Improving Security Through Egalitarian Binary Recompilation

David Williams-King

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

Improving Security Through Egalitarian Binary Recompilation

David Williams-King

In this thesis, we try to bridge the gap between which program transformations are possible at source-level and which are possible at binary-level. While binaries are typically seen as opaque artifacts, our binary recompiler Egalito (ASPLOS 2020) enables users to parse and modify stripped binaries on existing systems. Our technique of binary recompilation is not robust to errors in disassembly, but with an accurate analysis, provides near-zero transformation overhead.

We wrote several demonstration security tools with Egalito, including code randomization, control-flow integrity, retpoline insertion, and a fuzzing backend. We also wrote Nibbler (ACSAC 2019, DTRAP 2020), which detects unused code and removes it. Many of these features, including Nibbler, can be combined with other defenses resulting in multiplicatively stronger or more effective hardening. Enabled by our recompiler, an overriding theme of this thesis is our focus on deployable software transformation. Egalito has been tested by collaborators across tens of thousands of Debian programs and libraries.

We coined this term *egalitarian* in the context of binary security. Simply put, an egalitarian analysis or security mechanism is one that can operate on itself (and is usually more deployable as a result). As one demonstration of this idea, we created a strong, deployable defense against code reuse attacks. Shuffler (OSDI 2016) randomizes function addresses, moving functions periodically every few milliseconds. This makes an attacker's job extremely difficult, especially if they are located across a network (which necessitates ping time)—JIT-ROP attacks take 2.3 to 378 seconds to complete [1, 2]. Shuffler is egalitarian and defends its own code and target code simultaneously; Shuffler actually shuffles itself.

We hope our deployable, egalitarian binary defenses will allow others to improve upon state-of-the-art and paint binaries as far more malleable than they have been in the past.

Table of Contents

Acknowledgements

I am indebted to a larger number of people than most doctoral students. Many students worked with me via pair programming, usually in person on a second keyboard or laptop. I would like to thank, in roughly chronological order: James P. Blake, Graham Gobieski, Michelle Zheng, Abhishek Walia, Heather Preslier, Zixiong Liu, Phil Schiffrin, Seung Bin Lee, Yanbei Pang, Avijit Shah, Jessy Han, Qianrui (Cherish) Zhao, Mingjie Zhao, Graham Patterson, Frank Spano, Nick Bullard, Anthony Saieva, Serena Tam, Sarah Leventhal, Yu Jian Wu, Ziwei (Sara) Gong, Kang Sun, and Shubham Sinha. Apologies if I have left anyone off this list. Special thanks to James P. Blake, Hidenori Kobayashi, Graham "#1" Gobieski, Graham "#2" Patterson, Anthony Saieva, Yu Jian Wu, and Kent Williams-King for working with me for an extended period of time.

I would like to thank several people for encouraging my forays into accessibility via speech recognition. Number one is Professor Sang-Mook Lee, who hosted me in Korea multiple times and helped me see how many other people can benefit from accessibility. Thanks also to Professor Homayoon Beigi and the ACM ASSETS community. Alex Roper, David Zurow, Dane Finlay, and numerous others have built the voice recognition software I use every day.

My advisor, Junfeng Yang, always had helpful insights and supported me even when my health meant I had to take long hiatuses. Thank you to all of my co-authors and especially Vasilis Kemerlis for believing in my research. My friends kept me sane and introduced me to new perspectives on the world. My family likes to tally up the number of years I have spent in graduate school despite originally being homeschooled, but they were steadfastly supportive.

Thank you all.

Preface

"May you live in interesting times," says the apocryphal Chinese curse.

"Quick, fashion a climbing harness out of a cat-6 cable and follow me," replies XKCD.

Perhaps one would expect this section to be referring to the COVID-19 pandemic. True, I had to adapt quite quickly to an online lifestyle, fleeing New York City when my roommate fell ill. I was one of the first students in our department to have a paper at an online conference; I was one of the first to have an online defense (across two zoom rooms). One could say I should have seen it coming. I flew to Tokyo on December 30th, 2019, and there in Narita was a small but prominent sign saying something about a virus in Wuhan—additional screening if you had visited there. I remember the sign, but doubt I took a picture of it. I had been awake for 20+ hours and was focused on clearing customs and meeting my friend Hidenori for dinner. I usually travel a lot; the pandemic uprooted me. But I'm content to settle into a different kind of life for the time being.

In fact, the main topic I want to address in this section is not the pandemic, but rather something that had a much bigger impact on my doctoral studies. Since November 2014, I've had a medical condition that prevents me from typing on a computer keyboard properly. Essentially, I can work for a few minutes (or even hours in my good months), but if I exceed that limit, I experience severe inflammation in my hands that can take several days to recover. My ankles also become inflamed if I walk too much, and my nervous system is hypersensitive, which means (among other things) that I get an immediate headache when exposed to smells and thus can only eat the blandest of foods. My current diagnosis (it has changed a few times) is rheumatoid arthritis with fibromyalgia. These conditions can be treated but not cured.

As you can imagine, it has been a difficult adaptation for me, both mentally and physically. I wear compression gloves and socks and elbow padding, and use all kinds of ergonomic devices. Most significantly, I interact with computers (and phones) almost entirely by voice. I use custom voice grammars that are designed for programming and general-purpose, precise interactions.

The vast majority of this thesis document was dictated by voice. It is not my main research area, but I have done a little research in voice recognition systems. If you are curious, I gave a talk on coding by voice at HOPE XI in 2016 `https://youtu.be/YRyYIIFKsdU` and another at CCC in 2020 `https://youtu.be/t_nC4sUwXno`.

Speaking all the time was unnatural and tiring at first, but quickly became a way of life. When I was cramming for a paper submission to OSDI, I lost my voice after using voice recognition for about 16 hours straight. I called my collaborator, Bill Aiello, and croaked out the problem. He said "Well, that's an unexpected failure mode," and wrote the final paragraph for me. Later, I took vocal lessons to learn to use my voice effectively. I started recording all of my speech utterances in preparation for training new speech models. After a few months, I quickly accumulated 150,000 utterances. Let's just say, my B-grade superpower is that I never have trouble spelling things out over the phone (with the NATO phonetic alphabet).

The other factor that made my research possible, besides speech recognition, was extensive collaboration. I worked with many students mostly in a pair programming capacity, so that I would not have to type as much. I always explained what we were working on to keep students engaged and learning, and for me, it was a real lifesaver. For the last few years, I worked with 3-6 students per semester, and many worked with me semester after semester until they graduated. My advisor, Junfeng Yang, was also very understanding when I had to take time off or could not work as quickly as I would have liked. I will always be grateful for all the support I received.

Interesting life, indeed. I could go on—about growing up in a solar-powered house—about international science fairs—about bushwhacking in Hawaii, a con-man in Shanghai, engine fires in Africa—but for now, let's leave it at that.

—David Williams-King, August 2020

Chapter 1: Introduction

Software deployment pipelines are getting deeper and more complicated. They are optimized for straight-line development, writing code and running some tests, and delivering code to production as quickly as possible. It can be difficult to incorporate new compilers or other tools to improve security of production systems. Thus, in many contexts, from open source development to commercial DevOps environments, binary-level analysis and transformation tools are desirable (see Section 2). However, adding features, fixing bugs, and mitigating security vulnerabilities is difficult without replicating the full build environment. Fundamentally, binaries are opaque artefacts of an automated build, intended to be used as-is or recreated from scratch if their functionality is not as desired.

Nevertheless, several techniques to transform binary software have been created. Broadly, these techniques can be categorized as 1) binary rewriting, 2) process virtualization, and 3) binary recompilation. Binary rewriting preserves the structure of the executable, a fairly simple technique but difficult to apply at scale. Process virtualization has to dynamically discover code, emulating original addresses and instruction layout. Binary recompilation is the most efficient, as all changes in address layout are computed statically, and generated output programs can run baremetal with near zero transformation overhead. However, any small error in disassembly means recompilation will likely fail. Hence, recompilation cannot be reliably applied to every binary (unlike virtualization), but has excellent performance when disassembly is accurate. In this thesis, we describe the first general-purpose binary recompilation engine (Egalito).

Once an effective binary transformation mechanism (such as recompilation) is in place, there are myriad possible defenses. We wrote several demonstration tools with Egalito, including code randomization, control-flow integrity, retpoline insertion, and an AFL [3] fuzzing backend. We also wrote Nibbler, which detects unused code and removes it. Many of these features, including Nibbler, can be combined with other defenses resulting in multiplicatively stronger or more effective

hardening. Enabled by our recompiler, an overriding theme of this thesis is our focus on deployable software transformation. We focus on creating defenses that leave system components unmodified whenever possible. Our goal is to run defenses entirely in userspace, operating on binaries that would normally be present on a system, and to allow future researchers to build upon our systems.

We coined this term *egalitarian* in the context of binary security. Simply put, an egalitarian analysis or security mechanism is one that can operate on itself (and is usually more deployable as a result). In other words, no additional privilege level is required to enforce security; the security mechanism applies to itself. For a binary transformation tool to be egalitarian, it must be able to parse and transform itself, the same way a bootstrapping compiler can compile itself. As one demonstration of this idea, we created a strong, deployable defense against code reuse attacks, called Shuffler. Shuffler randomizes function addresses, moving functions periodically every few milliseconds. This makes an attacker's job extremely difficult, especially if they are located across a network (which necessitates ping time)—JIT-ROP attacks take 2.3 to 378 seconds to complete [1, 2]. Shuffler is egalitarian and defends its own code and target code simultaneously—Shuffler actually shuffles itself.

This thesis incorporates three major projects:

- Chapter 3: Egalito (binary recompilation), published in ASPLOS 2020 [4];
- Chapter 4: Nibbler (debloating), which appeared in ACSAC 2019 [5], and for which an expanded journal version appeared at DTRAP 2020 [6]; and
- Chapter 5: Shuffler (code re-randomization), published in OSDI 2016 [7].

The thesis also incorporates a workshop paper CodeMason [8] on profile-guided optimization (see Section 3.6.4). Two additional works, on disassembly with machine learning, are primarily authored by Kexin Pei and are not incorporated into the thesis. They are XDA [9], which appeared in NDSS 2021, and StateFormer [10], which will appear in IEEE S&P 2021.

1.1 Research Hypothesis

Using metadata in modern binaries, static binary analysis can recover all code and code pointers in a large set of binaries at least 90% of the time. With knowledge of all code and pointers, we can construct a binary recompilation framework (Egalito) that incurs less than 1% transformation overhead. We can then implement sophisticated defenses such as retpolines, CFI, code debloating (Nibbler), and continuous code randomization (Shuffler).

1.2 Contributions

The main contributions of Egalito are as follows:

- We define a binary transformation framework to be *layout agnostic* if it can individually relocate or resize each binary element, without reliance on patching or address virtualization.
- We present the *Egalito recompiler*, a binary transformation framework built around a layout-agnostic IR called *EIR*. Egalito-transformed binaries achieve excellent performance: SPEC CPU 2006 has a 1.7% speedup.
- We demonstrate the usefulness of binary-level architectural adaptation with a retpoline defense against Spectre [11] and a software implementation of Intel's CET [12] to augment hardware and compiler deployment as it rolls out.
- We demonstrate Egalito with a total of nine transformation tools, including a continuous code randomization defense (JIT-Shuffling [7]), which operates from a fully self-hosted environment where tool code is itself defended recursively.
- We present a large-scale study of 867 Linux executables/libraries, and show that Egalito can fully analyze and recover all cross-references 99.9% of the time. Furthermore, 90 of 149 Debian packages pass all tests after binary rewriting (40 straightforward known failures).

The main contributions of Nibbler are as follows:

- We design and develop a practical system, Nibbler, which removes bloat from binary shared libraries without requiring source code and recompilation. In addition, we devise a novel method for detecting unused address-taken functions, which allows Nibbler to eliminate, safely, even more unused code.

- We evaluate the debloating capabilities of Nibbler with real-world binaries and the SPEC CPU suite, and show that it removes 33%–56% of code from libraries, and 59%–82% of the functions in scope. We also apply Nibbler on ≈30K C/C++ binaries, and ≈5K unique dynamic shared libraries, as well as on nine official Docker images.

- We demonstrate the benefits of debloating to security schemes, like continuous code re-randomization, by integrating Nibbler with an existing system (namely Shuffler [7]), observing a 20% run-time improvement for Nginx compared with Shuffler on its own.

Our contributions with Shuffler are as follows:

- We design a re-randomization defense against JIT-ROP and code reuse, which runs without modification to the source, compiler, linker, or kernel, and with minimal changes to the loader.

- We introduce a real-time deadline on the order of milliseconds for any disclosure-based attack, using a new asynchronous re-randomization architecture that has low latency and low overhead.

- We describe how we bootstrap our defense into an egalitarian self-hosting environment, thus avoiding any expansion of the trusted computing base.

For more details on Egalito, see Section 3; for Nibbler, Section 4; and for Shuffler, Section 5.

Chapter 2: Background

2.1 Deployable Transformations

An overriding theme of this thesis is our focus on deployable software transformation. In the area of computer security, researchers constantly develop a wide range of interesting defenses, but they are clumsy and difficult to deploy outside of e.g. an exact image of the researcher's machine. Part of the problem is that prototypes often modify significant components of the system, such as the compiler [13, 14, 15, 16, 17], runtime environment [16, 17], operating system kernel [13, 18, 19], or hypervisor [18, 20, 19, 21]. This has three primary impacts. First, the prototypes cannot be maintained and quickly succumb to bit rot—no longer running correctly in the face of new software versions. Second, of course, real-world uptake of these defenses is slow or nonexistent. Third, and most significantly, it seems relatively uncommon for researchers to build upon prior work of other researchers.

Perhaps this is a natural state of affairs for academic projects, but it means that there is a lot of duplication of effort, especially in the area of binary analysis. Thus, in this thesis, we focus on creating deployable defenses that leave system components unmodified whenever possible. Our goal is to run defenses entirely in userspace, operating on binaries that would normally be present on a system. Real-world uptake may be unrealistic given the incompleteness inherent in binary recompilation, but we hope to at least allow future researchers to build on our software artifacts.

2.2 Why Binary-Level Transformation

Often, the simplest way to change how code is generated is to go to the origin: the compiler. The compiler has a complete picture of code structures and data flow within a program, and it is fairly easy (since the advent of LLVM [22]) to write compiler passes to change its behaviour.

However, the developer must then recompile the target source code with this modified compiler, a task which has a surprising number of challenges involved. Let's consider a few scenarios below.

Open source In the open-source setting, developers collaborate to push out new versions of the source. Individual developers might be responsible for compiling binaries sent to users, but in larger projects there is typically a continuous-integration build environment that produces binaries automatically (potentially for multiple architectures or platforms). In many cases, individual projects don't even control this build environment. For example, most Linux distributions generate binaries in large build farms—e.g. Debian uses the autobuilder network [23]. This means developers can't build their projects with a custom compiler; they must rely on mainlined features already included in the distribution's compiler packages. Distributions where users might compile their own packages like Arch and Gentoo have more flexibility, but relying on a frozen compiler version with precisely the right configuration is still not a good long-term solution.

In other words, to effectively transform open source code by modifying the compiler, those changes must be small and self-contained enough to be accepted into the mainline version of the compiler. This is not a problem for straightforward mechanisms, but radical security improvements are difficult to get merged, especially because such defenses often incur significant overhead.

Commercial In modern companies, the traditional separation between developers (who write code), testers (who check it), and operations engineers (who deploy it) is shrinking. With increased automation, all of these tasks are handled by one group, DevOps engineers [24]. Version-control commits are funnelled through a single pipeline which automatically builds executables and runs continuous-integration tests, and software deployment to production is simplified as much as possible. This strategy accelerates the software development lifecycle, but unfortunately, it places individuals in charge of the security of their own code. As a rule, unless the pipeline automatically applies a security mitigation, it won't be applied.

In this environment, since full source code is usually available at some point in the pipeline, source-level transformations and hardening do make sense. It can be difficult to get a particular

compiler to be used across different containers and buildsystems, but it is possible to insert another tool to do transformations. This is the approach taken by several commercial products such as Synk [25]. However, security experts that investigate and address security issues—"DevSecOps"—often work backwards and try to figure out how individual developers decided to build systems. Binary-level tools are the only option to provide fast turn-around time during investigations, but the focus is more on analysis rather than transformation. Binary hardening (especially of third-party libraries) may be relevant to integrate earlier in the continuous-integration pipeline.

Embedded/Legacy For embedded systems, especially Internet-of-Things devices, the vendor likely has significant pressures to deliver products, and may not have much focus on security. As a third party investigating embedded systems, dealing with binary programs and firmware is inescapable. Similarly, legacy systems, which may no longer build or for which the source code may have been lost, must also be analyzed in binary form. Users may wish to transform these systems to fix security problems.

We do not focus on embedded and legacy systems in this work, because significant effort is required to disassemble and decompile these types of binaries. The hardware running the code has fewer resources and as a result, compilers are more likely to take shortcuts such as intermixing data and code. Older code also has less metadata available. However, since binary-level analysis is the only option, users may still apply techniques like ours with the addition of some manual effort to achieve patching and hardening goals.

2.3 Egalitarian Security Mechanisms

We coined the term *egalitarian* in the context of binary security. Simply put, an egalitarian analysis or security mechanism is one that can operate on itself. In other words, no additional privilege level is required to enforce security; the security mechanism applies to itself. For example, if the security mechanism defends a target program's code, it must defend its own code equivalently. For a binary transformation tool to be egalitarian, it must be able to parse and transform itself, the

same way a bootstrapping compiler can compile itself. Obviously, this requires tools to be written in a compiled language like C++ and not an interpreted one like Python.

We care about egalitarian defenses because they are usually more deployable. The kernel does not need to be modified to defend userspace, and the hypervisor does not need to be modified to defend the kernel. Furthermore, when a transformation tool can transform itself, it becomes possible to do binary analysis at runtime. This allows dynamically-loaded code to be analyzed and defended, and even enables defenses that rely on code generation that would not otherwise be possible. Sometimes this recursive nature will simplify designs as well, while e.g. DynamoRIO ends up with three copies of glibc mapped per process.

2.4 Recompilation Related Work

Several techniques to transform binary software have been created. Broadly, these techniques can be categorized—see Figure 2.1—as 1) binary rewriting, 2) process virtualization, and 3) binary recompilation. Traditional binary rewriting preserves the structure of the executable, inserting detours to new code located at new addresses. Process virtualization transforms the code inline but emulates the original addresses and instruction layout. Finally, binary recompilation involves lifting binaries to an intermediate representation, then generating a new executable layout; no emulation of original addresses is needed. All of these techniques are discussed further below.

Rewriting Via Code Patching Statically rewriting a binary by installing hooks or trampolines in the original code (binary patching) is simple and efficient. Examples of patching-based rewriters include PEBIL [26], REINS [27], and RevARM [28]. Patching can also be deferred until runtime as in Dyninst [29]. These systems do not attempt to find and transform all pointers in a binary, and overhead increases with more modifications, limiting the scale of the approach.

Process Virtualization In dynamic binary translation (DBT), a process virtual machine transforms each basic block, just-in-time, before it is executed. Existing DBT systems include DynamoRIO [30], Pin [31], Valgrind [32], and (on ARM64) Mambo [33]. These systems are not designed for

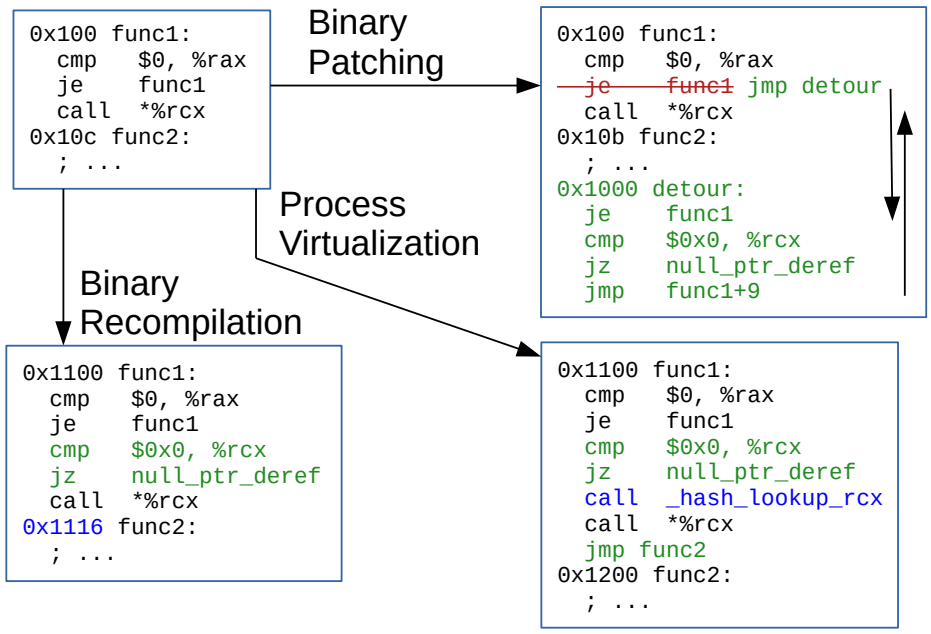


Figure 2.1: Examples of different forms of binary transformation. In each case, the code in the top left is transformed to check for a NULL pointer before the indirect call instruction. Binary recompilation results in the simplest code, but potentially all addresses may need to be reassigned.

security: for instance, a DynamoRIO tool that protects code pointers has no way to defend the code pointers in DynamoRIO's code cache or translation tables (and proof-of-concept exploits already exist [34]). Furthermore, DBT incurs substantial runtime overhead: 887% (DynamoRIO) and 3421% (Pin) on a large set of real-world x86_64 programs, and 28%-34% average (Mambo) on SPEC CPU. Valgrind, with its higher-level IR called VEX, has 330% overhead on SPEC CPU. Thus, existing DBT frameworks are ill-suited for binary hardening.

The PSI [35] platform is designed like a DBT system, implementing a process virtual machine and preventing control flow from escaping the instrumented version of the code. However, PSI operates through binary rewriting, statically inserting all the code necessary into an executable. It reuses a disassembler from prior work [36, 37]. PSI has a flexible architecture, and is designed for security, but does incur nontrivial virtualization overhead (53% on a large set of programs).

Multiverse [38] conservatively disassembles a binary starting from every offset within the

`.text` section. This guarantees a complete but imprecise disassembly. Hence, they cannot easily identify all valid code pointers. Accordingly, Multiverse virtualizes addresses and preserves input program layout, incurring 60.42% average overhead on SPEC CPU, with 288% overhead in the worst SPEC CPU case.

Recompilation/Reassembly Binary analysis suites [39, 40, 41, 42] often include sophisticated analyses to lift binaries without metadata into a high-level IR (such as VEX or BIR). The most popular is LLVM IR; frameworks that lift to LLVM IR include llvm-mctoll [43], fcd [44], and remill [45]. Most frameworks are not designed to regenerate code after analysis.

However, some works including SecondWrite [46] and McSema [47] lift binaries to LLVM intermediate language and regenerate a new executable using LLVM's standard backend. SecondWrite operates on arbitrary binaries with no metadata and speculatively disassembles and lifts all parts of an executable that could be code. Because it does not necessarily find all pointers, SecondWrite includes a full copy of the original binary mapped at the original address to service read requests.

Two recent works, Uroboros [48] and Ramblr [49], implement binary reassembly. Their aim is to fully solve the disassembly problem, and lift a binary into a `.s` file which can be processed by a standard assembler. These are the first works to recognize the potential of layout-agnostic binary rewriting. However, since they are targeting arbitrary binaries, these systems must use complex and expensive speculative disassembly techniques, with no guarantee of success. They also provide no IR beyond flat generated assembly.

Chapter 3 focuses on our implementation of binary recompilation.

2.5 Attack/Defense Related Work

Once a user has the ability to readily transform programs, for example at the binary level with binary recompilation, many applications become possible. We focus on the security context and on which defenses are possible to add to a binary after it has been compiled. Chapter 4 demonstrates one possible defense, and Chapter 5 focuses on one of the most sophisticated, a re-randomization

technique. The rest of this section details some of the history behind attacks and defenses on programs written in memory-unsafe languages like C and C++.

