Practical Software Specialization against Code Reuse Attacks

A Dissertation presented

by

Hyungjoon Koo

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2019

Stony Brook University

The Graduate School

Hyungjoon Koo

We, the dissertation committe for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dr. Michalis Polychronakis - Dissertation Advisor
Assistant Professor, Computer Science Department

Dr. R. Sekar - Chairperson of Defense
Professor, Computer Science Department

Dr. Nick Nikiforakis - Committee Member
Assistant Professor, Computer Science Department

Dr. Vasileios P. Kemerlis - External Committee Member
Assistant Professor, Computer Science Department
Brown University

This dissertation is accepted by the Graduate School

Richard Gerrig
Interim Dean of the Graduate School

Abstract of the Dissertation

Practical Software Specialization against Code Reuse Attacks

by

Hyungjoon Koo

Doctor of Philosophy

in

Computer Science

Stony Brook University

2019

Abstract

Software bugs are everywhere. Among them, exploitable bugs often threaten the security and privacy of users. The security community has been combating memory corruption vulnerabilities that can lead to code injection or code reuse attacks for several decades. Although the deployment of exploit mitigations (e.g., non–executable memory and address space layout randomization) in modern operating systems has raised the bar, recent adversarial advancements in code reuse attacks (e.g., disclosure–aided or just–in–time return oriented programming (JIT-ROP)) still allow adversaries to bypass these mitigations and achieve successful exploitation. Such sophisticated attacks can be mitigated further using *fine-grained* code diversification, as either a standalone defense or a prerequisite of other protections (e.g., execute–only memory). However, despite decades of research, software diversity has remained mostly an academic exercise for three main reasons: i) lack of a transparent and streamlined model for delivering diversified binaries to end users, ii) unaffordable cost and complexity for creating

diversified variants, and iii) incompatibility with well-established software build, regression testing, debugging, crash reporting, diagnostics, and security monitoring workflows and mechanisms.

In this dissertation, we present a *practical software specialization framework* against code reuse attacks, tackling various challenges that have so far prevented the practical deployment of code diversification and specialization. First, we propose *instruction displacement*, a practical code diversification technique for stripped binary executables, applicable even with partial code disassembly coverage. It aims to improve the randomization coverage and entropy of existing binary–level code diversification techniques by displacing any non-randomized gadgets to random locations. Second, we also explore *code inference attacks and defenses*: a novel code inference attack that can undermine defenses based on destructive code reads, and a practical defense against such inference attacks based on code re-randomization. Next, we present *compiler–assisted code randomization*, a compiler-rewriter cooperation approach that allows for practical, generic, robust, and fast fine-grained code transformation on endpoints. It is based on a hybrid model in which both vendors and endpoints jointly participate in creating specialized instances of a given application, satisfying four key factors for successful deployment: transparency, reliability, compatibility, and cost. To this end, we identify a minimal set of supplementary information for code diversification from the compilation toolchain (compiler and linker), and augment binaries with transformation-assisting metadata for on-demand rewriting on endpoints. The results of our experimental evaluation demonstrate the feasibility and practicality of this approach, as on average it incurs a modest file size increase and negligible runtime overhead. Lastly, we introduce *configuration-driven code debloating*, an approach that removes feature-specific shared libraries that are exclusively needed only when certain configuration directives are specified by the user, and which are typically disabled by default.

Thesis Advisor: Michalis Polychronakis

Dedicated to my wife, Sungmy, who always inspired and empowered me
to end a long journey and who overcame every obstacle beside me,
my little boy, Ian, who invited me into the world of true curiosity,
and my parents who have been endless supporters, no matter where I am

Table of Contents

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank my advisor, Michalis Polychronakis, for his fruitful guidance and continuous support throughout my Ph.D. journey. One of the most important lessons I have learned from him is to think critically and rationally about research problems. He also always encouraged me not to be intimidated and be patient when things were not going as well as expected. He not only taught me knowledge when I did not understand something clearly but also guided me in another direction when I was stuck on a particular subject. On top of that, his excellent writing skills always motivated me to improve my writing and my publications.

I am very happy to have become his first Ph.D. student. Looking back on the summer of 2014, I vividly remember my gratitude when he agreed to become my mentor at Columbia University even before he became a faculty member at Stony Brook University. The glory of crossing the finish line would never have happened without his commitment and efforts ever since.

I would like to thank my thesis committee members, R. Sekar, Nick Nikiforakis and Vasileios P. Kemerlis, who offered invaluable feedback. In particular, Vasileios helped me a lot when, as a co-author, he provided useful ideas for enhancing one of my papers. I also would like to thank my former advisor, Phillipa Gill, who gave me an opportunity to join her lab and study traffic differentiation and censorship. I was able to practice many fundamental research skills under her direction.

I should express my sincere gratitude to my Hexlab colleagues, Shachee, Phong, Tapti, and Seyedhamed. They are amazing labmates: intelligent, zealous, and cheerful. They made the lab lively, and I often enjoyed constructive discussions with them. It was also great to have friends and collaborators from other security labs: Panagiotis, Sergej, Manolis, Christine, Yaohui, Mingwei, Rui, Nahid, Oleksii, Rishab, Meng, Najmeh, Babak, Timothy, Rachee, Micah, Anke, Abbas, Ruhul, Dung, Javad, Jian, Mina, Junseok, Jihoon, Kiwon, Sungsoo, Heeyoung, Yongseo, Shinyoung, Haena and Sohee.

Last but not least, I can never thank my wife enough for her dedication and for going through this long journey with me. My little boy was a precious gift for my family during the journey. I thank my parents who, as always, silently rooted for me on the other side of the globe.

1 Introduction

Memory corruption attacks against software are a long–standing problem in the security field due to a lack of memory safety checks. A low–level programming language such as C or C++ offers a way to (in)directly access memory with a pointer. The pointer contains the address of a memory object and is used to dereference data and/or code. Because a programmer is responsible for offering appropriate memory safety checks (ensuring that a pointer access is within bounds and to a valid memory object), the absence of such checks may give rise to an exploitable memory vulnerability. This has remained a challenging task for software security experts to resolve for more than three decades.

In 1988, the Morris worm [1] was recorded as the first instance that enables memory corruption (i.e., buffer overflow) to be propagated via the internet. It infected thousands of real servers, crashing their running systems. Attacks that exploited memory corruption vulnerabilities became a mainstream threat after the disclosure of a buffer overflow operation by Aleph One in 1996 [2]. In the same vein, the Code Red worm [3] successfully attacked 250,000 machines all over the world that were running Microsoft's IIS web server over a period of 9 hours in 2001. Two years later, the SQL Slammer worm slowed down the internet by infecting 75,000 machines within a mere 10 minutes [4]. The Conficker worm, also known as DOWNAD or Kido, spread to millions of computer systems worldwide after being first spotted in 2008. Since its initial discovery, several different variants have been confirmed. Surprisingly, this notorious worm has been reported as actively self-propagating in the wild via shared networks for almost a decade [5]. One of the latest malware that hit end users around the globe was WannaCry ransomware in 2017. It has infected 200,000 computers across 150 countries [6], which leverages multiple zero-day vulnerabilities to exploit the victims [7]. Once infected with the ransomware, it encrypts user files and demands the ransom [8] for decryption through a cryptocurrency such as Bitcoin [9].

All the above malware took advantage of buffer overrun vulnerabilities, popularizing them. Traditional memory corruption exploits involve two steps for them to be successful: i) injecting carefully crafted code of an attacker's choice on the stack or heap (i.e., input manipulation) and ii) executing the injected code by diverting the original control flow within a vulnerable process (i.e., stack overflow, heap overflow, or format string) [2, 10]. Figure 1 shows

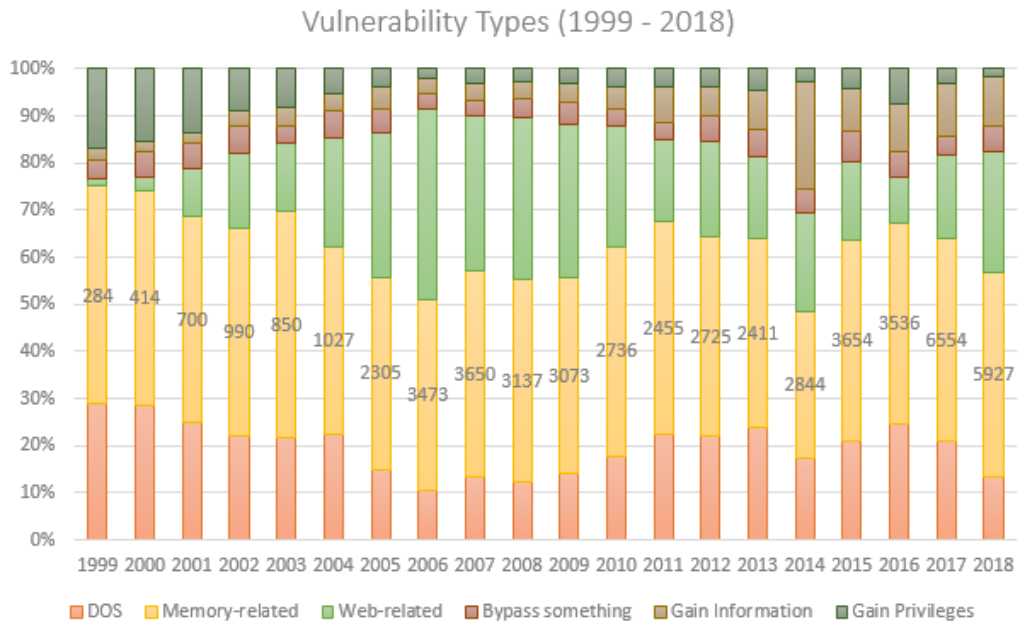


Figure 1. Vulnerability types for the last 20 years [11]

the distribution of types of vulnerability that have been recorded over the last 20 years [11] (from 1999 to 2018). Memory–related vulnerabilities include overflow, memory corruption, and code execution. It is noteworthy that the rate of memory–related vulnerabilities (i.e., yellow bars) has remained almost unchanged (i.e., 46% in 1999, 43% in 2008, and 43% in 2018) for the last two decades, in spite of significant research efforts to develop defensive solutions [12–14].

Various techniques have been proposed including stack canaries, stack cookies, shadow stacks, instruction set randomization (ISR) and data space randomization (DSR) [15–18] to prevent code injection attacks. A notable mitigation against a classic code–injection–based attack is to restrict the execute permission to a certain memory region so that no memory region can hold both writable and executable permissions at the same time (non-executable memory or data execution prevention) [19]. This model prevents adversaries from executing any injected code, thus leading to exploitation failure.

However, a novel exploitation technique, dubbed *ret2libc* [20], was in-

troduced to bypass execute-only memory page protection. The main idea behind this technique was to use existing functions in `libc`, but attackers devised a generic way to execute the arbitrary code they needed instead of borrowing whole functions [21]. This was later formalized as *return-oriented programming (ROP)* [22], which leverages multiple code chunks to generate a functional payload. Attackers select a sequence of instructions, termed a *gadget*, to perform a simple operation and carry out a full exploitation by chaining multiple gadgets together. This technique can circumvent a non-executable memory policy, which has shifted the paradigm from code injection to code reuse attacks.

A naive approach to preventing such attacks is to remove all potential bugs from memory. However, it is often an extremely challenging task to write a bug-free application due to the complexity of modern software and a lack of extensive tests for functionality and performance. For example, considering only lines of code (LOC) for the sake of simplicity, Linux Kernel 4.14 has over 25 million LOC [23] and the Chromium 69 browser has almost 33 million LOC [24]. Besides, modern applications tend to be built on top of heterogeneous frameworks, toolkits, and libraries in a highly modularized fashion, which makes tracking bugs even more difficult. Choosing a memory-safe language is a possible alternative, but it is impractical to re–implement a large volume of existing applications (i.e., libraries and device drivers) that are already written in C or C++, taking advantage of low–level programming features such as direct access to memory addresses. Another way to reduce memory corruption is to search for bugs beforehand, using static or dynamic bug–finding techniques [25–27] (i.e., fuzzing, symbolic execution, or concolic execution) to pinpoint the exact location of the cause of abnormal behavior in a program. Despite all the effort and time that has been expended, memory corruption is still prevalent. Thus, effective mitigation for undiscovered vulnerabilities is necessary.

1.1 Problem Statement and Approach

Early memory corruption attacks [2] predominately relied on injecting a specially crafted payload to exploit a vulnerability in a system's memory before diverting the original execution flow into the injected code. The $W \oplus X$ strategy constitutes a major step toward thwarting code injection into certain memory areas (i.e., the stack or heap) because it effectively blocks

the execution of injected malicious code. This feature, known as No-eXecute (NX) or data execution prevention (DEP) has become a standard defense mechanism used in modern operating systems, which has led to code reuse attacks gaining popularity.

Sophisticated code reuse attacks (i.e., return-oriented programming or ROP [22]) have emerged whereby adversaries are able to construct a functional payload by reusing existing code fragments instead of introducing external code. The main idea behind code reuse attacks is combining multiple sequences of legitimate code after hijacking the original control flow of a vulnerable process. Chaining a series of short instruction sequences (gadgets) enables the adversary to achieve arbitrary computation as a Turing-complete language, even in the presence of additional protection mechanisms such as control flow integrity (CFI) [28–35].

To obfuscate the location of code, address space layout randomization (ASLR) [36, 37] relocates the base addresses of main executables and shared libraries at each load to prevent the attacker from collecting useful gadgets. Yet, the current ASLR implementation in modern operating systems does not offer the level of entropy it is claimed to provide [38]. Moreover, incomplete ASLR coverage in a process memory leaves enough code mapped in static locations, allowing for the successful construction of functional ROP payloads [39–42]. Even worse, ASLR mitigation can be easily bypassed with a memory disclosure vulnerability under fully randomized address space, leveraging a single (disclosed) code pointer to reveal an entire code layout. This is because the relative distances between memory objects remain identical, allowing an attacker to pinpoint the exact location of the gadgets. Disclosure-aided ROP requires finer-grained code diversification to alter code structure, thus breaking the assumption that the attacker knows the location of the existing gadgets.

Recent adversarial advancement in code reuse attacks has seen the introduction of a *"Just-In-Time" return-oriented programming (JIT-ROP)* attack [43] that enables gadgets to be gathered and chained on the fly. Feature-rich applications, such as web browsers and PDF readers, support scripting languages (i.e., JavaScript, ActionScript), which enables JIT-ROP to be launched easily. JIT-ROP leverages information leaks (multiple pointers disclosed instead of a single pointer) to dynamically scan the code segments of a process, searching for useful gadgets to synthesize them into a functional

ROP payload at runtime. Such sophisticated attacks can defeat even fine-grained code randomization protections. In response to this new attack vector, defenders have introduced another mitigation primitive that removes readable permissions because a crucial requirement of a JIT-ROP exploit is the ability to read executable memory segments of the vulnerable process by exploiting a memory disclosure vulnerability. The enforcement of an "execute-no-read" policy [44–52] blocks the gadget discovery process efficiently. However, the executable–only memory scheme requires fine-grained code randomization as a prerequisite. Otherwise, an attacker can still harness useful gadgets from predictable locations.

1.1.1 Software Diversity

Software diversification is indisputably an important and effective defense against modern exploits and is a prevalent memory protection mechanism in the vast body of work in this area [53]. Surprisingly, however, despite decades of research [54, 55], only address space layout randomization (ASLR) [36, 37] (and lately, link-time coarse-grained code permutation in OpenBSD [56]) has actually seen widespread adoption. More comprehensive techniques, such as fine-grained code randomization [57–63], have mostly remained academic exercises for three main reasons: i) a lack of a transparent and streamlined model for delivering diversified binaries to end users, ii) the unaffordable cost and complexity of creating diversified variants, and iii) incompatibility with well-established software builds and other mechanisms that rely on software uniformity.

With regard to transparency, the vast majority of existing code diversification approaches rely on code recompilation [51, 58, 64, 65], static binary rewriting [52,60,61,63,66–68], or dynamic binary instrumentation [61,63,69,70] to generate randomized variants. The breadth of this spectrum of approaches stems from tradeoffs related to their applicability (source code is not always available), accuracy (code disassembly and control flow graph extraction are challenging for closed–source software), and performance (dynamic instrumentation incurs a high runtime overhead) [53]. It is worth noting that the possibility of generating unreliable mutations due to inaccurate disassembly or inadequate control flow graph reconstruction has also deferred the wide adoption of fine–grained code diversification.

