attacks work once more.

Current Implementation

Because of the security risk of having RWX pages, Shuffler now uses R–X pages (sandboxes) and RW– pages (stack) in the target process, all of which are mapped as RW– pages in the shuffling process. In this setup, writes may be delayed in the write-back cache indefinitely\(^7\). Implementing proper support for this environment would involve writing a generation counter whenever new functions are shuffled, and reading the counter from the target’s processor to see whether the writes have propagated or not (as outlined above).

However, we did not fully implement this. Instead, we just continue writing new code into shared memory maps, and trust that it will eventually propagate to

---

\(^6\) In Linux, by calling `flush_dcache_page()`.

\(^7\) In fact we observed the target processor seeming to enter an infinite loop, running old code, on multiple occasions. (Using ptrace to read the code also revealed old data.)
the target. After a memory fence, Shuffler’s writes should appear eventually. We found that the delay inherent in waiting for the next shuffling cycle before erasing old code is usually enough to have the writes propagate.

This is not a robust solution, and the system will crash at higher shuffling rates (compared with the RWX solution, which never does). Clearly we should implement a system that actually checks if the writes have appeared before erasing old code. But our current implementation works well enough to gather timing information and get an idea of whether shuffling is feasible or not.

Making Atomic Updates

One other potential issue is that the target processor may see half-updated code which, when decoded, yields an invalid (or undesirable) instruction stream. We get around this by making a new copy of code in advance, and then updating the live code path with quadword aligned writes to addresses or call instructions. On modern processors, quadword-aligned writes performed by a single store instruction are atomic and do not require special synchronization.

4.1.3 Bootstrapping Shuffler (and Attach/Detach)

Shuffler has two methods of shuffling programs when it first starts: 1) taking control of a program from its very beginning with a custom loader which forks into the target and Shuffler, and 2) using ptrace to attach onto a running program at any point. Shuffler can also detach from a program that is being shuffled. This means that Shuffler can be deployed only when a target program really needs defending (e.g., a new zero-day attack is discovered), attaching onto a running program, and detaching when the threat has passed. See Section 8.4 for further discussion of this idea.

Technically speaking, the main challenge with both of these methods is getting code injected into the target (a task which Shuffler must perform fairly frequently). Shuffler reads the instruction pointer to find a page in the target which is executable; it then uses ptrace to inject code which calls mmap and then traps. When the target is allowed to run, it maps in an executable “staging area” as a shared memory region. Shuffler restores the original code for the overwritten page and can in
future write directly to the staging area, warping the target’s instruction pointer to the beginning of the region. Code injection is required to call open, mmap, mprotect, etc from the target’s context. Finally, writing trampolines and new copies of functions is also injection of a sort, but it is easier because Shuffler does not have to coordinate taking control back after running the code once.

4.2 Code Discovery and Relocation

Code is not designed to be moved to different addresses as it is running. However, it seems like this must be possible, since the compiler build chain regularly moves (or relocates) code to different addresses during the compilation and loading process.

4.2.1 How the Build Chain Uses Relocations

Conceptually, when the compiler generates basic blocks it keeps the relocation information necessary to move that block to any address. Subsets of these relocations are stored in ELF files and kept around for other stages of the build/load process. In particular,

- The compiler generates object files (which are relocatable ELF files). Object files contain code which starts at address 0, and relocations sufficient to shift the code to a different base address.

- The linker takes object files and decides on an ordering for the code within, relocating everything to non-overlapping addresses. This creates executables or libraries (which are also ELF files).

- The loader copies code into memory at runtime, and if the code is position-independent, then the loader relocates it to another base address using the relocations in the ELF.

ELF executables have nearly all their relocations resolved, and position-independent code (used for shared libraries) is located at address 0 so that the loader can decide on the base address.

So as code is built, different aspects of the code are resolved (basic block ordering, function ordering, base address, etc) and relocations that become unnecessary
are discarded. However, if we could keep all these relocations around, then shuffling a program essentially means re-linking it, at runtime. There are compiler flags (like GCC’s --q or --emit-relocs) which preserve relocations, and an earlier Shuffler prototype made use of this information. But relocations are only available if one has access to the source code and can compile with relocations preserved, and we want Shuffler to work on binaries. So we need to find code, disassemble it, and try to reverse-engineer the relocations in order to move the code.

4.2.2 Code Discovery and Disassembly

In x86-compatible binaries, code and data are interleaved. Distinguishing between them is a non-trivial task, although certainly possible: the heuristic-based recursive disassembler from Zhang and Sekar [39] is 100% accurate at disassembling their chosen set of test cases, and Wartell et al [36] have performed a broader study with their disassembler. However, we want Shuffler to be applicable to a wide range of binaries without requiring complex heuristics. Thus, Shuffler assumes that a fully accurate disassembly may not be available.

The simplest discovery technique, which is used by the current Shuffler implementation, is to obtain the target’s list of functions from its symbol table. Shuffler then uses a disassembly library on each function to find the jump and RIP-relative instructions that need to be fixed when the function is moved (relocated). We benchmarked five popular open-source disassembly libraries and chose diStorm3 [8], which had the best performance (and furthermore, has complete 64-bit support). If some code is not discovered through the symbol table, that code simply will not be shuffled. Note that shared libraries have their own symbol tables which are searched separately.

Not all executables have a symbol table, however. In fact “stripped” executables are the default in most Linux distributions. To handle these, the best option is to use heuristics to disassemble the code segment, looking for embedded data to skip over. We decided not to implement this, because it is a tedious process.

---

8Unfortunately, this option is rarely used and we found bugs at the linker level [6], and in the compiler (GCC 4.8): gedit_text_region_intersect()’s calls to find_nearest_subregion() have no relocations since the latter is a static function. Furthermore, it is difficult to modify most build systems to emit full relocations (e.g., Firefox, Chromium). The option -ffunction-sections is a workaround for the gedit case, but is fairly heavyweight.
and there is extensive prior work in the area (including [39]). So at the moment, Shuffler will only move functions that are mentioned in a symbol table.