Attack taxonomy Many attacks seen in the wild against running programs are based on control-flow hijacking. An attacker uses a memory corruption vulnerability to overwrite control data, like return addresses or function pointers, and branches to a location of their choosing [50]. In the early days, that location could be a buffer where the attacker had directly written their desired exploit code, thus enacting a so-called *code injection* attack. Nowadays, the widespread deployment of Write-XOR-Execute (W^X) [51] ensures that pages cannot be both executable and writable, which has led to the effective demise of code injection.

In response, attackers began to create *code reuse* attacks, stitching together pieces of code already present in a program's code section. The first and simplest such attack was return-to-libc (`ret2libc`) [52, 53], where an attacker redirects control flow to reuse whole `libc` functions, such as `system`, after setting up arguments on the stack. A more sophisticated technique called Return-Oriented Programming (ROP) [54] was soon discovered, where an attacker stitches together very short instruction sequences ending with a return instruction (or other indirect branch instructions [55, 56])—sequences known as gadgets. The terminating return instruction allows the attacker to jump to the next gadget, and the attacker may set up the stack to contain the addresses of a desired "chain" of gadgets. ROP has been shown to be Turing-complete, and there are tools known as ROP compilers which can automatically generate ROP chains [2].

Control-Flow Integrity The research community has proposed three main categories of defenses against code reuse. The first is Control Flow Integrity (CFI) [57], which tries to ensure that every indirect branch taken by the program is in accordance with its control-flow graph. However, both coarse-grained CFI [58, 36] and fine-grained CFI [59] can be bypassed through careful selection of gadgets [60, 61, 62].

Code Randomization The second category of defense is code randomization, performed at load-time to make the addresses of gadgets unpredictable. Module-level Address Space Layout Randomization (ASLR) is currently deployed in all major operating systems [63, 64]. Fine-grained randomization schemes have been proposed at the function [65], basic block [37], and instruction [66] level. These defenses spurred a noteworthy new attack called Just-In-Time ROP (JIT-ROP) in 2013 [1]. In JIT-ROP, the attacker starts with one known code address, recursively reads code pages at runtime with a memory disclosure vulnerability, then compiles an attack using gadgets in the exfiltrated code. No load-time randomization scheme can stand against this attack.

Execute-Only Memory If the hardware supports execute-only memory, which cannot be read, this is a strong defense against JIT-ROP. On x86_64, execute access implies read access for historical reasons, and execute-only memory is only possible by using a hypervisor [18, 20]. It is also possible to emulate execute-only memory with software fault isolation, whether on x86_64 [19] or ARM64 [17]. Nevertheless, attacks are possible against execute-only memory, such as Address-Oblivious Code Reuse [67] and other techniques [68, 69].

Re-Randomization Re-randomization is seen as an alternative to execute-only memory. In this model, assume the attacker has access to memory disclosures and will eventually unravel any one-shot randomization. Some work rerandomizes on crashes [16], system calls [13, 70], or every N seconds [71]. However, the threat model must be carefully constructed to ensure that attackers cannot carry out an attack quickly or stealthily enough.

Although re-randomization is less powerful than execute-only memory, it is much easier to create a deployable implementation. This is the focus of Chapter 5.

Chapter 3: Binary Recompilation (Egalito)

3.1 Introduction

Software written in compiled languages ultimately runs in binary form, and the majority of software distributors provide binaries directly to their end-users. Since binaries are so widespread, it is desirable for many DevOps activities, including profiling, security hardening, and architectural adaptation, to apply directly to binaries. Applying changes to source code or compiler infrastructure has high turn-around time, requires the cooperation of many parties, and may not be possible in the case of commercial or third-party libraries and applications. Furthermore, security hardening and architectural adaptations must be applied comprehensively to all library dependencies, or risk compromise through an untransformed component; only at the binary level is it possible to transform all code that will actually run.

However, manipulating binaries directly is difficult. The developer who examines binaries feels more like an archaeologist than an engineer—dealing with artefacts of an unknown buildsystem, no blueprints, and many details lost to time. Automated binary rewriting tools must treat binary code as a black box that can be emulated but whose structure is unknown. The main difficulty of binary rewriting is in handling unidentified code and code references (pointers). Existing binary rewriting frameworks use either: 1) a process virtual machine to translate references on the fly; or 2) binary code patching, leaving code at original locations. Common frameworks like DynamoRIO [30] and Pin [31] use virtualization. Binary patching, on the other hand, often requires significant expertise with reverse-engineering tools like IDA Pro [72]. Both mechanisms incur performance overhead that is likely unacceptable for production environments. Generated outputs also do not behave like ordinary binaries, impeding debugging and compatibility with other tools.

In this work, we aim to create a binary rewriting framework that manipulates and outputs

ordinary-looking and *highly performant* binaries. Specifically, we aim to allow program code (and layout) to change arbitrarily without any constraints from the original binary—i.e., the framework is *layout agnostic*. This requires complete disassembly: we must have confidence that every pointer has been found and updated to allow the code to run bare-metal within a new address-space layout. Our observation is that while malware analysis is and will always be a herculean task [73], for binaries the user depends on, it is important simply to be able to handle the actual output of compilers. Furthermore, today's binaries have recently started to include more metadata, which can assist with analysis. Most importantly, major Linux distributions have shifted to position-independent binaries over the past few years [74, 75, 76, 77]. Note that position-independent is a much weaker property than layout-agnostic: the former allows a single linear shift through the selection of a base address, while the latter allows piecewise permutation or relocation of each instruction. Nevertheless, this additional (position-independent) metadata allows more powerful binary analyses than in the past.

We present *Egalito*: a binary transformation framework that performs *complete* and *precise* binary analysis, and can generate output binaries that do not use patching or virtualization. Egalito lifts (stripped) modern Linux binaries into a standalone, layout-agnostic intermediate representation (IR) that allows arbitrary modifications to program layout. Essentially, Egalito is a compiler backend in reverse, followed by transformations and then normal code generation. Hence, we call Egalito a *binary recompiler*. Tools written with Egalito are structured as modular *recompiler passes* that manipulate IR. Egalito is fully functional on x86_64 and ARM64 architectures, with RISC-V support underway.

Our IR is not a high-level representation like LLVM IR [78] that a compiler (i.e., LLVM) uses for optimization [22]; rather, it is lower-level, like LLVM's `MachineInstr` or GCC's Register Transfer Language (RTL) [79]. Lifting to a higher-level intermediate language would abstract each instruction into multiple operations, not tied together semantically, which might be modified, reordered, and optimized independently. Code generation would become difficult, and likely diverge significantly from the input assembly. However, for low-level binary instrumentation or

hardening, producing a differently-optimized version of the code is counterproductive. Instructions might have been carefully chosen to work around low-level architectural issues (e.g., Spectre [11]), or to perform additional security checks; an optimizing framework could undermine the defense. Hence, we deliberately use a lower-level IR, in order to make output code generation more predictable and more in-line with the original input.

As a result of our layout-agnostic design, binaries transformed by Egalito have excellent performance. On SPEC CPU 2006, Egalito incurs 0.46% overhead, which becomes a 1.7% performance *speedup* when we enable some simple binary-level optimizations. Sometimes, even when security hardening is applied, the transformed program may run faster than the original, e.g., we see a 1.4% performance speedup with lightweight control flow integrity. With our profile-guided optimization tool, performance can potentially become even better (the best SPEC CPU case observes 11.8% speedup). Our AFL fuzzing backend is 18-61x faster than other binary-level fuzzing. Hence, Egalito provides a realistic way to introduce binary modifications with production-level performance.

A binary rewriter that relies on completely accurate disassembly must have high confidence in its analyses. Thus, we tested Egalito thoroughly across multiple Linux distributions during its development, including Debian, Ubuntu, openSUSE, Fedora, and Gentoo. Our evaluation of 172 Debian packages containing 867 executables and libraries showed that in 866 cases (99.9%) Egalito correctly recovered all cross-references and jump table metadata (the most challenging aspect), including table bases, invocations, and bounds. After transformation, 90 of 149 Debian packages pass all tests—and fully 40 of the failures are due to a detectable binary-generation issue that can be addressed with additional engineering. As further evidence of completeness, Egalito is able to analyze and transform itself, analogous to a bootstrapping compiler. Our system is currently in use by researchers and instructors at several other institutions, and we hope that a wider userbase will give it even better robustness over time.

Our end goal is to have a full Linux distribution where every program can be readily transformed at the binary level to adapt to new attacks or architectural quirks [80]. A security-conscious user may be willing to trade off performance for security, enabling a suite of defenses on particular

applications. We implemented nine binary tools atop Egalito, most of which increase security—including a JIT-Shuffling defense [7] which relocates functions periodically to random addresses, in a fully self-hosted environment with no untransformed code. A hardware designer creating new hardware might wish to remove—or add—certain sequences of instructions, without having to modify a compiler and recompile userspace. Two of our Egalito tools perform such binary-level architectural adaptation. One is a retpoline defense against Spectre [11], which replaces problematic instructions with new sequences. Another is a software implementation of Intel's Control-flow Enforcement Technology (CET) [12], which augments hardware and compiler deployment by handling partially-present instrumentation. All of our tools run on stripped binaries, built with typical flags: i.e., those used to build distribution-standard `.deb` or `.rpm` files.

Egalito has been released [81] under an open source license (GPL v3) and may be found at `https://egalito.org/` [82].

3.2 Design

We designed Egalito according to the following goals:

- **Performance:** The framework should introduce near zero overhead, and therefore avoid using binary patching or address virtualization machinery.
- **Flexibility:** The framework should enable arbitrary code insertions and deletions without concern for address space layout (i.e., it should be layout-agnostic).
- **Deployability:** The framework should operate on ordinary (stripped) binaries, e.g., `.deb`/`.rpm` archives, without requiring special metadata or compiler flags.
- **Bootstrapping:** The framework should provide runtime support through (egalitarian) self-transformation.

The Binary Landscape Binary rewriting has historically been considered fragile: rewriting might work in some cases and fail in others. Traditional executables are stripped and position-dependent,

and are very difficult to analyze. Recently, however, executables have begun to include more metadata by default. Most importantly, Linux distributions have migrated to using position-independent binaries over the past few years [74, 75, 76, 77]. Position-independent code (PIC) may be loaded at any base address, a feature used to implement shared libraries for decades, but enabled now for executables to strengthen Address-Space Layout Randomization (ASLR) defense [83]. Position-independent binaries contain more metadata, though not enough to make analyses straightforward—e.g., jump tables have no associated metadata. Yet, this shift to PIC is what enables our analyses.

Our analyses are sufficient to handle our target binaries, the types of ELF binaries that appear in current `.deb` or `.rpm` archives: position-independent, optimized, and stripped. However, our analyses are relatively straightforward and could become simpler over time. It is always possible for compilers to provide disassembly ground truth with extra metadata [80]. We hope Egalito will widen the demand for binary transformation and help compiler writers judge what metadata is helpful to include for binary analysis. Since we use some heuristics for jump table analysis, Egalito gives up on completeness (see Section 3.7). In exchange, we obtain a much more powerful and complete binary intermediate representation that represents the internals of a binary program.

Some binary rewriting techniques over-approximate the code (e.g., by disassembling from every offset in `.text` [38]). Others wait and discover the true extent of code at runtime (an under-approximation). We aim to statically uncover the precise set of code and control-flow in a program, so that we can generate standalone code without virtualization machinery. We also insist on avoiding binary patching, which means that we must find all references within a program; we cannot leave trampolines behind at original function addresses to catch errant calls through untransformed pointer values. Any virtual address in the input binary can be repurposed in the output. This recompiler output should look rather closer to the output of a real compiler than that of a binary rewriter. There is no copy of the original `.text` to service reads and potentially provide security vulnerabilities; it is not overwritten with `hlt` instructions, but simply not included in our new ELF.

Egalitarian Capabilities In the tradition of bootstrapping compilers, we designed Egalito to be

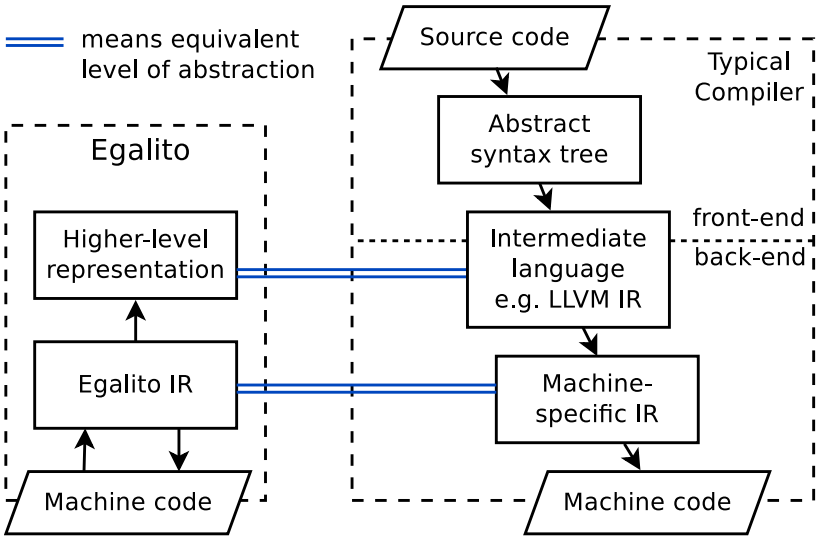


Figure 3.1: Egalito IR (EIR) design. Egalito reverses a compiler backend to obtain an IR similar to a machine-specific IR, and augments it with higher-level data structures.

able to analyze and transform itself. This is possible in part because Egalito is written in a compiled language (C++). We provide a custom loader which analyzes itself, the executable, shared libraries, and Egalito, at load-time; it then bootstraps into a fully self-hosted environment where the only code present in the address space is code that Egalito has generated (and the `vDSO` kernel interface). This enables transformations that need dynamic analyses or runtime code-generation. As an example of this, we provide JIT-Shuffling (based on an earlier egalitarian defense, Shuffler [7]), which continuously generates its own code at new random addresses, with no undefended (fixed-address) code. For details, see Section 3.5. From a security perspective, egalitarian transformation can often enforce security isolation without requiring additional levels of privilege (e.g., kernel assistance).

Intermediate Representation The defining design decision of Egalito is its choice of intermediate representation (IR). There are many possible types of IR, which can be roughly categorized as follows, from front- to back-end in the compilation process: 1) abstract syntax trees, closely tied to the original source language; 2) intermediate languages such as three-address code or LLVM IR, used for optimization [22]; and 3) low-level machine-specific intermediate representation for code generation and peephole optimization (LLVM's `MachineInstr`, GCC's RTL). Figure 3.1

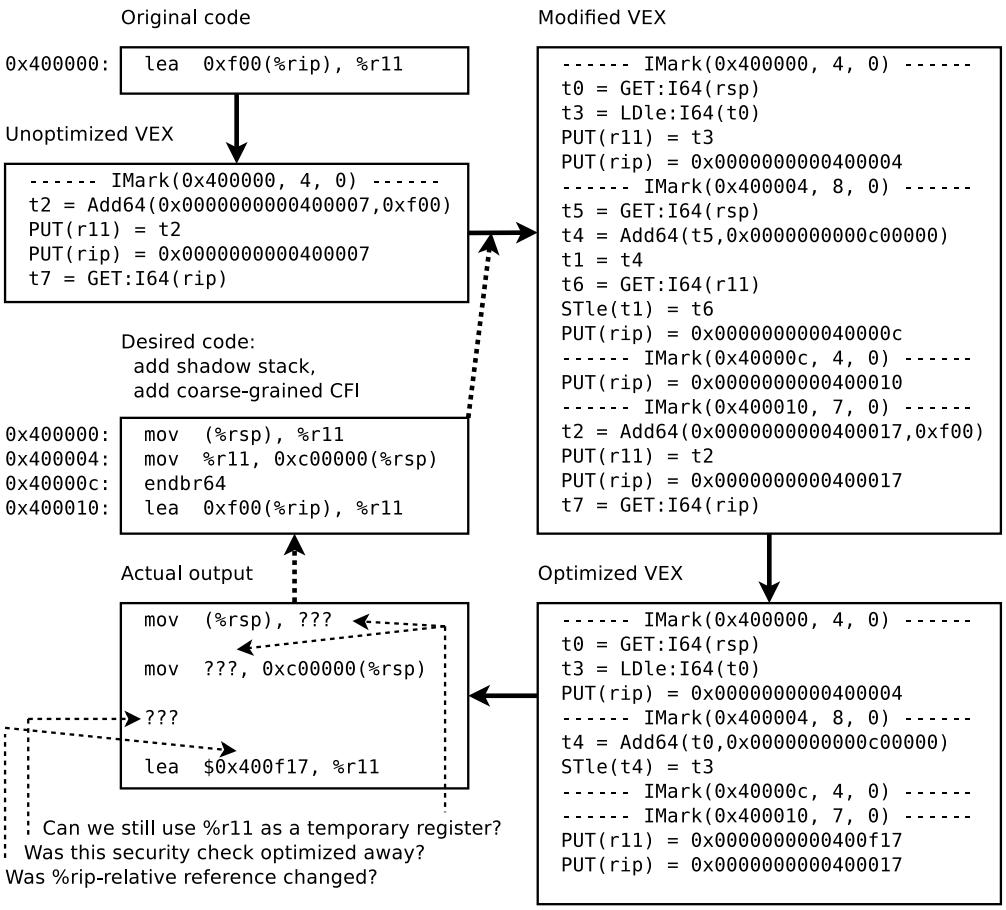


Figure 3.2: In-depth example of why VEX IR is not suited to binary recompilation.

shows the basic structure of a compiler's intermediate representations. A recompiler consists of two pieces: first, the inverse of a compiler backend, to turn binary code into an IR; second, a forward mechanism to turn the IR back into binary code.

Intuitively, lifting machine code to the lowest-level IR is simplest, while higher-level IR provides more expressive analysis and transformative power. With Egalito, we reverse the compilation process only to the machine-specific IR level (see Figure 3.1), instead of all the way to an intermediate language like VEX or LLVM IR. Lifting to a higher-level intermediate language would typically turn each assembly instruction into multiple operations in SSA (single static assignment) form. The set of operations created for an instruction would not be tied together semantically, and might be modified, reordered, and optimized by existing infrastructure. Code generation would then become difficult, and likely diverge significantly from the input assembly. Hence, we deliberately

chose a lower-level IR to make the output code generation more predictable and more in-line with the original input.

Here is an example of why existing IRs are unsuited to binary compilation. VEX (used by Valgrind [32] and angr [39]) uses a single representation for both relative and absolute references, making code regeneration difficult; inserting code in VEX also requires modifying the addresses of all subsequent instructions manually. Figure 3.2 shows an example of starting with one instruction, and trying to add three instructions to implement security techniques. If the code is lifted to VEX, and modifications are made in VEX, the first problem that arises is moving the original structure to a new address; there is no differentiation between addresses and constants. Then, if optimizations are applied (shown are default optimizations from Valgrind [32] and Angr [39]), there is not enough information to recreate instructions. Even if the code is left unoptimized, the fact that the original instruction was position-independent (%rip-relative) is lost in the VEX representation.

By using our own IR, we do give up on reusing the substantial existing code that operates on established IRs. One of the main benefits of LLVM IR is its suite of existing analyses, optimizations, and backends. However, for low-level binary instrumentation or hardening, producing a differently-optimized version of the code is counterproductive. The transformations may be working around low-level issues (e.g., Spectre [11]) that are invisible to the optimizing framework; or transformations might add checks that the framework would rather optimize away, undermining a defense. Furthermore, intermediate languages are simply not designed for code modification and regeneration. Hence, we designed a custom IR with precisely the properties we need for layout-agnostic binary recompilation. Details follow in Section 3.3.

3.3 Intermediate Representation (EIR)

Our Egalito IR (*EIR*) is a C++ class hierarchy that can be viewed as a complete abstract syntax tree of the ELF input binary. It stores instruction encodings placed in architecture-independent categories (e.g., `ControlFlowInstruction`). It also stores semantic information that we recover about an ELF, such as jump tables and control flow. EIR has two major innovations

compared to typical binary rewriters.

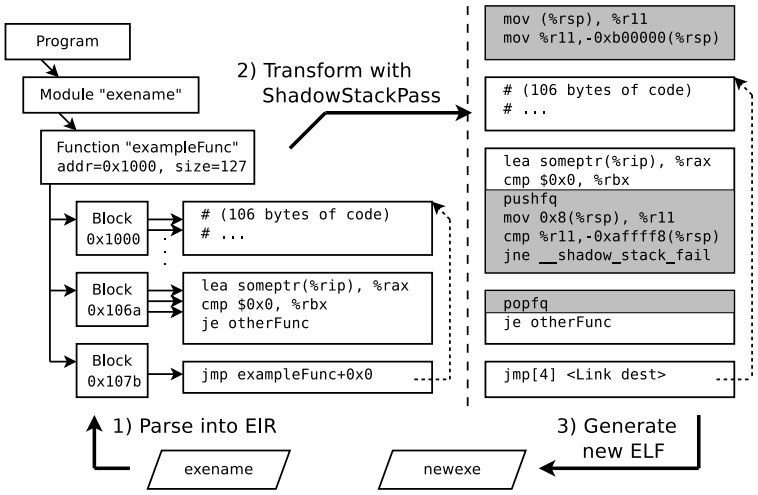
First, in a departure from normal parse trees, we abstract addresses into *links*, and then store both addresses and links. The addresses in each tree node allow EIR to represent an original ELF precisely, and enables minimal changes when performing ELF-to-ELF recompilation. Meanwhile, links are key to supporting layout-agnostic user modifications, since targets may be resolved even if addresses are reassigned or if new code is inserted. In the latter case, assigned addresses may overlap until they are recomputed during a "linking" step that moves nodes such as basic blocks to new non-overlapping addresses. Egalito tools can rearrange addresses directly, or rely on the framework to generate non-overlapping addresses after modifications.

Second, we extend EIR with high-level use-definition/def-use data structures. These data structures are read-only and ephemeral, meaning they are only valid for a specific EIR state but can be recreated at any point from EIR. They provide access for analyses possible only on higher-level IRs, such as our own jump table analyses, while EIR remains low-level to enable efficient code regeneration. EIR is the canonical representation on which all modifications must take place.

3.3.1 Shadow Stack Transformation Example

In Figure 3.3, we show an example transformation tool that adds a shadow stack to ordinary x86_64 binaries. This shadow stack is located at a constant offset (`-0xb00000`) from the real stack [84, 85]. The majority of the code is written in one recompiler pass, shown in Figure 3.3b. The code is simplified for brevity. Our full implementation creates `__shadow_stack_fail` with a single `hlt` instruction, and allocates the shadow stack memory region by adding a call at program start.

A recompiler pass is a `Visitor` as in the *Visitor design pattern* [86], able to access EIR nodes at any level of granularity by implementing `visit` functions. In this case, we visit each `Function` and add code in its prologue to save the return address to the shadow stack. We recurse on all (`Block`) children of the `Function`, and the default `Block` visitor recurses on all of its (`Instruction`) children. We visit each `Instruction` and look for ones that leave the



(a) EIR transformation example of adding a shadow stack. Input code is 1) disassembled, 2) transformed, and 3) regenerated.

```
class ShadowStackPass : public RecompilerPass {
public:
  virtual void visit(Function *function) {
    Instruction *instr1 = function->getChild(0)->getChild(0);
    Mutator::insertBefore(instr1, /*forceSameBlock=*/ false,
      std::vector<Instruction *>{
        ASM("mov (%rsp), %r11"),
        ASM("mov %r11, -0xb00000(%rsp)") });
    recurse(function);  // RecompilerPass::recurse
  }

  virtual void visit(Instruction *ins) {
    if(ins->isType<Return>()
      || (ins->isType<ControlFlow>() && ins->isExternalJump())
      || (ins->isType<IndirectJump>() && !ins->forJumpTable())) {

      // We clobber the RFLAGS register; Egalito saves it if
      // necessary, pushing stackAdded bytes onto the stack.
      auto addCheckLambda = [] (int stackAdded) {
        auto failFunc = Find::function("__shadow_stack_fail");
        return std::vector<Instruction *>{
          ASM("mov " << stackAdded << "(%rsp), %r11"),
          ASM("cmp %r11, " << -0xb00000+stackAdded << "(%rsp)"),
          ASM("jne " << new RelativeLink(failFunc)) };
      };
      Mutator::insertBefore(ins, /*forceSameBlock=*/ true,
        AddRegisterSaving({X86_REG_RFLAGS}, addCheckLambda));
    }
  }
};

int main(int argc, char *argv[]) {
  assert(argc == 3);  // usage: ./shadowstackify input output
  EgalitoInterface egalito;
  egalito.parse(argv[1]);
  egalito.getRoot()->accept(ShadowStackPass());
  egalito.getRoot()->accept(ReassignAddresses());
  return egalito.generate(argv[2]);
}
```

(b) C++11 code for shadow stack transformation.