In relation to deployment, *all* the above approaches share the same main drawback: the burden of diversification is placed on end users, who are responsible for carrying out a cumbersome process involving complex tools and operations (i.e., binary analysis, disassembly, or recompilation). The process inevitably requires a significant amount of computational resources and human expertise. An alternative would be to let software vendors carry out the diversification process by delivering pre-randomized executables to end users. Under this model, the availability of source code would make even the most fine-grained forms of code randomization easily applicable, and the distribution of software variants could be facilitated through existing "app stores" [71, 72]. Although it seems attractive due to its transparency, this deployment model is unlikely to be adopted in practice due to the increased cost that it would impose on vendors to generate and distribute the program variants across millions or even billions of users [53, 73].

Regarding compatibility, apart from the (orders-of-magnitude) higher computational cost for generating a new variant per user, the fact that software mirrors, content delivery networks, and other caching mechanisms involved in software delivery become useless is probably more challenging. Moreover, the mutated binaries become incompatible with the existing software build and maintenance processes including, but not limited to, code constructs, patching, crash reporting, whitelisting, regression testing, debugging, diagnostics, and security monitoring workflows.

Decades of research on code diversification, whether as a standalone defense or as a prerequisite for execute-only memory protections, has repeatedly shown its effectiveness against ROP exploits. From a practical point of view, however, the applicability of the existing techniques depends on the availability of source code [57–59, 74] or debug symbols [75, 76] or on the assumption of accurate code disassembly [60,61,77]. Unfortunately, precise disassembly with full coverage is a challenging proposition because i) the behavior of a program is undecidable at runtime and ii) distinguishing code from data is non–trivial.

1.1.2 Software Debloating

Another method of software protection employed against code reuse attacks is code slimming (debloating), which entails removing unneeded code [78–82]. Modern software development is greatly simplified by an abundance of freely

available frameworks, toolkits, and libraries. Shared libraries, in particular, are widely used due to their several benefits. They offer higher productivity by using ready–made third–party modules to carry out certain tasks, simple code maintenance, and bug fixes that do not require redistribution of the whole application. They also save space by avoiding multiple copies of the same code on disk and in memory. The downside of this flexibility, however, is that the whole library must be loaded even if just one of its functions is needed. This results in "code bloat" due to a large amount of code that is present but never exercised in memory.

From a security perspective, although a larger code base on its own may not be a significant drawback due to the ample resources of modern computing devices (except, perhaps, embedded systems and resource-constrained devices), the much larger attack surface is definitely not welcome. As the code base of a program grows, so does the likelihood of finding (exploitable) bugs. A larger code base also increases the odds of finding sufficient "gadgets" that can be strung together to mount ROP [22] or other types of code reuse attacks. The inclusion of more libraries also provides more ways to access private or security–sensitive data using rarely used or unneeded functionality that is still present.

The above observations have given rise to software debloating techniques that aim to reduce the attack surface by eliminating unused code. For most applications, the bulk of the code comes from shared libraries that are either bundled with the application or are provided by the operating system to expose system interfaces and services. Applications typically use only a fraction of the functions included in these general-purpose libraries, so an intuitive approach to reducing the attack surface of a process is to remove unneeded (i.e., non-imported) functions from all loaded libraries [78–80]. In some cases, a significant amount of code is relevant for certain features but is not needed by any other component and can thus be removed whenever the corresponding feature is disabled. Existing library customization approaches, however, cannot remove that code because of the presence of control flow paths to it from other parts of the program (i.e., the configuration parser).

1.1.3 Practical Software Specialization

In this research, we focus on *practical software specialization against code reuse attacks*. We question why almost no software vendors or operating systems have considered adopting hardening techniques, such as fine–grained code diversification or code debloating. To date, the only operating system in which such an effort has been made is OpenBSD [56], which has lately supported a unique kernel by re-linking object files with randomly sized gapping and randomly permuted symbols at either install or boot time.

First, we present an instruction displacement technique for practical code diversification that is applicable with partial code disassembly coverage. Next, we propose a compiler—rewriter code diversification model. We take a generic code transformation approach that depends on compiler—rewriter cooperation. Our hybrid solution meets all key requirements for practical deployment: transparency, reliability, compatibility, and cost. We also introduce code inference attacks to undermine state-of-the-art mitigations (i.e., execute-only memory and destructive code reads), suggesting another mitigation technique even under such an advanced adversarial environment. Lastly, we present a practical software debloating technique that relies on the configuration of an application to identify code that is not needed at runtime. The insight behind our approach stems from the fact that among the multitude of configuration directives provided by feature-rich applications, some of them are rarely used and are disabled by default.

1.2 Thesis Statement

This dissertation argues that compiler-–rewriter cooperation is an effective and practical approach for achieving transparent endpoint–side software specialization as a countermeasure against code reuse attacks. Extracting supplementary information from the compilation toolchain, and embedding it in the form of metadata in the binary executable, enables generic, reliable, and rapid fine–grained code transformation (such as code randomization or code debloating) on endpoints.

1.3 Contributions

The main contributions of this dissertation are as follows:

Instruction Displacement: We have introduced *instruction displacement*, a practical code diversification technique for stripped binary executables, which is applicable even with partial code disassembly coverage. We have designed and implemented a prototype of the proposed technique for Windows binaries. We have also experimentally evaluated our prototype implementation and demonstrated that it can reduce the number of remnant gadgets from the existing in-place code randomization technique [62]. We have shown that instruction displacement incurs a negligible runtime overhead for the SPEC CPU2006 benchmark programs.

Code Inference Attacks and Defenses: We have explored *code inference attacks* by implicitly inferring code without direct/indirect information leaks, which undermines state-of-the-art defense mitigations even when both fine-grained code transformation (i.e., in-place code randomization [62]) and destructive code reads are present. We have tackled this new threat by proposing a *practical mitigation against implicit code disclosure*.

Identifying a Set of Essential Information for Code Randomization: We have identified a *minimal set of crucial information for code transformation* to facilitate rapid fine-grained code randomization at the basic block level at installation or load time. We have designed an efficient data structure to serialize the information as metadata, which can be embedded into executable binaries to further help reconstruct code layout for code transformation.

Compiler–rewriter Code Transformation Model: We have proposed *Compiler–assisted code randomization (CCR)*, a practical and generic code transformation approach that relies on compiler—rewriter cooperation. It is a hybrid solution that combines producer–side and consumer–side efforts to satisfy the key requirements for successful deployment of code randomization, transparency, reliability, compatibility, and cost, filling a void in prior code randomization research. This approach enables robust and fast diversification

of binary executables on end-user systems, which does not require recompilation, disassembly, or binary analysis to produce mutations, thus preserving legacy software distribution channels.

CCR Prototype Implementation and Evaluation: We have designed and implemented a prototype of CCR by extending the LLVM/Clang compiler and the GNU `gold` linker to generate augmented binaries in Linux, and developing a binary rewriter that leverages the embedded metadata to generate hardened variants. Our prototype supports existing features of code constructs including (but not limited to) position independent code, shared objects, exception handling, inline assembly, lazy binding, link-time optimization, and even control flow integrity. We have experimentally evaluated our prototype and demonstrated its practicality and feasibility, resulting in a modest average file–size increase and a negligible average runtime overhead.

Configuration–driven Attack Surface Reduction: We have presented a novel approach, *configuration–driven code debloating*, that eliminates feature-specific shared libraries that are only needed when certain configuration directives are specified by the user and which are typically disabled by default. Our experimental evaluation demonstrates that this semi-automated approach can remove up to 77% of the code based on a default configuration. The technique can also be combined with other code debloating approaches, such as library customization [79].

Open–sourced Prototype Implementation: We have made our prototype implementation for both instruction displacement and compiler–assisted code randomization approaches publicly available in a `GitHub` repository so that our work can be leveraged to build a myriad of other interesting projects that require reliable and robust binary instrumentation.

- `https://github.com/kevinkoo001/ropf`
- `https://github.com/kevinkoo001/CCR`

1.4 Outline

The rest of this dissertation is structured as follows.

Chapter 2 surveys comprehensive background information on the evolution of code reuse attacks and defenses, followed by a variety of code transformation techniques focusing on static binary instrumentation. We also outline related work on attack surface reduction through the elimination of unneeded code.

Next, Chapter 3 demonstrates a practical code diversification technique, instruction displacement, which is applicable with incomplete code disassembly coverage for stripped binary executables. In Chapter 4, we introduce implicit code disclosure, which can significantly undermine even a state-of-the-art mitigation of destructive code reads, and propose a practical defense against this attack. In Chapter 5, we suggest a novel model for software diversity, compiler–assisted code randomization, which relies on compiler–rewriter cooperation to allow rapid diversification with full accuracy on the client–sides. Chapter 6 presents a software debloating technique based on configuration directives to reduce attack surface.

Finally, chapter 7 concludes this dissertation with suggestions for future research directions.

1.5 Publications

Parts of this dissertation have been published in the following international conference or workshop proceedings.

- Configuration-Driven Software Debloating, **Hyungjoon Koo**, Seyedhamed Ghavamnia, and Michalis Polychronakis. *In the 12th European Workshop on Systems Security (EuroSec)*, 2019
- Compiler-assisted Code Randomization, **Hyungjoon Koo**, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. *In the 39th IEEE Symposium on Security & Privacy (S&P)*, 2018
- Defeating Zombie Gadgets by Re-randomizing Code Upon Disclosure, Micah Morton, **Hyungjoon Koo**, Forrest Li, Kevin Z. Snow, Michalis Polychronakis, and Fabian Monrose. *In the 9th International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2017

- Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code-Inference Attacks, Kevin Z. Snow, Roman Rogowski, Jan Werner, **Hyungjoon Koo**, Fabian Monrose, and Michalis Polychronakis. *In the 37th IEEE Symposium on Security & Privacy (S&P)*, 2016

- Juggling the Gadgets: Binary-level Code Randomization using Instruction Displacement, **Hyungjoon Koo** and Michalis Polychronakis. *In the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016

I participated in the following publications during my Ph.D. study. They are relevant to traffic differentiation and internet censorship but are not part of this dissertation.

- The Politics of Routing: Investigating the Relationship between AS Connectivity and Internet Freedom, Rachee Singh, **Hyungjoon Koo**, Najmehalsadat Miramirkhani, Fahimeh Mirhaj, Leman Akoglu, and Phillipa Gill. *In the 6th USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2016

- Identifying Traffic Differentiation in Mobile Networks, Arash Molavi Kakhki, Abbas Razaghpanah, Anke Li, **Hyungjoon Koo**, Rajeshkumar Golani, David Choffnes, Phillipa Gill, and Alan Mislove. *In the 15th ACM Internet Measurement Conference (IMC)*, 2015

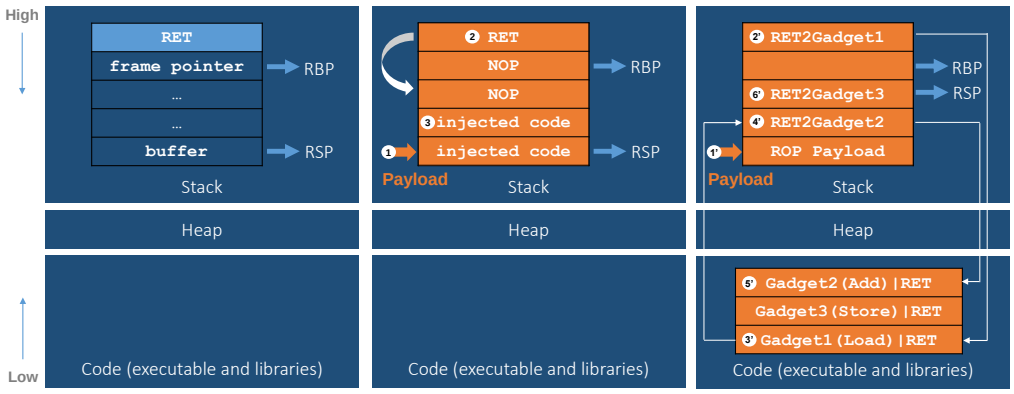


Figure 2. A comparison between the operation of code injection attacks (right) and that of code reuse attacks (middle). Both attacks initially take advantage of control flow hijacking. The main difference is that return–oriented programming (ROP) reuses existing code for an exploit payload without introducing new code to evade non-executable memory.

2 Background and Related Work

Figure 2 depicts how code reuse attacks (right) differ from code injection attacks (middle) that exploit a stack overflow vulnerability. The image on the left shows a process memory when a function call has been invoked. Suppose that return addresses (RET) ② and ②' have been hijacked by the crafted payload ① and ①' via user input, `buffer`, on the stack respectively. A classic code injection attack manipulates the return address so that it points to injected code such as shellcode ③. However, ROP redirects the hijacked return address ②', containing the next instruction to be executed, to the first gadget ③', which has been carefully selected by the attacker. Once the Gadget1 (i.e., LOAD a value to a register) is executed and returned, a current RSP (stack pointer) moves from ①' to ④' to point to the next instruction ⑤'. Likewise, when the next Gadget2 (i.e., ADD instruction) is returned, the stack pointer points to ⑥'. The process of controlling the stack is called *stack pivoting*. The attacker must carefully chain multiple gadgets (each gadget consists of a sequence of instructions) as well as perform appropriate stack pivoting for an exploit to be successful.

2.1 Arms Race: Code Reuse Attacks and Defenses

This section elaborates on how code reuse attacks and defenses have been evolving to defeat against each other. It is of importance to evaluate the effectiveness of each mitigation, necessitating a combination of defenses against cutting–edge attacks.

2.1.1 Code Reuse Attacks and Address Space Layout Randomization

The Emergence of Return-Oriented Programming (ROP): The $W \oplus X$ memory protection scheme was defeated using a novel exploitation technique named *return-into-libc* (`ret2libc`) [20] This technique transfers a control flow into an existing function defined in a `libc` library. Although it allows an adversary to jump into a set of useful functions, arbitrary code execution is limited. Krahmer [83] introduced the concept of the borrowed-code-chucks exploitation for the first time in 2005. The main idea behind it is to chain necessary code chucks together by controlling a stack, which forms a basis for code reuse attacks. In particular, he introduced a series of instruction sequences – gadgets – as a building block for a functional exploit payload. Two years later, in his seminal paper, Shacham presented *return-oriented programming (ROP)*, which generalizes a `ret2libc` attack [22]. He showed that ROP is very powerful because i) a set of gadgets can be generated as a Turing-complete language via a selection process and ii) there is a plethora of such useful gadgets in user applications. For example, it was discovered that one in every 178 bytes in the `libc` code segment was `0xc3`, which indicates "ret" in `x86`. In other words, gadgets are plentiful since the x86 instruction set is extremely dense and unaligned.

Various Types of Code Reuse Attacks: When the attack paradigm has shifted from code injection to code reuse attacks, researchers demonstrated that ROP poses a severe threat in various platforms and operating systems including SPARC, Mac OS X, and embedded systems [84–86]. Moreover, it turns out to be feasible to mount ROP attacks with other types of control flow instructions (i.e., jumps or calls) instead of a return instruction [30, 87–89]. *Jump-oriented programming (JOP)* [88] removes the reliance on stack pivoting and ret instructions for gadget discovery without losing expressiveness. Rather,

it merely depends on a so-called dispatcher gadget, a sequence of indirect jump instructions in a dispatcher table for building a functional payload. Checkoway et al. [87] leverage instruction sequences that behave like a return to yield Turing-complete functionality. Carlini and Wagner [30] first proposed an idea using call-ending gadgets, dubbed call-oriented programming (COP), and later Sadeghi et al. [89] demonstrated the feasibility of pure COP based solely on call gadgets. A number of automated tools for discovering and chaining gadgets have also been developed [90–94].

Address Space Layout Randomization against Code Reuse Attacks: Because the core element of code reuse attacks is the power to predict address space and divert control flow, ROP mitigation focuses on two main approaches: i) invalidating an attacker's knowledge of the layout through code rearrangement or code transformation and ii) restricting the use of the instructions for control flow against control hijacking. With regard to the first approach, even before code reuse attacks became prevalent, PaX Team [36] designed address space layout randomization (ASLR) to protect against buffer overflow attacks in 2001. ASLR is an efficient protection scheme with low overhead that assigns the base addresses of code segments to different locations at each load, rendering memory locations unpredictable. Beginning with Linux (since 2001 as a kernel patch and since 2005 in kernel 2.6.12) and OpenBSD (since 2003), an ASLR feature has been gradually integrated into most popular operating systems by default, including Microsoft Windows (since Windows Vista in 2007) [37], Mac OS X (since 10.5 Leopard in 2007), and mobile systems such as Apple iOS (since iOS 4.3 in 2011) and Google Android (since Android 4.0 Ice Cream Sandwich in 2011). We will revisit the second approach, control flow integrity (CFI), in a separate Section 2.1.3.