4.2.3 Relocating Code at Runtime

Migrating a program’s code while the program is executing is a challenging task, because the code will likely be incorrect when it is moved to a new location. For example, embedded call instructions need to be adjusted (because they add a constant to the current instruction pointer to find the target). In fact, all embedded instructions with RIP-relative arguments need to be updated. Fortunately, diStorm makes it easy to find and fix all such arguments.

But after the code itself is updated, there are still external references to the old version of the code (e.g., direct calls to the old code, function pointers). Shuffler builds an “adjustment list” for each function of memory locations that refer to the function and need updating when it moves. This includes jump tables, incoming calls, etc. The list may be incomplete because Shuffler may only have a partial disassembly available. Furthermore, some types of references (such as function pointers) may become propagated throughout memory at runtime, making it almost impossible to fix all references. Shuffler fixes all the references it knows about, and handles all other references at runtime when the old addresses are used; see Section 4.4.1 for details on handling stale addresses. To make this easier, we are careful to ensure that stale addresses only ever refer to original locations of functions, and not their shuffled variants.

Unfortunately, a function is free to refer to its current address by saving the RIP register. This is quite common in 32-bit x86 code, where RIP-relative addressing is not present; instead, GCC makes use of the idiom in Figure 4.4 to load eip into a register.

```
call next
next:   pop %ebx
```

Figure 4.4: Idiom for retrieving EIP (32-bit x86 code).

This is dangerous for code movement, since an unknown set of registers may contain EIP-relative addresses; worse still, these may be pushed onto the stack.
We do not know whether this practice is common in 64-bit GCC-generated code, but to be safe, we avoid moving whichever function is currently executing. We assume that a function may perform RIP-relative computations, but does not need this state once it calls another function (an assumption which seems to be borne out in practice).

4.3 Trampolines

4.3.1 How to Update the Stack

We want to be able to move functions at arbitrary time intervals throughout the target program’s execution. We have described how to statically relocate functions and find references to them. But the biggest barrier to moving functions are the runtime artefacts of their execution: in particular, return addresses get pushed onto the stack, and these return addresses refer to the new copies of the code. When we attempt to move these functions, the return addresses will be stale and likely cause a program crash if actually used.

The obvious way to fix this is to unwind the stack, find all the return addresses, and update them to new locations. However, in x86 the stack frame for a function may be optimized out, making it very difficult to unwind the stack with 100% accuracy. Even libraries such as libunwind [20] must sometimes resort to heuristics. Given access to source code, compiler flags like GCC’s -fno-omit-frame-pointer and -fasynchronous-unwind-tables or information from custom LLVM compiler passes (as in [15]) can guarantee accurate stack unwinding. But Shuffler targets binaries.

The approach taken by Shuffler is to hoist each call instruction into a trampoline. As shown in Figure 4.5, each existing call instruction is transformed into a jump to the trampoline, which calls the target; on the return path, control flow passes through the trampoline before returning to the original caller. In other words, this ensures that return addresses on the stack (for shuffled functions) always refer to trampolines, allowing the caller and callee functions to be moved without having to update the return address.

Now the issue is that trampolines themselves are never moved. Shuffler extends
the basic trampolines to remember locations of return addresses using a separate shadow stack (see Figure 4.6). Then, trampolines can be moved as well, with the additional step of walking the shadow stack and updating any references to old trampolines.

It would be possible to rewrite functions directly with this extra shadow stack instrumentation, and avoid the use of trampolines. However, this involves inserting a lot of code around each call site, which may negatively impact code caching, alignment, and other parameters set up by compiler optimizations. With trampolines, the majority of calls turn into jumps of the same size, without impacting code size at all. Trampolines also add one more layer of indirection to the actual addresses of functions; the attacker cannot, for example, read the stack and immediately find addresses of functions.

**Figure 4.5:** Transforming a call instruction with a trampoline.
4.3.2 Trampoline Implementation

As shown in Figure 4.7, basic trampolines, without the shadow stack “bookkeeping”, are very simple. We use RIP-relative call and jmp instructions to refer to 64-bit constants located immediately following them in memory. The constants are aligned on 64-bit boundaries and can be updated with atomic 64-bit writes\(^9\). They do not have to be atomically updated together since both the old and new code are valid at the moment of update.

\[
\begin{align*}
call & \quad \ast0x6(%rip) \\
jmp & \quad \ast0x8(%rip) \\
.quad & \quad 0x2000 \quad ; \text{dest function (for call)} \\
.quad & \quad 0x1000 \quad ; \text{source address (for jmp)}
\end{align*}
\]

\[\text{Figure 4.7: Trampoline assembly code (AT&T syntax).}\]

The bookkeeping adds some complexity to the trampolines, increasing the size\(^9\) on modern 64-bit CPUs, 64-bit writes aligned to 64-bit boundaries are atomic—see Section 8.1.1 of [18].
of a trampoline to 64 bytes (including the two embedded addresses at the end). We use a simple stack for bookkeeping at a fixed point in memory, the address of which is embedded into trampolines when they are generated. See Figure A.1 in Appendix A for the full code for bookkeeping trampolines.

Finally, functions which contain calls must be updated to use trampolines. Most call instructions are 32-bit relative, which can be replaced with direct 32-bit jumps to trampolines (since both instructions are 5 bytes long and we try to place trampolines within 32 bits of their functions). However, inevitably Shuffler sometimes needs to patch in more bytes of code than there is room for: this happens if 64-bit jumps are required, and for indirect calls (which are usually less than 5 bytes long).

We handle this by trying to rewrite the source function, expanding its code to make space for the new jumps. Internal jumps within the function and all RIP-relative instructions must be fixed up, and potentially some internal jumps may need to be upgraded to wider variants (if their target is now too far away to reach). Our implementation attempts to perform rewriting once at code discovery time; all future shuffled copies of a function will be based on the rewritten code. But our implementation gives up on rewriting if opcode substitution is necessary, falling back instead on a 1-byte \texttt{int3} trap (when this is hit, Shuffler warps the instruction pointer to the trampoline). The traps are slower but always work. The full rewriting process could easily be fully implemented, and there are also existing libraries like DynamoRIO \cite{4} that do full rewriting, but we found that opcode substitution is very rarely required.