Figure 3.3: Adding a constant-offset x86_64 shadow stack.

function: returns, external (tail) jumps, or indirect jumps/calls. At each exit point, we insert code to verify the return address against the saved shadow stack value. The inserted code modifies the flags register (`%rflags`), so we ask Egalito to save it if necessary—`stackAdded` indicates how many bytes Egalito pushed to the stack, and we emit code appropriately.

In the `main` function, we parse an input ELF file, run two passes, and then generate an output ELF. The first pass is the one in this example, while `ReassignAddresses` chooses non-overlapping addresses for all code—necessary since we insert instructions and increase the size of blocks and functions.

Figure 3.3a shows an example EIR tree structure, and the code transformations that `ShadowStackPass` performs. Its seemingly simple code insertions trigger operations of significant complexity. First, notice the `forceSameBlock` flag in the code. This controls whether instructions inserted at the very beginning of a block should be part of the same block (i.e., whether incoming jumps will run the new code). The initial shadow stack save uses `forceSameBlock=false`; it should execute only once when the function is first invoked. If any jumps target the first instruction in the block, Egalito creates a new earlier basic block for inserted instructions (as happens with the `jmp` in the last Block in Figure 3.3a). Conversely, incoming jumps to a block containing an exit jump should run the new code to check the return address before exiting. With `forceSameBlock=true`, Egalito reuses the same basic block for insertions. Links that originally targeted the first instruction in the block are automatically updated in constant time to point to the new first instruction.

The original `jmp` instruction had a one-byte displacement, since its offset was `-127` (the size of `exampleFunc`); the assembler uses the shortest possible encoding. However, after inserting some intervening code, a (signed) one-byte displacement no longer reaches the target. Egalito automatically re-encodes one-byte jumps that no longer reach their target to use 4-byte displacements in the `PromoteJumps` pass. Since one jump promotion can cascade and cause others to need promotion, this pass runs iteratively until a fixed point.

Next, consider the registers used by this example. We use `%r11` as a temporary register, but

do not ask Egalito to save this register, because `%r11` is callee-saved and may be overwritten by a function call. We do ask Egalito to save the flags register `%rflags` which is overwritten by our `cmp` instruction. `%rflags` is not expected to be preserved across function calls, but it must be preserved across conditional tail recursion (Figure 3.3a shows such an example with `je`). Egalito finds all jumps that perform tail recursion. Often, transformations will need to handle these cases specially, e.g., to prevent tail recursion from causing two shadow-stack pushes in a row without an intervening pop. In this simple shadow stack, however, pushing (a memory write) is idempotent.

Egalito saves registers with `push`/`pop` instructions, and can analyze a function to see if register saves are really necessary (i.e., the value is used by some successor instruction). Egalito will also identify whether a function uses the red zone instead of a stack frame: leaf functions on x86_64 may access 128 bytes beyond the top of the stack in lieu of moving `%rsp`, which is slightly more efficient, but means that an inserted `push` will overwrite actual program data. Hence, Egalito may automatically add up to `128` to the stack pointer before saving registers. In our example, we save `%rflags` which involves adding 8 bytes to the stack pointer, and this information is passed to the code-generation lambda so it can adjust its offsets. Egalito, and user tools, can also inject additional global data, thread local storage, or `%gs` variables.

3.4 Binary Analysis

We use several analyses to recover static control-flow graphs and obtain complete and precise disassembly. We focus on modern Linux binaries, which are position-independent [74, 75, 76, 77], optimized, and stripped. See Section 3.7 for limitations. Our focus on PIC binaries is unlike most related works: most focus on position-dependent binaries, and conversely, some rely on additional compiler flags (e.g., `-Wl,-q` and `-ffunction-sections`) for extra metadata [7, 87].

3.4.1 Disassembling Code, Not Data

Binary analysis in general is undecidable [88]—e.g., classifying bytes as code or data after a loop is equivalent to the halting problem. The standard technique for binary rewriting is recursive

disassembly, which gives *sound* but *incomplete* results. Many tools conservatively overapproximate—in this example, treating the bytes both as potential code (if they disassemble without errors) and also as data, leaving the original `.text` section in place. Dynamic binary translation can delay the decision to runtime, and only treat bytes as code when when a jump to the memory is actually observed.

Modern binaries separate code and data sections, and refrain from using embedded constants. x86_64 code has no need since RIP-relative literal loads can reach +/- 2GB, and ARM64 constant pools (`-mpc-relative-literal-loads`) are disabled except under the tiny memory model [89] (address space $\leq$ 1MB). This is consistent with prior work which finds that while embedded constants are present in real-world x86 Windows binaries [88], all GCC- and Clang-generated binaries can be linearly disassembled on x86 [90]. Thus, linear disassembly of code sections will suffice.

3.4.2 Reconstructing Functions

Egalito operates on stripped binaries, where function boundaries are not specified. We approximate function boundaries with a coarse-grained heuristic based on direct call targets, which may conservatively lump functions together. We then analyze jump tables. The final control flow graph of each function is split into disjoint connected components to accurately reconstruct function boundaries. There is one further special case: non-returning functions. After a call to a function that never returns, the compiler will place the next basic block immediately afterwards, knowing execution will never fall-through. GCC tracks functions like `exit` with an attribute `noreturn`, recursively propagating it to functions that always call `exit` etc; Egalito uses a similar analysis.

Frame unwind information, created for C++ exceptions and debugging, is sometimes present. When it is, we use it to precisely identify function boundaries. On ARM64, this info is only present for functions that throw exceptions; on x86_64, stripped binaries contain frame unwind information for every function (`-funwind-tables` is enabled by default).

3.4.3 Identifying Code Pointers

A binary is full of values that might be constants or might be pointers. It is not sufficient to simply consider all values to be pointers if they lie within the valid range for code virtual addresses—even SPEC CPU is disassembled incorrectly under such a heuristic (§6.1 of [48]). In PIC, all absolute pointers will have relocations [91]. Relative pointers in the data section are typically used only used for jump tables, covered next. Relative pointers in x86_64 code sections are a `%rip`-relative constant in a single instruction, and are easy to identify.

The situation is more complicated on RISC architectures such as ARM64, however. A single fixed-length instruction cannot encode a PIC reference. The compiler uses: 1) an `adrp` instruction to load the upper bits; and 2) an `add`, `ldr`, or `str` instruction to load the page offset. These instructions form a PC-relative load, so there is no relocation metadata. Furthermore, the compiler's optimizer may place these logically related instructions far away from one another, in different basic blocks, hoisted outside loops, etc. We leveraged data flow analysis to find such split-pointer loads.

3.4.4 Reconstructing Jump Tables

There is no metadata which can assist in locating jump tables (no relocations in PIC). Standard ARM64 jump tables can use 4-, 2- or even 1-byte offsets—but ARM64 does not even define standard relocation types for 1-byte values. Nevertheless, we aim to recover all jump table addresses, invocation locations (indirect jumps), and table bounds (number of entries). Although bounds checks may be optimized away by the compiler, determining the bound is essential: if it is underestimated, some edges in the function's control flow graph will be unidentified, while if it is overestimated, arbitrary data will be corrupted during recompilation.

Detection Procedure Our solution leverages sophisticated data analysis techniques including use-def chains. We will use Figure 3.4 as a running example. First, we consider every indirect jump in the program ①. We look for expressions that flow into the jump computation ②, pattern-matching

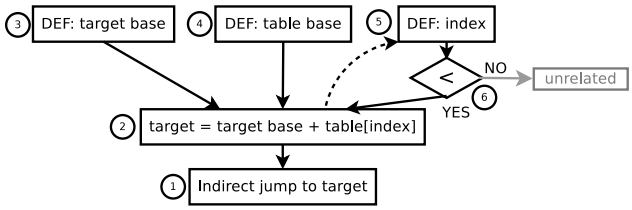


Figure 3.4: Jump table reconstruction steps.

against structures that the compiler uses to implement jump tables. These patterns are independent of exact instructions, registers, operand order, flow through repeated `movs`, basic block structure, etc. We extract the address of the sequence of table entries (the table base ③), and the value added to each table entry to compute its destination (the target base ④, same as table base on x86_64). Finally, we look for the table bound. We extract the indexing register or memory expression, find the definition of its value ⑤, and iterate over all uses of this value. One or more uses will flow back to the jump instruction, and there may be bounds checks along those paths ⑥. We select the tightest comparison bound.

Bounds Not all bounds are enforced with a straightforward comparison instruction, however. We implemented many special cases, including: 1) subtraction/bitwise test, then check flags against zero; and 2) bitwise `and` with a constant bound (e.g., for hash computations). We also encountered sophisticated tables which we call multistage jump tables: one table is indexed into to determine an index within a second table, which finally contains a target address. We handle this by parsing the first table as usual, with the correct striding. We examine every value in the table and find the maximum index—thus deducing the bound for the second-stage table.

Adjacency Heuristic In some cases, we simply cannot determine the bound, and we use the following heuristic. We observed that current compilers (GCC and Clang) place all jump table contents sequentially. We expand each table as much as possible without including entries whose computed target lies outside the source function, and stop at the end of the section or when another link occurs. This heuristic can fail in real-world cases such as tables that partially overlap each others' data (e.g., `glibc` hand-coded assembly, which luckily contains explicit bounds checks), but it is a useful fall-back. Our evaluation shows that across thousands of jump tables, the adjacency

heuristic is only used 6.52% of the time. Considering that not all jump tables even include a bounds check, we believe Egalito does very well in its analyses.

3.5 Egalito Tools

Nine tools follow, ordered from simple to sophisticated.

Counter-Based Profiling This tool instruments EIR (functions or basic blocks) with counter increments. Each `Chunk` is given a separate global variable, and its Egalito name is written into a data section. Counter values are appended to a binary file at program exit. We provide a `gprof` look-alike, which prints accumulated statistics from past runs.

Profile-Guided Optimization We implemented a profile-guided optimization tool which modifies a program's layout for the best performance given knowledge about the input. Given function-call counts from a representative execution, recorded by our profiling tool, this tool arranges functions from most common to least for better caching performance.

Debloating We implemented a function-level debloating [92, 5] tool for x86_64 and ARM64. Starting from the program entry point and from every function whose address is taken, this tool iterates over our control-flow graph and finds all reachable code. Any unreachable functions are removed, as they represent bloat and cannot be called (barring reflection).

Instruction Reordering In-place randomization [93] consists of four techniques, of which we implemented two: instruction reordering within basic blocks, and reordering register saves/restores. We use dataflow information to create a graph of all instruction dependencies within each basic block. We then choose a new order for the entire block, maintaining a set of valid next instructions at each point and selecting one at random. Note that this algorithm, while simple, does not guarantee uniformly random permutation selection. Within function prologues and epilogues, we reorder register saves/restores by eliminating the ordering constraint between `push`/`pop` instructions that would otherwise hold (due to contention on `%rsp`). In the prologue, we choose a new register save order; in each epilogue, we restrict the ordering between pops to follow the reverse of the save

order. Non-`push`/`pop` instructions may still be randomly intermingled as dependencies allow.

Data Execution Prevention On x86_64, we wrote a sandbox to enforce `W^X` memory: no memory page may be writable and later executable. We find all `syscall` instructions, leveraging dataflow analyses to deduce the system call number (`%rax`). We instrument `mmap`, `mprotect`, and `munmap` to track whether each mapped memory page has ever been writable. If the program tries to make such a page executable, this constitutes a sandbox violation and the program is terminated. We combine this sandbox with our control flow integrity from CET. The control flow integrity prevents an attacker from jumping over the system call instrumentation, while the data structure that tracks writable pages cannot be reliably corrupted (located at a random address, only referred to by `%gs`). Hence, this sandbox prevents any code-injection attack, even if the attacker can corrupt data and call `mprotect`.

Retpolines The recent Spectre [11] vulnerability exploits a hardware bug, an inconsistency in the way current CPUs (Intel, AMD, and ARM) implement speculative execution. One available software fix for Spectre Variant 2 is to transform all indirect jumps into *retpolines* [94], which force speculative execution into safely contained infinite loops. We created an x86_64 Egalito tool that transforms every indirect jump into a retpoline. We outline retpolines to avoid duplicating their code, generating a new function for each unique indirect target expression (e.g., `%rax`, `0x10(%rax,%rbx,2)`, etc).

Software Implementation of Intel's CET Intel has recently announced an x86_64 hardware extension called CET or Control-flow Enforcement Technology [12, 95]. This extension specifies a set of hardware instructions that will be made available in future CPUs, and map to no-ops on current processors. CET consists of a) control-flow integrity (CFI) for indirect branches, and b) a hardware shadow stack to protect return statements. We implemented this defense in software in August 2018 when the `libstdc++.so` in Ubuntu 18.04 began to include `endbr64` instructions (but other libraries did not). Our tool may be applied comprehensively across a system directly to the binary code, and when hardware support becomes available, the instrumentation can be removed.

Under CET's CFI scheme, the target of every indirect control flow (call or jump) is marked with a specific instruction `endbr64`. Our CFI pass iterates over each function whose address is taken and adds `endbr64` to its prologue (if not already present). Finally, we instrument each indirect call with a runtime check to verify that its target is an `endbr64`. If not, our error handler raises a fatal `SIGILL` signal.

We developed two shadow stack implementations. The first is a simple constant-offset shadow stack. The second, a more faithful reproduction of CET, stores the shadow region at `%gs` with a shadow stack pointer at `%gs:0x0`. Pushing and popping involves incrementing/decrementing this pointer, which means that a push without a corresponding pop will cause a detectable fault. CET also specifies hardware instructions that modify the top-of-stack pointer directly, for stack unwinding and exception handling. We leave these for future work.

Egalito-AFL Egalito-AFL is a binary-level backend for the AFL fuzzing framework [3]. It adds the instrumentation necessary for AFL to determine coverage (i.e., the set of branches that are taken). We integrate with the AFL forkserver, which forks new targets to avoid `exec` calls. The forkserver creates a System V shared memory segment and passes it to the initial target via the `__AFL_SHM_ID` environment variable. We inject code at program start to detect this and map a 0x10000-size memory region with `shmat`.

We instrument every basic block with a coverage-recording snippet based on `bin_coverage.c` from drAFL [96]. Each block is assigned a random (constant) ID, and an accumulator tracks history of the past few branches, hashing into the shared memory region [97]. Each time, the accumulator is right-shifted by one, and XOR'd with the random block ID; the index in the shared memory region corresponding to the lower 16 bits of the accumulator is incremented. We perform this update using only a single register (plus `%rflags`).

Just-In-Time Shuffling JIT-Shuffling is a novel continuous code randomization defense, based on prior work (Shuffler [7], TASR [13]). Like Shuffler, JIT-Shuffling is x86_64-only and transforms every code pointer into an index within a runtime dispatch table. Return addresses become a pair of numbers, a function index and a byte offset into that function. The table is stored using the

ID	Arch	Linux Distribution	Machine	RAM	GCC
M1	x86_64	Debian buster	4c/8t i7-4770	32GB	7.2.0
M2	x86_64	Debian stretch 9.6	8c/16t X5550x2	24GB	6.3.0
M3	x86_64	Debian testing	4c/8t i7-2600	16GB	8.2.0
M4	x86_64	Devuan ascii	8c/8t W-2145	64GB	6.3.0
M5	x86_64	openSUSE*	4c/8t i7-4770	32GB	7.3.0
M6	x86_64	Ubuntu 18.04.1†	6c/12t X5550x2	10GB	7.3.0
M7	x86_64	Fedora 31	8c/16t X5550x2	24GB	9.2.1
M8	ARM64	openSUSE Leap†	8c/8t ThunderX	8GB	7.1.1
M9	ARM64	openSUSE*	Raspberry PI 3	1GB	7.2.1

Figure 3.5: Machines used for Egalito testing and evaluation. *=openSUSE Tumbleweed rolling release. †=Virtual Machine.

unused `%gs` segment register [7, 98], to prevent an attacker from performing memory disclosure on the table. Direct function calls, tail recursive calls, indirect function calls, returns, etc, are replaced with `%gs`-relative jumps, while pointer initializations are changed into indices—addresses are never used as code pointers.

In a departure from Shuffler, JIT-Shuffling operates synchronously. Function `%gs`-table entries initially point to the address of an Egalito *resolver* function that instantiates the function at a new address, similar to the way a lazy PLT resolver computes addresses on the fly. Periodically, a "reset" callback erases all functions, and points their table entries back to the Egalito resolver. If control flow returns to a function which has been erased, it will be reinstantiated. As in Shuffler, JIT-Shuffling makes use of two code sandboxes during execution, and migrates between them while leaving no fixed code in the address space—even Egalito code. JIT-Shuffling supports `fork` and multiple threads: each execution context uses its own sandboxes (threads share EIR).

3.6 Evaluation

Our evaluation uses the machines in Figure 3.5. Unless otherwise noted, we used M1 for x86_64, and M8 for ARM64.

3.6.1 Correctness of Binary Analysis

Function Boundaries On ARM64, we transformed all 105 GNU Coreutils binaries (stripped). Egalito identified all function boundaries; however, in 11 cases we split the code into additional functions, when the (non-returning) `error()` function is called with a constant argument. (Separating such functions is an accurate representation of control flow.)

Code and Data Pointers We validated that Egalito can detect all pointers using relocation ground truth. We compiled GNU Coreutils and `glibc` on both x86_64 and ARM64 with `-ffunction-sections` and `-Wl,-emit-relocs` (`-Wl,-q`), to include as many relocations as possible in the output. Egalito creates links for precisely the set of code and data pointers in the ground truth (plus additional links for jump table entries, which have no relocations).

Inline Assembly Egalito handles many assembly functions correctly, e.g. those in `glibc`. However, some hand-coded assembly (`libffi`, crypto code) embeds jump table values into `.text` symbols. By design, Egalito trusts function boundary metadata, but to handle non-standard cases we provide an override settings file for users to define code/data boundaries. Our evaluation does not rely on any such parse overrides.

Jump Table Study We verified Egalito's detected jump tables against ground truth obtained from the compiler, using GCC's `-fdump-final-insns`, which outputs the RTL intermediate language while producing the corresponding object files. This data includes the number of jump tables per function as well as the number of entries in each table. On both x86_64 and ARM64, we programmatically verified jump tables in `glibc` and Coreutils. We manually verified `glibc` jump tables written purely in assembly.

To test jump table detection at scale, we ran a large-scale experiment on x86_64 Debian packages. We built each package from source with the additional `dpkg-buildpackage` option `DEB_CFLAGS_APPEND=-fdump-final-insns`. We preserved every `.o` object file created during the build process by overriding `rm` and various variants of `gcc`. Finally, we extracted the built package (and its debug package) and found all executables and libraries therein. We

used `FILE` symbols to map functions back to object files and hence to jump table information. For functions not within a `FILE`, we searched for functions with the same name in all object files. We also considered function aliases and `*.lto_priv.NN` symbols generated by link-time optimizations. We analyzed each case with Egalito and compared our list of recovered jump tables with the ground truth. In case of multiple ground-truth options (multiple functions of the same name), we considered Egalito to be correct if it matched one option.

Overall, we ran the experiment on 172 packages from Debian's `popcon` list [99] on M3. We excluded large packages like `systemd` and packages that would likely not contain C/C++ executables. These 172 packages produced 867 executables and shared libraries, and in 866 cases, Egalito successfully reproduced the ground truth jump table list. (Parsing these 867 ELFs took Egalito 55 minutes on a single core.) The single failure is `sftp` which contains heavily nested jump tables that our control-flow graph does not yet capture. Hence, Egalito correctly recovered jump tables 99.9% of the time. Nearly half of all executables included complex stack-variable bounds checks that required tracking data flow through memory, not just registers. Of the 3970 recovered jump tables, Egalito analytically determined table bounds in 93.48% of cases (relying on the table adjacency heuristic in the remaining 6.52%). Since not all jump tables even include a bounds check, Egalito's analyses do very well.

Distribution-Scale Analysis We have since analyzed 34301 executables in a full-scale analysis of Debian sid packages. 92.5% of them parsed correctly. See Appendix C for details.

3.6.2 Correctness of Binary Transformation

To gain confidence in Egalito's generated code, we executed and successfully passed the test suites for Coreutils, `ffmpeg`, and `sqlite` on M2 (`sqlite` includes 2,583,067 tests). We manually tested 13 arbitrary programs; many succeeded, including `dpkg`, `make`, `tmux`, and `vim`, while 4 failed. Two failed due to some position-dependent code linked in with PIC, one failed due to Egalito features not yet implemented (`aptitude` throws an exception), and Google's V8 contains embedded data after every function (within symbol boundaries).

Debian Package Tests We ran a second large-scale analysis on Debian binaries, running test suites associated with Debian packages on Egalito-transformed binaries. We found that some 9904 Debian packages are registered in the Debian continuous integration system, and we targeted the 308 packages that have the tag `implemented-in` set to `c` or `c++`. We transformed all executables in the distribution's `.deb` files with Egalito, then ran the package tests in a `chroot` with `autopkgtest`. 149 packages contain tests and build correctly in the `chroot`. 90 out of 149 (60%) packages have *all* tests pass. 38 failures can be detected by Egalito, they are because we do not yet support generating symbol versions (this merely requires additional engineering effort). At least one throws an exception (not supported), and one has a too-strict test that looks for sequences of zeros in the executable. If we take this into consideration, 90/109 (82.6%) of the packages pass all tests (4 additional packages pass at least one).

Compiler Versions To test binaries from different compilers, we built many versions of GCC from source. With each GCC, we verified that `sqlite` (compiled position-independent) still passed all tests. Specifically, we tested the following versions of GCC (earlier compilers no longer built cleanly): 4.9.4, 5.3.0, 5.4.0, 5.5.0, 6.1.0, 6.2.0, 6.3.0, 6.4.0, 6.5.0, 7.1.0, 7.2.0, 7.3.0, 7.4.0, 8.1.0, 8.2.0. We tested Clang 5.0.1 by transforming and successfully running SPEC CPU ref on M5 (loader mode).

Linux Distributions We verified that Egalito-transformed SPEC CPU ref runs correctly on four different distributions: Debian (M1), Ubuntu (M6), openSUSE (M5) and Fedora (M7). (All in 1-1 ELF mode except M5, where we used loader mode).

Go Binaries Egalito cannot transform Go binaries correctly, because they contain `vtable`-like structures in `go.itab.*` without relocations. We would need to represent these data structures in EIR in order to transform Go programs.

Chromium Libraries V8 and Chromium contain an unusual form of DWARF that we do not support. However, we transformed Chromium's shared libraries in 1-1 mode. Of 177 libraries, 59 use symbol versions in a way we do not support (a low-level ELF generation, solvable with engineering effort). Of the 118 remaining, 12 cause Chromium to fail, and `libffi` fails disassembly

due to embedded jump tables in assembly code. 105 of 118 (89%) transform correctly and using `LD_LIBRARY_PATH` to load all transformed libraries, Chromium can browse to news sites and display videos.

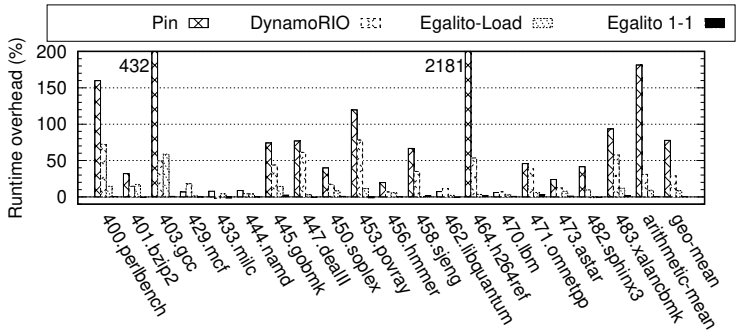
`/usr/bin/` smoke test We tried to transform all the executables in `/usr/bin/` on M4 (1-1 mode), then ran each program with `-help`, comparing against baseline. There were 1379 executables; 90 (6.5%) failed during transformation because they were position-dependent, and 11 failed transformation for other reasons (Egalito was aware of a problem). Of the remaining 1278 executables, 1256 (96.6%) produced identical output and exit code. Of the remaining 33 failures, 25 were due to lack of `RUNPATH` support, and 8 from fatal signals.