ASLR Effectiveness and Weaknesses: Although ASLR hinders the types of attacks that require precise address prediction, researchers [95, 96] have demonstrated that the use of ASLR on a 32-bit architecture does not provide a high degree of entropy due to the limited number of bits available for address randomization. Findings have shown that ASLR just slows down attack time to compromise a victim, failing to prevent the attack itself. Ganz et al. [38] measured the robustness of the randomness in different Linux systems. Only a 64-bit Debian distribution has the 28 bits it is claimed to have; no other Linux system (i.e., 32-bit Debian, 32-bit or 64-bit OpenBSD or

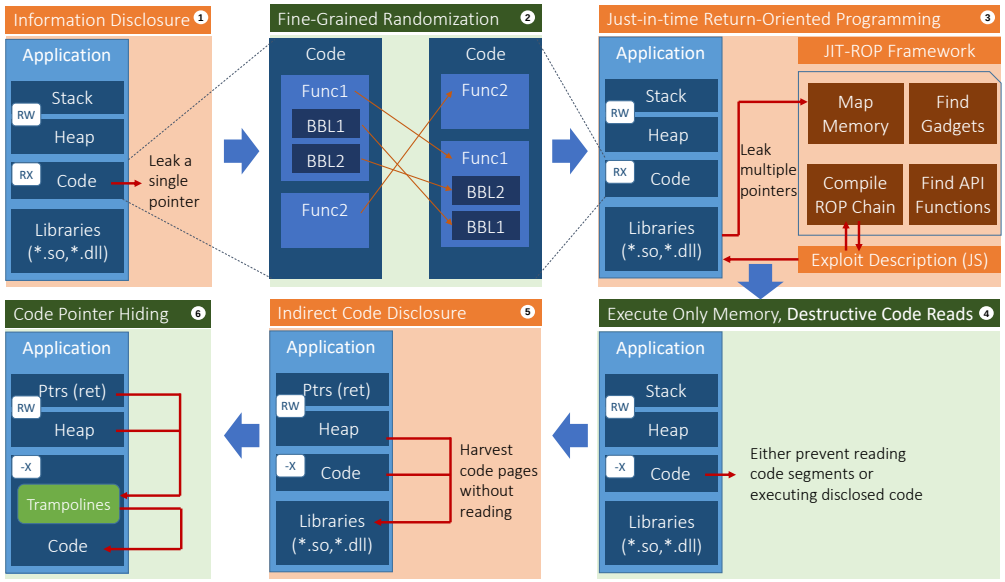


Figure 3. Summary of the arms race between code reuse attacks and defenses. The square areas in orange (①, ③ and ⑤) represent attackers' tactics to defeat existing defense mechanisms, whereas the areas in green (②, ④ and ⑥) represent defenders' strategies to mitigate newly introduced threats. Following the blue arrows (from ① to ⑥), each side has been cumulatively equipped with previous tools. For example, A JIT ROP attack (③) can be performed on top of a previous mitigation (②). The initial setting assumes that the attackers have a control flow hijacking vulnerability and the defenders have both non-executable memory and ASLR in place.

HardenedBSD) has sufficient randomness. In the case of 32-bit HardenedBSD, the measured randomness was just 8 bits, far from the 14 bits it is claimed to have. In general, brute force [95, 96] defeats ASLR. However, the difficulty obviously increases as more entropy becomes available, as on a 64-bit architecture. Besides the issue of low entropy [38, 95, 96], incomplete ASLR coverage [39–42] has weakened its effectiveness.

2.1.2 Disclosure-Aided ROP and Fine-grained Code Transformation

Memory disclosure vulnerabilities have been the target of a new class of attacks that thwart even the combination of $W \oplus X$ and ASLR protection

schemes [43, 97–102] currently available in all major operating systems. These bugs enable an adversary to leak the base address of loaded modules [43, 97–102]. Furthermore, a single code-pointer leak can help an attacker to compute other code locations because coarse-grained randomization such as ASLR keeps the relative distances between objects (i.e., functions) the same in memory. In Figure 3, the top-left box ① illustrates the fact that a single pointer leak is sufficient to learn the entire code layout. The pointer can be easily found from import/export tables, global offset tables (GOT), stack frames, virtual tables, or even exception handling information. Finer-grained randomization approaches have been proposed to diversify software into a different granularity (i.e., function, basic block, or page) to limit the effectiveness of memory leak as illustrated at the top-middle box ② in Figure 3. Section 2.2 deals with software diversity in detail.

2.1.3 Control Flow Integrity and its Effectiveness

Another approach to undermining code reuse attacks is *control flow integrity (CFI)* [69, 77, 103–119], which enforces the control flow graph (CFG) to perform as intended. The property, which has been formalized by Abadi et al. [103], seeks to ensure that any target address must follow a valid path in the original CFG at each indirect instruction (i.e., `jmp`, `call` and `ret`). This is because ROP often results in an unintended flow as it takes advantage of indirect branches for arbitrary computation. CFI checks can be made either statically [77, 104] or dynamically [105, 112–119]. In order to avoid performance degradation of such checks, state-of-the-art fine-grained CFI takes advantage of modern hardware features [1] such as branch tracing stores (BTS) and last branch records (LBR) [105, 118, 119].

Static CFI: Static CFI checking confines execution flow within the boundary of allowed control paths, focusing on statically determining valid targets. Mingwei et al. [104] have presented a novel way to apply CFI to stripped binaries without source code, compiler support, debugging information, or the presence of relocation. They developed robust techniques for disassembly, static analysis, and transformation of a given binary, which can work against

[1] Modern CPU supports a built-in performance monitoring unit (PMU) for performance parameter measurement (i.e., instruction cycles, cache hits, cache misses, etc.), which includes BTS, LBR, event filtering, and conditional counting.

control flow hijacking even for complex COTS products. Zhang et al. [77] have proposed a practical CFI with binary rewriting, named CCFIR (compact control flow integrity and randomization). CCFIR collects all legitimate indirect control flow instructions and limits their destinations to known locations, compiled as a white list, which facilitates random shuffle while protecting against the possibility of a hijacked flow.

Dynamic CFI: Dynamic CFI checking monitors program execution at runtime. Monitoring "return" instructions, DROP [112] checks whether returning addresses are in the range of `libc` to detect the building of malicious code using ROP. Davi et al. [113] have proposed DynIMA for dynamic CFI checks with the help of a trusted computing mechanism, which verifies the integrity of executables. However, it suffers from a high performance overhead because dynamic taint analysis requires any untrusted data to be marked as tainted and then propagated. Bletsch et al. have suggested the concept of control flow locking [114], which is like mutex and asserts the correctness of the original CFG with a lock code insertion. ROPdefender [115] is another ROP detection tool that uses dynamic binary instrumentation. It maintains a shadow stack that is updated using `call` and `ret` pairs. If the pair does not match the addresses on the shadow stack, further execution is suspended. However, ROPdefender cannot detect any gadget ending with indirect jump or call instructions, and it imposes high overheads (2x on average). ROPGuard [116] works on the premise that critical API functions are often invoked to launch successful code reuse attacks. During runtime it verifies that: i) a return address is executable, ii) the instruction at the return address is preceded by a call instruction, and iii) the call instruction goes back to the current function. [2] Kayaalp et al. [117] have proposed branch regulation that enforces control flow rules at the function level and only checks unintended branches instead of the whole CFG construction.

Hardware-assisted Dynamic CFI: Hardware features can reduce performance overheads for full CFI implementation. CFIMon [105] leverages the BTS area to analyze runtime traces on the fly. In essence, it collects legitimate control transfers to detect CFI violation. kBouncer [118] and ROPecker [119] leverage LBR to keep track of branch history in order to

[2]The idea has been incorporated with Microsoft EMET tool [120]

monitor branches. Both approaches introduce two heuristic thresholds, gadget chain and gadget size (20 and 8, 6 and 11 respectively), to suspend ROP attacks. A shorter gadget chain or a longer gadget size makes it harder for an adversary to construct an exploitable payload. Thus, ROPecker has stricter CFI policy than kBouncer. However, its reliance on the very limited size of the LBR stack (holding only 16 records) has again been defeated by relaxed assumptions [29,30,33–35], where many different evasion techniques and niche attack vectors (i.e., NOP operation gadgets, long gadgets, and call-preceded gadgets) nullify even hardware-assisted CFI defense mechanisms.

2.1.4 Just-In-Time ROP Attacks and Mitigations

Snow et al. [43] have demonstrated another level of code reuse attacks, JIT-ROP, that leverage JIT (Just-In-Time) engines to an exploit, which is available in the applications that understand expressive scripts such as JavaScript in a web browser or ActionScript in an Adobe reader. As shown in the top-right illustration 3 in Figure 3, the main processes are: i) leaking multiple pointers, ii) scanning all mapped memory pages through all leaked pointers, iii) disassembling the discovered code and finding useful gadgets for an exploit, and iv) compiling an ROP payload on the fly. As JIT-ROP attacks construct a dynamic ROP payload, randomization schemes render them ineffective because the attacker does not rely on the knowledge of code layout any longer.

The emergence of JIT-ROP attacks [43] has led to the development of execute-only memory protections [44–48, 50–52]. A prerequisite for these techniques is that the protected code must have been previously diversified using fine-grained randomization. Backes et al. [45] first introduced a basic approach for removing the root cause of memory disclosure exploits and JIT-ROP attacks in general. A new primitive called *Execute-no-Read (XnR)* ensures that code cannot be read as data but can only be executable (bottom-right box ④ in Figure 3). Heisenbyte [49] introduces destructive code reads to prevent attackers from executing disclosed code rather than from reading code segments.

Although execute-only memory prevents code discovery, adversaries can still harvest code pointers from (readable) data sections and indirectly infer the location of code fragments [121–123] or, in some implementations [44, 49],

achieve the same by partially reading or reloading pieces of code [124] (bottom-middle box ⑤ in Figure 3). As a response, leakage-resilient diversification [48, 125] combines execute-only memory with code-pointer hiding using additional control flow indirection (bottom-left box ⑥ in Figure 3).

Shuffler [68] has introduced a re-randomization scheme that re-randomizes function locations at runtime on the order of milliseconds. It does so asynchronously in a separate thread to ensure that virtual addresses are only valid for a short time period. RuntimeASLR [126] also re-randomizes the address space of every child process to prevent clone-probing attacks. Similarly, TASR [127] adopts compiler-based re-randomization, but it is not able to protect data pointers.

2.1.5 Other Code Reuse Attacks Oblivious to Gadget Locations

Even when the harvest of indirected pointers is prevented from revealing anything useful about the immediate surrounding code area of their targets, attackers may still be able to reuse whole functions, e.g., using harvested pointers to other functions of the same or lower arity [32, 128, 129]. For example, an address-oblivious code reuse attack [128] bypasses leakage-resilience defenses by profiling and reusing protected code pointers without having direct knowledge of the code layout.

Bittau et al. [130] first demonstrated that it is possible to remotely discover ROP gadgets on the fly without knowing their locations beforehand. It leverages two properties to probe the gadgets: i) a child process inherits the certain state (i.e., memory layout) from a parent process and ii) a server daemon re-launches the child process when it is crashed.

2.2 Software Diversity via Static Binary Instrumentation

In this section, we survey related research on software diversity. In general, code randomization (complementary to ASLR) aims to diversify even further the layout and structure of a process's code, which involves a binary instrumentation (rewriting) process. Here we mostly focus on static binary instrumentation.

2.2.1 Software Diversity in the Security Field

Software diversity has been studied for decades in the context of security and reliability. Early works on software diversification focused on building fault-tolerant software for reliability purposes [131, 132]. Changing the location of code can also improve performance, especially when guided by dynamic profiling [65, 75, 133, 134]. In the security field, software diversification has received attention as a means of breaking software monocultures and mitigating mass exploitation, the concept of which has been the basis for a wide range of software protections against code reuse attacks [53–55].

2.2.2 Various Types of Code Randomization

Code randomization often involves complex binary code analysis, which brings significant challenges when it comes to accuracy and coverage, especially when supplemental information (e.g., relocation or symbolic information) is not available [135–140]. Static binary rewriting of stripped binaries is still possible in certain cases, but it involves either code-extraction heuristics [52, 60–63, 66, 69] or dynamic binary instrumentation [61, 63, 69, 70]. Other implementation approaches include compile-time [48,51,58,64,65], link-time [59], load-time [52, 60, 61, 63, 66, 141], and runtime [67, 68, 127, 142, 143] solutions. Note that all the above approaches assume that a diversification process is performed on the client side. By contrast, the concept of server–side diversification has been only briefly explored, most notably as part of "app store" software distribution models [71, 72].

From a deployment perspective, most of the techniques that fully randomize all code segments depend on the availability of source code [57–59, 74, 121] or debug symbols [75, 76], the use of heavyweight dynamic binary instrumentation [61, 121], or the assumption of accurate code disassembly [60, 77, 144]. In contrast, in-place code randomization [62], can be applied on stripped binaries even with partial disassembly coverage. Randomization granularity varies from the function [57–59,74], memory page [144], basic block [60,75,76], to the instruction level [61–63], thus breaking attackers assumptions about the location and structure of gadgets based on the original code image.

Among the above, the techniques proposed by Bhatkar et al. [58] and Selfrando [141], both of which rely on a single compilation to generate self-randomizing binaries, are probably the closest in spirit to our compiler–

assisted randomization. Selfrando, for instance, uses a linker wrapper script to extract function boundary information from object files, which is kept in the resulting executable. However, these approaches are limited to function–level permutation, which is not enough to thwart exploits that depend on code-pointer leakage to infer the location of gadgets within functions [39, 123, 125, 145, 146].

2.2.3 Limitations of Existing Approaches

Code Randomization with Re-compilation: Bhatkar et al. [57,58] have attempted to apply an address obfuscation technique when source code is available. This technique involves i) randomizing the base addresses of the stack, heap, shared objects, and code segments, ii) permuting the order of variables and routines, and iii) introducing random gaps between objects. Another range of compile-time approaches prevent the construction of an ROP payload by generating machine code that does not contain unintended gadgets and that safeguards any remaining intended gadgets using additional indirection [147, 148]. In particular, G-Free [147] eliminates all unaligned free-branch instructions to protect the other aligned ones. It avoids `ret` instructions and other types of opcodes that can be used as gadgets. However, the main drawback of this approach is that it not only requires source code but also has to re–compile a program to produce a new variant, rendering code randomization impractical.

Code Randomization with Binary Analysis: Binary analysis is essential for rewriting code that is semantically equivalent to the original code when source code or additional information is unavailable.

XIFER [63] suffers from the problem of imprecision because it relies on the accuracy of disassembly and of building a control flow graph. This is evident in the results of Andriesse et al. [139, 149] and in the multitude of heuristics employed by Shuffler [68] because fully accurate disassembly and CFG extraction for complex C/C++ binaries (even when relocation information is available) is not feasible.

Chongkyung et al. have proposed address space layout permutation (ASLP) [59], a binary rewriting tool that places the static code and data segments in arbitrary locations and performs function-level permutation up

to 29 bits of randomness. ASLP requires relocation information in a binary or source code for a re-compilation and re-linking process.

Hiser et al. have suggested another approach called instruction location randomization (ILR) [61] to invalidate prior knowledge of gadget locations at the instruction level. In a nutshell, it statically randomizes instruction addresses coupled with dynamic control flow in a virtual machine, producing a fall-through map for reassembling code fragments. A disassembly engine analyzes indirect branch targets and call sites using pre-defined rules (i.e., instruction orders).

Pappas et al. [62] have proposed in-place code randomization (IPR) that works with a partial disassembly, introducing four different transformation techniques of different spatial granularity (instruction, basic block, and whole function): i) instruction substitution, ii) intra basic block instruction reordering, iii) instruction reordering with register preservation and iv) register reassignment.

Wartell et al. [60] have proposed binary stirring, a high-level architecture that includes static rewriting and stirring at load time. It transforms legacy `x86` application binaries into self-randomizing instruction addresses without source code or debug information and statically randomizes basic blocks in each invocation at load time.

2.3 Attack Surface Reduction

Another technique to mitigate against code reuse attacks is to reduce the attack surface. In general, eliminating unneeded functionality or unused code offers the benefits of i) reducing exploitable bugs from vulnerable processes, ii) lowering the number of potential gadgets available in legitimate code, iii) eliminating potentially harmful functionality or sensitive data access, and iv) making other mitigation techniques (i.e., CFI) simpler.