\subsection*{4.3.3 Memory Protection}

Trampolines provide a fair amount of information to an attacker: they contain the addresses of both source and destination functions. However, this information must be placed somewhere in the target’s address space for the target to be able to run at near-native speed without Shuffler’s intervention. So there is not much that can be done about this, with current hardware and operating system support.

In an ideal world, hardware would support pages with write and execute permissions, but no read permission (e.g., \texttt{-WX}). This is currently impossible to achieve
except without delving into Intel’s Virtual Machine Extensions [18] (i.e., the hypervisor level). Having such pages would make implementing a secure shuffling mechanism much easier: simply write addresses or jumps straight into this region, and jump to the (now disguised) function location. We can approximate this mechanism with current systems, however. Shuffler implements an optional protection layer on top of the shuffling mechanisms, wherein all code pages have no execute permission by default. Whenever the instruction pointer reaches a protected page, execute permission is enabled on that page; if the program tries to read the page, it must be an attack.

Ideally memory access checks would occur every time a page is used, but that is far too slow, and we are forced to “cache” the execute permission on a page. This provides some level of protection—an attacker can no longer simply read all existing pages looking for code—but frequently-executed pages almost always have the execute bit enabled, so it is possible to sidestep this mechanism. Still, perhaps in the future, $wX$ pages will be accessible from userspace, and this scheme will become relevant.

4.4 Completeness

The number of constructs that are available to userspace programs is large, and in this section we discuss how Shuffler handles some of these cases.

4.4.1 Handling Stale Addresses

At some point, the target may jump to a code address which is stale. We discuss how to deal with addresses that are generated dynamically and may refer to shuffled copies of functions (e.g., return addresses on the stack) above in Section 4.3. The only other case is unanticipated references to the original function location. We must be somewhat careful here and not just trap and warp such references to the current function location; this works, but defeats the purpose of shuffling, as an attacker can jump to any piece of code simply by jumping to its original location.

So we handle stale addresses as special cases, according to how they arise. Here are the cases we came across, and how Shuffler handles them:

- **Indirect calls** Indirect calls (such as `jmp *%rax`) are replaced with jumps
to trampolines. However, indirect call instructions have very short encodings, and jump-to-trampoline instructions (which are at least five bytes long) normally will not fit in their place. Thus, we try to use function-rewriting to make space for the new instruction, or fall back on a (1-byte) trap instruction.

- **Function pointers** Function pointers occur in the data section as literal addresses or in code as constants, and it is impossible to tell that they are code pointers until they are called. Shuffler traps on initial function pointer invocations, warping the instruction pointer to the current function location, and inserts a trampoline to speed up subsequent calls. We do not attempt to modify function pointer values when a function is moved, as the pointer may be propagated throughout the heap and stack.

- **Jump tables** Jump tables contain absolute code addresses and are embedded into the data section, with no clear delineations (although relocations, if present, will indicate jump tables [29]). Currently, Shuffler uses a simple heuristic\(^\text{10}\) to detect jump tables and replace the table entries with trampoline addresses. For more complete and advanced heuristics, see Cifuentes [5].

- **Setjmp/longjmp** These functions allow the stack state of a program to be saved and restored, enabling non-local gotos across function calls. GCC does not inline setjmp/longjmp even at high optimization levels, so any longjmp will restore within a stack frame for the setjmp function. We could simply avoid shuffling the setjmp function, so that any longjmp will work correctly, and additionally fix bookkeeping trampoline information (see below). However, this special case is not yet implemented.

- **C++ exceptions** Exception handling involves unwinding the stack when an exception is thrown. Discarding stack frames causes issues with bookkeeping trampolines, but there is a simple fix (which we did not implement): simply check for all bookkeeping entries which do not follow in descending

\(^{10}\) We look for JMP instructions with a scale factor of 8 and a constant offset which points inside the code segment, to a table of addresses which also point inside the code segment. We assume the first word which does not look like an address demarcates the end of the jump table. This is very accurate at picking out GCC-generated jump tables, even at high optimization levels.
order from the previous entries, or which refer to memory beyond the current stack pointer. Such bookkeeping entries are out-of-date and can be ignored.

There are other unusual cases which we do not handle\[11\], but we have listed above the majority of cases that appeared in our testing.

### 4.4.2 Handling Shared Libraries

Shuffler finds shared libraries through the same mechanism as GDB\[12\], by walking the dynamic loader’s list of libraries. New shared libraries are noticed by inserting a breakpoint at `dl_debug_state`, which is a hook called by dlopen after every shared library load or unload. Each shared library is its own ELF file, and is shuffled independently. Each process also gets its own private re-randomized copy of a library; any attempt to share the memory used by libraries across processes would increase the attack surface against randomization, since multiple processes could examine the code at once.

### 4.4.3 Handling Threads and Forks

Support for threads and forks is not fully implemented in Shuffler; refer to the discussion in Section 8.1.

### 4.5 Overall Operation

Once Shuffler can write code into a target process, redirect function calls through trampolines so that functions can be moved, and save return addresses on a shadow stack so that trampolines can be moved, the main remaining task is to actually move all this code.

As shown in Figure 4.8, Shuffler consists of a main thread and one or more worker threads. The main thread catches signals in one or more target processes under its control, can move targets’ instruction pointers and inject code, and is responsible for detecting new shared libraries and forks. The worker threads erase old shuffled functions, and create new shuffled copies asynchronously; the main

---

\[11\] For example, the unportable GCC extension called *labels as values*, which allows pointers to goto labels to be taken with the unary \& operator.

\[12\] See `enable_break()` in `gdb/solib-svr4.c`.
thread periodically stops the target processes and marks most old functions as being ready for erasure (with the single exception of whichever function/trampoline is currently executing, as described in Section 4.2.3). The target process may spend nearly all of its time actually executing, and will transparently migrate execution to the new copies of functions (as the worker threads atomically update old trampolines, or trampoline addresses on the stack).