Kernels We transformed a Fuchsia [100] microkernel (for ARM64) with Egalito. In lieu of PIC, we leveraged full relocations with the `-q` linker flag. We added function call logging into a ring buffer, generated a flat binary image, and successfully booted it on a bare-metal Raspberry PI 3 (M9).

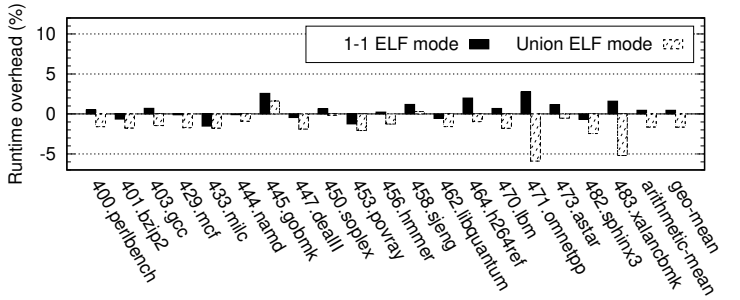
3.6.3 SPEC CPU Performance Evaluation

Egalito supports three major execution modes: 1-1 ELF, union ELF, and loader mode. We present Egalito's performance on SPEC CPU in each mode, utilizing binary optimizations. In 1-1 ELF mode, we transformed only the main executable and not shared libraries; in union ELF and loader mode, we transformed all code. Since Egalito does not yet support C++ exceptions, we modified SPEC CPU by replacing exceptions with conventional control flow in `omnetpp` (20-line change) and `povray` (15 lines). We also fixed a compile error in `soplex` (1 line) in recent GCC versions.

Comparison with DynamoRIO and Pin As shown in Figure 3.6a, Egalito has much better performance than existing DBT-based tools DynamoRIO and Pin (measured on M1). DynamoRIO geo mean overhead is 28.8%, Pin is 77.7%, while Egalito in 1-1 mode is only 0.46% (executables pre-transformed). For a fairer comparison, Egalito in loader mode also parses all executables at load-time, incurring 8.7% slowdown (1.1% slowdown with caching; see loader mode below).



(a) Comparison of DynamoRIO, Pin, Egalito (loader and 1-1 mode).



(b) Performance of Egalito-generated ELF outputs.

Figure 3.6: Egalito runtime overhead on SPEC CPU 2006, in different modes, compared with DynamoRIO and Pin.

1-1 ELF Mode This is Egalito's default mode, reading in a single ELF and outputting a single ELF. Since EIR represents and recreates each input instruction, we can expect near zero performance overhead (given a no-op transformation tool). We measured this baseline overhead in 1-1 ELF generation mode by running SPEC CPU 2006 on machine M1, and results are shown in Figure 3.6b. Egalito incurs only 0.46% overhead.

Union ELF Mode In union ELF mode, Egalito combines the input executable and all its shared libraries into one output ELF, essentially transforming a dynamically linked program into a statically linked one. We measured the baseline overhead in union ELF mode (also on M1), and Egalito achieves a 1.7% geo mean *speedup*. We use original function order and 2-byte alignment as in 1-1 ELF mode; the speedup is because union ELF mode collapses PLT calls into direct calls.

The memory and space overhead of Egalito-generated ELF files is minimal. On SPEC CPU, union ELF outputs were on average only 0.44% larger (low variance) than the sum of input ELFs. At program entry, the mapped memory of union ELFs is 79%-95% (average 88.4%) of baseline. At

program exit, file-backed resident memory use is 152KB-1.25MB less (598KB average). So union ELFs are memory efficient, but will not share library code pages with different programs. Code sections, augmented with library code, are 540KB-2.56MB larger (1.7MB average)—an upper bound on memory wasted.

Loader Mode Next, we investigated Egalito's baseline overhead in loader mode. The unoptimized raw overhead of the Egalito loader is 8.7% geo mean; using HOBBIT files (a binary serialization of EIR: Hierarchical Object Built for Binary Transformation) gives 1.1% geo mean overhead. Much of this is load-time overhead, spent parsing ELF files (9.4 seconds average, max 15.9) or loading HOBBIT files (2.8 seconds, max 4.1). Also, the SPEC CPU harness invokes the target more frequently in certain cases—e.g., `gcc` is invoked nine times per run while some binaries are only invoked once. By subtracting all load-time cost (amortized in long-running programs), we see a 1.4% geo mean speedup. (Some speedup is again expected due to collapsed PLTs.)

3.6.4 Binary Optimizations

The results presented in this section appeared in CodeMason [8] at FEAST 2019.

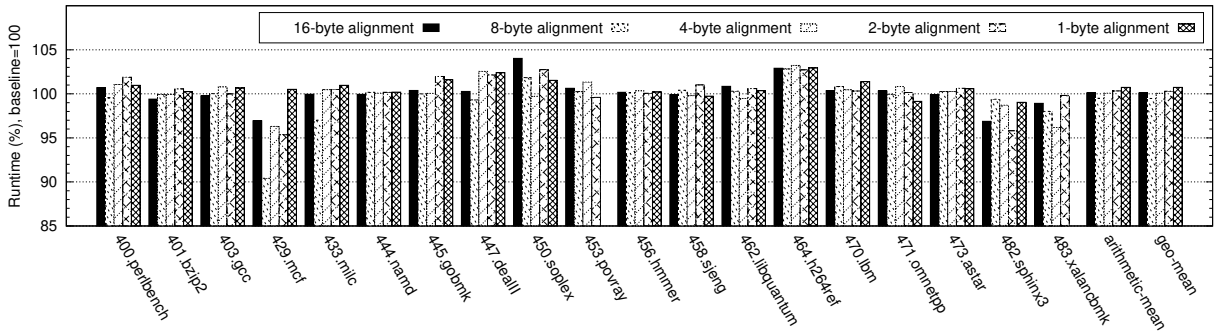
Motivating Example When experimenting with Egalito on binaries installed on our Debian system, we transformed `/usr/bin/perl`. We used the following simple micro-benchmark:

```
perl -e 'for(1..10000000){$x+=$_}print $x'
```

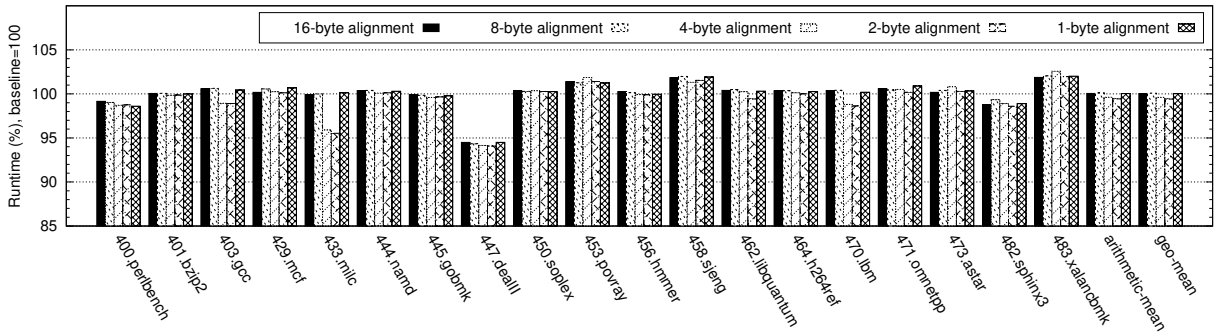
Simply by transforming the binary with Egalito (default 16-byte function alignment), we saw a 2.22% speedup (over a multi-second execution). But we observed even faster performance with smaller alignment, the fastest being a reproducible *13.6% speedup* at 2-byte alignment. We believe that the smaller alignments coincidentally moved functions close enough together that the code used in this loop would all fit into the CPU's code cache.

ID	CPU	Microarch	L1 code	L1 data	L2	L3
M1	Intel Core i7-4770	Haswell (22nm)	4x32KB 8-way	4x32KB 8-way	4x256KB 8-way	1x8MB 16-way
M2	Intel Xeon X5550 (x2)	Nehalem (45nm)	4x32KB 4-way*	4x32KB 8-way*	4x256KB 8-way*	1x8MB 16-way*

Figure 3.7: Details of the CPUs in each machine used for optimization testing. *=per socket.



(a) SPEC CPU overhead for different function alignments, measured on machine M1.



(b) SPEC CPU overhead for different function alignments, measured on machine M2.

Figure 3.8: Egalito SPEC CPU overhead with different function alignments (`nop` padding). For readability, these graphs show overhead relative to baseline=100 instead of relative to baseline=0.

SPEC CPU Performance Evaluation

Here, we evaluate Egalito profile-guided optimization performance on SPEC CPU 2006. Unless otherwise noted, we transformed only the main executables and not shared libraries.

Function Padding The compiler uses 16-byte padding between functions, but we try 16, 8, 4, and 2-byte padding. (1-byte function padding interferes with exception handlers; the least significant bit is set to indicate thrown exceptions.) Compact code may utilize instruction caches better, so Egalito defaults to 2-byte padding. The best padding varies according to test case and machine; for in-depth analysis see [8]. Results are shown in Figure 3.8a and Figure 3.8b. On

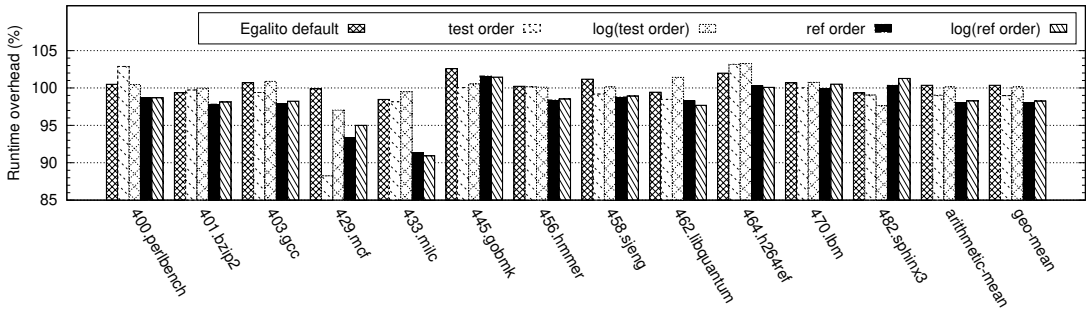


Figure 3.9: Performance obtained by ordering functions according to a captured execution profile (PGO). Run on M1 on SPEC CPU C programs only due to time constraints.

M1 (Figure 3.8a), the winning parameter value is 8-byte alignment. Note that 1-byte alignment fails in two cases (povray and xalancbmk): `libstdc++` exception-handling code uses the least-significant bit of a function address to indicate that an exception has been thrown, and so odd addresses cause phantom exceptions in try-catch blocks. Meanwhile, on M2 (Figure 3.8b), the best alignment is 2-byte alignment. With a different (older) version of `libstdc++`, 1-byte alignment runs in all cases.

Finally, when we select the best alignment for each SPEC case on M1, we obtain 1.2% performance speedup simply by modifying function alignment, while preserving original function order. On M2, 2-byte alignment is a 0.59% speedup, and selecting the best alignment for each case is only 0.63% speedup. So on M2, a good strategy would be to always use 2-byte alignment. We believe this is because the instruction cache on M2 is only 4-way, whereas M1's 8-way instruction cache gives the processor more leeway to keep code around for longer, benefitting more from the degree of freedom given by adjusting function alignment.

PLT Collapsing To support dynamically linked libraries, calls from one library to another go through indirection via the Procedure Linkage Table (PLT). But in union ELF or loader mode, Egalito knows all the code that will be executed. So we collapse PLT calls into direct calls, and place code from all libraries into a single text section, essentially transforming a dynamically linked program into a statically linked one.

Function Permutation We performed several profile-guided optimization experiments based on hot functions. These experiments were only performed on SPEC C programs, due to lack of time. Some C++ programs (especially xalancbmk, with its many tiny virtual functions) will likely benefit even more from function permutation. The experiments, shown in Figure 3.9, were:

1. Instrument SPEC with our profiling code, count function calls under SPEC's "test" input size. Order functions according to number of calls. Then run and benchmark the "ref" input.

2. Count calls on "test"; order according to $floor(log_2(N))$ of the number of calls N. If $N < 3$, place into bucket zero. Then benchmark on "ref".

3. Count calls on "ref"; order functions according to exact number of calls. Then run the "ref" input and benchmark.

4. Count calls on "ref"; order functions according to the log of the number of calls. Then run the "ref" input and benchmark.

(The overhead of our function-call instrumentation was only 0.75% on "ref".) The final speedup in the first case is 1.03%. This shows that profiling on one input (the "test" inputs are really very small) yields a useful speed improvement for another input. The second case shows 0.13% slowdown, so either the buckets were too coarse-grained or the "test" input size did not generate enough calls. When training on "ref" in the third case, we observe 1.98% speedup; using buckets slows performance slightly, to 1.76% speedup. Hence, we observe 1-2% speedup overall, with increased performance when the exact input is known in advance. Precise function ordering works better than bucketing by the log of the counts.

Note that the last two cases, trained on "ref" counts, were repeated three times each for reproducibility. Variance is fairly low, averaging 1.93%, 2.39%, 1.98% in the former, and 2.42%, 1.76%, 1.55% in the latter (all numbers are speedups). The graph shows the results from the mid-performing run.

Egalito Tool	Mode	Overhead	Worst case
No-op 1-1 mode	1-1	0.46%	2.8% omnetpp
No-op union mode	union	-1.7%†	1.6% gobmk
No-op loader mode	loader	-1.4%†	2.6% povray
Profiling	1-1	0.16%, 1 fail	3.3% povray
Profile-guided opt	1-1	-1.0%†, C only	3.2% h264ref
Retpolines	1-1	6.9%	63.0% povray
CET CFI only	union	-1.4%†	-0.57%† gobmk
CET CFI + const stack	union	4.4%	22.2% povray
CET CFI + `%gs` stack	union	9.8%, 1 fail	40.7% povray
Instruction reordering	loader	-2.7%†	3.8% povray

Figure 3.10: Geometric mean overhead of Egalito tools (loader mode has load time excluded). †=performance speedup.

3.6.5 Egalito Tool Performance

Egalito is 51759 lines of code (determined by sloccount [101]); each tool is between 157-306, except JIT-Shuffling with 1510.

Tools on SPEC CPU Figure 3.10 shows the performance of several Egalito tools. Retpolines on M1 has geo mean overhead 6.9%, in line with other published numbers (e.g., Bullet [102]). However, it is prohibitively expensive for `povray` and `xalan`, which use large numbers of virtual function calls and hence incur the indirect-jump overhead repeatedly. The instruction reordering tool observed a speedup over the baseline, likely noise from loader mode. In our CET tool, the simpler constant-offset shadow stack is more efficient than `%gs`. The `%gs` shadow stack fails in one case (`gcc`) due to a bug with our conditional tail recursion detection. Profile-guided optimization, trained with call counts on "test" and evaluated on "ref", shows a 1.0% speedup (best: 11.8% speedup for dealII) [8].

Egalito-AFL We measured the fuzzing throughput of Egalito-AFL, in comparison with a DynamoRIO-based fuzzer called drAFL [96]. Both tools operate directly on binaries, unlike the original AFL which requires source-level instrumentation with an appropriate compiler. We fuzzed `readelf` and `libpng` on M2 for 5 minutes, and Egalito-AFL obtained a 15.6x and 64.2x speedup respectively over drAFL; when run for 10 minutes, these speedups became 18.0x (60060 vs 3353 executions) and 61.4x (526149 vs 8566). Egalito-AFL is 18-61x faster because 1) Egalito-AFL outputs one

ELF binary, while drAFL runs DynamoRIO to rebuild the code cache for each execution; and 2) Egalito-AFL integrates with AFL's forkserver (drAFL does not), allowing minimal `exec` syscalls.

Just-In-Time Shuffling To evaluate JIT-Shuffling, we defended Nginx 1.11.3 on x86_64 (M5). We tested four workers serving ten concurrent clients from the `wrk` tool over one minute, for 612B, 100KB, and 1MB requests. Results in multithreaded and multiprocess mode are nearly identical.

We first erase all functions after each Nginx HTTP request, as in TASR [13], but this is prohibitively expensive (5-50x). TASR advocates this design but does not measure its performance (Shuffler [7] would have 1.5% throughput with 100KB requests). We can trade performance for security by resetting less frequently; instead of after every request, we can reset after every 10, or every 100, etc. Nginx achieves 50-90% of the original throughput when resetting after every 100 requests. This policy is still orders of magnitude ahead of leakage attacks like Blind ROP, which requires a total of 11,070 Nginx requests [103]. It also results in re-randomizing every 1.8-4.9 ms, much faster than Shuffler's 50 ms interval.

3.6.6 Comparison with Other Frameworks

Ramblr [49] We installed Ramblr on M2, and tried to transform `/bin/ls` with the simplest `stackretencryption` reassembler backend. It failed during code generation, as did a) hello world, b) hello with `-no-pie`, c) hello with `-static`, d) programs without `glibc` (complained about empty input).

Multiverse [38] On a 32-bit VM, `pwntools` dies (dependency). We tested further on a 64-bit system (M3). Multiverse does not work on position-independent executables (generates invalid ELF headers), statically linked executables, or `glibc` (jump mapping table contains an invalid offset). On hello world compiled with `-no-pie`, `_start` is transformed, but the RIP-relative `__libc_start_main` pointer is unmodified and the code runs the original `main`. By mapping the original `.text` with executable permissions, Multiverse allows the original code to execute. There are also several `RWX` memory regions in which Multiverse writes and then runs code, opening

the door to code injection attacks.

PSI [35] PSI is distributed on a 32-bit Ubuntu 12.04 VirtualBox VM. We measured the runtime of Egalito (M2) and PSI (VM) against baselines on their respective machines. On `zip`, Egalito had -3.07% overhead vs PSI's 8.26%; on `python`, 3.33% vs 44.6%; on `perl`, -13.6% vs 108%; on `vim`, success vs crash. PSI's higher overhead is due to its address virtualization.

3.7 Limitations and Discussion

Egalito works across many binaries and architectures, but it:

1. requires inputs to be position-independent code (PIC) for sufficient metadata (discussed in Section 3.2);

2. uses data-flow analysis techniques that infer missing metadata (see Section 3.4.4), and so is not complete and cannot guarantee full disassembly;

3. cannot handle obfuscated code, nor inline assembly which embeds data—like jump table values—into `.text` symbols (discussed in Section 3.6.1);

4. does not yet support some implementation-level features (described in Section 3.6.2), such as C++ exceptions and atypical metadata (e.g., Go binaries or V8).

As discussed earlier, non-PIC code lacks sufficient metadata to perform a principled disassembly. However, our disassembly of PIC code is quite principled. We rely on metadata to make disassembly decisions, and there are only two cases where metadata is unavailable: 1) identifying split-pointer loads (only on RISC architectures), and 2) identifying jump tables. We describe these as patterns in data-flow graphs, which are by nature very general (e.g. independent of registers used, intermediate movs, block structure, instruction variations, etc). However, the set of all patterns is necessarily empirically determined; on a large enough set of binaries (such as the 34,000-binary experiment in Appendix C), there may be cases that we have not encountered before.

Of course, there are small issues with disassembly that we consider a matter of implementation details. For example, our low-level disassembly library Capstone [104] has some x86_64 instructions that it does not support. On other instructions, the library reports incorrect register-flow properties (which we work around). We do not implement every x86_64 instruction in our data-flow analysis, and uncommon instructions occasionally crop up. Less than half a dozen assembly functions in the GCC C runtime, linked into almost all executables, must be identified through examining static libraries. And of course, there are binary features that we do not support, such as C++ exceptions. However, addressing all of these cases is only a matter of engineering effort.

We have a few issues with regenerating output executables (such as regenerating symbol versions and symbol hash tables). All such issues are a matter of engineering effort. As a recompiler, Egalito fully disassembles an input binary, and as long as that process was correct, it can in principle support arbitrary modifications and produce correct output code.

3.8 Conclusion

We have presented the Egalito recompiler, a layout-agnostic binary rewriting framework. We implemented nine tools with Egalito including a novel defense JIT-Shuffling, an AFL backend, and a software version of Intel's CET. Egalito is very efficient, observing a substantial performance speedup of 1.7% on SPEC CPU thanks to binary optimizations. It successfully runs on programs from hundreds of Debian packages. We open-sourced Egalito [81, 82] to aid researchers in creating robust and efficient binary transformations.

3.9 Future Work

Our main goal now is to increase the adoption of Egalito in other universities and research labs, including by creating a 64-bit RISC-V port. We are also porting Egalito to work on Windows PE binaries, which may prove especially beneficial for industry. Egalito is a powerful tool and we'd like to explore some of its possible applications. Finally, we are exploring the possibility of using machine learning to aid in disassembly, e.g. for function boundary identification.

Chapter 4: Shared Library Debloating (Nibbler)

Rather than reinvent the wheel, modern developers typically leverage collections of existing code during application development. Code is typically imported by linking with a whole shared library. When an application loads a shared library, the entirety of that library's code is mapped into the address space, even if only a single function is actually needed. The unused portion of code is referred to as *bloat*, and it can negatively impact software defenses by unnecessarily inflating their overhead or increasing their attack surface. Debloating is the process of removing unneeded functionality. We investigate whether debloating is possible and practical at the binary level.

To this end, we present *Nibbler*: a system that identifies and erases unused functions within shared libraries. Nibbler works in tandem with defenses like continuous code re-randomization and control-flow integrity, enhancing them without incurring additional run-time overhead. Nibbler reduces the size of shared libraries and the number of available functions, for real-world binaries and the SPEC CINT2006 suite, by up to 56% and 82%, respectively. We also demonstrate that Nibbler benefits defenses by showing that: (i) it improves the deployability of a continuous re-randomization system for binaries, namely Shuffler (Chapter 5), by increasing its efficiency by 20%, and (ii) it improves certain fast, but coarse and context-insensitive control-flow integrity schemes by reducing the number of gadgets reachable through returns and indirect calls by 75% and 49% on average.

4.1 Design

Figure 4.1 depicts a high-level overview of Nibbler. Given a set of binary applications, Nibbler processes the shared libraries they use, disassembles them, and statically analyzes them to reconstruct the FCG of each library. Then, the functions required by applications and the already-extracted

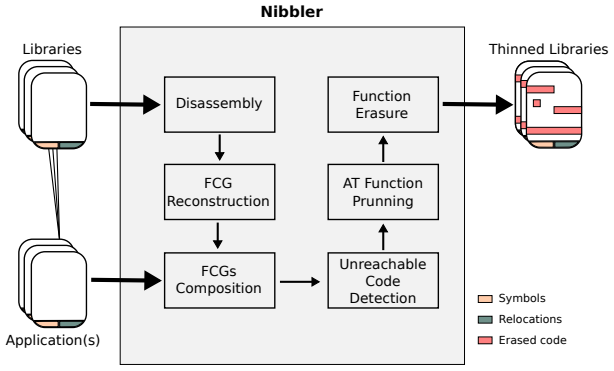


Figure 4.1: Overview of Nibbler.

library FCGs are composed to determine functions that are never called (*i.e.,* unreachable code), by any of the applications of the set. At this point, Nibbler considers all functions that may be called *indirectly* (*i.e.,* through a function pointer) as used. The analysis over-approximates the set of functions that could (potentially) be used to *eliminate* the possibility of error, assuming no manually-loaded libraries (*e.g.,* via `dlopen()` or `dlsym()`). We then perform an iterative analysis that detects address-taken functions that can never be used and also remove them. Finally, Nibbler produces a set of new (thinned) libraries that can be used with the input binaries, where the extra code has been erased by overwriting it with a trapping instruction.

4.2 Debloating Evaluation

Nibbler [5] was originally written as a standalone tool. It was reimplemented with Egalito and further evaluated, and will appear as a DTRAP 2020 journal article [6]. The results presented here are from the latter publication (Section 4.2.3 is present in both).

4.2.1 Distribution-Scale Analysis

[Note: experiments in this section were performed by my collaborators.]

(For further details on how Egalito performs across all packages in Debian, see Appendix C).

To better understand code bloat at large, we ran Nibbler on (almost) all C/C++ applications in Debian `sid`, with the goal of reporting on unused code across a *complete* Linux distribution.

We chose Debian `sid` (*i.e.,* the development distribution of Debian) due to its high availability of debug symbol packages [105].