Earlier works have explored various other debloating approaches applied at different levels, including library customization [78–80], function argument specialization [150], feature-driven customization [81, 82], and kernel customization [151–153]. Other software debloating approaches specialize code for specific languages or environments, including Java [154–156], mobile systems [157,158], containers [159], or even network protocols [160,161]. Most of the above approaches follow one of two main strategies for identifying the

code to be removed: i) deterministically identifying code that is guaranteed to be unneeded, e.g., through static code analysis, or ii) profiling the application using representative workloads, and keeping only the exercised code. This section provides a brief overview of existing code debloating studies.

2.3.1 Library Customization

One of the earliest library specialization approaches to defending against exploitation was presented by Mulliner and Neugschwandtner [80]. Their code stripping and image freezing techniques, which operate on closed-source binaries, identify and remove all non-imported functions at load time and then "freeze" the remaining code by modifying certain memory allocation routines to prevent the loading or injection of additional code.

Quach et al. [79] have proposed Piece–Wise compilation, which leverages a modified compiler and loader to perform shared library specialization. Information regarding call dependencies and function boundaries is embedded as metadata into the binary at compilation time. The loader then takes two steps to invalidate unneeded code: i) if the whole code of a memory page can be removed, then the page is set as non-executable; ii) if only a subset of the code in a page needs to be removed, then the loader overwrites the unneeded functions with invalid opcodes.

Song et al. [78] have demonstrated the potential benefits of fine-grained library customization for statically linked libraries using data dependency analysis. Shredder [150] has aimed to further specialize any remaining functions after applying one of the above library specialization approaches. This is achieved by restricting the scope of critical system API functions and allowing only the subset of argument values that are needed by the benign code.

2.3.2 Feature–oriented Software Customization

Feature–oriented software specialization aims to remove unused functionality across the whole program, depending on its intended use. DamGate [81] uses both static and dynamic analysis to construct a call graph according to a set of seed functions that are given as input and pinpoint the required code. TRIMMER [82] relies on user–defined configuration data to remove unneeded feature–related code. The code is identified using inter-procedural analysis

based on the entry points specified in the initial configuration. CHISEL [162] has proposed a reinforcement-learning-based approach that allows a developer to generate a reduced version of a program based on a set of example runs using the desired options.

2.3.3 Kernel Debloating

Another line of code debloating research focuses on the kernel. FACE-CHANGE [151] generates a customized kernel view for each application to reduce the exposed kernel code based on profiling. It facilitates dynamic switching at runtime by identifying a process context. Kurmus et al. [152] have presented an automated approach for generating kernel configurations adapted to particular workloads, which can be used to compile a specialized kernel tailored for a given use case. The proposed model facilitates quantitative measurement of an attack surface using call graphs.

2.3.4 Other Types of Code Specialization

Other types of code specialization focus on different languages [154–156], environments [157–159], or protocols [160, 161]. Jred [154] removes unused methods and libraries based on static analysis of Java code. It lifts Java bytecode into `Soot` intermediate representation to produce a light-weighted transformed version. Similarly, Bhattacharya et al. [155] have introduced a technique to detect bloated sources in Java applications. Jiang et al. [156] have presented a technique for Java bytecode customization using static data flow analysis and program slicing. RedDroid [157] eliminates unneeded methods and classes from Android, and Apple has introduced "app thinning" [158] to deliver optimized versions of apps for different devices.

Cimplifier [159] performs container debloating based on system call analysis to detect unnecessary resources on a running instance of a `Docker` image. David et al. [160] have observed that vulnerabilities related to protocol implementations often reside in code that is not frequently used. TOSS [161] is an approach for automated customization of client–server systems that removes code related to unneeded network protocols. It leverages tainting–guided symbolic execution and program tracing to identify desired functionalities.

3 Instruction Displacement

3.1 Motivation

The complexity of static binary code analysis when dealing with complex stripped executables poses challenges for code diversification protections. Being a provably undecidable problem [163], accurate code disassembly and complete control flow graph extraction is complicated due to intermixed code and data, jump tables, computed jumps, callback and exception handling routines, and other code intricacies. Although at the source code level (or when debug symbols are available) it is possible to perform extensive transformations that effectively randomize all available gadgets [57–59, 74, 121], at the binary level it is challenging to apply aggressive fine-grained code diversification, such as randomizing the location of functions or basic blocks.

Existing attempts to achieve this, such as Binary Stirring [60], rely on various heuristics to fully and precisely extract all code and code references, so that after randomization all appropriate points can be fixed appropriately. Unfortunately, however, although such approaches may work well for relatively simple executables, they do not scale for large and complex COTS software, such the vulnerable Windows browsers and document viewers that are being targeted in the wild. Indeed, Wartell et al. [60] evaluate Binary Stirring using only main executables (not dynamic libraries) taken from simple utility programs. Introducing a runtime component after static analysis [61], on the other hand, can allow for the randomization of arbitrarily complex programs, in the expense though of increased runtime overhead.

From a practical perspective, a different compromise can be made by accepting the imprecision of static code analysis, and developing binary-compatible code diversification techniques that can tolerate partial code extraction in the expense of the achieved randomization coverage. In-place code randomization (IPR) [62], for instance, uses four different narrow-scoped code transformations that probabilistically alter the functionality of (or eliminate completely) short instruction sequences that can be used as gadgets.

Specifically, *instruction substitution* replaces existing instructions with functionally-equivalent ones (of the same or smaller length), to alter any overlapping instructions that may be part of a gadget. *Basic block instruction reordering* changes the order of instructions within a basic block according to

an alternative, functionally equivalent instruction scheduling, again affecting any overlapping gadgets. *Register preservation code reordering* changes the order of the `push <reg>` and `pop <reg>` instructions that are often used at function prologues and epilogues, respectively, to alter the semantics of any useful "`pop; pop; ret;`" gadgets that are often found at function epilogues. Lastly, *register reassignment* swaps the register operands of instructions throughout overlapping live ranges, again with the goal to alter the semantics of any gadgets that involve those registers.

By not altering the location and size of basic blocks and functions, IPR diversifies only the accurately extracted parts of the code, enabling compatibility with third-party stripped binaries. The achieved *partial* code randomization, however, unavoidably leaves a fraction of gadgets completely unaffected by the applied randomization. Specifically, Pappas et al. [62] report that on average, 18% of the gadgets located in the extracted code regions remained unmodified. When also considering the executable regions that were left out due to incomplete disassembly coverage, this percentage increases to 23.1% of all gadgets in the binary. Although the authors demonstrate that two automated ROP payload construction tools did not manage to construct a functional ROP payload using solely the remaining 23.1% of the gadgets, as they admit, this does not preclude that an attacker could manually construct a robust payload using solely unmodifiable gadgets.

Furthermore, some of the randomized gadgets are affected only in a minimal and predictable way that may still allow for their use. For instance, an attacker could still use a reordered function epilogue gadget by initializing the register operands of all `pop` instructions in the gadget with the same value, and then reliably using any one of the initialized registers. Consequently, it is also desirable to increase the entropy of randomization, so that guessing or inferring the state of a randomized gadget becomes much harder.

In this work, we aim to improve both the *coverage* and *entropy* of binary-level code diversification, so that the percentage of any reliably usable (i.e., non-randomized) gadgets is reduced even further.

3.2 Threat Model

Code diversification techniques rely on the assumption that an attacker cannot read or leak a diversified instance of the protected code. Experience though

has shown that under certain conditions this is possible by reading [43], leaking [130], or inferring [164] the code of a vulnerable process. Although instruction displacement makes the gadgets "disappear" from their original locations, they are still available in some other random location. Consequently, as any code diversification technique, it cannot defend against JIT-ROP [43] and other code leakage attacks.

These can be tackled by recent execute-only memory protections [44, 45, 47–50], which operate under the assumption that protected code has been properly diversified. For binary-compatible approaches [44, 45, 49], instruction displacement can be crucial in ensuring that adequate randomization coverage has been achieved. When execute-only memory enforcement is implemented using the concept of "destructive reads" [44, 49], however, an attacker may be able to infer the structure of a randomized gadget by (destructively) reading a few preceding bytes [165]. As is also the case with previous in-place code transformations [62], in such a setting where an attacker can disclose arbitrary bytes of the randomized code, instruction displacement can be undermined. For instance, by (destructively) reading the bytes of the inserted jump instructions, a JIT-ROP exploit can pinpoint at runtime the actual address of the displaced gadgets and then use them as part of a dynamically constructed ROP payload. [165]

3.3 Instruction Displacement

The goal of instruction displacement is to randomize the locations of gadgets so that their starting addresses become unknown to an attacker. In contrast to in-place code randomization, which leaves the randomized instructions in their original locations, instruction displacement relocates sequences of instructions that contain gadgets from their original locations to a newly allocated code segment. Due to ASLR and additional random padding, the base address of this separate segment in the address space of a process is completely random, and thus the locations of all displaced gadgets become unpredictable.

In the rest of this section, we first provide an overview of the overall displacement approach and various constraints that must be satisfied, and then describe in detail the displacement strategy that we follow.

3.3.1 Overall Approach

Any code diversification approach must maintain the semantics of the original program. In addition, given our assumption that parts of the original code may not have been extracted or disassembled properly, an additional constraint that must be followed is that the location and size of any correctly identified basic blocks must not be altered.

Changing the location of a basic block requires adjusting *all* instructions in the rest of the code that transfer control to that basic block—including computed jumps—to point to the new location. In our case, given that a complete view of the control flow graph is not available, moving a basic block may break the semantics of the code, since any control transfer from non-extracted code to that basic block will become stale, as it will still point to the original location. Similarly, changing the size of a basic block, e.g., in order to add more instructions for diversification purposes, requires shifting any code that immediately follows the expanded basic block. In essence, this means that all basic blocks following the modified one must be moved, which again is not possible.

Given the above constraints, we observe that although we cannot change the boundaries of a basic block, we can still perform arbitrary modifications *within a basic block*, as long as the semantics of the code remain the same (e.g., as is the case with the intra basic block instruction reordering transformation of in-place code randomization [62]). Furthermore, although patching an arbitrary location of a binary executable is not possible, we can safely patch any location within a basic block (assuming there is enough space), as long as the basic block's boundaries have been properly identified.

Based on these two observations, instruction displacement uses code patching to selectively relocate (some of) the instructions of a basic block to a random location. The overall approach is illustrated in Figure 4. The upper part of the figure shows the original code of a basic block (rectangles represent variable-length instructions), and the lower part shows the modified version of the code, with some of its instructions displaced into a new code region. In this example, the basic block contains two ROP gadgets, G1 and G2, located at addresses `addr1` and `addr2`, respectively. The first (G1) is an *unintended* gadget that begins from the middle of an existing instruction and ends with a `ret` instruction (opcode 0xC3) located again within an existing instruction. The second (G2) is an *intended* gadget that ends with a `call`

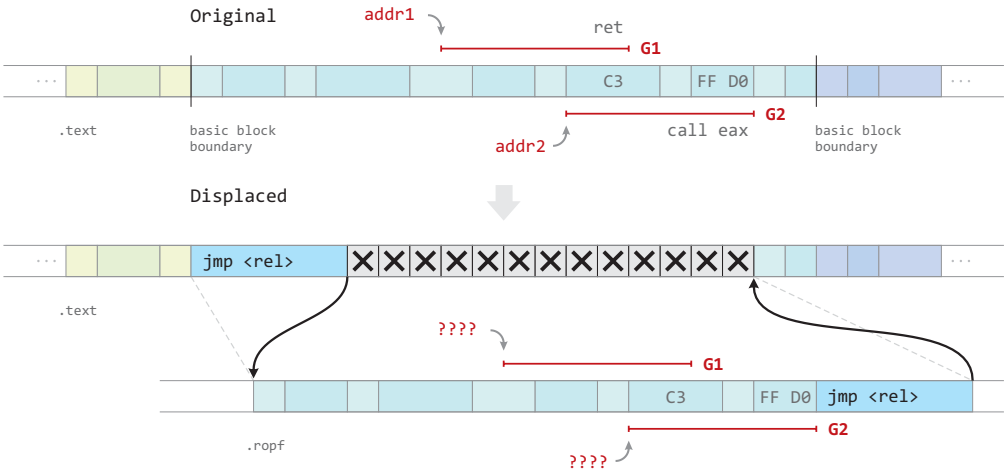


Figure 4. High-level view of instruction displacement. By moving part of the original basic block's code in a random location, the starting addresses of the two gadgets become unpredictable. To maintain the original semantics of the code, the displaced instructions are linked with the rest of the code using relative jumps.

`eax` instruction that is part of the program's code.

In the modified version of the code, the instructions at the beginning of the basic block have been overwritten by a relative `jmp` instruction that points to the overwritten instructions, which have now been copied into a random location, along with some of their following instructions. Note that the `jmp` instruction takes five bytes (one-byte opcode plus four bytes for its immediate operand), and thus instructions contained in basic blocks shorter than five bytes cannot be displaced in the general case. Although a smaller 2-byte relative `jmp` instruction could be used, its 8-bit displacement usually cannot "reach" far enough for transferring control to the area that contains the displaced instructions. For such cases, an alternative approach would be to insert a smaller trap instruction and achieve indirection through an appropriate handler routine. Unfortunately, the associated runtime overhead of such a solution would be prohibitively high. As we discuss in Section 3.5.3, the percentage of gadgets in such small basic blocks is very low, in the order of 0.83%, and thus we have chosen to ignore them.

Recall that a basic block is defined as a straight-line sequence of instructions with only one entry point and only one exit. Consequently, we can

safely patch with a `jmp` instruction *any* location within a basic block that corresponds to the address of an existing instruction. To preserve the semantics of the basic block's code, all that remains to be done is to transfer control back to the original location after the execution of the displaced instructions. This can be achieved again with a relative `jmp` placed right after the final displaced instruction.

By moving the instructions that contain the two gadgets in a randomly chosen location, an attacker cannot rely on them anymore based on their original addresses. The original code right after the patched location is overwritten with instructions that will crash the program or trap execution (e.g., privileged or interrupt instructions), and thus any transfer to the original locations of the gadgets (`addr1` and `addr2`) is ruled out. At the same time, the starting addresses of the displaced gadgets are now random, so an attacker cannot guess them (proper ASLR implementations, e.g., the one used in the latest versions of Windows, and additional random padding at the beginning of the segment that contains the displaced code fragments achieve enough entropy for that purpose).

3.3.2 Displacement Strategy

Although the address of a displaced gadget is not known to an attacker, the location of the inserted `jmp` can be easier to predict, and thus an attacker can still use it as the starting address for reaching a gadget. Depending on whether a gadget is intended or unintended, we can follow a different displacement strategy while trying to minimize the number of displaced instructions.

3.3.3 Intended Gadgets

Due to the inserted `jmp`, among all intended gadgets in a displaced code region, the one that (in the original code) begins with the patched instruction is still usable—the attacker can still rely on its original address, and the inserted `jmp` at that location will unconditionally transfer control to it. Depending on the location of the gadget within the basic block, however, this means that the attacker now must use a longer gadget, which is likely to have many more side effects in terms of register and memory state changes (a given indirect branch instruction is generally the "end" of several nested gadgets extending backwards from it). Although the use of longer-than-usual gadgets

is possible [30, 33–35, 166], it complicates significantly the construction of ROP payloads due to the additional side effects of the extra non-essential instructions.

To increase the complexity of any remaining usable gadgets, a displaced sequence of instructions begins as "far" as possible from any contained gadgets—in most cases, this means the beginning of the respective basic block. Given that it is desirable to minimize the number of displaced instructions, for very large basic blocks, we have set a limit of displacing up to 20 instructions from the end of a target gadget. In the example of Figure 22, the `jmp` is inserted at the beginning of the basic block, and the three instructions of gadget G2 can now be used only if seven additional instructions are executed before them.

Given that the percentage of the remaining usable gadgets by following the "entry point" of a displaced region is very low (0.6% in our experiments, as discussed in Section 3.5.3), we have chosen to not take any further action about them. We should note, however, that instruction displacement opens up more possibilities for randomizing or eliminating altogether the displaced gadgets. Indeed, once a sequence of instructions has been displaced, further transformations on the displaced gadgets can be applied. Fortunately though, in contrast to the general case, we can now fully disassemble the displaced instructions, and there is no space constraint due to previous or following basic blocks, as we have full control over the region where the displaced instructions are copied, and the placement of individual code fragments within that region. This means that we can apply more aggressive code transformations, beyond what is possible using in-place code randomization, such as splitting an existing instruction into two or more instructions. As an alternative example, we can apply transformations similar to the ones used by G-Free [147] to completely eliminate the displaced gadgets.