Figure 4.8: Overall architecture of Shuffler.
Chapter 5

Performance Evaluation

We performed most of the experiments in this section on a system running Debian jessie with the Linux 3.13 kernel, with a quad-core 3.20 GHz Intel Core i5-4570 (Haswell) processor and 8 GB of RAM. Some experiments (clearly marked) were run on our “alternative” system, an ASUS Zenbook UX32VD with a dual-core 1.90 GHz Intel Core i7-3517U (Ivy Bridge) processor and 6 GB of RAM.

All results are reported as the average of three runs.

5.1 Shuffler Microbenchmarks

5.1.1 Memory Access Overhead

To compare the different methods of accessing another process’s memory, we wrote test programs to transfer 10GB in 1MB chunks. The timing results are shown in Table 5.1 (as usual, the average of three runs).

<table>
<thead>
<tr>
<th>Type</th>
<th>Runtime (sec)</th>
<th>Runtime (normalized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local memory access</td>
<td>0.607s</td>
<td>1.000x</td>
</tr>
<tr>
<td>Shared memory map</td>
<td>0.604s</td>
<td>0.995x</td>
</tr>
<tr>
<td>process_vm_readv</td>
<td>0.630s</td>
<td>1.038x</td>
</tr>
<tr>
<td>/proc/pid/mem</td>
<td>0.947s</td>
<td>1.561x</td>
</tr>
<tr>
<td>Vanilla ptrace</td>
<td>19.916s</td>
<td>32.811x</td>
</tr>
</tbody>
</table>

Table 5.1: Performance of reading 10GB of memory between processes.
Shared memory maps provide identical performance to local memory accesses (though of course it may take a short while for the writes to be committed). Both the new system call `process_vm_readv` and reading `/proc/pid/mem` with `pread` provide worse performance. In both cases, the overhead of just the system calls is negligible: invoking the same number of system calls, but only transferring 1 byte, takes 0.003 seconds. On the other hand, using `ptrace` directly requires a huge number of system calls and slows down unacceptably. Finally, we were unable to benchmark the proposed `PTRACE_MULTI` patch [11], but its performance would likely be comparable to `process_vm_readv`, because their interfaces are similar.

Thus, although shared memory maps are complicated to set up, their additional flexibility and superior performance provide a compelling reason to use them (as Shuffler currently does).

5.1.2 Context Switch Overhead [on Ivy Bridge]

To measure the overhead of a context switch between the target program and Shuffler (and back again), our test program dereferences a function pointer to an empty function repeatedly, causing a million switches to Shuffler and back on the indirect jump trampoline. This results in an average time of 6.9 µs per context switch. This is slightly higher than the expected context switch cost of modern processors because the shuffler makes a number of `ptrace` system calls in handling the trampoline. As a point of comparison, a function call on the same system takes only 0.002 µs.

5.1.3 Code Injection Overhead [on Ivy Bridge]

As a simple code injection example, injecting an `mprotect` involves four context switches and the actual call to `mprotect`. This is due to the target program trapping to Shuffler, which injects the `mprotect` and switches back to the target program. The target program executes the `mprotect` call and traps back to Shuffler, which then switches back to the target program. The context switch after the `mprotect` is a callback mechanism that Shuffler uses to perform any additional operations.
Since the cost of a context switch between the target program to Shuffler and back takes 6.9 µs, the expected cost of an `mprotect` call plus the additional context switch back and forth should be at least twice that of just context switching. However, the cost is only 9.1 µs. The reduction in expected cost is likely due to the effects of caching.

### 5.1.4 Cache Trampoline Optimization [on Ivy Bridge]

<table>
<thead>
<tr>
<th>Description</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.003s</td>
</tr>
<tr>
<td>Shuffling without optimizations</td>
<td>13.887s</td>
</tr>
<tr>
<td>With func ptr caching</td>
<td>6.946s</td>
</tr>
<tr>
<td>With func ptr caching and rewriting</td>
<td>0.222s</td>
</tr>
</tbody>
</table>

**Table 5.2:** Function pointer overhead on a million calls.

To measure the performance of function pointers, we created a synthetic benchmark that dereferences a function pointer to an empty function a million times. The simple step of introducing trampolines (without any optimization) causes a huge performance penalty. This is due to every function pointer call turning into two separate traps, one at the indirect call site (because the `call *%rax` instruction is so small it needs a trap) and the other at the original function location.

The first optimization is to turn on function pointer caching, which writes a trampoline at the original function location to remove one of the traps on the critical path, cutting the runtime in half. The second optimization is to rewrite the source function so that the indirect call can be replaced with a jump, removing the other trap from the critical path. The combination of these optimizations results in good performance. Though the baseline is still much faster at 0.003s, that measurement does not include the constant startup-time overhead required to shuffle a process.

### 5.1.5 Jump Table Optimization [on Ivy Bridge]

Figure 5.3 shows the effect of jump table optimizations on the sjeng SPEC CPU2006 benchmark with a 10 ms shuffle rate. Introducing jump table optimizations realizes a 171x performance improvement. This is because originally the
Table 5.3: Performance improvement for jump table optimizations on \textit{sjeng} (10ms shuffle rate).

<table>
<thead>
<tr>
<th>Description</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>3.61s</td>
</tr>
<tr>
<td>Shuffling without optimizations</td>
<td>939.00s</td>
</tr>
<tr>
<td>Shuffling with optimizations</td>
<td>5.48s</td>
</tr>
</tbody>
</table>

jump table refers to the original function location, and invoking it requires a context switch. With jump table optimizations, this context switch is replaced by a trampoline.

5.2 Macrobenchmarks on SPEC CPU2006

We ran Shuffler on a representative selection of SPEC CPU2006 benchmarks. Figure 5.1 shows the overhead of shuffling compared with native performance, with shuffling rates varying between every 100, 50, 20, and 10 milliseconds. Performance generally decreases at higher rates of shuffling.