At the time of writing, Debian `sid` contained over 50K x86-64 packages, distributed over three major repositories: `main`, `contrib`, and `nonfree`. The first step in our analysis involved identifying the list of packages that contained executable C/C++ binaries. To build a list of candidate packages, we excluded packages that did not contain executable code by definition, including: documentation packages (`*-doc`); development headers (`*-dev`); debug symbols (`*-dbg`, `*-dbgsym`); metapackages and virtual packages; architecture-agnostic packages (architecture type '`all`'), which cannot include x86-64 binaries; kernel packages; and packages that contain only shared libraries. Note that shared libraries and other excluded packages can be installed during processing as dependencies of candidate packages.

We performed our analysis using the Egalito-based implementation of Nibbler, on a single host, armed with an 8-core AMD Ryzen 2700X 3.7GHz CPU with 64GB of RAM, running Arch Linux (kernel v5.2). For each candidate package, we install the package itself, its dependencies, and all required debug symbols, and run Nibbler on each binary included. The median time to process each binary is around 20s, with 90% of binaries completing in under 200s—this is a *substantial* improvement over our older, Python-based prototype, which required more than 60s per binary [5]. In total, we processed 34537 binaries across 8641 packages, of which 30631 ($\approx$ 91.7%) could be analyzed successfully (the rest 8.3% corresponds to binaries with missing symbols, non-C/C++ code, *etc.*). Across the set of all the successfully-analyzed binaries, we observed 5295 unique dynamic shared libraries.

Bloat in Popular Libraries For each binary, Nibbler reports the set of libraries used by the application, the total set of functions in each library, and the set of functions that are unused and can be removed. We aggregate these results across all the $\approx$30K considered binaries to *determine* and *characterize* code bloat at large (*i.e.,* across the whole Debian distribution).

Figure 4.2 shows the distribution of the amount of code removed (x-axis) for the top 20 libraries

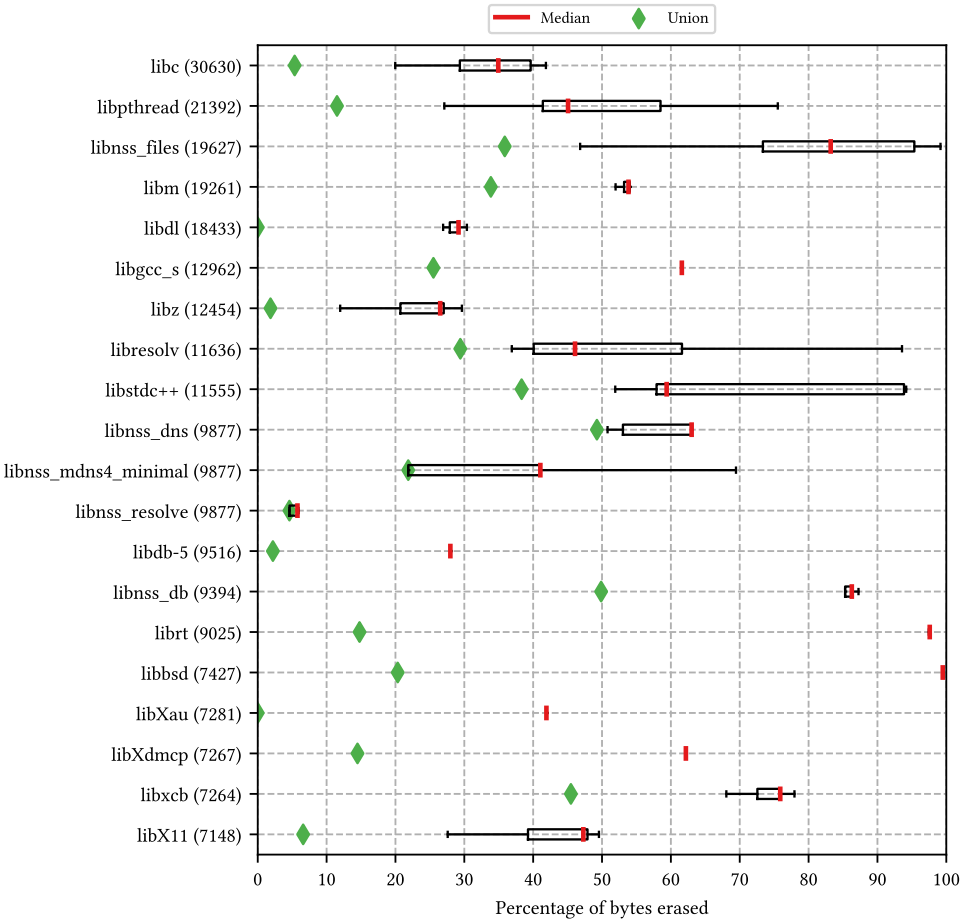


Figure 4.2: Unused code (x-axis) in the 20 most popular libraries (y-axis) in Debian `sid`. Numbers in parenthesis indicate how many binaries (out of 30631) use the respective library.

observed (y-axis), as a percentage of their total size. (Shared libraries are ranked based on the number of packages they link-with.) With one exception, all the binaries we analyzed (*i.e.,* 30631) link-with `libc`; the exception is the binary `mtcp_restart`, an executable with no shared library dependencies.

For `libc`, we observe that the median amount of code removed, per binary, is ≈35%, with most binaries individually not requiring 20%–42% of its code. The next most popular libraries are `libpthread` and `libnss_files`, used by ≈20K and ≈19K binaries, respectively, demonstrating much wider distributions (*i.e.,* 27%–76% and 47%–99%), similarly to `libresolv` and `libstd++` (37%–99%, 52%–94%). In contrast, `libnss_resolve` (≈9K% binaries) has a very tight distribution with a median of only ≈5% of code removed, owing to the relatively small number of functions it

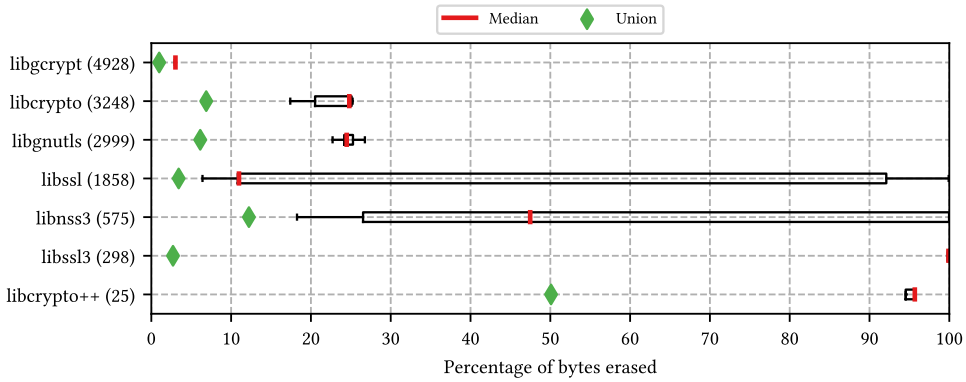


Figure 4.3: Unused code (x-axis) in cryptographic libraries in Debian `sid`. Libraries (y-axis) used by $\leq$ 20 packages are excluded. Numbers in parenthesis indicate how many binaries (out of the 30631 analyzed) use the respective library.

exports and their tight coupling (*i.e.,* all variations of `gethostbyname` and `gethostbyaddr`). `libm`, `libdl`, `librt`, and `libbsd` also exhibit tight distributions, with a median of ≈54%, ≈29%, ≈97%, and ≈99% of code removed, respectively.

Beyond per-binary metrics, we can gain further insights by considering the *union* of all functions required across all binaries in the distribution, which identifies code not used by (almost) *any* binary in the system. Accordingly, the union amount of code to remove is always lower than the median reported in Figure 4.2. For `libc`, we observe that ≈5% of its code is unused across all binaries. As another example, the popular NSS [106] libraries `libnss_dns` and `libnss_db` have nearly 50% unused code across all binaries; likewise, `libxcb` and `libstd++` exhibit ≈46% and ≈38% unused code across all binaries.

Lastly, Figure 4.3, Figure 4.5, and Figure 4.4 illustrate the effect of Nibbler on popular cryptographic, boost, and X11 libraries, across the whole Debian distribution. More importantly, our results indicate that Nibbler can be used to replace the 5295 unique dynamic shared libraries, in Debian `sid`, with their "unionized" equivalents, without affecting the functionality of any binary (out of the 30631 analyzed), while removing ≈420MB (≈1.6M functions) of code in total.

4.2.2 Docker Container Debloating

[Note: experiments in this section were performed by my collaborators.]

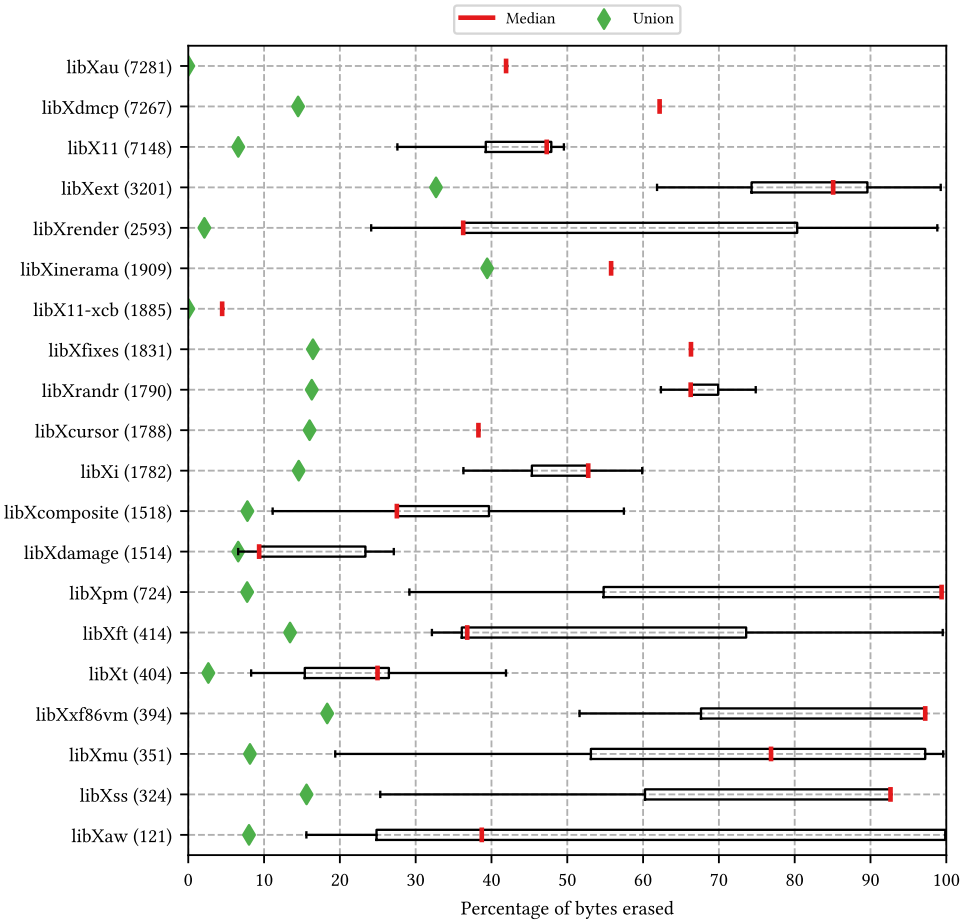


Figure 4.4: Unused code (x-axis) in X11 libraries in Debian `sid`. Libraries (y-axis) used by ≤ 20 packages are excluded. Numbers in parenthesis indicate how many binaries (out of the 30631 analyzed) use the respective library.

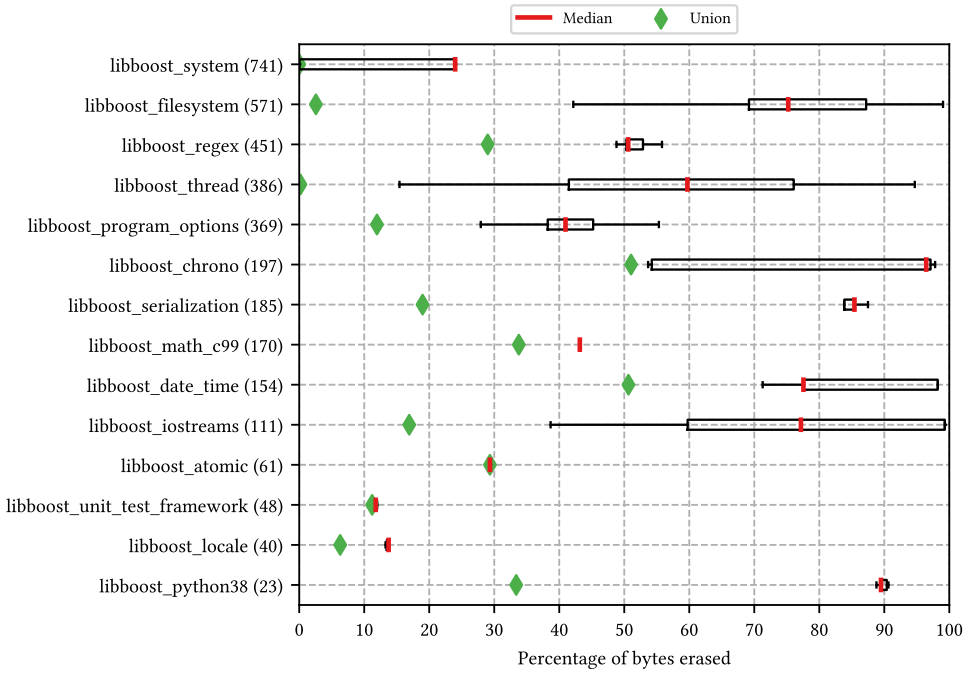


Figure 4.5: Unused code in Boost libraries in Debian `sid`. Libraries used by fewer than 20 packages are excluded. Numbers in parenthesis indicate how many binaries (out of the 30631 analyzed) use the respective library.

To assess the amount of code bloat in commonly-distributed sets of Linux binaries, we used Nibbler to examine packages from popular container images published in the Docker Hub repository [107]. Rather than analyzing binaries in each image directly, we leverage our existing dataset and infrastructure for processing Debian `sid` packages (Sec. 4.2.1) to provide an equivalent analysis. Specifically, we used the container images to identify *sets* of binaries to consider from our Debian `sid` dataset. By examining the union of library functions removed, across all binaries in each set, we can we can quantify the total set of unused code across all libraries in the published container image.

We selected 9 official Docker images for popular C/C++ applications, each with more than ten million downloads, on Docker Hub. To simplify the process of translating packages in the container's base distribution to Debian package names, we selected container images that were built from distributions using the `apt` package manager. For each container image, we mapped each binary to its providing package using `dpkg`, and then looked up the equivalent `sid` package in our dataset. Across all images, we considered a total of 255 unique packages. Mappings for

			Bytes (KB)			Functions		
Image	**Bins**	**Libs**	**Total**	**Erased**	**%**	**Total**	**Erased**	**%**
`haproxy`	293	62	15247.06	3565.83	23.39	46822	17000	36.31
`influxdb`	309	83	16778.82	3397.27	20.25	52240	16822	32.20
`mariadb`	394	84	17466.90	3790.69	21.70	53231	18364	34.50
`memcached`	292	61	13683.17	3393.47	24.80	39621	14606	36.86
`nginx`	294	59	13425.71	3263.02	24.30	38531	13953	36.21
`postgres`	363	83	21047.71	4808.17	22.84	69125	23817	34.45
`redis`	295	68	15886.73	3649.33	22.97	47964	17367	36.21
`spiped`	296	60	14828.60	3590.70	24.21	44949	17048	37.93
`ubuntu`	310	61	13549.62	3292.48	24.30	39077	14261	36.49
Arithmetic mean			15768.26	3638.99	23.20	47951	17026	35.69
Geometric mean			15613.71	3615.18	23.15	47177	16818	35.65

Table 4.1: Erasure for Docker images (using packages from Debian `sid`).

all but 11 packages could be resolved automatically—we manually resolved the remainder, which differed only by version numbers (*e.g.,* `mariadb-server-10.3` vs. `mariadb-server-10.4`). While this mapping between Linux distributions does not provide an exact representation of the unused code included in the container images, it does provide a *tight* approximation of code bloat across a commonly-installed set of packages in popular Docker images.

Table 4.1 shows the union amount of code removed across all libraries in each container image. Every image used roughly 300 binaries and less than 100 shared libraries. Across all images, we observe that an average of 23.2% of code (in terms of by bytes) is unused. All images exhibited roughly the same amount of bloat, with image-wide union values within 4% of each other. The majority of this bloat is due to common system libraries used in each image, rather than application-specific libraries—we leave the analysis of comparing ancillary utilities and binaries required for the application's use-case as a future study. Interestingly, in 7 out of the 9 images, the library with the most code removed (> 94%) was `libunistring` [108], a common library for handling Unicode. Conversely, all images utilized one library which had no code removed, *i.e.,* `libdebconfclient` [109], which is used by `dpkg`. Overall, our results demonstrate opportunities for reducing code bloat, using Nibbler, in commonly-used container images.

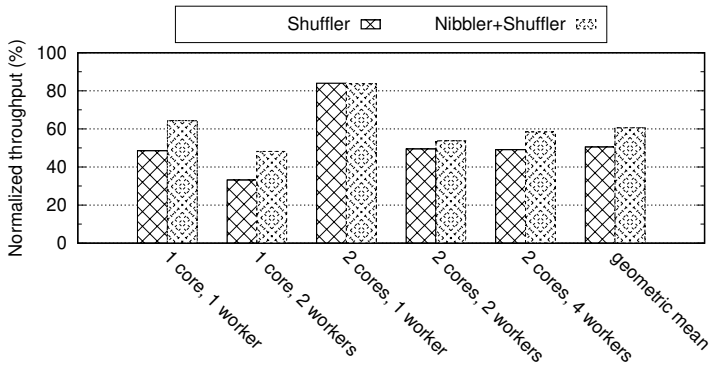


Figure 4.6: Nginx throughput for vanilla Shuffler, and Nibbler+Shuffler, over an undefended baseline.

4.2.3 Combining with Continuous Code Re-Randomization

By identifying and removing unused code, Nibbler can reduce the overhead of certain security techniques, thereby easing their adoption and improving software security. Shuffler (Chapter 5) is a system realizing such a technique: continuous re-randomization of a target program and all its libraries, including its own code. It does so asynchronously, from a background thread, preparing a new copy of the code every 20 ms (in case of Nginx), and then signaling all other threads to migrate to this new copy. Because of this asynchronous design, all functions must be re-randomized, during each shuffle period, as the system cannot determine in advance what will be required. Nibbler's library thinning can combine excellently with Shuffler's defense. We fused Nibbler with Shuffler, on Nginx 1.4.6, trimming functions that would never be used during execution, reducing the amount of work that Shuffler must perform. Overall, we nibbled ≈1.6K functions (out of ≈6.2K) or 26%.

In our experiment, we used 4 Nginx worker processes, pinned to 2 CPU cores; Shuffler threads (one per worker) were also pinned to the same cores. Shuffler's asynchronous overhead will take CPU time away from the target program, reflecting in a throughput drop. We ran 32 client threads (using the benchmark tool Siege [110]) pinned to 4 other cores on the same system, which was sufficient to saturate the server. This experiment is a smaller scale version of the Nginx experiment included in the original Shuffler paper [7].

Results are shown in Figure 4.6. Nibbler+Shuffler performance improves substantially when there are more Shuffler workers, and hence more CPU time is being spent on asynchronously copying code. In the 2-cores, 1-worker case, one core runs the Nginx worker and one executes the Shuffler thread, so Nibbler has little impact. However, if we assume a carefully provisioned system with few resources to spare, Nibbler can improve Shuffler's performance significantly. Overall, the geometric mean of Nginx throughput improved from 50.51% to 60.54% while being shuffled, a relative increase of 19.87%.

This makes sense since we trimmed 26% of the code; due to Shuffler's design, we expect a linear increase in performance as the amount of code decreases. Additionally, we expect these results to scale to larger experiments, since Shuffler spends very little time on coordination (0.3% of overall runtime [7]). If the server is multiprocess, every process gets its own independent Shuffler thread, and Nibbler still reduces the overhead linearly. If the server is multithreaded, Shuffler's overhead decreases as more cores become available, and Nibbler will still reduce the overhead proportionally.

4.3 Conclusion

We presented Nibbler, a system which demonstrates that debloating binary-only applications is possible and practical. Nibbler identifies unused code in shared libraries and erases it. We use a conservative FCG reconstruction algorithm to initially only remove functions without pointers to them, which we refine by introducing an optimization for eliminating functions with unused pointers. We evaluated the debloating capabilities of Nibbler with real-world binaries and the SPEC CINT2006 suite, eliminating 56% and 82% of functions and code, respectively, from used libraries. Nibbler is able to correctly analyze binary software, by only leveraging symbol and relocation information produced by existing compilers. In addition, we applied Nibbler on ≈30K C/C++ binaries and ≈5K unique dynamic shared libraries (*i.e.,* almost in the complete Debian `sid` distribution), as well as on 9 official Docker images (with millions of downloads in Docker Hub), reporting findings regarding code bloat at large.

Nibbler, and debloating generally, improves security of software indirectly, by benefiting defenses. Continuous code re-randomization systems get a performance boost, which we demonstrated by integrating Nibbler with Shuffler to lower Shuffler's overhead by 20%. Lower overheads make such defenses more attractive for deployment on production systems, or can be used to provide stricter security guarantees (*e.g.,* by raising re-randomization frequency) in critical systems. Control-flow integrity defenses also benefit, because we remove code involved in allowable control-flows. Nibbler reduces the number of gadgets reachable through returns and indirect calls by 75% and 49% on average (see [111]).

Chapter 5: Defense via Re-Randomization (Shuffler)

5.1 Introduction

We propose a system, called *Shuffler*, which provides a deployable defense against JIT-ROP and other code reuse attacks. Other such defenses have appeared in the literature, but all have had significant barriers to deployment: some utilize a custom hypervisor [18, 20, 19, 21]; others involve a modified compiler [13, 14, 15, 16, 17], runtime [16, 17], or operating system kernel [13, 18, 19]. Note that there is a security risk in any solution that requires additional privileges, as an attacker can potentially gain access to that elevated privilege level. Also, modified components present a large barrier to the adoption of the system and have less chance of incorporating upstream patches and updates, so users may continue to run vulnerable software versions. In comparison, Shuffler runs in userspace alongside the target program, and requires no system modifications beyond a minimal patch to the loader. Shuffler can be deployed amongst existing cloud infrastructure, adopted by software distributors, or used at small scale by individual security-conscious users.

Shuffler operates by performing continuous code re-randomization at runtime, within the same address space as the programs it defends. Most defenses operating at the same level of privilege as their target do not consider defending their own attack surface. In contrast, we bootstrap into a self-hosted and self-modifying *egalitarian* environment—Shuffler actually shuffles itself. We also defend all of a program's shared libraries, and handle multithreading and process `fork`s, shuffling each child independently. Our current prototype does not handle certain hand-coded assembly, but in principle, *all* executable code in a process's address space can be shuffled.

With Shuffler, we aim to rapidly obsolete leaked information by rearranging memory as fast as possible. Shuffler operates within a real-time deadline, which we call the *shuffle period*. This deadline constrains the total execution time available to any attack, since no information about the

memory layout transfers from one shuffle period to the next. We achieve a shuffle period on the order of tens of milliseconds, so fast that it is nearly impossible to form a complete exploit. Shuffler creates new function permutations asynchronously in a separate thread, and then atomically migrates program execution from one copy of code to the next. This migration requires a vanishingly small global pause time, as program threads continue to execute unhindered 99.7% of the time (according to SPEC CPU experiments). Thus, if the host machine has a spare CPU core, shuffling at faster rates does not significantly impact the target's performance. Shuffler's default behaviour is to use a fixed shuffling rate, but it can work with different policies. For instance, if the system is under reduced load, a new vulnerability is announced, or an intrusion detection system raises an alarm, the shuffling rate can be increased dynamically.