3.3.4 Unintended Gadgets

Unintended gadgets begin only from unaligned instructions, and may end with an either aligned or unaligned instruction (if the first instruction is an aligned one, then there is no way to "escape" from the intended instruction stream due to the unambiguous nature of instruction decoding). Consequently, the "predictable entry point" issue discussed above does not apply when a displaced

instruction sequence contains solely unintended gadgets—by following the inserted `jmp`, an attacker still cannot reach the unintended gadget (as is the case with gadget G1 in Figure 22). This makes the decision on which location to patch much simpler: it is enough to patch the intended instruction that contains the opcode byte of the first unintended instruction of the gadget. The location of that opcode byte in the displaced instruction will be random, and by following the `jmp` the attacker will be forced to execute the intended instruction stream, without being able to reach the unintended gadget.

Especially for unintended gadgets, this approach is quite effective even when a gadget spans two consecutive basic blocks. In such cases, although we cannot displace the whole gadget (due to our restriction in maintaining basic block boundaries intact), it is enough to displace even just the first instruction of the gadget to make the whole gadget unusable. This is possible when the first overlapping instruction is located towards the end of the first basic block, in which case it can be safely displaced.

In essence, instruction displacement enforces a coarse-grained control flow integrity constraint in a probabilistic and selective way. For intended gadgets, control flow is allowed to reach only the entry point of the basic block that contains a gadget (or, for very large basic blocks, the first of a sufficiently large number of instructions preceding the gadget). For unintended gadgets, control flow cannot "escape" from the intended instruction stream and reach any of the unaligned instructions of the gadget.

3.3.5 Combining Instruction Displacement with In-Place Code Randomization

Each displaced code region results in a slight increase in memory space and CPU overhead, due to the copied code, the extra indirection, and the disruption of code locality (although the latter some times has a positive impact, as discussed in Section 3.5.7). It is thus desirable to minimize the number of displaced regions whenever possible. Given that the end goal of the proposed technique is to improve the coverage and entropy achieved by existing code diversification techniques, we can combine instruction displacement with in-place code randomization, and apply the former only for gadgets that cannot be randomized by any of the existing code transformations of IPR (and optionally, also for gadgets that are randomized with insufficient entropy).

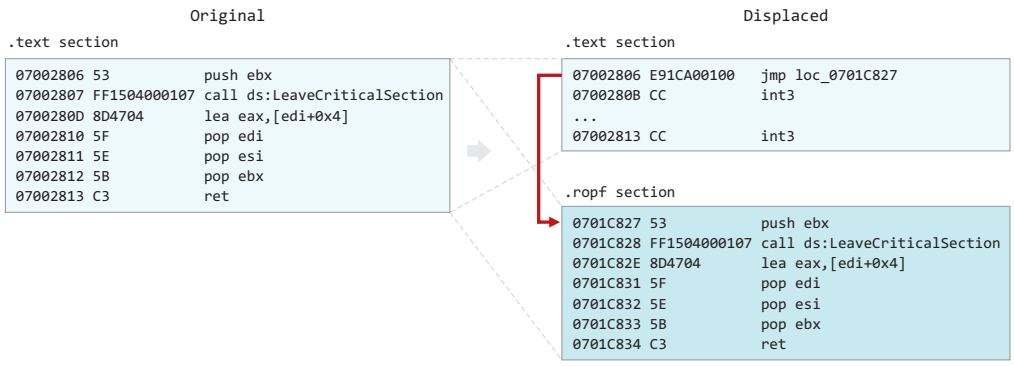


Figure 5. A real example of gadget displacement taken from Adobe Reader's BIB.dll module.

To that end, each binary is first analyzed to pinpoint all existing gadgets, and IPR is applied to randomize or eliminate as many gadgets as possible. Then, a second instruction displacement pass considers all remaining unmodifiable gadgets, and attempts to displace them whenever possible. In many cases, a basic block might contain several gadgets, some of which might be affected by IPR, and some not. To increase randomization coverage as much as possible, we follow a conservative approach and apply displacement even if only a single out of several gadgets within the same instruction sequence cannot be randomized by IPR.

3.3.6 Putting It All Together

We discuss a few remaining issues and optimizations by looking at a real example of applying instruction displacement. Figure 5 shows a basic block from Adobe Reader's BIB.dll that contains several (nested) gadgets ending with a `ret` instruction. In particular, "`pop; pop; ret;`" gadgets are quite useful in assembling ROP payloads, while the `call`-preceded gadget starting with the `lea` instruction can be used to bypass coarse-grained CFI protections [30, 33–35, 166]. After instruction displacement, the `push` instruction at address 0x7002806 has been replaced by a direct `jmp` to the displaced instructions, which now reside at a random location within a new `.ropf` section of the binary (detailed in the following section). All remaining original instructions are overwritten with `int3` instructions. The only option that is

now left for an attacker is to use the code of the whole basic block, starting with the `push` instruction. This might not be desirable, as it involves the execution of another function, which may have disastrous side effects. All other (intended and unintended) gadget starting locations within the basic block become unpredictable.

This example illustrates a common case in which the ending instruction of a gadget is also the final instruction of a basic block. We can exploit this fact to reduce the number of indirections needed due to instruction displacement. Depending on the type of branch at the end of a basic block, a `jmp` back to the original location may not be needed at all. As the most common case, all indirect branch instructions (i.e., those that can be the ending instructions of gadgets), will transfer control to the intended target no matter whether they have been displaced or not. In this example, the `ret` instruction will always transfer control to the return address that will be read from the stack, irrespectively of the actual location of the `ret`. Consequently, an extra `jmp` for transferring control back to the original location is not needed. The same is true for any unconditional branches, but care must be taken to adjust any relative displacement operands accordingly. Unfortunately, the same strategy cannot be applied for conditional branches, as we do not have control of the fall-through target.

Any other instructions that involve relative address operands must also be adjusted accordingly after the randomly chosen location of the displaced code region is picked. Besides relative `call` instructions and the like, this includes PC-relative memory accesses for 64-bit programs.

3.4 Implementation

To demonstrate the effectiveness of instruction displacement, we have developed a prototype implementation for Windows binaries. Our prototype supports 32-bit PE binaries (both main executables and dynamic link libraries), without relying on any debug or symbolic information (e.g., PDB files). To randomize a binary, a three-phase process is followed: i) identification of candidate gadgets for displacement, ii) modification of the PE executable to add a new code section for the displaced instructions, and iii) binary instrumentation for actually displacing the selected gadgets. In the following, we discuss these three phases in detail.

3.4.1 Gadget Identification

The first phase aims to identify the code regions that will be displaced. A necessary condition for any candidate region is to fall within the boundaries of a basic block, and thus a first necessary step is to extract the code and identify as many functions and basic blocks as possible. This is achieved using IDA Pro [167], a state-of-the-art code disassembler that achieves decent accuracy when dealing with regular (non-obfuscated) PE executables. IDA Pro leverages the relocation information present in Windows DLLs, and identifies compiler-specific code constructs and optimizations, such as basic block sharing [168]. We should note, however, that as in previous works [62], we do not take into account IDA Pro's speculative disassembly results, e.g., for embedded data and code regions that are reached only through computed jumps or which are part of signal handling routines. These rely on heuristics that are prone to errors, and thus we follow a conservative approach to prevent any correctness issues with the instrumented code due to falsely identified code regions.

Our code extraction module is based on the open-source implementation of in-place code randomization (IPR) [169], which we also use to pinpoint all remaining gadgets after the application of IPR. We have extended the implementation to consider gadgets comprising up to 15 instructions, from the just five instructions in the original implementation. We use IPR with maximum coverage settings, so as to reduce the number of displacements. An analysis pass then identifies all remaining unmodified gadgets and calculates the appropriate code regions to displace as many gadgets as possible. Gadgets contained in basic blocks smaller than five bytes are left intact, as they cannot be safely patched. Depending on the proximity of different gadgets (ending with different indirect branch instructions) within the same basic block, separate candidate regions are merged to minimize the required instrumentation in terms of additional `jmp` instructions. The final boundaries of each region are computed based on the strategy described in Section 3.3.2.

3.4.2 PE File Layout Modification

Once all to-be-displaced code regions have been identified, the PE file is augmented with a new code section, named `.ropf`, in which the displaced regions will be moved. The executable is modified using the `pefile` python

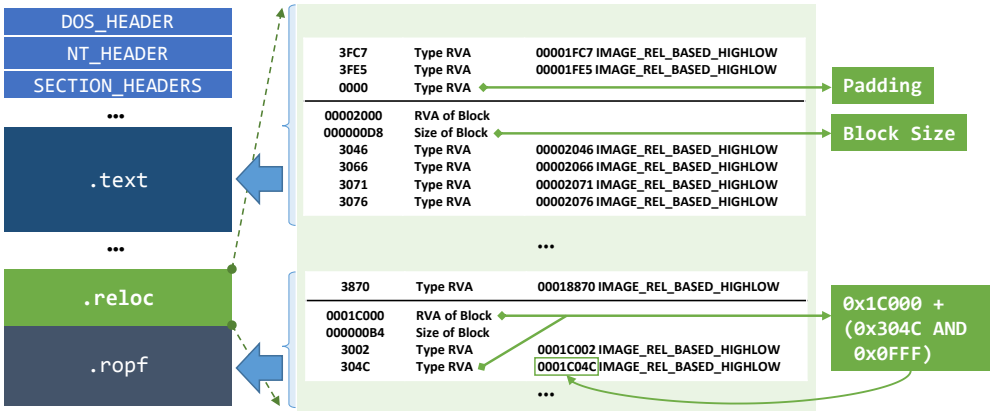


Figure 6. Rewriting the relocation section of a PE file for both the original (`.text`) and the new (`.ropf`) code sections.

library [170]. First, we define a new section header in accordance with the `IMAGE_SECTION_HEADER` structure, which is inserted into the section headers array, between the last existing header and the first data section. For simplicity, the new section is appended at the end of the file, so that the rest of the sections remain intact. Although more complex layouts could be studied to keep displaced instructions closer to their original locations and facilitate patching using two-byte `jmp` instructions (e.g., by identifying and reusing any unused regions within existing code segments), the resulting increase in coverage would still be minimal (due to the small percentage of less-than-5-byte basic blocks, as well as the limited reach of the 8-bit displacement), so the added complexity is not justified.

Besides the addition of the above entry, some existing information related to the overall PE image must be updated accordingly. Specifically, the following entries in the `IMAGE_OPTIONAL_HEADER` structure need to be updated: size of code and image, size of initialized data and uninitialized data, and the checksum of the binary. The size of the `.ropf` section is calculated based on the identified code regions, and by provisioning some extra room for the added `jmp` instructions for transferring control back to the original code, as well as some padding space.

3.4.3 Binary Instrumentation

With the `.ropf` section ready to host the displaced instructions, the actual patching of the original code and the copying of the displaced instructions can begin. The identified code regions are copied and placed in the `.ropf` section in a randomly chosen order (an additional small random gap can be added between successive regions if needed). As regions located within the same basic block or function of the original code end up in close proximity after displacement, this some times has a positive impact in terms of runtime overhead due to code locality, as discussed in Section 3.5.7. More sophisticated ordering schemes could also be explored, especially when taking into consideration hot spots and code locality, e.g., based on prior profiling information. To diversify the locations of gadgets even further, a large padding area of a randomly selected size is allocated at the beginning of the `.ropf` section.

For the code disassembly and reassembly operations needed to patch the original code locations, adjust the operands of displaced instructions, and insert additional `jmp` instructions at the end of displaced regions (whenever necessary), we use the Capstone framework [171]. We have also employed several optimizations using bit-level operations to speed up the instrumentation phase. Care must be taken while generating the `jmp` instructions for patching the original code so that any immediate operands do not result in accidental generation of new potentially useful gadgets (e.g., due to embedded 0xC3 bytes). This is avoided by adjusting the destination address of the displaced instructions by a few bytes in case an immediate contains an indirect branch opcode.

Finally, a final important step for ensuring the correct operation of the resulting binary is to update the PE file's relocation information for all affected code locations. To enable loading of modules at arbitrary addresses, PE files contain a `.reloc` section that contains a list of offsets (relative to each PE section), known as "fixups" [172, 173]. At load time, these entries specify the absolute code or data addresses within the module that must be adjusted according to the module's load address (which is usually randomly selected, due to ASLR).

As Figure 6 shows, the relocation table consists of a series of blocks grouped according to their relative virtual address (RVA). Each block begins with the RVA, the size of the block, the actual relocation entries, and some padding bytes for alignment. Each relocation entry consists of two bytes. The first

Applications		Gadget Distribution			Randomized Gadgets			Other
Name	Files	Total	Unintended	Unreachable	IPR	Disp.	Both	File Increase
Adobe Reader	50	677,689	55.24%	4.61%	82.16%	88.98%	96.69%	2.18%
MS Office 2013	18	195,774	55.04%	4.93%	83.02%	88.71%	97.25%	2.98%
Windows 7	1,224	5,595,031	53.97%	6.11%	83.95%	89.11%	97.41%	1.94%
Windows 8.1	1,341	6,077,543	63.46%	6.90%	86.43%	91.14%	97.15%	1.42%
Various	62	496,749	55.15%	5.79%	83.23%	89.21%	96.83%	1.79%
Total	2,695	13,042,786	58.52%	6.37%	84.96%	90.04%	97.23%	1.68%

Table 1. Data set of PE files used for randomization coverage analysis.

four bits of the entry are set to 0x3, which represents the most common type of fixup transformation (`IMAGE_REL_BASED_HIGHLOW`). The following 12 bits represent the offset from the RVA of the corresponding block. The relocatable address can be calculated by adding the RVA and the offset, making it relative to the new base address of the segment instead of its original (preferred) one [172].

A crucial detail here is that any relocation entries regarding locations in the original code regions (that have now been displaced) must be removed from the respective block. The reason for this is that any stale entries can lead to corruption of the inserted `jmp` instructions, e.g., in case any of the overwritten instructions happened to involve RVAs with corresponding `.reloc` entries. Thus, not only new entries for the `.ropf` section must be created, but also the corresponding entries for the `.text` section must be removed, resulting in a total number of relocation entries equal to the number of entries in the original binary.

3.5 Experimental Evaluation

In this section we present the results of the experimental evaluation of our prototype implementation in terms of randomization coverage, file size increase, correct execution, and performance overhead. Our tests were performed on a 64-bit Windows 10 system equipped with an Intel Core i5-4590 3.3GHz processor and 16GB of RAM. For the evaluation of randomization coverage, we used a set of 2,695 PE files (both main executables and DLLs) from two different versions of Adobe Reader (Reader v9.3 and Acrobat Reader DC), Microsoft Office 2013, two Windows 7 and Windows 8.1 installations, and other programs and utilities, as detailed in Table 1. For correctness and

performance evaluation, we used a set of core Windows DLLs, as well as the Windows version of the SPEC CPU2006 benchmark suite.

3.5.1 Randomization Coverage

We begin our evaluation with the goal of assessing the improvement in terms of randomization coverage that instruction displacement can achieve. To that end, we compare the randomization coverage of in-place code randomization [62], instruction displacement, and the combination of the two techniques, as described in Section 3.3.5. In our initial experiments we use a maximum gadget length of five instructions, so that our results are comparable with the results reported by Pappas et al. [62]. In Section 3.5.4, we present further results when considering gadgets of size up to 10 and 15 instructions.

Table 1 summarizes key statistics about the distribution of gadgets in the tested binaries, and the randomization coverage of the two techniques. The 2,695 executables contain a total of approximately 13 million gadgets, 58.52% of which are unintended. The "unreachable" column refers to gadgets located in regions that cannot be properly disassembled, and thus are left untouched (by both techniques). These amount to 6.37% of all gadgets on average. In the rest of this section, unless specified otherwise, percentages of randomized gadgets are calculated over the number of gadgets located only within the properly disassembled code regions.

3.5.2 Coverage Improvement

Figure 7 shows the percentage of randomized gadgets in each PE file achieved by in-place code randomization, instruction displacement, and the combination of both techniques, as a cumulative fraction of all PE files in our data set. Although both techniques achieve comparable coverage, their combination manages to randomize a greater number of gadgets, and this is true for most executables, as evident from the slightly steeper curve. Specifically, as shown in Table 1, in-place code randomization on average affects 84.96% of the gadgets, instruction displacement affects 90.04% of the gadgets, while their combined use randomizes 97.23% of all gadgets in the properly disassembled code regions.