The performance characteristics are clearest in the libquantum case: here, any shuffling at all has an overhead of roughly 10% (because of the constant overhead introduced by trampolines). Then, performance decreases as the shuffling rate increases, because the target program spends more time stopped and its CPU’s caches have to contend with more cross-modifications from the shuffling processor.

Some cases, especially milc and hmmer, have large codebases. Thus, more work needs to be done in each shuffling period, which either causes performance overhead (as in milc) or even causes writes to be delayed too long for shuffling to operate correctly (as in hmmer at high shuffling rates). The scheme presented in Section 4.1.2 would need to be properly implemented for cases like hmmer to work. The other C benchmarks in SPEC CPU fail for what is probably the same reason, with other explanations in some cases (e.g., perl uses threads, which would need to be supported as described in Section 8.1).
Figure 5.1: Performance of shuffler mechanisms and periodic shuffling on SPEC CPU2006 Benchmarks.
Table 5.4: Average overhead of SPEC CPU2006 benchmarks with different shuffling rates.

<table>
<thead>
<tr>
<th>Shuffle Rate</th>
<th>Average Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>100ms</td>
<td>10.3%</td>
</tr>
<tr>
<td>50ms</td>
<td>11.3%</td>
</tr>
<tr>
<td>20ms</td>
<td>13.4%</td>
</tr>
<tr>
<td>10ms</td>
<td>21.1%</td>
</tr>
</tbody>
</table>

Collecting the average runtimes into Table 5.4, we see an average overhead of 10-21% on the SPEC benchmarks that successfully run.

5.3 Running Nginx [on Ivy Bridge]

We ran the web server nginx (with optimizations and threads disabled) under Shuffler. We created a 90-byte index.html page and fetched it 1000 times sequentially. Without Shuffler, this took 7.136 seconds, and with shuffling every 100ms, it took 8.525 seconds (an overhead of 19%). This is not bad considering the complexity of a web server.
Chapter 6

Security Evaluation

6.1 Memory Disclosure Attacks

The primary motivation for binary shuffling is to mitigate sophisticated memory disclosure attacks, such as the one presented by Snow et al [31]. As discussed in Section 2.3, we assume a remote attacker who knows one valid code address, and who may repeatedly invoke a memory access vulnerability to report back memory pages, tracing the control flow of the program so as to always request valid pages. The attacker can then compile a customized ROP attack by searching for gadgets within the retrieved code, and specifying control flow between these gadgets.

For this type of attack to succeed, the gadgets that are used by the exploit must be located at the same place in memory when the exploit is executed. In other words, if one or more gadgets are shuffled away before the exploit can be compiled and executed, the attack will have been foiled. We wish to show that shuffling occurs at a fast enough rate that such attacks will always be foiled with high probability.

6.1.1 Analysis of a Memory Disclosure Attack

For our analysis, we define the following variables:

• let $T_s$ be the time between shuffles;

• let $T_D$ be the time taken to discover and retrieve code pages; and
• let $T_E$ be the time required to execute the customized ROP attack.

However, in a departure from Snow, we assume a stronger adversary who has access to an unshuffled binary of the target program and can analyze it statically in advance of the attack. Thus, the abstract control flow graph between possible gadgets is known, although it does not contain actual memory addresses. The memory discovery process is used to fill in the abstract control flow graph with actual addresses to create exploit code. We further assume the attacker may have many such potential exploits available, requiring different gadgets. Assuming that crafting exploits has been done in advance, the time to complete an attack is given by $T_D + T_E$.

We assume that all functions are shuffled atomically. This means that if even a single gadget remains to be executed after a shuffle, the gadget attack code will not run to completion. We assume that if the gadget attack code does not run to completion, then it will not succeed. Under these assumptions, an attack that fills the abstract gadget control graph directly with memory locations read during the discovery process will not succeed if a shuffle occurs before the attack completes both the memory discovery phase and the gadget execution phase. That is, the attack fails if $T_D + T_E > T_S$.

Note that $T_S$ is a tunable parameter of the Shuffler and can be set by the user based on current estimates of $T_D + T_E$ of known memory disclosure exploits in the wild. Prior work on just-in-time gadget compilation suggests that $T_D$ may be quite high (discussed below). However, if we can demonstrate that even the most pessimistic $T_D$ is greater than $T_S$, then this is sufficient and $T_E$ can be ignored (we believe $T_E$ to be quite low in practice).

### 6.1.2 Why the Attacker Has Insufficient Time

An attacker needs several distinct gadgets to craft an exploit. Polychronakis et al [26] find the number of distinct gadgets to be between 8 and 21 in an analysis of twelve real-world ROP exploits. As we are considering the pessimal case, we assume an attacker only needs to find 8 gadgets. In an analysis of common Windows DLLs, Snow et al [31] were able to discover a total of 413 gadgets across 307 pages, yielding an average density of 1.35 gadgets per page. Therefore, in order to
obtain 8 gadgets, we estimate that an attacker will need a minimum of 6 pages.

The fastest demonstrated remote exploit (a JavaScript page loaded in a web browser which contains their exploit framework) \cite{31} harvests pages at a rate of 84 pages/second\footnote{We assume that their hardware, which is not described, is equivalent to our own.}. Therefore, in the most pessimistic case, if the first 6 pages discovered happen to have all 8 gadgets that the attacker needs, the time required is 71 ms. As long as \( T_S \) is less than this, the attack will fail.

In reality, \( T_D \) is likely to be quite a bit larger than in this pessimistic analysis. In the fastest attack described by Snow \textit{et al}, it takes 2.3 seconds to traverse pages, collect gadgets, resolve APIs, and compile an attack. Examining their graph which shows a breakdown of time spent on each aspect of the attack, it takes approximately 500 ms for the gadget discovery phase. Therefore, a \( T_S \) of 500 ms is likely sufficient to defend against most attacks, while a \( T_S \) of 50 ms is likely sufficient to defend against even the most pessimistic attacks.