Our system operates on program binaries, analyzing them and performing binary rewriting. This analysis must be complete and precise; missing even a single code pointer and failing to update it upon re-randomization can cause correctness issues. Because of the difficulty of binary analysis, we leverage existing compiler and linker flags to preserve symbols and relocations. Some (but not all [112]) vendors strip symbol information from binaries to impede reverse engineering, but reversing stripped binaries is still feasible using disassemblers like IDA Pro [72]. We anticipate that vendors would be willing to include (obfuscated) symbols and relocations in their binaries, given the additional defensive possibilities. For instance, relocations enable shuffling but are also required for executable base address randomization on Windows. In the open-source Linux world, high-level build systems are already designed to support the introduction of additional compiler flags [113], allowing distribution-wide security hardening [76, 74, 75].

Evaluation shows that our system successfully defends against all known forms of code reuse, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We ran Shuffler on a range of programs including web servers, databases, and Mozilla's SpiderMonkey Javascript interpreter. We successfully defend against a Blind ROP attack on Nginx, and against a JIT-ROP attack on a toy web server. Shuffler incurs 14.9% overhead on SPEC CPU when shuffling every 50 ms, and has good scalability on Nginx when shuffling up to 24 workers every 50 ms. We show that a 50 ms

shuffle period is orders of magnitude faster than the time required by existing JIT-ROP attacks, which take 2.3 to 378 seconds to complete [1, 2].

5.1.1 Threat Model

Shuffler is built upon continuous re-randomization. We aim to defend against all known forms of code reuse attacks, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We assume that protection against code injection (W^X) is in place, and that an x86-64 architecture is in use. Our system does not require (and, in fact, is orthogonal to) other defensive techniques like intra-function ASLR, stack smashing protection, or any other compiler hardening technique.

On the attacker's side, we assume:

1. The attacker is performing a code reuse attack, and not code injection (handled by W^X [51]) or a data-only attack [114] (outside the scope of Shuffler).

2. The attacker has access to 1) a memory disclosure vulnerability that may be invoked repeatedly to read arbitrary memory locations, and 2) a memory corruption vulnerability for bootstrapping exploits.

3. Any memory read or write that violates memory permissions (or targets an unmapped page) will cause a detectable crash, and the attacker has no meta-information about page mappings.[1]

4. The attacker knows the re-randomization rate and can time their attack to start at the very beginning of a shuffling period, maximizing the time that code addresses remain the same.

Our technique is particularly effective when defending long-lived processes and network-facing applications, such as servers. Note that network-based attackers have additional latency induced by communication delays, each time they invoke a vulnerability; see Section 5.5.3 for details.

[1] Such as access to `/proc/<pid>/maps`.

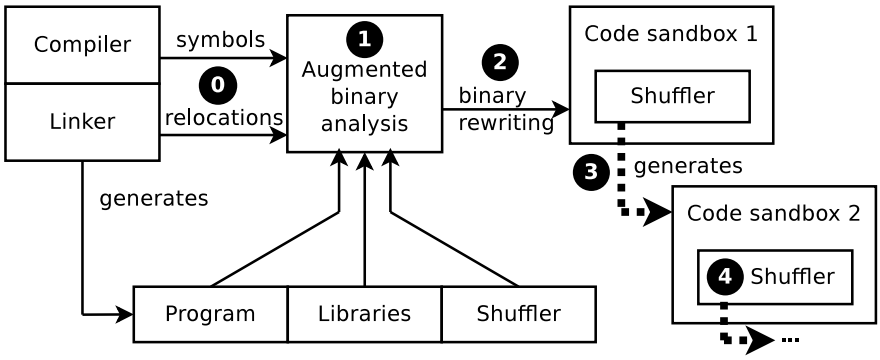


Figure 5.1: Shuffler architecture. We use symbols and relocations (0) for augmented binary analysis (1), rewrite code into shufflable form (2), and asynchronously create new code copies at runtime (3), while self-hosting (4).

5.2 Design

This section presents the design goals of Shuffler, along with its architecture, and outlines significant technical challenges. The main goals of Shuffler are:

- **Deployability:** We aim to reduce the burden on end-users as much as possible. Thus, we require no direct access to source code, no static binary rewriting on disk, and no modifications to system components (except our small loader patch).
- **Security:** Our goal is to defeat all known code reuse attacks, without expanding the trusted computing base. We constrain the lifetime of leaked information by providing a configurable shuffling period, mitigating code reuse and JIT-ROP attacks.
- **Performance:** Because time is an integral part of our security model, speed is of the essence. We aim to provide low runtime overhead, and also low total shuffling latency to allow for high shuffling rates.

5.2.1 Architecture

Shuffler is designed to require minimal system modifications. To avoid kernel changes, it runs entirely in userspace; to avoid requiring source or a modified compiler, it operates on program binaries. Performing re-randomization soundly requires complete and precise pointer analysis.

Rather than attempting arbitrary binary analysis, we leverage symbol and relocation information from the (unmodified) compiler and linker. Options to preserve this information exist in every major compiler. Thus, we are able to achieve completely accurate disassembly in what we call *augmented binary analysis*—as shown in Figure 5.1 part (1) and detailed in Section 5.2.2.

At load-time, Shuffler transforms the program's code using binary rewriting (Figure 5.1 part (2)). The goal of rewriting is to be able to track and update all code pointers at runtime. We avoid the taint tracking used by related work [13, 16] because it is expensive and would introduce races during asynchronous pointer updates. Instead, we leverage our complete and accurate disassembly to transform all code pointers into unique identifiers—indices into a *code pointer table*. These indices cannot be altered after load time, but they trade off very favorably against performance and ease of implementation. We handle return addresses (dynamically generated code pointers) differently, encrypting them on the stack rather than using indices, thereby preventing disclosure while maintaining good performance.

Our system performs re-randomization at the level of functions within a specific *shuffle period*, a randomization deadline specified in milliseconds. Shuffler runs in a separate thread and prepares a new shuffled copy of code within this deadline, as shown in Figure 5.1 part (3). This step is accelerated using a Fenwick tree. The vast majority of the re-randomization process is performed asynchronously: creating new copies of code, fixing up instruction displacements, updating pointers in the code table, *etc.* The threads are globally paused only to atomically update return addresses. Since any existing return addresses reference the old copy of code, we must revisit saved stack frames and update them. Each thread walks its own stack in parallel, following base pointers backwards to iterate through stack frames (a process known as *stack unwinding*); see Section 5.2.2 for details.

Shuffler runs in an *egalitarian* manner, at the same level of privilege as target programs, and within the same address space. To prevent our own code from being used in a code reuse attack, Shuffler randomizes it the same way it does all other code (Figure 5.1 part (4)). In fact, our scheme uses binary rewriting to transform all code in a userspace application (the program, Shuffler, and all

shared libraries) into a single code sandbox, essentially turning it into a statically linked application at runtime. Bootstrapping from original code into this self-hosting environment is challenging, particularly without substantially changing the system loader.

5.2.2 Challenges

Changing function pointer behaviour Normal binary code is generated under the assumption that the program's memory layout remains consistent and function pointers have indefinite lifetime. Re-randomization introduces an arbitrary lifetime for each block of code, and so re-randomization becomes an exercise in avoiding dangling code pointers. Failing to update even one such pointer may cause the program to crash, or worse, fall victim to a use-after-free attack.

Hence, we need to accurately track and update every code pointer during the re-randomization process. We opt to statically transform all code pointers into unique identifiers—namely, indices into a hidden *code pointer table*. Relying on accurate and complete disassembly (discussed next), we transform all initialization points to use indices. Then, wherever the code pointer is copied throughout memory, it will continue to refer to the same entry in the table. This scheme does not affect the semantics of function pointer comparison. Iterating through and updating the pointer values stored in the table can be done quickly and asynchronously.

Some code pointers are dynamically generated, in particular, return addresses on the stack. We could dynamically allocate table indices, but on the x86 architecture, `call`/`ret` pairs are highly optimized, and replacing them with the table mechanism would involve a large performance degradation [84, 115]. Instead, we allow ordinary calls to proceed as usual, and at re-randomization time we unwind the stack and update return addresses to new values. Rather than leave return addresses exposed on the stack, we encrypt each address with an XOR cipher. Every callee is responsible for disguising the return address on the top of the stack, encrypting it at function entry and decrypting before any function exit. Callers, meanwhile, are responsible for erasing the (now unencrypted) return address immediately after the called function returns. Even though the address is never used by the program, it is still a (leakable) dangling reference. The encryption key can be

unique to each function and changed during each stack unwind.

Augmented binary analysis The commonly accepted wisdom is that program analysis can be performed at the source level (requiring access to source code) or at the binary level (plagued with completeness issues). In this initial Shuffler work, we propose a middle ground, *augmented binary analysis*: we analyze program binaries that have additional information included by the compiler. We use existing compiler flags and have no visibility into the source code, and yet can achieve complete disassembly.

The common problems with binary analysis are distinguishing code from data, and distinguishing pointers from integers. To tackle these issues, we require that (a) the compiler preserve the symbol table, and (b) that the linker preserve relocations. The symbol table indicates all valid `call` targets and makes disassembly straightforward—we iterate through symbols and disassemble each one independently; there is no need for a linear sweep or recursive traversal algorithm [116]. Relocations are used to indicate portions of an object file (or executable) that need to be patched up once its base address is known. Since each base address is initially zero, every absolute code pointer must have a relocation—but as object files are linked together, most code pointers get resolved and their relocations are discarded. We simply ask the linker to preserve these relocations.

Bootstrapping into shuffled code As stated above, Shuffler defends its own code the same way it defends all other code—leading to a difficult bootstrapping problem. Shuffled code cannot start running until the code pointer table is initialized, requiring some unshuffled startup code. Shuffled and original code are incompatible if they use code pointers; the process of transforming code pointers to indices overwrites data that the original code accesses, and then the original code will no longer execute correctly. For example, if Shuffler naïvely began fixing code pointers while making code copies with `memcpy`, it would at some point break the `memcpy` implementation, because the latter uses code pointers for a jump table.[2] Hence, we would have to call new functions as they became available, and carefully order the function-pointer rewrite process to avoid invalidating any

[2]This crash took place in an earlier prototype of Shuffler.

functions currently on the call stack.

Instead, we opted for a simpler and more general solution. Shuffler is split into two stages, a minimal and a runtime stage. The minimal stage is completely self-contained, and it can safely transform all other code, including `libc` and the second-stage Shuffler. Then it jumps to the shuffled second stage, which erases the previous stage (and all other original code). The second stage inherits all the data structures created in the first so that it can easily create new shuffled code copies. From this point on, Shuffler is fully self-hosting.

5.3 Implementation

Shuffler runs in userspace on x86-64 Linux. It shuffles binaries, all the shared libraries that a binary depends on, as well as itself. The shuffling process runs asynchronously in a thread, without impeding the execution of the program's threads. Figure 5.2 shows a running snapshot of shuffled code. Code pointers are directed through the code pointer table and return addresses are stored on the stack, encrypted with an XOR cipher. In each shuffle period, Shuffler makes a new copy of code, updates the code pointer table and sends a signal to all threads (including itself); each thread unwinds and fixes up its stack. Shuffler waits on a barrier until all threads have finished unwinding, then erases the previous code copy.

Our Shuffler implementation supports many system-level features, including shared libraries, multiple threads, forking (each child gets it own Shuffler thread), `{set,long}jmp`, system call re-entry, and signals. Shuffler does not currently support `dlopen` or C++ exceptions. Yet, it does expose several debugging features, notably, exporting shuffled symbol tables to GDB and printing shuffled stack traces on demand.

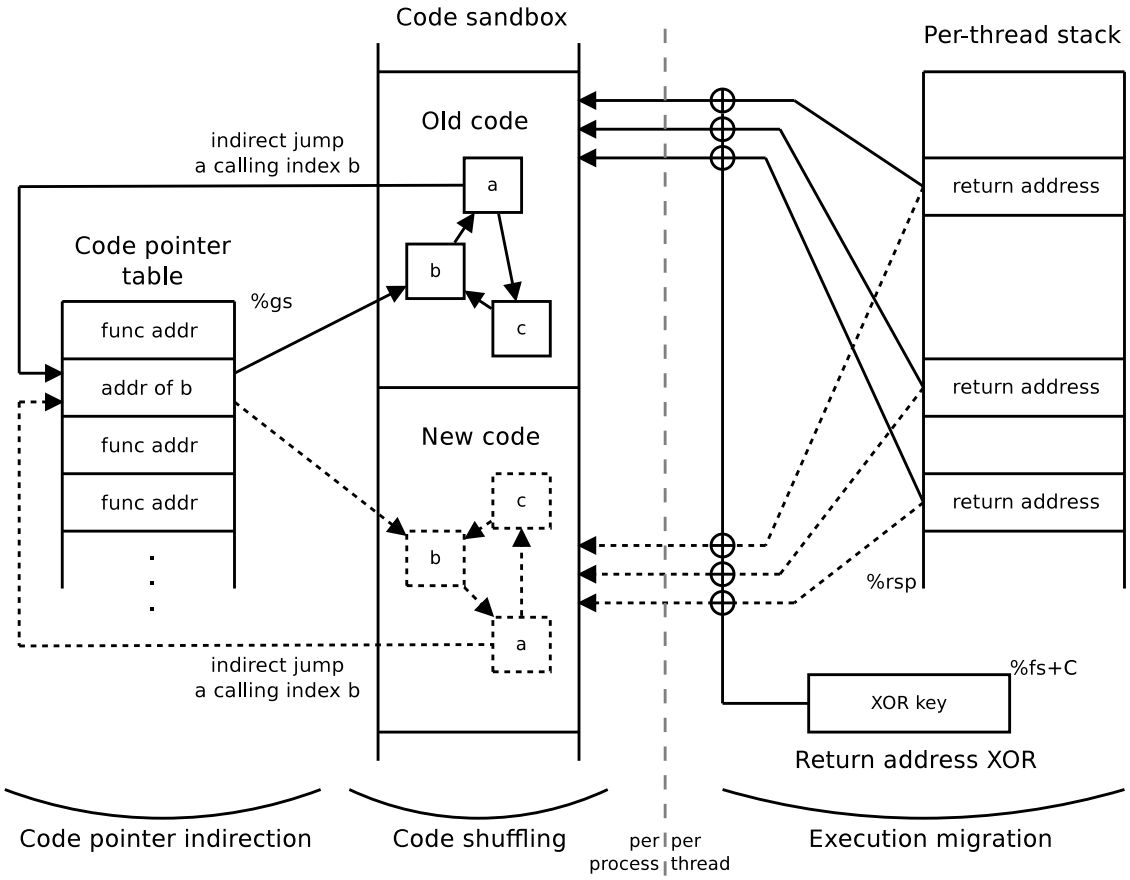


Figure 5.2: Overview of shuffled code at runtime, as Shuffler executes a shuffle pass. The old code is shown with solid lines and the new code with dotted lines.

5.3.1 Transformations to Support Shuffling

Code pointer abstraction We allocate the code pointer table at load-time and set the base address of the `GS` segment (selected by the `%gs` register) at it. Then, we transform every function pointer at its initialization point from an address value to an index into this table. We use relocations generated by the compiler and preserved by the linker flag `-q` to find all such code pointers. Pointer values are deduplicated as they are assigned indices in the table, for more efficient updating. Jump tables are handled similarly, with indices assigned to each offset within a function that is used as a target. Note that indices may also be assigned dynamically by Shuffler (*e.g.,* so that `setjmp` works across shuffle periods).

We must also transform the code so that indices are invoked properly. As shown in the

Source instruction		Transformation
`lea funcptr, %rax`	→	`lea index, %rax`
`call *%rax`	→	`callq *%gs:(%rax)`
`callq *(%rax,%rbx,8)`	→	`mov (%rax,%rbx,8),%r11` `callq *%gs:(%r11)`
`jmp *%rax`	→	`jmpq *%gs:(%rax)`

`jmpq *(%rax,%rbx,8)`	→	`mov %r11, %fs:0x88`
		`mov (%rax,%rbx,8),%r11`
		`mov %gs:(%r11),%r11`
		`xchg %r11, %fs:0x88`
		`jmpq *%fs:0x88`

(a) Transforms to support the code pointer table.

Source instruction		Transformation

`# function begin`	→	`mov %fs:0x28,%r11`
		`xor %r11,(%rsp)`
		`# function begin`

`ret / jmp *%rax`	→	`mov %fs:0x28,%r11`
		`xor %r11,(%rsp)`
		`ret / jmp *%rax`

`call anything`	→	`call anything` `mov $0x0, -8(%rsp)`

(b) Transforms to support return address encryption.

Figure 5.3: Binary rewriting transformations performed by Shuffler. `%fs:0x28` is the stack canary, `%r11` is a scratch register, and `%fs:0x88` is a scratch variable.

Figure 5.3a, every instruction which originally used a function pointer value is rewritten to instead indirect through the `%gs` table. This adds an extra memory dereference. Since x86 instructions can contain at most one memory reference, if there is already a memory dereference, we use the caller-saved register `%r11` as scratch space. For (position-dependent) jump tables, there is no register we can safely overwrite, so we use a thread-local variable allocated by Shuffler as a scratch space (denoted as `%fs:0x88`).

Return-address encryption We encrypt return addresses on the stack with a per-thread XOR key. We reuse the stack canary storage location for our key; our scheme operates similarly to stack canaries, but does not affect the layout of the stack frame. As shown in Figure 5.3b, we add two instructions at the beginning of every function (to disguise the return address) and before every exit jump (to make it visible again); after each `call`, we insert a `mov` instruction to erase the

now-visible return address on the stack. We again use `%r11` as a scratch register, since it is a caller-saved register according to the x86-64 ABI, and thus safe to overwrite.

Displacement reach A normal call instruction has a 32-bit displacement and must be within ± 2GB of its target to "reach" it. Shared libraries use Procedure Linkage Table trampolines to jump anywhere in the 64-bit address space. We wish to use only 32-bit calls and still enable function permutation; thus, we place all shuffled code at most 2GB apart, and transform calls through the PLT into direct function calls. Essentially, we convert dynamically linked programs into statically linked ones at runtime.

5.3.2 Completeness of Disassembly

We demonstrate the complete and precise disassembly of binaries that have been augmented with a symbol table and relocations. The techniques shown here are sufficient to analyze `libc`, `libm`, `libstdc++`, the SPEC CPU binaries, and the programs listed in our performance evaluation section. While shuffling these libraries and programs, we encountered myriad special cases. Figure 5.4 lists the main issues we faced, which would also need to be handled by other systems performing similar analyses. The issues boil down to: (a) dealing with inaccurate/missing metadata, especially in the symbol table; (b) handling special types of symbols and relocations; and (c) discovering jump table entries and invocations.

Jump tables One major challenge is identifying whether relocations are part of jump tables, and distinguishing between indirect tail-recursive jumps and jump-table jumps. If we fail to realize that a relocation is part of a jump table, we will calculate its target incorrectly and the jump will branch to the wrong location; if we decide that a jump table's jump is actually tail recursive, we will insert return-address decryption instructions before it, corrupting `%r11` and scrambling the top of the stack.

GCC generates jump tables differently in position-dependent and position-independent code (PIC). Position-dependent jump tables use 8-byte direct pointers, and are nearly always invoked

Issue	Description	How to handle
Missing symbol sizes	Internal GCC functions have a symbol size of zero.	Hard-code sizes; `_start` is 42 bytes.
Fall-through symbols	Functions implicitly fall through to the following function.	Attach a copy of the following code.
Overlapping symbols	Some functions are a strict subset of an enclosing function.	Binary search for targets very carefully.
Symbol aliases	Symbol tables have many names for the same function.	Pick one representative name.
Ambiguous names	One LOCAL name, multiple versions (`bsloww` in libm).	Look up address resolved by the loader.
Pointers to static functions	For pointers to functions within the same module, the offset is known, and object files contain no relevant relocations.	Determine if `lea` instructions target a known symbol (not completely sound).
`noreturn` function calls	GCC always generates a NOP after calls to `noreturn` functions like `longjmp`, but omits unwind information.	Detect when at a NOP following a call and use unwind info from at the call.
COPY relocations	Object initialized in one library, then `memcpy`'d to another.	Track data symbols, not just code.
IFUNC symbols	Return pointer to actual function to call (cached in PLT).	Statically evaluate from `lea` refs.
Conditional tail recursion	Does not appear in normal GCC-generated code. Used in hand-coded assembly by glibc (`lowlevellock.h`).	Can do XOR'ing both before and after, works whether or not the jump is taken.
Indirect tail rec.	Difficult to tell apart from jump-table jumps.	Use a function epilogue heuristic.
Finding jump tables	Jump tables are not clearly delineated.	See the text for a discussion on this.

Figure 5.4: Special cases in augmented binary disassembly.

by an instruction of the form `jmpq *(%rax,%rbx,8)` at any optimization level. PIC jump tables use 4-byte relative offsets added to the address of the beginning of the table—and the `lea` that loads the table address may be quite distant from the final indirect jump. To find PIC jump tables, we use outgoing `%rip`-relative references from functions as bounds and check if they point at sequences of relocations in the data section.[3] Note that `R_X86_64_PC32` relocations must have 4 bytes added to their value (the displacement size) if present in an instruction, and they must not if present in a jump table.

It is difficult to tell whether a `jmpq *%rax` instruction is used for indirect tail recursion, or a PIC jump table. In our system, we must distinguish these to decide whether to decrypt the return address or not. We do this with a heuristic that pairs function epilogues with function prologues. We use a linear sweep to record `push` instructions in the function's first basic block, and keep a log of the `pop` instructions seen since the last jump (within a window size). If an indirect jump is preceded by `pop` instructions that are in the reverse order of the `push` instructions, we assume we have found a function epilogue and that the jump is indirect tail recursive.

5.3.3 Bootstrapping and Requirements

We carefully bootstrap into shuffled code using two libraries (stage 1 and stage 2) so that the system never overwrites code pointers for the module that is currently executing. These libraries are injected into the target using `LD_PRELOAD`.[4] Rather than reimplement loader functionality, we defer to the system loader to create a valid process image, and then take over before the program—or even its constructors—begin executing.

The constructor of stage 1 is called before any other via the linker mechanism `-z initfirst`.[5] Then, by setting breakpoints in the loader itself, stage 1 makes sure all other constructors run in shuffled code. The last constructor to be called (a side effect of `LD_PRELOAD`) is stage 2's own constructor; stage 2 creates a dedicated Shuffler thread, erases the original copy of all other code,

[3] Fortunately, GCC only emits jump tables of size five or more, which makes this heuristic very accurate.

[4] `LD_PRELOAD=./libshuffle0.so:./libshuffle.so`

[5] We require a patch to fully use this mechanism; see Section 5.3.3.

and resumes execution at the shuffled ELF entry point.

Full Shuffling Requirements

Compiler flags We require the program binary and all dependent libraries to be compiled with `-Wl,-q`, a linker flag that preserves relocations. Since we require symbols and DWARF unwind information, the user must avoid `-s`, which strips symbols, and `-fno-asynchronous-unwind-tables`, which elides DWARF unwind information. For simplicity, we do not support some DWARF 3 and 4 opcodes, so the user may need to pass `-gdwarf-2` when compiling C++. Finally, we found that some SPEC CPU programs required `-fno-omit-frame-pointer`, due to a limitation in our DWARF unwind implementation.

System modifications The `-z initfirst` loader feature currently only supports one shared library, and `libpthread` already uses it. To maintain compatibility with `libpthread`, we patched the loader to support constructor prioritization in multiple libraries. Our 24-line patch transforms a single variable into a linked list. (We have submitted our patch to `glibc` for review.)

Since shuffled functions must be within ± 2GB of each other, we simplify Shuffler's task and map all ELF `PT_LOAD` sections into the lower 32 bits of the address space (1-line change to the loader). Since `glibc` and `libdl` refer directly to variables in the loader with only 32-bit displacements, we also place the loader itself into that region, preresolving its relocations with `prelink` [117]. Finally, we disabled a manually-constructed jump table in the `vfprintf` of `glibc`, which used computed `goto` statements (1-line change). No other library changes were necessary.

5.3.4 Implementation Optimizations

Generating new code The Shuffler thread maintains a large code *sandbox* that stores shuffled (and currently executing) functions. In each shuffle period, every function within the sandbox is duplicated and the old copies are erased. The sandbox is split in half so that one half may be easily

erased with a single `mprotect` system call.[6] Performance suffers if each function is written to an independent location in the sandbox. The bottleneck is in issuing many `mprotect` system calls (we do not want to expose the whole sandbox by making it writable). Instead, we maintain several *buckets* (64KB–1MB) and each function is placed in a random bucket; when a bucket fills up, it is committed with an `mprotect` call and a fresh bucket is allocated. The Memory Protection Keys (MPK) feature on upcoming Intel CPUs [119] may allow buckets to be created even more efficiently.