The Venn diagram in Figure 8 sheds more light into how each of the

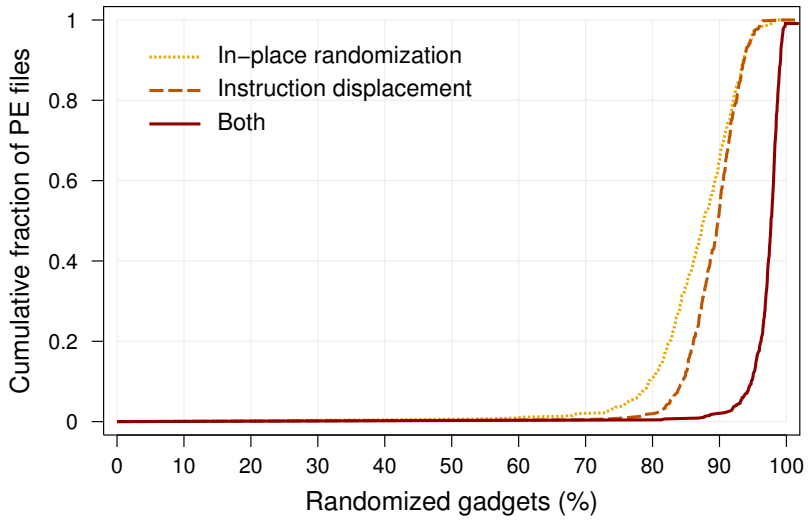


Figure 7. Randomized gadgets per PE file due to in-place code randomization, instruction displacement, and the combination of both techniques.

two techniques contributes in randomizing gadgets. While a majority of 77.78% can be randomized by both techniques, instruction displacement affects an extra 12.27%, increasing significantly the overall randomization coverage. When considering the whole binary, including the areas that cannot be disassembled, the randomization coverage is improved from 79.55% to 91.04%.

3.5.3 Gadget Analysis

The achieved randomization coverage of 97.23% leaves only a remaining 2.77% of gadgets that cannot be randomized by either of the two techniques. There are several reasons why instruction displacement cannot affect those gadgets. Among them, 0.83% are contained within basic blocks smaller than five bytes, and thus cannot be displaced due to the restriction of having to use a 5-byte `jmp` instruction for patching. Another 0.6% correspond to "basic block entry" gadgets that remain usable by following the inserted `jmp` instruction. The rest 1.34% cannot be displaced due to various other corner cases related to basic block alignment.

We also looked specifically into `call`-preceded gadgets, as they can be

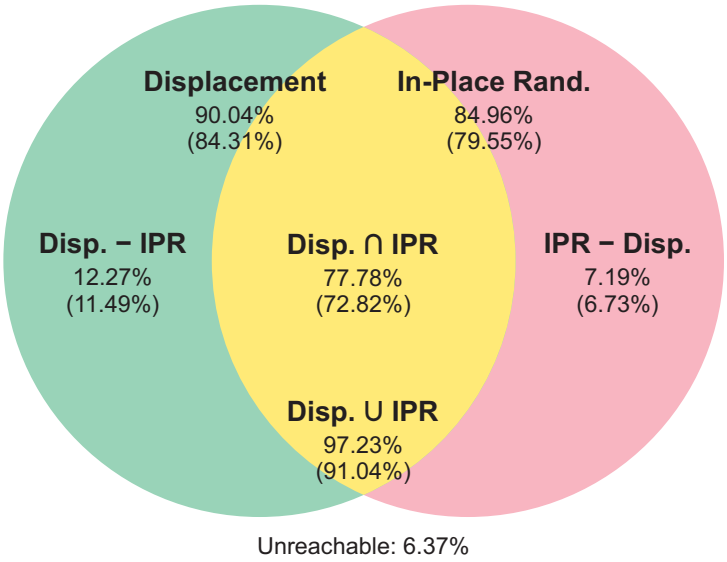


Figure 8. Randomization coverage of each technique in relation to each other. Instruction displacement increases the coverage achieved by in-place randomization alone from 84.96% to 97.23%.

particularly useful for an attacker that wants to bypass any deployed coarse-grained CFI protections [30, 33–35, 166]. The percentage of `call`-preceded among all gadgets, including the areas that cannot be disassembled, is 5.76%. After randomization using both techniques, their number is reduced to just 0.16% of all gadgets, with 0.14% being located in unreachable regions. This means that the achieved randomization coverage is enough to affect the vast majority of `call`-preceded gadgets.

3.5.4 Longer Gadgets

Given that an attacker may be able to use longer gadgets by spending some additional effort, we explored how the randomization coverage is affected when considering longer gadgets. To that end, we repeated our experiments using a maximum gadget length of 10 and 15 instructions. In all cases we follow the same approach as before, i.e., we first apply each diversification technique separately, followed by their combination.

Although the total number of discovered gadgets when considering a maximum length of 10 instructions increases by about 18%, the percentage

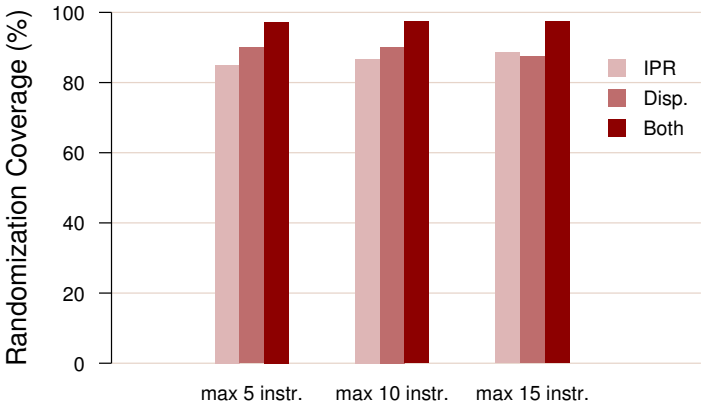


Figure 9. Randomization coverage for different maximum gadget lengths.

of randomized gadgets using both techniques remains almost the same, at around 97.4%, and so does also for 15-instructions-long gadgets, as shown in Figure 9. As the gadget size increases, IPR affects a slightly larger percentage of gadgets, moving from 84.96% to 86.78% and 88.59%, respectively. This result is expected, as the longer the gadget, the more opportunities for the different code transformations of IPR to affect some of its instructions. In contrast, as longer gadgets are more likely to span consecutive basic blocks, the coverage of instruction displacement drops slightly from 90.11% for 10 instructions to 87.42% for 15 instructions. It still contributes though an additional 9% in coverage when combined with IPR.

3.5.5 File Size Increase

Instruction displacement unavoidably incurs an increase in the size of the randomized PE files. Based on our experiments, the size of the `.ropf` section that hosts the displaced gadgets was verified to increase proportionally to the ratio of displaced code regions. As shown in Table 1 (last column), the average increase over the original PE file is minimal, at about 2.35%.

The total size of the displaced code regions is slightly larger than the original displaced code due to the additional *jmp* instructions that sometimes are appended at the end of displaced regions, and more rarely, due to larger displacement offsets in some operands. From all displaced regions, only 43.54% require a pair of jumps—in the rest of the cases, the region ends with

an indirect branch instruction that takes care of transferring control to the appropriate location. Some additional spaced is also consumed to the random padding at the beginning of the `.ropf` section.

3.5.6 Correctness

Any static binary instrumentation technique should preserve the original semantics of the instrumented program. To ensure that our transformations do not break the functionality of complex binaries, we first performed some manual testing with real-world applications, such as Adobe Reader. After randomization, we verified that a variety of PDF documents would open and render properly. Furthermore, when running diversified versions of the SPEC benchmarks, as described below, we did not encounter any issues with erroneous output.

As an attempt to exercise a more significant amount of code, we also used an automated testing approach based on the test suite of Wine [174], as similarly done by previous works [62,118]. Wine is a compatibility layer capable of running Windows applications on several POSIX-compliant operating systems. To validate that the ported APIs provided by Wine function as expected, Wine comes with an extensive test suite that covers the implementations of most functions exported by the core Windows DLLs. We ported to Windows some of Wine's test suites for 27 system DLLs, comprising a total of 10,036 different test cases, and used them repeatedly with randomized versions of those 27 actual Windows DLLs. By checking the outcome between various inputs and expected outputs, we could confirm that the randomized versions of the DLLs always worked correctly.

3.5.7 Performance Overhead

Finally, our last set of experiments focused on evaluating the performance overhead of instruction displacement. Since the technique involves extensive code patching and indirection, we expect to observe an increase in CPU overhead due to the extra executed `jmp` instructions and different code locality patterns. To get a better understanding of the performance implications, we performed two sets of experiments. First, we used a subset of the DLLs and Wine test cases used for the correctness evaluation, leaving aside any tests that involved the creation of files and other operations that would mask

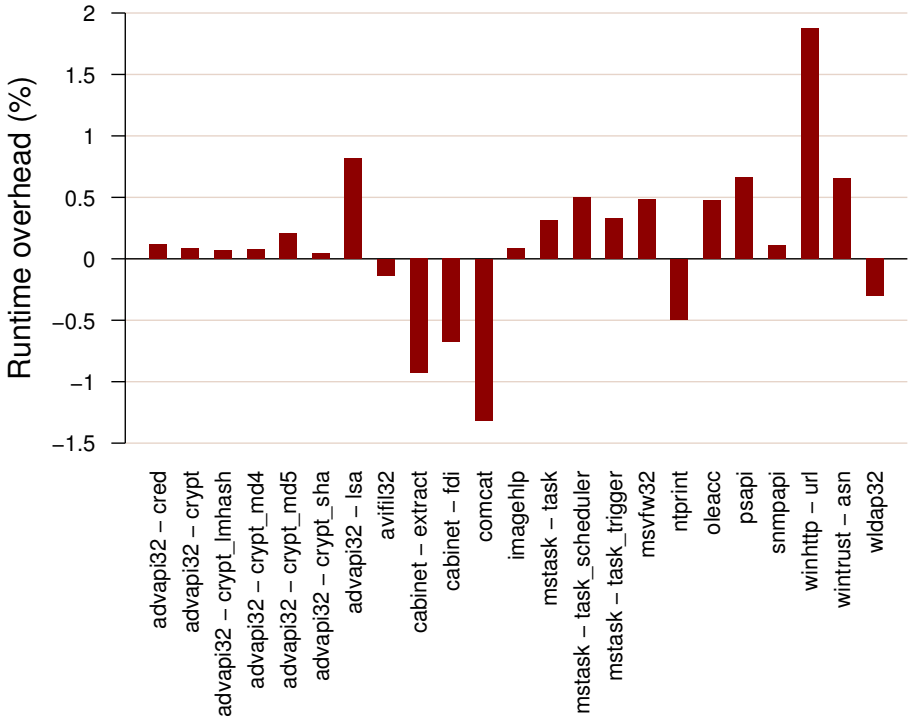


Figure 10. Runtime overhead over native execution for diversified versions of Windows system DLLs, driven by test cases ported from Wine's test suite. The average overhead across all tests is 0.48%.

out any CPU overhead. For each DLL, we measured the overall CPU user time for the completion of all relevant tests by taking the average time across multiple runs, using both the original and the randomized versions of the DLL. Second, we used the Windows-compatible subset of the standard SPEC CPU2006 benchmark suite.

Figures 10 and 11 show the runtime overhead of instruction displacement (when used in conjunction with in-place code randomization) over native execution for the Wine and SPEC experiments, respectively. The average overhead across all Wine tests is 0.48%, with a maximum of 1.87%. Surprisingly, some of the test cases exhibit a negative overhead, meaning that the diversified code ran faster than the original. We observed the same behavior in a stable and repeatable way across many iterations of the same experiment, with different instances of randomized binaries. We attribute this speedup in

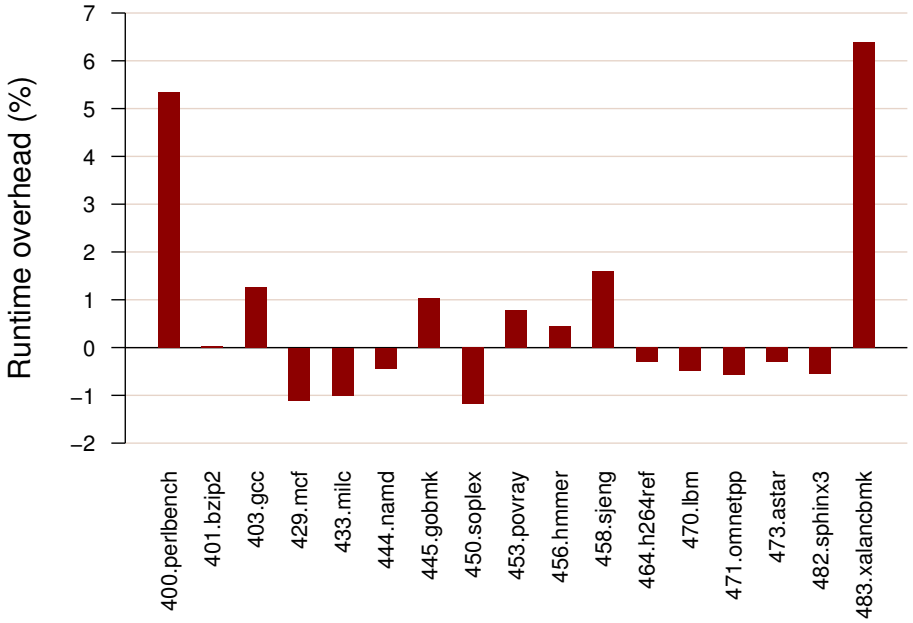


Figure 11. Runtime overhead for the SPEC CPU2006 benchmarks. The average overhead across all benchmarks is 0.36%.

better caching behavior due to better code locality in the `.ropf` section, as different "hot" basic blocks may now be brought in close proximity.

For the SPEC benchmarks, the average overhead was 0.36%. The two benchmarks with the highest overhead are `xalancbmk` and `perlbench` (6.38% and 5.34%, respectively), which is expected given that they are among the largest and more complex ones. A few other benchmarks exhibited the same negative overhead behavior that was also observed before, again in a consistent way across many repetitions.

We analyzed further the Wine and SPEC test cases that exhibited negative overheads using statistical hypothesis testing. With the null hypothesis that the mean CPU times for the original and randomized binaries are identical, Welch's two-sample t-test failed to reject it. That is, the means of the two distributions of CPU times for the original and randomized binaries in each case are not significantly different from each other with a 95% confidence interval, implying that these differences fall within the margin of measurement error.

We also explored the overhead of instruction displacement when used as a standalone technique, without the prior application of IPR. That is, when all 90.04% of the gadgets that can potentially be displaced are actually displaced, as opposed to just 12.27% when used in conjunction with IPR. The average overhead across all SPEC benchmarks in that case was just 2.06%, denoting that even extensive but focused patching can still incur a minimal performance overhead.

3.6 Discussion and Limitations

The two main limiting factors for instruction displacement in terms of randomization coverage are the precision of code extraction, and the size of existing basic blocks. Even when using a state-of-the-art disassembler like IDA Pro, some parts of the code cannot be extracted, and thus any gadgets in those regions remain unmodified. As our experiments have shown (Figure 8), when considering all available gadgets in a binary, instruction displacement reduces the number of unmodifiable gadgets from 21.45% for standalone in-place randomization, to 8.96% for the combination of both techniques. Given that the majority of them are located in unreachable regions (6.37%), a more accurate code extraction technique would allow for improved coverage. On the other hand, only a fraction (0.83%) of all extracted gadgets could not be displaced because they reside in small basic blocks. For "entry point" gadgets that still remain available after displacement, we plan to explore further transformations that can be applied on the displaced instructions, as discussed in Section 3.3.3.

Given the best-effort nature of our approach, we still cannot exclude the possibility of an attacker being able to assemble a functional ROP payload using solely the remaining fraction of unmodifiable gadgets. An indication about the complexity of ROP payload construction when working with a limited set of gadgets was provided by Pappas et al. [62], who showed that two automated ROP payload construction frameworks were unable to construct a functional payload using only the remaining unmodifiable gadgets by IPR. With the application of instruction displacement on top of IPR, this set of gadgets is significantly reduced even further (from 21.45% to 8.96%), and thus it is reasonable to assume that automated construction becomes even harder.