Finally, all of this discussion has been assuming the attacker can directly interact with the target program. More typically, the user is concerned with remote exploits. In this case, even after the memory vulnerability disclosure has been used to read data from the target process, that data must be communicated back to the attacker. The attacker will need to overcome the intervening network latency, which is likely on the order of a millisecond (at minimum). And unless the disclosure vulnerability can be run in batches, an attacker may need to make hundreds of calls for I/O across the network just to read a handful of code pages (since each read would normally retrieve only a small amount of data).

### 6.2 Other Types of Attacks

Adversaries are not limited to the type of attack given by Snow \textit{et al}. We discuss other possible ways to subvert dynamic shuffling.

**Brute force**

As with any randomization technique, the attacker can simply guess at the memory locations of the gadgets required for an attack. We assume that at least one shuffle has been performed (the target may be vulnerable before this, but Shuffler can
easily perform a shuffle when the target is first started). Then each function will be
placed at a random byte-aligned address somewhere in the shuffling sandbox (such
that it does not overlap with any other function). Currently we use 256MB sand-
oxes, which provides 28 possible bits of entropy. If at most half of the sandbox
is used (code sizes are usually much smaller than this), then there are at least 27
bits of entropy in choosing any new function location. So correctly guessing the
location of a particular block of code will only work one in $2^{27}$ times. So pinning
down the location of (say) eight unique gadgets would only succeed one time in
$2^{27\cdot8} = 2^{216}$. This appears very unlikely to be a successful attack vector.

**Guessing the Random Sequence**

An attacker may find greater utility in trying to discover the random seed used by
Shuffler. If Shuffler’s operation were completely deterministic with respect to the
random number generator’s seed, then a well-provisioned attacker could simply
run (or simulate) Shuffler on the target program with each of the possible seeds.
In fact, Shuffler’s random number generator only has $2^{32}$ possible seeds, so an
attacker could also randomly guess the seed, providing much better odds for the
attacker than guessing at gadget locations.

The correct solution here is to use a random number generator with a seed wider
than 32 bits. We did not implement this; however, it is a fairly simple change. The
performance of Shuffler does not depend much on the speed of the random number
generator, because random numbers are generated asynchronously in the worker
threads (which have more time than they need).

**Using the Shadow Stack to Find Valid Addresses**

The shadow stack used by bookkeeping trampolines contains a list of pointers to
return addresses, which are themselves inside trampolines. So this gives an attacker
access to additional valid code locations (albeit doubly indirected). Much the same
information is already available by reading the program’s real stack, but with the
shadow stack, no stack unwinding is necessary. However, we are assuming that the
attacker knows some valid code locations and simply cannot read them fast enough
to execute an attack. So this additional information is not particularly helpful.
Using Cached Calls to Find Valid Addresses

For performance reasons, when Shuffler sees the target invoking a function pointer and raising a trap at the original location of the function, a cache trampoline is placed at the original location which simply jumps to the current location of the real function. This represents a much more significant advantage for the attacker: it becomes possible to read the original location of a function, and find the current location of that particular function. What’s more, if the cache trampoline is not present, the attacker will simply read a series of trap instructions instead of crashing the target. So an attacker can collect a set of gadgets that are necessary for an attack, continuously re-read the original locations of the functions containing those gadgets, and spring the exploit only when all functions have cache trampolines.

However, this attack only works for functions which are referenced by a function pointer. It also requires an attacker to read at least sixteen pages (original function locations and then current locations) to find eight gadgets; the current locations are dependent on the values of the first reads and cannot all be batched together. So our analysis of a memory disclosure attack given above still holds, although this essentially gives the attacker their best case scenario for number of pages to read. It is also possible to disable cache trampolines, and simply take the performance hit on function pointer calls.

In future we would like to have a stronger solution to this problem. The best option seems to be to start randomizing function pointers; i.e., create cache trampolines but at random locations, and occasionally invalidate them. Function pointers can be updated by using the Intel Last Branch Record to find the source instruction, and changing the register or memory location which contained the function pointer. This will obviously miss many occurrences of pointers, but should provide good performance for tight loops.

Jumping to Original Function Locations

Shuffler traps all jumps that reach within an original function’s code (once the function has been shuffled away). This could be a legitimate jump, usually from dereferencing a function pointer (or undetected jump table), in which case the instruction pointer will be warped to the current function location. The jump could
also be the result of an attacker trying to jump into the middle of a function. At the moment, Shuffler cannot differentiate and assumes that the jump is legitimate. It is likely that a simple heuristic, like allowing jumps to the start of the function but not the middle, will suffice. In any case, some policy must be implemented before real-world Shuffler deployment. This is discussed further in Section 8.6.

**Trampoline Re-Reading Attacks**

One technique an attacker might try is to continuously re-read trampolines. This is particularly effective if trampolines are not shuffled; the trampolines themselves may not contain any gadgets, but a simple re-read of the trampoline will reveal the current location of its source and destination functions. Once a trampoline is discovered, the attacker can always read that location (leaking information between shuffling periods). Furthermore, the attacker can read a trampoline to discover whether functions have been shuffled or not, providing a potential way to synchronize an attack within shuffling periods.

Needless to say, this is a fairly powerful attack which is outside of our original threat analysis. We avoid the issue by shuffling trampolines just as much as we move functions; when bookkeeping trampolines are used, re-reading a trampoline is no more effective than re-reading a function.

**Compromising the Shuffler Process**

Shuffler itself is a natural point of attack given its privileged position with respect to target code. However, there is no opportunity for the attacker to exploit Shuffler through the target program. While Shuffler does read from memory locations in the target code, this is treated only as data and is stored in no-execute pages of memory within Shuffler. The behaviour of Shuffler is not based on the contents of those reads during runtime. The behaviour is based entirely on the analysis of the symbol tables and dynamic disassembly when Shuffler launches the target program. In particular, Shuffler knows exactly how large each function that it reads should be and only copies that amount of memory. It is possible that a malicious executable could be crafted that causes Shuffler to misbehave, but we are assuming the attacker has no control over the executable which is given to Shuffler.
Chapter 7

Related Work

7.1 Return-Oriented Programming

Return-Oriented Programming (ROP) began as return-into-libc attacks [30], and was generalized to finding gadgets in any existing code base. Recently, ROP compilers can automatically find gadgets and create a desired exploit using the available gadgets; ROP compilers can even harden attacks for the presence of ASLR and other defenses [28].