Generating function addresses with high entropy (*i.e.,* uniformly at random) is a challenging task. The simplest allocator would pick random addresses repeatedly until a free location is found, but this may require many attempts due to fragmentation. Instead, we use a Fenwick Tree (or Binary Indexed Tree) [120, 121] for our allocations. Our tree keeps track of all valid addresses for new buckets, storing disjoint intervals; it also tracks the sum of interval lengths (*i.e.,* the amount of free space). We can select a random number less than this sum and be assured that it maps to some valid free location, and compute this mapping in logarithmic time. This guarantees that each allocation is selected uniformly at random.

Stack unwinding Stack unwinding is performed by parsing the DWARF unwind information from the executable. This information is used by exception handling code, and by the debugger to get accurate stack traces. We found that the popular library `libunwind` [122] was quite unwieldy, used unwind heuristics, and made it difficult to add an address-translation mechanism. Hence, we wrote a custom unwind library with a straightforward DWARF state machine, using binary search to translate between shuffled and original addresses. We generate DWARF information for new code inserted through binary rewriting, and also record the points where return addresses are (or are not) encrypted.

[6] This also clears the old code from the instruction cache, since Linux's updates to the Translation Lookaside Buffer (TLB) flush the appropriate cache lines as per Section 4.10.4 of the Intel manual [118].

Binary rewriting Shuffler's load-time transformations are all implemented through binary rewriting. We disassemble each function with diStorm [123] and produce intermediate data structures which we call *rewrite blocks*. Rewrite blocks are similar to basic blocks but may be split at arbitrary points to accommodate newly inserted instructions. Through careful block splitting, we can choose whether incoming jumps execute or skip over new instructions as appropriate. This data structure also allows fast linear updates of internal offsets for jump instructions. We promote 8-bit jumps to 32-bit jumps (iteratively) if the jump targets have become too far away. Once jumps and other data structures are consistent, the final code size is known and we create the first shuffled copy of a function. The runtime shuffling process copies the shuffled version of each function to a new location and patches it without invoking the rewriting procedure.

5.4 Performance Evaluation

Unless otherwise noted, performance results were measured on a dual-socket 2.8GHz Westmere Xeon X5660 machine, with 64GB of RAM and 24 cores (hyperthreading enabled), running Ubuntu 16.04 with GCC 4.8.4.

5.4.1 SPEC CPU2006 Overhead

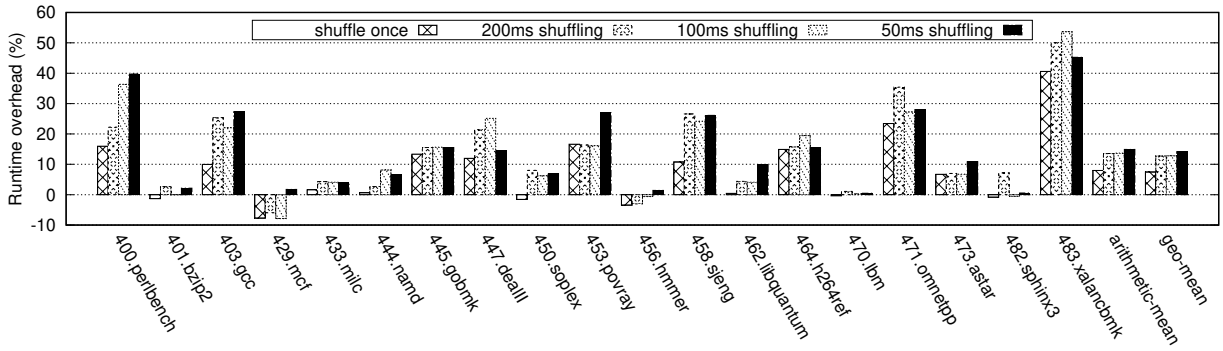


Figure 5.5: Shuffler performance (shown as overhead percentage) on SPEC CPU2006 at different shuffling rates.

We ran Shuffler on all C and C++ benchmarks in SPEC CPU2006, over a range of different shuffling periods. The SPEC baseline was compiled with its default settings (`-O2`). The shuffled

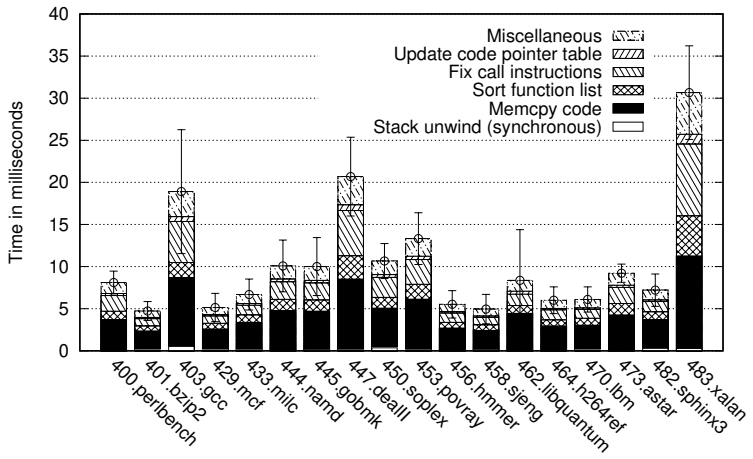


Figure 5.6: SPEC CPU continuous shuffling breakdown. Synchronous (stack unwind) overhead is barely visible at the bottom. Data for `omnetpp` was not gathered.

versions were compiled the same way with the addition of `-Wl,-q`, and additionally with `-fno-omit-frame-` due to a limitation in our DWARF unwind implementation. Since Shuffler does not yet support C++ exceptions, we replaced exceptions with conventional control flow in `omnetpp` (20-line change) and `povray` (15 lines).

Effect of shuffling rate Figure 5.5 shows the overhead observed by the single-threaded SPEC benchmarks at different shuffling rates, excluding the overhead of the Shuffler thread. The average overheads are 7.99% (shuffling once), 13.5% (200ms shuffling), 13.7% (100ms shuffling), and 14.9% (50ms shuffling). Considering that thousands of shuffles were performed in each case (the runtime per program is from 3.5–10 minutes), the observed overhead is acceptable. Note that faster shuffling rates do not cause significant slowdown, because the static code rewriting cost is paid only once (up-front).

Asynchronous overhead By design, Shuffler offloads the majority of the shuffling computations onto another CPU core (see Figure 5.6). We assume that the protected system is not at full capacity and has sufficient cycles to execute the Shuffler thread concurrently.

We can, however, approximate the shuffling overhead: the asynchronous shuffling time divided by the shuffling period yields the CPU load. Assuming `gcc` asynchronously shuffles in 25 milliseconds,

it would use 50% of the offload core in a shuffle period of 50 milliseconds, and 25% in a shuffle period of 100 milliseconds. We confirmed this approximation by measuring the reported CPU usage once per second, as each SPEC CPU program ran. The true overheads were within a few percentage points of the approximation. For instance, `xalancbmk` was predicted to use 61.31% of the CPU in the Shuffler thread and in fact used 58.64%.

Synchronous overhead The only synchronous work in Figure 5.6 is the short time when the program thread is interrupted via a signal to perform stack unwinding. Shuffler's stack unwind performance is linear in the call stack depth, processing 3247 stack frames per millisecond (including the thread barrier synchronization time between Shuffler and the program threads). Most SPEC programs have modest call stack depths, except `xalancbmk`, where certain stages have call stacks at least 20,000 deep (up to 45,000), and take up to 6 ms to unwind. The highest average unwind time is 0.53 ms for `gcc`; the Shuffler thread unwinds itself in ~0.025 ms.

5.5 Security Analysis

In this section we show how Shuffler defends against existing attacks assuming all its mechanisms are in place, including code pointer indirection, return address encryption, and continuous shuffling every r milliseconds. Then we discuss other possible attacks against the Shuffler infrastructure, and follow up with some case studies.

5.5.1 Analysis of Traditional Attacks

Normal ROP It is fairly obvious that a traditional ROP attack will fail when the target is being shuffled, because the addresses of gadgets are hard-coded into the exploit. Shuffler's code sandbox currently has 27 bits of entropy (a 31-bit sandbox should be possible as per Section 5.3.1) and gadgets could be anywhere in the sandbox. Thus, if the ROP attack uses N distinct gadgets, the chance of it succeeding is approximately 2^{-27N}. Any attack which desires better odds needs to incorporate a memory disclosure component to discover what Shuffler is doing.

Indirect JIT-ROP Indirect JIT-ROP relies on leaked code pointers and computes gadgets accordingly. Because code pointers are replaced with table indices, the attacker cannot gather code pointers from data structures; nor can the attacker infer code pointers from data pointers, since the relative offset between code and data sections changes continuously. While the attacker can disclose indices, these are not nearly as useful as addresses: they can only be used to jump to the beginning of a function, and they cannot reveal the locality of nearby functions. We assume indices are randomly ordered at load time, with gaps (traps) in the index space to prevent an attacker from easily brute-forcing it [68]. The table itself is a potential source of information, but the table's location is randomized and it is continuously moved (see Section 5.5.2 below). Return addresses are encrypted with an XOR cipher, so disclosing them does not reveal true code addresses. In fact there are no sources of code pointers accessible to an attacker by way of memory disclosure, and so indirect JIT-ROP is impossible by construction.

Direct JIT-ROP In direct JIT-ROP [1], the attacker is assumed to know one valid code address, and employs a memory disclosure recursively, harvesting code pages and finding enough gadgets for a ROP attack. A control flow hijack is used to kick off the exploit execution.

Our argument against JIT-ROP is threefold. First, the attacker must be able to obtain the first valid code address, and as described for indirect JIT-ROP, there is no accessible source of code pointers in the program. Thus the attacker must resort to brute force or side channels (as for Blind ROP below). Second, once an attack has been completely constructed, there is no easy way to jump to an address of the attacker's choosing: indirect calls and jumps treat their operands as table indices, not addresses, while return statements mangle the return address before branching to a target. The attacker must therefore use a partial return address overwrite (described below in Section 5.5.2), which itself has a significant chance of failure.

Thirdly, and most importantly, the entire attack must be completed within the shuffle period of r milliseconds. No useful information carries over from one shuffle period to the next, and all previously discovered code pages and gadgets are immediately erased. If the attacker can do

everything in r milliseconds, they win; thus, the defender should select a small enough r to disrupt any anticipated attacks. We discuss the attack time required in Section 5.5.3. The fastest published attack times are on the order of several seconds, not tens of milliseconds.

Blind ROP Blind ROP [103] tries to infer the layout of a server process by probing its workers, which are `fork`ed from the parent and have the same layout. The attack uses a timing channel to infer information about the parent based on whether the child crashed or not. Shuffler easily thwarts this attack because it randomizes child and parent processes independently.

5.5.2 Shuffler-specific Attacks

Breaking XOR encryption Our XOR encryption is less vulnerable to brute force than typical XOR ciphers. Leaking multiple return addresses does not allow the attack to easily construct linear relations, because there are two unknowns: random values (addresses) encrypted under a random key. The addresses are re-randomized during each shuffle period, and the XOR key could be too. If every function uses it own key, the attacker's task becomes even harder [17]. The keys are stored at unknown addresses in thread-local storage. While there is a small window of two instructions after calls during which the unencrypted return address is visible on the stack, this would be difficult to exploit because the attacker cannot insert any intervening instructions—though a determined attacker might try to do so from another thread.

It is possible to bypass XOR in other ways. For example, an attacker might partially overwrite an encrypted return address, attempting to increment the return address by a small amount without knowing the plaintext value. This could be used to initiate execution of a misaligned gadget, or to trampoline through a return instruction and jump straight to an attacker-controlled address. Such an attack would be difficult; the attacker would need to find a function on the call stack with appropriate known code layout, and then brute-force several bits of the canary.

Ciphertext-only attacks The attacker could attempt to swap valid code pointer indices. This allows an attacker to jump to the beginning of functions whose address is taken, similar to the

restrictions under coarse-grained Control Flow Integrity (CFI) [58, 36]—and such defenses have been bypassed [56, 60]. The mapping between indices and functions would have to first be discovered (subject to permutation and traps). We consider this a data-only attack [114]. As per Section 5.1.1, we do not attempt to add to the literature for data-only attacks.[7]

The attacker might swap valid encrypted return addresses on the stack. This is equivalent to jumping to call-preceded gadgets (as in coarse-grained CFI), but using only those functions which occur on the call stack. While such an attack may be theoretically possible, it has not been demonstrated in the literature—especially within the constraints of a single shuffle period, where return addresses change every r milliseconds.

Parallel attacks When Shuffler is defending a multithreaded program, every thread uses the same shuffled code layout. Thus, an attacker might run a parallel disclosure attack, multiplying the information that may be gathered from a single-threaded program. However, parallel disclosure is limited by dependencies—often one page's address is computed from another's content, so the disclosures are not parallelizable. In the worst case, defending a parallel attack requires a linearly faster shuffling rate. Currently, the user can run a multiprocess program instead (like Nginx) to avoid this issue. We also used the `%gs` register to store our code pointer table intentionally so that code could be shared between threads. It would be fairly straightforward to use the thread-local `%fs` register instead to maintain separate code copies and pointer tables for each thread, at a corresponding increase in memory and CPU use.

Exploiting the Shuffler infrastructure Since Shuffler runs in an egalitarian manner in the same address space as the target, it may be vulnerable to attack. Shuffler's code is shuffled and defended in the same way as the target, and any specific functionality (*e.g.,* dynamic index allocation) is not accessible through static references. However, Shuffler's data structures might be disclosed at runtime—*e.g.,* to reveal the location of every chunk of code. We are careful to place sensitive information in exactly one data structure, the list of chunks, which is itself destroyed and moved

[7] Thwarting this means updating indices at runtime; see Section 5.2.1.

in each shuffle period. There is a single global pointer to this list, which is stored in the `%gs` table along with code pointers.

Shuffler's code pointer table might itself be used to execute functions, or read or write function locations. As described earlier in Section 5.5.1, we assume that the table contains traps or invalid entries. This impedes execution of gadgets and requires the index-to-code mapping to be unravelled first. However, the table can be read and written directly with `%gs`-relative gadgets—which are not used by shuffled code but may occur at misaligned offsets. Writes can be disallowed using page permissions. Reads yield information that is only useful for one shuffle period; it is also a "chicken-and-egg" problem to rely on such a gadget to find one's gadgets.

Although the table contains many addresses that the attacker would like to disclose, we assume that the table location is randomized and is continuously moving during the shuffling process. The table's location is only stored in kernel data structures and the inaccessible model-specific register `%gs`. While x86 has a new instruction to read `%gs`, called `RDGSBASE`, it must be enabled through processor control flags (Linux v4.6 does not support that feature). Thus, the attacker must find the table's location through cache timing attacks or allocation spraying [124, 125], which has not been shown to be effective against a continuously moving target.

Finally, even if all of Shuffler's data is disclosed, the addresses for the next shuffle period can be made unpredictable by reseeding Shuffler's random number generator with the kernel-space PRNG `/dev/urandom`.

Shuffler thread compromise If the Shuffler thread crashes for whatever reason, the target program could continue executing its current copy of code unhindered (and undefended). To guard against this, we install signal handlers for common fatal signals. Our default policy is to terminate the process if a crash occurs in Shuffler code. We could also attempt to restart the Shuffler thread (as is done on fork). Instead of causing an outright crash, the attacker could attempt to hang the Shuffler thread, *e.g.,* by pretending that another thread has been created through data structure corruption. This particular technique would cause all threads to hang in the post-unwind synchronization

barrier, inside Shuffler code, which is not very useful for an attacker. Still, if a user is concerned that the Shuffler thread may be compromised, an external watchdog can periodically ensure (*e.g.,* by examining `/proc/<pid>/maps`) that shuffling is still occurring.

5.5.3 Case Study

When conducting a JIT-ROP attack, the attacker has a tradeoff: either quickly scan memory pages for desired gadgets, which may require many source pages; or, spend more time looking for gadgets in a small number of pages, which can be computationally prohibitive. The original JIT-ROP [1] attack searches through 50 pages to find the gadgets for an attack, and takes 2.3–22 seconds to carry out a full exploit. The ROP compiler Q [2] can attack executables as small as 20KB, but due to their use of heavyweight symbolic execution and constraint solving, their published real-world attack computation times are 40–378 seconds.

Fetching pages takes time because real memory disclosures do not execute instantaneously. The original JIT-ROP [1] attacks can harvest 3.2, 22.4, and 84 pages/second (*e.g.,* requiring between 12 and 312 milliseconds per page). We reproduced Heartbleed on OpenSSL 1.0.1f using Metasploit [126] and found that the attack takes 60ms to complete (17.2ms per additional disclosure), when the attacker is on the local machine.

5.6 Conclusion

We present Shuffler, a system which defends against all forms of code reuse through continuous code re-randomization. Shuffler randomizes the target, all of the target's libraries, and even the Shuffler code itself—all within a real-time shuffling deadline. Our focus on egalitarian defense allows Shuffler to operate at the same level of privilege as the target, from within the same address space, enabling deployment in environments such as the cloud. We require no modifications to the compiler or kernel, nor access to source code, leveraging only existing compiler flags to preserve symbols and relocations. Many defensive techniques exist outside the infrastructure they defend, or declare themselves part of the trusted computing base. We hope that Shuffler's design will

inspire more egalitarian techniques.

For the best possible performance, we perform shuffling asynchronously, making use of spare CPU cycles on idle cores. Programs spend 99.7% of their time running unhindered, and only 0.3% of their time running stack unwinding to migrate between copies of code. Shuffler can randomize SPEC CPU every 50 milliseconds with 14.9% overhead. We shuffled real-world applications including MySQL, SQLite, Mozilla's SpiderMonkey, and Nginx. Finally, Shuffler scales well on Nginx, up to a full system load of 24 worker processes on 12 cores.

5.7 Future Work

We reimplemented a simplified version of Shuffler in Egalito (JIT-Shuffling). We may experiment with other triggers for re-randomization instead of simply time elapsed and/or system call boundaries. Nibbler [111] showed that Shuffler can combine with other systems for even greater security.

Chapter 6: Conclusion

Although the simplest way to change how code is generated is to modify compilers, this can be surprisingly difficult in contexts from open source development to commercial DevOps environments. Only changes that are small, self-contained, with near-zero overhead are likely to be accepted into mainline compilers. Radical security improvements are especially difficult to get merged, because such defenses often incur significant overhead.

Binary software can be transformed using either 1) binary rewriting, 2) process virtualization, or 3) binary recompilation. Binary recompilation can be sensitive to errors in disassembly, but is the most efficient and powerful rewriting technique. When using binary recompilation, a user trades away the ability to reliably operate on any binary (the main advantage of virtualization), and gains complete flexibility over layout at a 0% performance cost. In this thesis, we describe the first general-purpose binary recompilation engine (Egalito [4, 81, 82]).

Egalito leverages metadata present in modern binaries, statically recovering all code and code pointers with very good accuracy (99.9% out of 867 cases, 92.5% out of 34,000 cases in additional experiments in Appendix C). It then enables very efficient binary recompilation, observing a substantial performance *speedup* of 1.7% on SPEC CPU thanks to binary optimizations. Egalito is currently in use by security researchers and instructors at several institutions, and we hope that a wider userbase will give it even better robustness over time. Egalito tries to represent every aspect of a binary to enable thorough understanding, analysis, and rewriting.

Egalito focuses on enabling deployable software transformation, as our goal is to run defenses entirely in userspace, operating on typical binaries that would normally be present on a system. We wrote several demonstration tools with Egalito, including code randomization, control-flow integrity, retpoline insertion, and an AFL [3] fuzzing backend. We also wrote Nibbler, which detects unused code and removes it. As one demonstration of an *egalitarian* mechanism, we

created a strong, deployable defense against code reuse attacks, called Shuffler. Shuffler randomizes function addresses, moving functions periodically every few milliseconds. Shuffler defends its own code and target code simultaneously—Shuffler actually shuffles itself. This makes an attacker's job extremely difficult, since JIT-ROP attacks can take many seconds to complete, or even longer if the attacker is located across the network.

Our hope is that this work takes a step towards binary recompilation becoming more well-known and widespread. Perhaps a positive feedback cycle will develop, where future compilers generate additional metadata to make binary analysis simpler, causing binary recompilation to become even more popular. Regardless, we hope this work inspires and supports researchers in creating robust, deployable next-generation software defenses.

References

[1] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization", in *Proc. of IEEE SOSP*, 2013.

[2] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy", in *Proc. of USENIX Security*, San Francisco, CA, 2011, pp. 25–25.

[3] M. Zalewski, *AFL*, `http://lcamtuf.coredump.cx/afl/`, 2019.

[4] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-Agnostic Binary Recompilation", in *Proc. of ASPLOS*, 2020.

[5] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating Binary Shared Libraries", in *Proc. of ACSAC*, 2019, pp. 70–83.

[6] I. Agadakos, N. Demarinis, D. Jin, K. Williams-King, J. Alfajardo, B. Shteinfeld, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Large-scale Debloating of Binary Shared Libraries", *Digital Threats: Research and Practice*, vol. 1, no. 4, pp. 1–28, 2020.

[7] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization", in *Proc. of USENIX OSDI*, 2016, pp. 367–382.

[8] D. Williams-King and J. Yang, "CodeMason: Binary-Level Profile-Guided Optimization", in *Proc. of ACM FEAST*, 2019, pp. 47–53.

[9] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning", *arXiv preprint arXiv:2010.00770*, 2020.

[10] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "StateFormer: Fine-Grained Type Recovery from Binaries Using Generative State Modeling", in *IEEE S&P*, 2021 (to appear).

[11] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwartz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution", in *Proc. of IEEE S&P*, 2019, pp. 1–19.

[12] Intel, *Control-flow Enforcement Technology Preview*, https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017.

[13] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely Rerandomization for Mitigating Memory Disclosures", in *Proc. of ACM CCS*, 2015, pp. 268–279.

[14] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand Live Randomization", in *Proc. of ACM CODASPY*, 2016, pp. 50–61.

[15] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity", in *Proc. of USENIX OSDI*, 2014, pp. 147–163.

[16] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization", in *Proc. of NDSS*, 2016.

[17] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices", in *Proc. of NDSS*, 2016.

[18] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure", in *Proc. of IEEE S&P*, 2015, pp. 763–780.

[19] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code", in *Proc. of ACM CCS*, 2014.

[20] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads", in *Proc. of ACM SIGSAC*, 2015, pp. 256–267.

[21] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities", in *Proc. of ACM CODASPY*, 2015, pp. 325–336.

[22] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in *Proc. of CGO*, 2004, pp. 75–86.

[23] Debian Project, *Debian – Autobuilder network*, https://www.debian.org/devel/buildd/, 2019.

[24] S. Guckenheimer, *What is DevOps?*, https://docs.microsoft.com/en-us/azure/devops/learn/what-is-devops, 2018.

[25] Synk, *Synk – Developer Security*, https://snyk.io/, 2020.

[26] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient Static Binary Instrumentation for Linux", in *Proc. of ISPASS*, 2010, pp. 175–183.

[27] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing Untrusted Code via Compiler-Agnostic Binary Rewriting", in *Proc. of ACSAC*, 2012, pp. 299–308.

[28] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, "RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications", in *Proc. of ACSAC*, 2017, pp. 412–424.

[29] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching", *IJHPCA*, vol. 14, no. 4, pp. 317–329, 2000.

[30] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization", in *Proc. of CGO*, 2003, pp. 265–275.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", in *Proc. of ACM SIGPLAN PLDI*, 2005, pp. 190–200.

[32] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", in *ACM SIGPLAN Notices*, vol. 42, 2007, pp. 89–100.

[33] C. Gorgovan, A. D'antras, and M. Luján, "MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM", *ACM TACO*, vol. 13, no. 1, p. 14, 2016.

[34] C. Gorgovan, *Escaping DynamoRIO and Pin - or why it's a worse-than-you-think idea to run untrusted code or to input untrusted data*, `https://github.com/lgeek/dynamorio_pin_escape`, 2016.

[35] M. Zhang, R. Qiao, N. Hasabnis, and R Sekar, "A Platform for Secure Static Binary Instrumentation", *ACM SIGPLAN Notices*, vol. 49, no. 7, pp. 129–140, 2014.