Besides the significant increase in coverage, instruction displacement also offers an additional benefit over in-place randomization in terms of the achieved randomization entropy. Although 77.78% of the gadgets can be randomized by both techniques, the randomization achieved through instruction displacement is qualitatively different. For some gadgets, IPR affects only a few of their instructions (or even just some of the instructions' operands), and often gadgets may exist in one out of just two possible states, leaving open the possibility of them being still usable after making the right assumptions. On the other hand, displaced gadgets end up in random locations that are infeasible to predict. Although in this work we have restricted the use of instruction displacement only for gadgets that are not randomized at all by IPR, in the future we plan to explore more aggressive combinations of the two techniques to improve randomization entropy even further. As we showed in Section 3.5.7, the associated overhead when displacing all possible gadgets is still modest, at 2.1%, so a small increase in the current number of displaced regions would have a negligible impact in the overall overhead.

Availability

Our prototype implementation is publicly available at: `https://github.com/kevinkoo001/ropf`

4 Code Inference Attacks and Defenses

4.1 Background

Fine-grained randomization [57–63,175] attempts to address well-known weaknesses [176–181] of contemporary ASLR by not only randomizing the locations of memory regions, but also shuffling functions, basic blocks, instructions, registers, and even the overall structure of code and data. The outcome of this diversification process is that the locations of any previously pinpointed gadgets are arbitrarily different in each instance of the same code segment. Even so, Snow et al. [43] showed that fine-grained randomization alone, even assuming a perfect implementation, does not prevent all control-flow hijacking attacks that leverage code reuse. Consider, for instance, a leaked code pointer that is not used to infer the location of gadgets, but is rather used along with a memory disclosure vulnerability to reveal the actual code bytes at the referenced location. In response to a new attack vector, two major mitigations have been proposed by preventing either code disclosure or code execution after being disclosed.

Preventing Code Disclosure (Execute Only Memory): The approaches that attempt to stop attack progression of disclosing code are, for the most part, instantiations of the longstanding idea of execute-only memory (XOM) [182] but applied to contemporary operating systems. For example, Oxymoron [144] proposed an approximation of XOM by eliminating all code-to-code pointer references—thus preventing the recursive code memory mapping step of just-in-time code reuse. To do so, a special translation table mediates all function calls. Unfortunately, such an approach requires heavy program instrumentation, and the necessary program transformations are only demonstrated to be achievable given source-level instrumentation. Additionally, Isomeron [121] later showed that just-in-time payloads can still be constructed even in the absence of code-to-code pointers (i.e., indirect code pointers).

Crane et al. [48,125] and Braden et al [50] approach the problem of XOM by starting with the requirement of source-level access. Hence, the many challenges that arise due to computed jumps and the intermingling of code and data in commodity (stripped) binaries, are alleviated. Readactor [48],

for example, relies on a modified compiler to ensure that all code and data is appropriately separated. Execute-only memory is then enforced by leveraging finer-grained memory permission bits made available by extended page tables (EPT) in recent processors. Likewise, HideM [47] also suggested XOM with consideration for intermixed code and data. To do so, data embedded in code sections is first identified via binary analysis (with symbol information), then split into separate code-only and data-only memory pages. At runtime, instruction fetches are directed to code-only pages, while reads are directed to data-only pages. Conditional page direction is implemented by loading split code and data translation look-aside buffers (TLBs) with different page addresses. One drawback of this approach is that all recent processors now make use of a unified second-level TLB, rendering this approach incompatible with modern hardware. Moreover, static code analysis on binaries is a difficult problem, leaving no other choice but to rely on human intervention to separate code and data in closed-source software. Thus, the security guarantees imbued by that approach are only as reliable as the human in the loop.

Preventing Disclosed Code Execution (Destructive Code Reads): Heisenbyte [49] takes a radically different approach that instead focuses on the concept of destructive code reads, whereby code is garbled after it is read. By taking advantage of existing virtualization support (i.e., EPT) and focusing solely on thwarting the execution of disclosed code bytes, Heisenbyte's use of destructive code reads sidesteps the many problems that arise due to incomplete disassembly in binaries, and thereby affords protection of complex close-sourced COTS binaries. Similarly, NEAR [44] implements a so-called no-execute-afterread memory primitive using EPT on x86 Windows and other hardware primitives on x86-64 and ARMv8 which, instead of randomly garbling code, substitutes fixed invalid instruction(s), hence ensuring that subsequent execution always terminates the application. NEAR also demonstrates how valid data within code sections can be automatically and reliably relocated onload, without the use of source code or debug symbols.

Both Heisenbyte [49] and NEAR [44] provide an excellent overview of how destructive code reads can be implemented by leveraging EPT and conservatively relocating intermingled code and data during an offline analysis phase. When a protected application loads, a duplicate copy of its executable memory pages is maintained, and that copy is used in the event of a memory read operation.

4.2 Threat Model

In a code inference attack scenario, we assume the following threat model.

Defenders' Assumptions: We assume the following mitigations are in place: i) non-executable memory, that is, the stack(s) and heap(s) of the protected program are non-executable, thus preventing an attacker from directly injecting new executable code into data regions, ii) fine-grained randomization using in-place randomization [62] to achieve binary compatibility, as other proposed approaches require auxiliary information (i.e., source code or debug symbols) for complex COTS software, iii) JIT mitigations against browser-specific attacks such as JIT–spraying instructions (i.e., Internet Explorer includes countermeasures that share commonalities with Librando [183]), and iv) destructive code reads that the act of reading any byte of code immediately precludes that specific byte of code from being executed later.

Adversaries' Assumptions: We assume that an adversary can read and write arbitrary memory of a vulnerable process. In addition, the adversary is capable of running scripted code within the limits of the target application (e.g., JavaScript or ActionScript code) and storing the gathered information either locally, e.g., in cookies or in HTML5 Local Storage [184], or on a remote server. To be specific, the concept of destructive reads works [49] in cases where the following (implicit) assumptions hold: i) *code persistence* where code may not be loaded and unloaded so that the adversary may not restore destroyed code after learning its layout, ii) *code singularity*, that is, the process may not contain any duplicate code sections so that the adversary may not infer any information about the code in process memory by reading another existing copy of that code, and iii) *code dis–association* that any information discovered during an attempted attack cannot be relied upon in subsequent attacks, ensuring that the adversary cannot mount an incremental attack against an application disclosing partial information and then reusing it in the next stage of the attack.

4.3 Code Inference Attacks Undermining Destructive Code Reads

In this section, we focus on how to undermine destructive code reads via implicit reads to just–in–time disclose a usable code reuse payload in face of destructive code reads. This strategy breaks the third implicit assumption for destructive code reads, code dis–association. Code inference attack entails implicit reading of code, thus avoiding code destruction altogether. This attack turned out to be more powerful than we first envisioned, and motivates the need for a stronger dis-association property that is not present in any binary–compatible fine–grained randomization scheme we are aware of.

4.3.1 Attack Approach

To allow for precise differentiation between code and data embedded within code segments, execute–only memory using destructive reads is enforced at a byte-level granularity. Although this approach effectively prevents the execution of code that has been previously read, its implications regarding an attacker's ability to infer the layout of code that follows already disclosed bytes requires careful consideration. It is conceivable that depending on the applied code randomization strategy, reading only a few bytes of existing code might be enough for making an informed guess about the instructions that follow the disclosed code without actually reading them. In particular, we call such a guessable gadget a *zombie gadget* that can be inferred through our implicit reads.

Here we describe how implicit reads undermines a set of narrow-scoped code transformations, in-place code randomization, applicable to binary-compatible fine-grained randomization [62]. Specifically, it applies the following four code transformations: i) instruction substitution that replaces existing instructions with functionally-equivalent ones, ii) basic block instruction reordering that changes the order of instructions within a basic block according to an alternative, functionally equivalent instruction scheduling, iii) register preservation code reordering that reorders the `push` and `pop` instructions of a function's prologue and epilogue, and iv) register reassignment that swaps the register operands of instructions throughout overlapping live regions. The following describes how code inference attacks can defeat each transformation technique.

Instruction Substitution: Given that the original binary code of a program and the sets of equivalent instructions are common knowledge, an attacker knows a priori all instructions that are candidates for substitution. By just reading the opcode byte of a candidate-for-substitution instruction in the randomized instance of a program, an attacker can precisely infer the sequence of bytes that follow the opcode byte (i.e., the instruction's operands), and consequently, the state of any overlapping randomized gadget. If the disclosed opcode is also part of the randomized gadget, however, the part of the gadget that starts after the opcode byte will remain usable.

Basic Block Instruction Reordering: By precomputing all possible orderings of a given basic block, an attacker may be able to infer the order of instructions towards the end of the block by just reading a few instructions from the beginning of the block. The feasibility of this inference approach for a given gadget depends on the size of the basic block in which the gadget is contained, the location of the gadget within the block, and the number of possible instruction orderings.

Register Preservation Code Reordering: An adversary could use code inference to implicitly learn the precise structure of a gadget that has been randomized, which involves reordering the `push` and `pop` instructions of a function's prologue and epilogue. By (destructively) reading instructions in the prologue that are affected by the transformation, but which are not part of the actual gadget, an attacker can accurately infer the structure of the gadget in the function epilogue. Concretely, if the attacker knows that registers are saved onto the stack by a function, the order by which these registers are popped in the epilogue is the reverse order in which they were pushed during the prologue, so reading the prologue allows the adversary to infer the exact gadget contained in the function epilogue. Since the actual disclosure by the adversary was aimed at the prologue, destructive read enforcement will only protect those bytes, leaving the epilogue to be freely used as a useful gadget for the adversary.

Register Reassignment: Given that an attacker can precompute all live regions in the original code, reading even a single instruction at the beginning of a live region might be enough to infer the structure of gadgets towards the

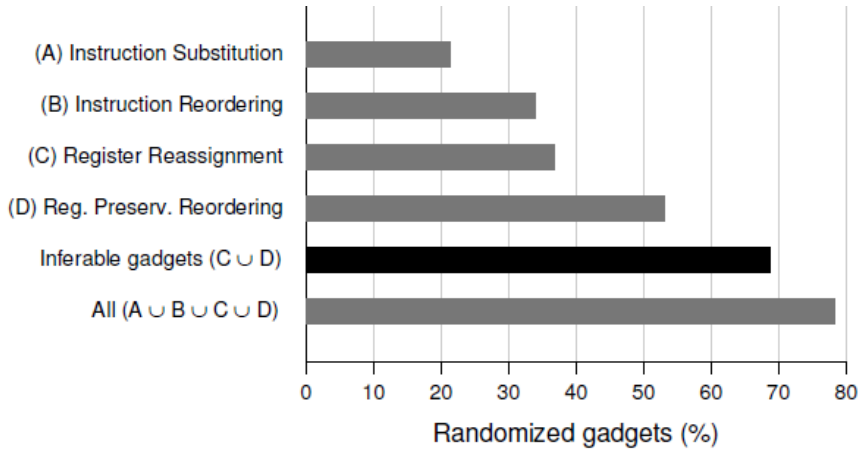


Figure 12. Randomization coverage achieved by the different transformations of in-place code randomization. The state of randomized gadgets due to register reassignment (C) and register preservation code reordering (D) can always be inferred through indirect disclosure. This means that an extra 68.72% of all infereable gadgets can be safely used by an attacker.

end of that region. Register reassignment has the second highest coverage among the four transformations, altering more than 40% of the gadgets in a code segment [62].

4.3.2 Evaluation on Code Inference Attacks

For our empirical analyses, we used a set of 47 libraries from Adobe Reader v9.3 and Adobe Acrobat Reader DC, which in total contain 628,907 gadgets. We used the publicly available implementation of in-place code randomization [169] to randomize the libraries. Figure 12 shows the percentage of gadgets that can be randomized by each of the four randomization techniques. Note that a given gadget can be randomized by more than one technique. The combination of all techniques randomizes 78.28% of all gadgets found in the analyzed code. We found that similar to the results reported by Pappas et al. [62], instruction substitution and basic block instruction reordering achieve the lowest randomization coverage (21.43% and 33.98%, respectively). The two more effective transformations, which happen to always allow for implicit code disclosure, achieve a combined coverage of 68.72%. In other words, by focusing only on register reassignment and register preservation

code reordering, an attacker can infer the state of 90.44% of all randomized gadgets (i.e., including the 21.72% of the gadgets that cannot be randomized by any of the transformations of Pappas et al. [62]). Based on these results, and considering that further inference against instruction substitution and basic block reordering is likely possible, we conclude that in-place code randomization is not sufficient for use in conjunction with binary-level execute-only memory protections.

4.4 Code Inference Defenses

Both Heisenbyte [49] and NEAR [44] provide solid foundations for detecting the most straightforward way an attacker can learn the values of code bytes (i.e., by directly reading their values in memory) and prevent the execution of those exact bytes at a later time. However, attacks are still possible when an adversary leverages the use of implicit code reads to infer the values of code bytes indirectly, based on the directly read values of related code bytes, rendering destructive code reads ineffective as shown in Section 4.3. In this section, we describe a practical defense against just–in–time code reuse attacks that take advantage of an adversary's ability to disclose and execute code bytes whose values were learned by so-called code inference attacks.

4.4.1 Defense Approach

At a high-level, our approach centers around the ability to place randomized versions of code in a process at key trigger points during the execution of a just–in–time code reuse attack. Specifically, we replace the code upon which an attack relies with logically equivalent code of a different form that will break the attacker's ROP payload. To achieve this, we apply binary-compatible in-place randomization to code modules in order to obtain multiple diversified copies of the code which are kept in kernel-space memory where they are not accessible to user-level processes. With swappable versions of a module available at our disposal, disclosed code can be efficiently replaced at runtime with minimal complexities while assuring correct program execution. Specifically, when a module is loaded into memory from disk, we ensure that a randomized copy of that module is mapped into the user-space process. Furthermore, individual reads to executable addresses in the module trigger our system to swap localized code sequences within functions for semantically-

equivalent randomized code sequences from one of the alternate versions maintained in kernel-space. This technique prevents adversaries from making use of individually disclosed gadgets, while not requiring any re-routing of control flow or swapping of entire code modules.

One of countermeasures to defend against code inference attacks is to adopt runtime re-randomization [68, 126, 127], which requires unsound and cumbersome pointer tracking. Re-randomization schemes can introduce stale pointers into a program if they do not carefully adjust every pointer that references a given code section when that section is relocated at runtime. Our approach, as an alternative, focuses on moving around large chunks of code in process memory to avoid complexity of tracking pointers. We opted for a more localized solution that guarantees that any disclosed bytes (either explicitly or implicitly) are rerandomized in response to disclosure, simplifying assurance of correct program continuation. When a code disclosure occurs, we detect and replace only the part of the code that was disclosed with a different randomized version. Because our approach maintains k different randomized versions of the program, we can randomly select from k different versions (which reside in kernel module memory) of the disclosed code to swap in at runtime.

4.4.2 Evaluation on Code Inference Defenses

The in-place code randomization of Pappas et al. [62] uses a combination of four different transformation techniques of different spatial granularity (instruction, basic block, and whole function) to generate alternative representations of a program's code. For a given code disclosure, multiple transformations may have been applied to the code area surrounding the address which caused an EPT fault. We use the coarsest randomization scope (i.e., the function that contains the disclosed code bytes) as the unit of re-randomization because function scope randomizations tend to offer the highest level of variability for a given range of code. That said, when we swap the bytes of a given function for a randomized copy, the contained bytes may have been altered by any combination of the four transformation techniques, so our solution still fully benefits from all four transformation tactics. It is crucial to evaluate whether re-randomization at the function level allows for enough randomization variability to prevent attackers from guessing or inferring the structure of the code to be swapped in. Specifically, according to the definition

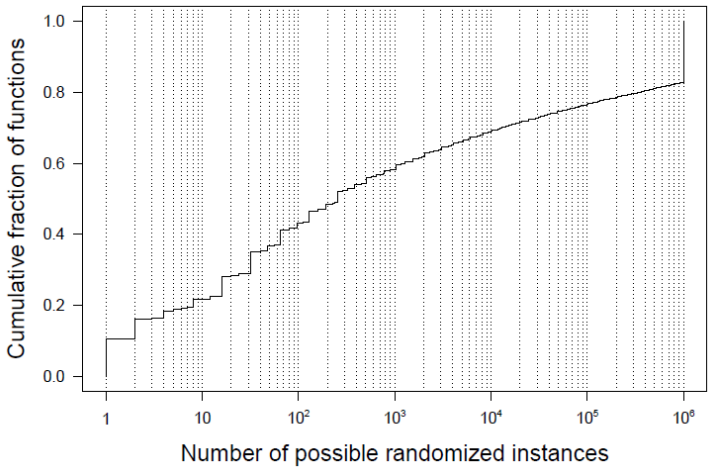


Figure 13. Function randomization variability

by Pappas et al. [62], we define function randomization variability to be the number of possible randomized instances that can be generated for a given function.