7.2 Static Randomization

Address Space Layout Randomization (ASLR) [25, 38] randomizes the locations of base addresses of the program (e.g., stack, heap, and libraries). Thus, an attacker no longer knows where in memory specific code resides and therefore can no longer reliably overwrite the return address with a valid location. However, since only base addresses are randomized, the relative position of code within each module remains constant. This opens the door for a memory disclosure attack. If an attack can reveal even a single address within a module, it can determine the location of all the code in that module [32, 35].

Building upon the basic concept of ASLR, fine-grained ASLR takes randomization one step further: instead of just randomizing the base address of a code module, the code within the module itself is randomized. Bhatkar et al [2] random-
ize at the granularity of functions, while also randomizing the location of stack-
resident variables and static data. However, their approach is a source-to-source
transformation. Binary stirring [37], on the other hand, applies randomization to
binaries, at the granularity of basic blocks. Instruction Location Randomization
(ILR) [16], as its name suggests, provides even finer grained randomization by
placing each instruction at an arbitrary address. In each of these systems, however,
a static randomization is applied at load time. That is, a program is rerandomized
each time it is run, but this randomization exists for the lifetime of the program.

In the face of multiple memory disclosure attacks in conjunction with just-in-
time exploit generation [31], these static randomizations are insufficient. A single
memory disclosure vulnerability can be exercised multiple times in order to reveal
multiple addresses. Once an attacker has gathered enough information, an exploit
can be compiled and executed.

### 7.3 Just-In-Time Code Reuse Attacks

The attack outlined by Snow *et al* [31] is able to subvert fine-grained code random-
ization and construct a successful exploit. The attack requires a memory disclosure
vulnerability that can be invoked repeatedly. Conceptually, this vulnerability could
be invoked to read the entire code segment of the target program, and then construct
a ROP attack based on the code that is present (no matter what randomizing permu-
tation has been used). However, in practice attempting to read addresses arbitrarily
will cause page faults on unallocated pages, terminating the target program.

Thus, the Snow attack is predicated on knowing one valid code address. The
memory access vulnerability can be invoked to read the whole page containing
that address (since memory protection is done at page granularity). The code is
disassembled, and any `call` or `jmp` instructions point to other pages which also
contain code and may be disassembled (the next page is another potential can-
didate, if control flow falls off the end of the previous page). By repeating this
process, the attacker can often recursively disassemble a significant portion of the
target’s code.

After discovering valid code pages and retrieving their contents, the attacker
can search these pages for ROP gadgets. If sufficient gadgets are found, a ROP
compiler can be used to target an exploit to the existing gadgets. Finally, the attacker can use a separate vulnerability (such as a stack smashing bug) in the target program to inject the custom-compiled ROP attack into the target, and successfully run an exploit.

7.4 Dynamic Randomization

In order to defend against multiple memory disclosure attacks, the layout of a program needs to be re-randomized as it executes [31]. This dynamic randomization can only succeed in defending against these attacks if the re-randomization occurs quickly enough.

Stabilizer [7] improves the consistency of performance measurement techniques by removing the side-effects of static randomizations on cache and branch predictor performance. Stabilizer is implemented as a compile-time transformation, necessitating source code. However, rather than continuously re-randomizing everything, they instead mark functions for re-randomization every 500 ms. The first time a function is executed after being marked, it is re-randomized. Thus, rarely used functions can exist at fixed locations for long periods of time. Further, a copy of the code remains at its original location, easily exploitable by an attacker. Stabilizer, however, was not designed for security purposes.

Giuffrida et al [15] propose a fine-grained re-randomization strategy for securing operating system components. Implemented as an LLVM link-time transformation, it also requires access to source code. At every re-randomization step, a new process is linked and launched and live state is migrated to the new process. The overhead of this approach is quite high, 42.7% average overhead for a one-second re-randomization period.

7.5 Control-Flow Integrity

Control-Flow Integrity (CFI) is a defense against ROP, where every indirect jump is validated before it is taken, so as to prevent control-flow hijacking. The original CFI implementation [1] provides strong correctness properties but has high overhead. Recent research has focussed on enforcing less strict (or “coarse-grained”) CFI, trading some security for a much faster implementation. CFI techniques have
been proposed that work just with debug information [13] and even on arbitrary binaries [39].

Unfortunately, a recent work by Davi et al [10] showed that most of these looser versions of CFI provide insufficient defense. The authors found a set of specific gadgets (located after call instructions, to look like valid return locations) such that recent CFI mechanisms would allow control-flow transfer and a return-oriented attack. These gadgets are Turing-complete.

Works such as XFI [13] still provide security against ROP attacks, but further work will need to be done to find a solution which works on arbitrary binaries, or a solution which has reasonable performance overhead.
Chapter 8

Future Work and Discussion

8.1 Shuffler Completeness

Although Shuffler handles many ways that a userspace target program might create stale address references, discussed in more detail in Section 4.4.1, there are other cases (like C++ exceptions) which we do not handle. These cases are likely to involve most implementation work for special-casing, but our overall design should remain unchanged. To truly handle unmodified program binaries, Shuffler should support as many of these cases as possible.

In addition, Shuffler does not handle several legitimate features of userspace processes, including threads and forks.

- **Threads** While not fully implemented, handling threads is straightforward. Whenever an old copy of a function is erased, Shuffler would simply check whether any thread is still executing the function, rather than just the main thread. Additionally, bookkeeping trampolines would need to store their saved return addresses in a thread-safe way instead of just appending to a stack. This could be implemented using a hash table, or by allocating thread-local storage for the bookkeeping stacks.