[36] M. Zhang and R Sekar, "Control Flow Integrity for COTS Binaries", in *Proc. of USENIX SEC*, 2013, pp. 337–352.

[37] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code", in *Proc. of ACM CCS*, 2012, pp. 157–168.

[38] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics", in *Proc. of NDSS*, 2018, pp. 40–47.

[39] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. V. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis", in *Proc. of IEEE S&P*, 2016, pp. 138–157.

[40] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform", in *Proc. of CAV*, 2011, pp. 463–469.

[41] A. Di Federico, M. Payer, and G. Agosta, "REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries", in *Proc. of CC*, 2017, pp. 131–141.

[42] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis", in *Proc. of ICISS*, 2008, pp. 1–25.

[43] Microsoft, *llvm-mctoll*, `https://github.com/Microsoft/llvm-mctoll`, 2020.

[44] F. Cloutier, *fcd – An optimizing decompiler*, `https://zneak.github.io/fcd/`, 2016.

[45] Trail of Bits, *remill*, `https://github.com/lifting-bits/remill`, 2020.

[46] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A Compiler-level Intermediate Representation based Binary Analysis and Rewriting System", in *Proc. of ACM EuroSys*, 2013, pp. 295–308.

[47] A. Dinaburg and A. Ruef, "Mcsema: Static translation of x86 instructions to llvm", in *ReCon 2014 Conference, Montreal, Canada*, 2014.

[48] S. Wang, P. Wang, and D. Wu, "UROBOROS: Instrumenting Stripped Binaries with Static Reassembling", in *Proc. of IEEE SANER*, 2016, pp. 236–247.

[49] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again", in *Proc. of NDSS*, 2017.

[50] AlephOne, *Smashing the stack for fun and profit*, `https://users.ece.cmu.edu/~adrian/630-f04/readings/AlephOne97.txt`, 1997.

[51] J. Corbet, *x86 NX support*, `http://lwn.net/Articles/87814/`, 2004.

[52] Solar Designer, *lpr LIBC RETURN exploit*, `http://insecure.org/sploits/linux.libc.return.lpr.sploit.html`, 1997.

[53] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)", in *Proc. of USENIX ACSAC*, 2009, pp. 60–69.

[54] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", in *Proc. of ACM CCS*, 2007, pp. 552–61.

[55] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack", in *Proc. of ACM CCS*, 2011, pp. 30–40.

[56] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity", in *Proc. of IEEE SOSP*, 2014.

[57] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity", in *Proc. of ACM CCS*, 2005, pp. 340–353.

[58] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables", in *Proc. of IEEE SOSP*, 2013.

[59] B. Niu and G. Tan, "Modular Control-flow Integrity", in *Proc. of ACM PLDI*, 2014.

[60] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection", in *Proc. of USENIX Security*, Aug. 2014.

[61] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity", in *Proc. of USENIX Security*, 2015, pp. 161–176.

[62] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity", in *Proc. of ACM CCS*, 2015, pp. 901–913.

[63] PaX Team, *PaX address space layout randomization (ASLR)*, `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[64] J. Xu, Z. Kalbarczyk, and R. Iyer, "Transparent runtime randomization for security", in *Proc. of IEEE SRDS*, 2003, pp. 260–269.

[65] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits", in *Proc. of USENIX Security*, 2005, pp. 271–286.

[66] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd My Gadgets Go?", in *Proc. of IEEE SOSP*, 2012, pp. 571–585.

[67] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, *et al.*, "Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity.", in *NDSS*, 2017.

[68] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks", in *Proc. of ACM CCS*, 2015, pp. 243–255.

[69] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware", in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 437–452.

[70] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W.-C. Hsu, and Y. Cheng, "Reranz: A light-weight virtual machine to mitigate memory disclosure attacks", in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2017, pp. 143–156.

[71] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the code space to counter disclosure attacks", in *Proceedings of the European Symposium on Security and Privacy (EuroS&P)*, 2017.

[72] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011.

[73] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis", in *Proc. of IEEE S&P*, 2007, pp. 231–245.

[74] Debian, *Hardening - Debian Wiki*, `https://wiki.debian.org/Hardening`, 2015.

[75] Ubuntu, *Security/features - Ubuntu Wiki*, `https://wiki.ubuntu.com/Security/Features#Userspace_Hardening`, 2016.

[76] Fedora, *Harden All Packages - Fedora Project*, `https://fedoraproject.org/wiki/Changes/Harden_All_Packages`, 2016.

[77] M. Meissner, *openSUSE Tumbleweed now full of PIE*, `https://lists.opensuse.org/opensuse-factory/2017-06/msg00403.html`, 2017.

[78] LLVM, *LLVM Language Reference Manual*, `https://llvm.org/docs/LangRef.html`, 2019.

[79] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Return-to-user Attacks", in *Proc. of USENIX SEC*, 2012, pp. 459–474.

[80] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted Code Randomization", in *Proc. of IEEE S&P*, 2018, pp. 461–477.

[81] D. Williams-King *et al.*, *columbia/egalito*, https://github.com/columbia/egalito, 2020.

[82] ——, *Egalito*, https://egalito.org, 2020.

[83] M. Richtarsky, *Hardening C/C++ Programs Part II - Executable-Space Protection and ASLR*, https://www.productive-cpp.com/hardening-cpp-programs-executable-space-protection-address-space-layout-randomization-aslr/, 2017.

[84] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries", in *Proc. of ACM CCS*, 2015, pp. 555–566.

[85] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks", in *Proc. of IEEE S&P*, 2019, pp. 985–999.

[86] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, India, 1995.

[87] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto, "Katana: A Hot Patching Framework for ELF Executables", in *Proc. of ARES*, 2010, pp. 507–512.

[88] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating Code from Data in x86 Binaries", in *Proc. of ECML PKDD*, 2011, pp. 522–536.

[89] G. C. Collection, *Using the GNU Compiler Collection (GCC): AArch64 Options*, https://gcc.gnu.org/onlinedocs/gcc/AArch64-Options.html, 2017.

[90] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries.", in *Proc. of USENIX SEC*, 2016, pp. 583–600.

[91] E. Bendersky, *Position Independent Code (PIC) in shared libraries on x64*, https://eli.thegreenplace.net/2011/11/11/position-independent-code-pic-in-shared-libraries-on-x64, 2011.

[92] Y. Chen, T. Lan, and G. Venkataramani, "DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries", in *Proc. of ACM FEAST*, 2017, pp. 23–29.

[93] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming using In-Place Code Randomization", in *Proc. of IEEE S&P*, 2012, pp. 601–615.

[94] P. Turner, *Retpoline: a software construct for preventing branch-target-injection*, https://support.google.com/faqs/answer/7625886, 2018.

[95] Intel, *Intel is innovating to stop cyber attacks*, https://blogs.intel.com/blog/intel-innovating-stop-cyber-attacks/, 2016.

[96] M. Shudrak, *drAFL*, https://github.com/mxmssh/drAFL, 2019.

[97] T. Huet, *AFL*, https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt, 2017.

[98] M. Backes and S. Nürnberger, "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing", in *Proc. of USENIX SEC*, 2014, pp. 433–447.

[99] A. Pennarun, B. Allombert, and P. Reinholdtsen, *Debian Popularity Contest*, https://popcon.debian.org/, 2019.

[100] Google, *fuchsia Git repositories*, https://fuchsia.googlesource.com/, 2018.

[101] A. Danial, *AlDanial/cloc*, https://github.com/AlDanial/cloc, 2017.

[102] M. Larabel, *Benchmarking Retpoline-Enabled GCC 8 With -mindirect-branch=thunk*, https://www.phoronix.com/scan.php?page=article&item=gcc8-mindirect-thunk&num=2, 2018.

[103] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking Blind", in *Proc. of IEEE S&P*, 2014, pp. 227–242.

[104] N. A. Quynh, "Capstone: Next-gen disassembly framework", *Black Hat USA*, 2014.

[105] D. Wiki, *Using Symbols Files*, https://wiki.debian.org/UsingSymbolsFiles.

[106] T. G. C. Library, *System Databases and Name Service Switch*, https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html.

[107] Docker, *Docker Hub*, https://hub.docker.com.

[108] F. S. Foundation, *libunistring*, https://www.gnu.org/software/libunistring/.

[109] Debian, *Package: cdebconf*, https://packages.debian.org/sid/cdebconf.

[110] J. Fulmer, *Siege Home*, https://www.joedog.org/siege-home/, 2012.

[111] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries", in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.

[112] MSDN, *Symbols and Symbol Files - Windows 10 hardware dev*, https://msdn.microsoft.com/en-us/library/ff558825.aspx, 2016.

[113] Debian, *sbuild - Debian Wiki*, https://wiki.debian.org/sbuild, 2016.

[114] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats.", in *Proc. of USENIX Security*, 2005.

[115] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture.", in *Proc. of USENIX Security*, 2006.

[116] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited", in *Proc. of IEEE WCRE*, 2002, pp. 45–54.

[117] Arch Wiki, *Prelink*, https://wiki.archlinux.org/index.php/Prelink, 2015.

[118] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, 2010.

[119] J. Corbet, *Memory Protection Keys [LWN.net]*, https://lwn.net/Articles/643797/, 2015.

[120] P. M. Fenwick, "A new data structure for cumulative frequency tables", *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.

[121] GeeksForGeeks, *Binary indexed tree or Fenwick tree*, http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/, 2015.

[122] GNU, *The libunwind Project*, http://savannah.nongnu.org/projects/libunwind/, 2014.

[123] G. Dabah, *diStorm3*, http://ragestorm.net/distorm/, 2003–2012.

[124] E Göktas, R Gawlik, B Kollenda, E Athanasopoulos, G Portokalidis, C Giuffrida, and H Bos, "Undermining information hiding (and what to do about it)", in *Proc. of USENIX Security*, 2016.

[125] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding", in *Proc. of USENIX Security*, 2016.

[126] H. Moore *et al.*, *The Metasploit Project*, http://www.metasploit.com/, 2009.

[127] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, “sysfilter: Automated System Call Filtering for Commodity Software”, in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020 (to appear).

Appendix A: Egalito Artifact (Virtual Machine)

Artifact guidelines and methodology: *http://cTuning.org/ae/submission.html*

A.1 Abstract

We provide a virtual machine image which contains data needed to replicate some Egalito experiments. The machine contains the following:

- Egalito source repository (pre-built).
- Instructions to build Egalito from scratch.
- Scripts for several SPEC CPU 2006 experiments.
- Scripts to run Egalito and DynamoRIO AFL fuzzing.
- Large-scale jump table analysis for Debian packages.
- Large-scale Debian package tests.

Some experiments need internet access; in particular, the large-scale experiments rely on access to a Debian mirror. At `http://doi.org/10.17605/OSF.IO/KDUZG` we provide the virtual machine image and a `README.txt` file (which includes credentials).

A.2 Artifact check-list (meta-information)

- **Algorithm: binary recompilation**
- **Program: SPEC CPU 2006 v1.1 (must be obtained separately)**
- **Compilation: GCC 6.3.0**

- **Transformations: binary-to-binary recompiler**
- **Run-time environment: Debian stretch 9.11 + internet connection**
- **Hardware: 20GB disk, 8GB RAM, 4 core virtual machine**
- **Execution: 24+ hour runtime for complete experiments**
- **Output: performance numbers and transformation pass/fail**
- **Experiments: via scripts in virtual machine**
- **How much disk space required? 20GB (fixed)**
- **Publicly available?: Yes**
- **Code licenses?: GNU GPL v3**
- **Archived? Yes**
- **Artifacts publicly available?: Yes**
- **Artifacts functional?: Yes**
- **Artifacts reusable?: Yes**
- **Results validated?: No (as per conference policy)**

A.3 Description

How delivered

Please go to `http://doi.org/10.17605/OSF.IO/KDUZG` and download the archive `egalito-artefact.tar.gz`. The archive is a 1.5GB download and requires 20GB of space once extracted. It contains the QEMU/KVM-compatible virtual machine image, a KVM machine definition XML file, and a copy of the `README.txt` with username/password and basic instructions. Further instructions on each experiment are included in additional `README` files in the home directory of the VM.

Hardware dependencies

The VM requires: 20GB disk space, 8GB RAM, 4 CPU cores. It should run on any x86_64 system (tested on Debian Linux only).

Software dependencies

We recommend using KVM with hardware acceleration enabled to run the virtual machine with optimal performance.

Data sets

SPEC CPU 2006 v1.1 (`SPECcpu2006-1.1.iso`) is required to replicate the SPEC CPU experiments. The reader must obtain this on their own.

A.4 Installation

Extract the `.tar.gz` (needs 20GB disk) to obtain the following: `egalito-artefact.qcow2`, `machine.xml`, `README.txt`. Then, create a new virtual machine with the `qcow2` disk image, or import the existing machine. To import the existing machine, update the disk image path in `machine.xml` and import it into KVM with: `virsh define machine.xml`. You may get errors about unsupported CPU features depending on your CPU; the machine was created for an Intel i7-4770 host (M1). In that case, simply select a different virtual CPU.

After you've booted the machine, log in with the credentials in the `README.txt`. You can get a `TTY` console with `virsh console egalito-artefact`. You can also find the IP address of the virtual machine by running `ip addr show` on the guest, and then `ssh` in. If you created a new machine, you may need to run `sudo dhclient DEVICE` on the guest to get network access, where `DEVICE` is the name Linux chooses for your new network card.

If you wish to replicate the SPEC CPU results, find the IP address of the virtual machine, then `scp` your `SPECcpu2006-1.1.iso` to the VM's home directory from your host machine.

Continue to follow the instructions in `README-speccpu.txt`.

A.5 Experiment workflow

Described in individual `README` files in the VM's home directory: `README-manual.txt`, `README-speccpu.txt`, `README-afl.txt`, and `README-largescale.txt`.

We recommend running experiments from within `tmux` because they can take a while. Also, you may wish to delete past experiments before running new ones to avoid running out of disk space (the VM is limited to 20GB disk).

A.6 Evaluation and expected result

Our SPEC CPU experiments should be able to successfully run `all_c` and `all_cpp` targets on `ref` size. We provide `mirrorgen` (1-1), `uniongen`, `retpolines`, `endbr` (Intel CET), `ss-const` (Intel CET + const shadow stack), and baseline configurations. Performance numbers collected in a virtual machine cannot be relied upon, but we observed similar results to our baremetal experiments:

The AFL experiment should run approximately 25x faster for Egalito-AFL than for DynamoRIO-based fuzzing. The target is `/usr/bin/readelf`; we are not aware of any bugs in this program.

The large-scale experiments rely on accessing packages, sources, and build dependencies from a Debian mirror. These will necessarily change over time, so different numbers are to be expected. We successfully analyzed 1207 executables in the jump table analysis with 17 skipped and 0 failures; full details are included in the `README`. In the large-scale package tests, we saw 90 pass, 59 fail, and 140 skipped (due to compilation errors and/or lack of tests).

A.7 Experiment customization

The user can manually invoke Egalito on any executable, with various transformations. Any binary can be fuzzed with our AFL tool. Individual SPEC CPU targets can be run. We provide a

SPEC diffing script to compare multiple runs. The large-scale tests are reusable, using a `chroot` environment to build and test packages. We believe the scripts may also be useful in other contexts.

Appendix B: Egalito Large-Scale Transformation Results

This appendix presents the results of two large-scale Egalito experiments on Debian packages. The first is Figure B.1, run on Debian stretch (current stable), where the P/F (Pass/Fail) numbers indicate the number of executables in each package where Egalito was able to identify the same jump tables as in the ground truth. The second experiment is in Figure B.2, run on all packages in Debian buster (current testing) in the Debian CI system marked as having tests and as being written in C/C++. We ran these autopkgtests after transforming the executables with Egalito in 1-1 ELF mode, on executables only. Entries are marked as follows: fail-1 for at least one test failing after transform, fail-T for failures where the source code uses C++ exceptions, skip-1 where autopkgtest fails to install dependencies, skip-C for instances where tests fail in the chroot with a certain runtime error but may pass outside when run manually (we tested some successfully).

Package	P/F	Package	P/F	Package	P/F	Package	P/F
alsa-utils	13/0	dconf-gsettings-backend	4/0	iw	1/0	screen	1/0
anacron	1/0	dconf-service	4/0	kbd	3/0	sed	1/0
apache2-bin	108/0	debianutils	3/0	less	2/0	shared-mime-info	1/0
apt	21/0	desktop-file-utils	1/0	login	4/0	sudo	12/0
apt-utils	21/0	dirmngr	1/0	logrotate	1/0	system-config-printer-udev	1/0
avahi-daemon	9/0	dmsetup	13/0	lsof	1/0	sysvinit-utils	6/0
bash	28/0	e2fsprogs	9/0	lynx	1/0	tmux	1/0
bc	2/0	file	1/0	man-db	6/0	udev	260/0
bluez	3/0	fortune-mod	3/0	modemmanager	18/0	udisks2	2/0
bsdmainutils	12/0	fuse	4/0	mount	6/0	unzip	2/0
bsdutils	6/**1**	gawk	13/0	ntfs-3g	9/0	upower	1/0
colord	0/**1**	gir1.2-glib-2.0	1/0	openssh-client	2/0	usbmuxd	1/0
coreutils	1/0	glib-networking	3/0	openssh-server	2/**1**	util-linux	7/**1**
cracklib-runtime	5/0	gnupg	1/**3**	openssl	3/0	vim	1/0
crda	3/0	gnupg-agent	1/**1**	p11-kit	1/0	whiptail	2/0
cron	2/0	gpgv	1/**1**	packagekit	3/0	x11-utils	9/0
cups	14/0	gpm	2/0	parted	2/0	x11-xkb-utils	6/0
cups-browsed	3/0	gtk-update-icon-cache	95/0	passwd	4/0	x11-xserver-utils	16/0
cups-daemon	16/0	gzip	1/0	pinentry-gnome3	4/0	xdg-user-dirs	1/0
cups-filters	3/0	hdparm	1/0	policykit-1	8/0	xxd	1/0
dash	1/0	hostname	1/0	procps	7/0	xz-utils	2/**2**
dbus	60/0	ifupdown	1/0	rsyslog	48/0		
dbus-user-session	60/0	iproute2	2/0	rtkit	2/0		

Figure B.1: Large-scale Debian jumptable results from common Debian `stretch` (stable) packages.

Package	Result	Package	Result	Package	Result	Package	Result
aegean	skip-C	exim4	fail-1	ocrad	pass	since	pass
akonadiconsole	fail-1	exonerate	pass	okular	pass	snp-sites	pass
amap-align	pass	ffmpeg	skip-C	onscripter	pass	snpomatic	pass
apt	fail-T	fityk	skip-C	parley	fail-1	spades	fail-T
aragorn	pass	freecad	pass	pbsim	pass	spline	pass
aria2	skip-C	freeradius	skip-C	pdf2djvu	skip-C	subversion	skip-C
atk1.0	pass	gdisk	pass	pgbouncer	fail-1	suricata	skip-C
augustus	fail-T	genometools	pass	pgpool2	pass	svn-all-fast-export	skip-C
bcftools	pass	gjs	skip-C	pgtap	pass	swig	pass
bedtools	fail-T	gnome-photos	skip-C	phylip	fail-1	syslog-ng	skip-C
bibtool	pass	gnudatalanguage	pass	phyml	pass	t-coffee	fail-1
binutils	pass	gnuplot	skip-C	plink	pass	tantan	pass
bio-eagle	pass	graphviz	skip-1	plink1.9	pass	tetgen	skip-C
biosquid	pass	grep	fail-1	postfix	skip-C	theseus	skip-C
bomstrip	pass	gwenview	fail-1	poxml	fail-1	tigr-glimmer	pass
bonnie++	fail-1	haproxy	fail-1	primer3	pass	timelimit	pass
bowtie	pass	haveged	pass	prips	pass	tracker	fail-1
bowtie2	fail-T	hmmer	pass	privoxy	pass	umbrello	fail-T
boxes	pass	hugin	skip-C	probcons	pass	upower	fail-1
boxshade	pass	imagemagick	pass	pulseaudio	skip-1	util-linux	fail-1
bs1770gain	skip-C	infernal	fail-1	pyrit	skip-1	valgrind	pass
bwa	pass	inspircd	fail-T	python-coverage	pass	velvet	pass
cd-hit	pass	ioping	pass	python-crypto	skip-1	vifm	pass
cdebootstrap	skip-C	limereg	skip-1	python-cups	pass	vim	skip-1
clamav	skip-1	linuxinfo	pass	python-magic	skip-1	vlc	skip-C
clustalw	fail-T	mafft	pass	python-pyxattr	pass	vorbis-tools	skip-C
cmake	pass	maq	pass	python-scipy	pass	wireshark	skip-C
colord	fail-1	marble	pass	python2.7	skip-1	wise	pass
confget	pass	memcached	fail-1	pyxplot	fail-1	x264	skip-C
coz-profiler	fail-T	miniasm	pass	quagga	fail-1	yapet	pass
cproto	pass	minimap	pass	radvd	fail-1	yorick	pass
crash	fail-T	mp4h	pass	rc	pass	zsh	pass
cups	skip-C	mscgen	skip-C	reapr	pass		
cura-engine	skip-1	mummer	pass	rna-star	skip-C		
dbus	pass	muscle	pass	rnahybrid	skip-C		
dcraw	skip-C	mustang	pass	rocs	pass		
deal	pass	nagios-plugins-contrib	pass	samba	skip-1		
dialign	pass	ncbi-blast+	skip-C	samtools	fail-1		
dolphin	pass	ncompress	pass	saods9	pass		
doxygen	fail-T	njplot	skip-C	screen	fail-1		
emboss	pass	nodejs	skip-1	sextractor	skip-C		
esorex	skip-C	notify-osd	fail-1	sim4	pass		

Result	#
pass	80
fail-1	23
fail-T	11
skip-1	12
skip-C	32
(no tests)	143
total	301

Figure B.2: Large-scale Debian `autopkgtest` results on Debian `buster` (testing).

Appendix C: Egalito Distribution-Scale Parsing Results

This appendix presents evaluation results of Egalito parsing nearly all Debian `sid` packages, with detailed diagnoses of the problems encountered. This experiment was conducted by Nick DeMarinis et al in the course of preparing the Nibbler journal paper [6], and while preparing sysfilter [127], which appeared in RAID 2020. Results included here with permission, and were reformatted independently.

Of all the results, Figure C.1a shows the total cases skipped; the most common situation is lack of support for RPATH to resolve libraries, which we consider an implementation detail. Figure C.1b shows results with skipped cases removed. The most common failure case is due to jump table analysis, which we may improve over time. However, currently 92.5% of these applicable cases are parsed correctly.

Description	Count	Absolute %
Skipped cases	816	2.32%
RPATH resolution	715	2.04%
Out of memory	95	0.27%
Missing symbols	4	0.01%
Disassembly error	2	~0%
Failing cases	2575	7.33%
Successful cases	31726	90.34%
Total cases	35117	100%

(a) Full experiment: 35117 executables across 9599 packages. We skipped 816 executables.

Description	Count	Absolute %
Failing cases	2575	7.51%
Potential bugs	59	0.17%
UseMem: Bad opcode	48	0.14%
BuildDataRegionList	11	0.03%
Real bugs	1878	5.48%
Jump tables	1136	3.31%
Overlapping data symbols	586	1.71%
Can't find base of marker	105	0.31%
DWARF parsing	51	0.15%
Unclassified bugs	638	1.86%
Unknown SIGABRT	491	1.43%
Unknown SIGKILL	89	0.26%
Unknown SIGSEGV	58	0.17%
Successful cases	31726	92.49%
Parse successful	31722	92.48%
Required parse override	4	0.01%
Total applicable cases	34301	100%

(b) Results across 34301 executables (after skipped cases removed). 92.5% parsed correctly.

Figure C.1: Distribution-scale amd64 (x86_64) Debian package parsing.

This experiment scanned all Debian packages, with the following exceptions: a) packages that do not contain scannable binaries by definition, including: documentation packages (*-doc), headers (*-dev), debug symbols (*-dbg, *-dbgsym); b) packages that only reference other packages which are scanned separately, such as metapackages and virtual packages; c) standalone library packages (those tagged 'libs'), which are scanned alongside packages that use them; and d) packages not for the amd64 architecture. For the amd64 architecture, Debian's repository (main, contrib, non-free) contains about 48k packages ignoring the other restrictions.