---

1On x86_64, GNU libc dedicates $fs$ to point to thread-local storage, and ld.so allocates $TLS\_SLOTINFO\_SURPLUS$ (62) extra TLS entries just in case. We could easily use one entry to point to a bookkeeping data structure (different offsets into a memory map). If the original ELF has no TLS program header then the $fs$ register is available for our own use.
• **Fork/exec** Intercepting fork and exec calls is easy with `ptrace`. But fork calls are difficult to handle because of the shared memory map technique we use to access memory in the target. These shared memory maps are preserved across forks, and the parent and child end up sharing their code and stack—essentially turning each fork call into a `vfork` call. Thus, the first action after a fork needs to be injecting the same code that was initially used to set up shared memory maps, to unmap and remap the memory regions backed by new files. We have not fully implemented this, but it is the same mechanism as originally used by the loader. Finally, exec calls can be handled by simply throwing away the existing structures and starting over, as if attaching onto a new process.

We would, of course, like to have had the time to fully implement these features, and hope to do so in the future.

### 8.2 Code Disassembly

Although Shuffler is targeted at common off-the-shelf binaries, our current implementation relies on having a symbol table to discover code in the target. We do not consider this a serious limitation, because there has been a great deal of prior work in recursive disassembly [29]. It is very hard to do fully accurate disassembly, but fortunately Shuffler does not need a complete disassembly; if some functions are not found, they will simply not be shuffled. We would like to incorporate one of these existing disassembly projects into Shuffler so that the latter will work on stripped binaries.

### 8.3 JIT Code

One major advantage of dynamic randomization is the ease of supporting dynamically-generated code (simply add the new code to the list of shufflable units). Thus, one potential future direction for this project is to support JIT code from language runtimes. Shuffler could simply remove execute permission on suspected code pages (e.g., all unknown pages allocated by the program), wait for a jump into them, and then assume a function begins at that point and disassemble from there. This is
almost the same process of dynamic code discovery discussed in Section 4.2.2. We have not implemented this technique, but we believe it speaks to the flexibility of randomization techniques that they can be easily extended to JIT code.

8.4 Hot Patching

The current Shuffler can dynamically attach onto and detach from running processes. This is very good for defending against zero-day vulnerabilities, because a layer of protection can be added to a running service which is known to be vulnerable, without bringing that service down. Because Shuffler is already dynamically copying in new versions of functions, it would be straightforward to extend Shuffler to incorporate hot patches that fix a vulnerability before detaching. Thus, a user could attach, wait for a patch, apply the patch, and detach from the target process without ever stopping it. This means Shuffler is poised to be an excellent zero-day defense. However, there is currently a small performance degradation over the original when Shuffler detaches, so this would ideally be fixed first.

8.5 Shuffling an Entire System

Given Shuffler’s ability to hook into a system loader, it would be quite possible to attempt to shuffle every userspace program running on a system. The way that forks would be handled allows one Shuffler process to handle multiple target processes, if they are spawned from the same source (e.g., a shell). And Shuffler can easily create additional worker threads to handle the new processes. In other words, potentially a small number of Shuffler processes (running on a small number of excess CPU cores) could shuffle an entire system at once. It would be interesting to see whether this works.

8.6 Shuffling Policy

Shuffler takes a very straightforward approach to shuffling, simply moving every function sequentially within a fixed time period. It might make sense to find functions that are higher-risk, perhaps because they contain obvious gadgets or system calls, and shuffle these at a higher rate. As mentioned in Section 3.1.4, perhaps the
shuffling rate should be customizable based on resource load and other features of the supporting system. Shuffler could be extended to have a shuffling policy that affects the rate at which different functions are shuffled, and when to dynamically adjust these rates. A policy would also be useful to decide whether a stray jump is the result of an attack or not.
Chapter 9

Conclusion

The two main classes of defenses against ROP, namely control-flow integrity and randomization, have both recently been shown to have weaknesses [10, 31]. We aim to resurrect randomization techniques as a viable defense against ROP. We do this by introducing shuffling, a high-frequency dynamic re-randomization technique.

Our system, Shuffler, implements binary shuffling (i.e., shuffling for program binaries without access to source code). Shuffler runs as a separate userspace process, for easy deployability. It can run with a custom program loader, or attach onto running programs (optionally even detaching at a later point), shuffling all the functions in a program as quickly as every ten milliseconds. In the future, a platform like Shuffler could provide on-the-fly defense against zero-day vulnerabilities, and operate on dynamically generated JIT code from language runtimes.

We benchmarked Shuffler on the benchmark suite SPEC CPU 2006, and found an average overhead of 10-21% depending on the shuffling rate. This overhead is small enough that shuffling appears to be a viable defense against memory disclosure attacks and ROP.
Bibliography


[18] Intel. Intel 64 and ia-32 architectures software developer’s manual volume 3a: System programming guide, part 1, Mar 2010. → pages 4, 16, 24, 26


53


Appendix A

Injected Code

Figure A.1 shows the code for our full bookkeeping trampolines (refer back to Figure 4.6 for a higher-level description of bookkeeping). The address 0x7fffffff (a 31-bit constant) is replaced with the address of the shadow stack region, and the addresses of the source and destination functions are embedded at the end of the trampoline. Every time the trampoline is called, the current stack pointer %rsp is saved on the shadow stack, and on return, popped from the shadow stack. The shadow stack itself is a flat memory region, the first 8 bytes of which are a pointer to the current top of the stack.

The two NOPs at the beginning serve to align the final embedded addresses to 8-byte boundaries, so that they can be updated atomically with normal writes (as described in Section 4.1.2). The incoming jump can safely point beyond the NOPs. Note that when the instruction pointer is set with ptrace, we sometimes observed it warping backwards two bytes, perhaps due to canceling speculative execution or some other hardware feature; thus, although trampolines are never warped to, we try to have two leading NOPs (which are normally skipped over) at the beginning of any injected code segment.

Finally, the whole trampoline is exactly 64 bytes, so that is the memory size allocated for each trampoline within the code sandboxes.
Figure A.1: Trampoline code with bookkeeping.