To gain a better understanding of the resulting randomization variability, we performed an empirical evaluation based on more than 1.5 million functions from 2,566 PE files from both Windows 7 and Windows 8.1. Figure 13 shows the number of possible randomized instances of a function (including its original form), as a cumulative fraction of all 1.5M functions contained in the analyzed PE files. Notice that 10% of the functions have a variability value of one (i.e., just their original instance), meaning that in-place randomization cannot generate any variants for them. The next 4% have only two possible instances, and then the variability for the rest of the functions increases exponentially. For ease of exposition, we cap the calculation of all possible variants to 100,000. Note that just two versions of a function could be enough to foil an attacker, since randomly choosing which version of the code to swap at runtime means that the success rate for the attacker diminishes rapidly.

In general, these 10% of functions cannot be randomized due to their tiny size, often a consequence of compiler intricacies such as basic block sharing, wrapper functions, and other performance optimizations. In fact, our data on Windows binaries shows that about 15% of functions are at most 10 bytes in size, whereas only half of them are larger than 50 bytes.

Moreover, 40% of functions consist of a single basic block, while 62% have five or fewer basic blocks. Our findings confirm the observations of Pappas et al. [62] in that the 10% of non-randomizable functions consists mostly of such tiny functions. Overall, we found that roughly 80% of gadgets can be probabilistically broken.

5 Compiler-Assisted Code Randomization

5.1 Motivation

Code randomization techniques can be categorized into two main types according to their deployment model—more specifically, according to *who* is responsible for randomizing the code (software vendor vs. end user) and *where* the actual randomization takes place (vendor's system vs. user's system). In this section, we discuss these two types of techniques in relation to the challenges that so far have prevented their deployment, and introduce our proposed approach.

5.1.1 Diversification by End Users

The vast majority of code randomization proposals shift the burden of diversification solely to end users, as *they* are responsible for diversifying the obtained software on *their* systems. For open-source software, this entails obtaining its source code, setting up a proper build environment, and recompiling the software with a special toolchain [48, 51, 57–59, 74, 125]. For closed-source software, this entails transforming existing executables using static binary rewriting, sometimes assisted by a runtime component to compensate for the imprecisions of binary code disassembly [60–62, 68, 69, 121, 144, 185]. Interested readers are referred to the survey of Larsen et al. [53] for an extensive discussion on the many challenges that code randomization techniques based on static binary rewriting face.

From a deployment perspective, however, both compiler-level and rewriter-level techniques share the same main drawbacks: end users (or system administrators) are responsible for diversifying an obtained application through a complex and oftentimes cumbersome process. In addition, this is a process that requires substantial computational and human resources in terms of the system on which the diversification will take place, as well as in terms of the time, effort, and expertise needed for configuring the necessary tools and performing the actual diversification. Consequently, it is unrealistic to expect this deployment model to reach the level of transparency that other diversification protections, like ASLR, have achieved.

At the same time, these approaches clash with operations that rely on

software uniformity, which is an additional limiting factor against their deployment [53, 72]. When code randomization is applied at the client side, crash dumps and debug logs from randomized binaries refer to meaningless code and data addresses, code signing and integrity checks based on precomputed checksums fail, and patches and updates are not applicable on the diversified instances, necessitating the whole diversification process to be performed again from scratch.

5.1.2 Diversification by Software Vendors

Given that expecting end users to handle the diversification process is a rather unrealistic proposition for facilitating widespread deployment, an alternative is to rely on software vendors for handling the whole process and distributing already diversified binaries—existing app store software delivery platforms are particularly attractive for this purpose [71]. The great benefit of this model is that it achieves complete transparency from the perspective of end users, as they continue to obtain and install software as before [186]. Additionally, as vendors are in full control of the distribution process, they can alleviate any error reporting, code signing, and software update issues by keeping (or embedding) the necessary information for each unique variant to carry out these tasks [53].

Unfortunately, shifting the diversification burden to the side of software vendors also entails significant costs that in practice make this approach unattractive. The main reason is the increased cost for both *generating* and *distributing* diversified software variants [53]. Considering that popular software may exceed a billion users [73], the computational resources needed for generating a variant per user, per install, upon each new major release, can be prohibitively high from a cost perspective, even when diversification happens only at the late stages of compilation [72]. Additionally, as each variant is different, distribution channels that rely on caching, content delivery networks, software mirrors, or peer-to-peer transfers, will be rendered ineffective. Finally, at the release time of a new version of highly popular software, an issue of "enough inventory" will arise, as it will be challenging for a server-side diversification system to keep up with the increased demand in such a short time span [53].

5.1.3 Compiler–Rewriter Cooperation

The security community has identified compiler–rewriter cooperation as a potentially attractive solution for software diversification [53], but (to the best of our knowledge) no actual design and implementation attempt has been made before. We discuss in detail our design goals and the benefits of the proposed approach in Section 5.3.1.

Note that our aim is not to enable reliable code disassembly at the client side (which Larsen et al. [53] suggested as a possibility for a hybrid model), but to enable *rapid* and *safe* fine-grained code randomization by simply treating code as a sequence of raw bytes. In this sense, our proposal is more in line with the way ASLR has been deployed: developers must explicitly compile their software with ASLR support (i.e., with relocation information or using position-independent code), while the OS (if it supports ASLR) takes care of performing the actual transformation (i.e., the dynamic linker/loader maps each module to a randomly-chosen virtual address).

This flexibility and backwards compatibility is an important benefit compared to the alternative approach of self-randomizing binaries [58, 141]. According to the characteristics of each particular system, administrators may opt for randomization at installation or load time (or no randomization at all), and selectively enable or disable additional hardening transformations and instrumentation that may be available. On systems not equipped with the rewriter component, augmented binaries continue to work exactly as before.

5.2 Background

To fulfill our goal of generic, transparent, and fast fine-grained code randomization at the client side, there is a range of possible solutions that one may consider. In this section, we discuss why existing solutions are not adequate, and provide some details about the compiler toolchain we used.

5.2.1 The Need for Additional Metadata

Static binary rewriting techniques [60, 69, 144] face significant challenges due to indirect control flow transfers, jump tables, callbacks, and other code constructs that result in incomplete or inaccurate control flow graph extrac-

tion [163,187,188]. More generally applicable techniques, such as in-place code randomization [62,185], can be performed even with partial disassembly coverage, but can only apply narrow-scoped code transformations, thereby leaving parts of the code non-randomized (e.g., complete basic block reordering is not possible). On the other hand, approaches that rely on dynamic binary rewriting to alleviate the inaccuracies of static binary rewriting [61, 63, 69, 70] suffer from increased runtime overhead.

A relaxation that could be made is to ensure programs are compiled with debug symbols and relocation information, which can be leveraged at the client side to perform code randomization. Symbolic information facilitates runtime debugging by providing details about the layout of objects, types, addresses, and lines of source code. On the other hand, it does not include *lower-level* information about complex code constructs, such as jump tables and callback routines, nor it contains metadata about (handwritten) assembly code [189]. To make matters worse, modern compilers attempt to generate cache-friendly code by inserting alignment and padding bytes between basic blocks, functions, objects, and even between jump tables and read-only data [190]. Various performance optimizations, such as profile-guided [191] and link-time [192] optimization, complicate code extraction even further—Bao et al. [138], Rui and Sekar [193], and others [135, 149, 194], have repeatedly demonstrated that accurately identifying functions (and their boundaries) in binary code is a challenging task.

In the same vein, Williams-King et al. [68] implemented Shuffler, a system that relies on symbolic and relocation information (provided by the compiler and linker) to disassemble code and identify all code pointers, with the goal of performing live code re-randomization. Despite the impressive engineering effort, its authors admit that they *"encountered myriad special cases"* related to inaccurate or missing metadata, special types of symbols and relocations, and jump table entries and invocations. Considering that these numerous special cases occurred just for a particular compiler (GCC), platform (x86-64 Linux), and set of (open-source) programs, it is reasonable to expect that similar issues will arise again, when moving to different platforms and more complex applications.

Based on the above, we argue that relying on *existing* compiler-provided metadata is not a viable approach for building a *generic* code transformation solution. More importantly, the complexity involved in the transformation

process performed by the aforementioned schemes (e.g., static code disassembly, control flow graph extraction, runtime analysis, heuristics) is far from what could be considered reasonable for a *fast* and *robust* client-side rewriter, as discussed in Section 5.1.1. Consequently, we opt for augmenting binaries with just the necessary *domain-specific metadata* needed to facilitate safe and generic client-side code transformation (and hardening) without any further binary code analysis.

5.2.2 Fixups and Relocations

When performing code randomization, machine instructions with register or immediate operands do not require any modification after they are moved to a new (random) location. In contrast, if an operand contains a (relative or absolute) reference to a memory location, then it has to be adjusted according to the instruction's new location, the target's new location, or both. (Note that a similar process takes place during the late stages of compilation.)

Focusing on LLVM, whenever a value that is not yet concrete (e.g., a memory location or an external symbol) is encountered during the instruction encoding phase, it is represented by a placeholder value, and a corresponding *fixup* is emitted. Each fixup contains information on how the placeholder value should be rewritten by the assembler when the relevant information becomes available. During the *relaxation* phase [195, 196], the assembler modifies the placeholder values according to their fixups, as they become known to it. Once relaxation completes, any unresolved fixups become *relocations*, stored in the resulting object file.

Figure 14 shows a code snippet that contains several fixups and one relocation. The left part corresponds to an object file after compilation, whereas the right one depicts the final executable after linking. Initially, there are four fixups (underlined bytes) emitted by the compiler. As the relocation table shows, however, only a single relocation (which corresponds to fixup ①) exists for address `0x5a7f`, because the other three fixups were resolved by the assembler. Henceforth, we explicitly refer to relocations in object files as *link-time relocations*—i.e., fixups that are left unresolved after the assembly process (to be handled by the linker). Similarly, we refer to relocations in executable files (or dynamic shared objects) as *load-time relocations*—i.e., relocations that are left unresolved after linking (to be handled by the dynamic

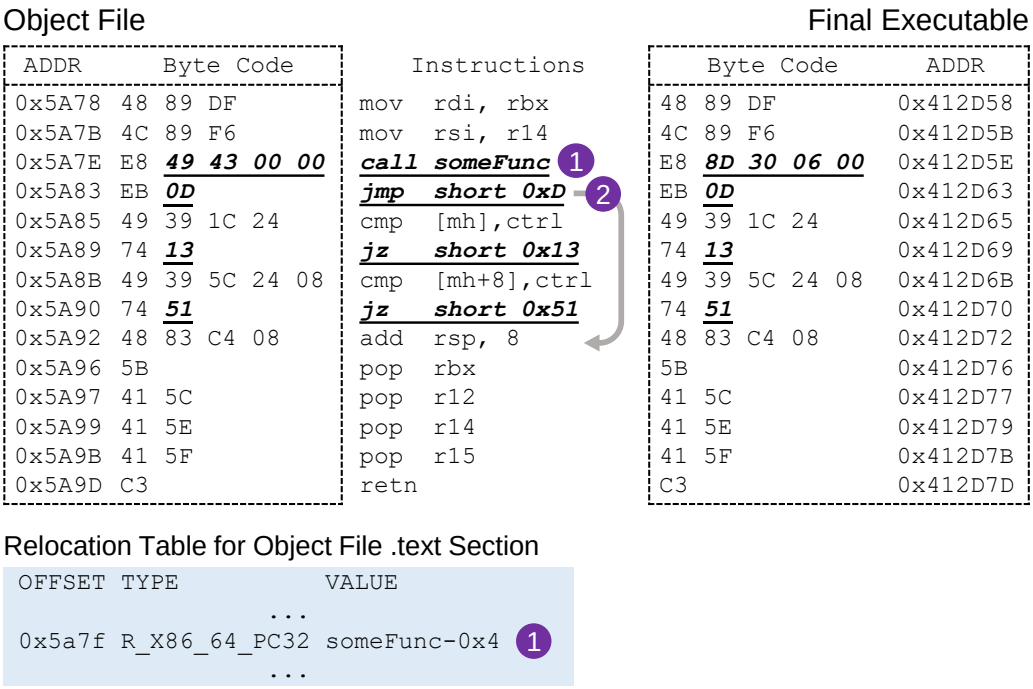


Figure 14. Example of the fixup and relocation information that is involved during the compilation and linking process.

linker/loader). Note that in this particular example, the final executable does not contain any load-time relocations, as relocation ① was resolved during linking (`0x4349`→`0x6308d`).

In summary, load-time relocations are a subset of link-time relocations, which are a subset of all fixups. Unfortunately, even if link-time relocations are completely preserved by the linker, they are not sufficient for performing fine-grained code randomization. For instance, fixup ② is earlier resolved by the assembler, but is essential for basic block reordering, as the respective single-byte `jmp` instruction may have to be replaced by a four-byte one—if the target basic block is moved more than 127 bytes forward or 126 bytes backwards from the `jmp` instruction itself. Evidently, comprehensive fixups are *pivotal* pieces of information for fine-grained code shuffling, and should be promoted to first-class metadata by modern toolchains in order to provide support for generic, transparent, and compatible code diversification.

5.3 Enabling Client-side Code Diversification

5.3.1 Overall Approach

Our design is driven by the following two main goals, which so far have been limiting factors for the actual deployment of code diversification in real-world environments:

Practicality: From a deployment perspective, a practical code diversification scheme should not disrupt existing features and software distribution models. Requiring software vendors to generate a diversified copy per user, or users to recompile applications from source code or transform them using complex binary analysis tools, have proven to be unattractive models for the deployment of code diversification.

Compatibility: Code randomization is a highly disruptive operation that should be safely applicable even for complex programs and code constructs. At the same time, code randomization inherently clashes with well-established operations that rely on software uniformity. These include security and quality monitoring mechanisms commonly found in enterprise settings (e.g., code integrity checking and whitelisting), as well as crash reporting, diagnostics, and self-updating mechanisms.

Augmenting compiled binaries with metadata that enable their subsequent randomization at installation or load time is an approach fully compatible with existing software distribution norms. The vast majority of software is distributed in the form of compiled binaries, which are carefully generated, tested, signed, and released through official channels by software vendors. On each endpoint, at installation time, the distributed software typically undergoes some post-processing and customization, e.g., its components are decompressed and installed in appropriate locations according to the system's configuration, and sometimes they are even further optimized according to the client's architecture, as is the case with Android's ahead-of-time compilation [197] or the Linux kernel's architecture-specific optimizations [198]. Under this model, code randomization can fittingly take place as an additional post-processing task during installation.

As an alternative, randomization can take place at load time, as part of the modifications that the loader makes to code and data sections for processing relocations [199]. However, to avoid extensive user-perceived delays due to

the longer rewriting time required for code randomization, a more viable approach would be to maintain a supply of pre-randomized variants (e.g., an OS service can be generating them in the background), which can then instantly be picked by the loader.

Note that this distribution model is followed *even for open-source software*, as installing binary executables through package management systems (e.g., `apt-get`) offers unparalleled convenience compared to having to compile each new or updated version of a program from scratch. More importantly, under such a scheme, each endpoint can choose among different levels of diversification (hardening vs. performance), by taking into consideration the anticipated exposure to certain threats [200], and the security properties of the operating environment (e.g., private intranet vs. Internet-accessible setting).

The embedded metadata serves two main purposes. First, it allows the safe randomization of even complex software without relying on imprecise methods and incomplete symbolic or debug information. Second, it forms the basis for *reversing* any applied code transformation when needed, to maintain compatibility with existing mechanisms that rely on referencing the original code that was initially distributed.

Figure 15 presents a high-level view of the overall approach. The compilation process remains essentially the same, with just the addition of metadata collection and processing steps during the compilation of each object file and the linking of the final *master* executable. The executable can then be provided to users and endpoints through existing distribution channels and mechanisms, without requiring any changes.

As part of the installation process on each endpoint, a *binary rewriter* generates a randomized version of the executable by leveraging the embedded metadata. In contrast to existing code diversification techniques, this transformation does not involve any complex and potentially imprecise operations, such as code disassembly, symbolic information parsing, reconstruction of relocation information, introduction of pointer indirection, and so on. Instead, the rewriter performs simple transposition and replacement operations based on the provided metadata, treating all code sections as *raw binary data*. Our prototype implementation, discussed in detail in Section 5.4, currently supports fine-grained randomization at the granularity of functions and basic blocks, is oblivious to any applied compiler optimizations, and supports

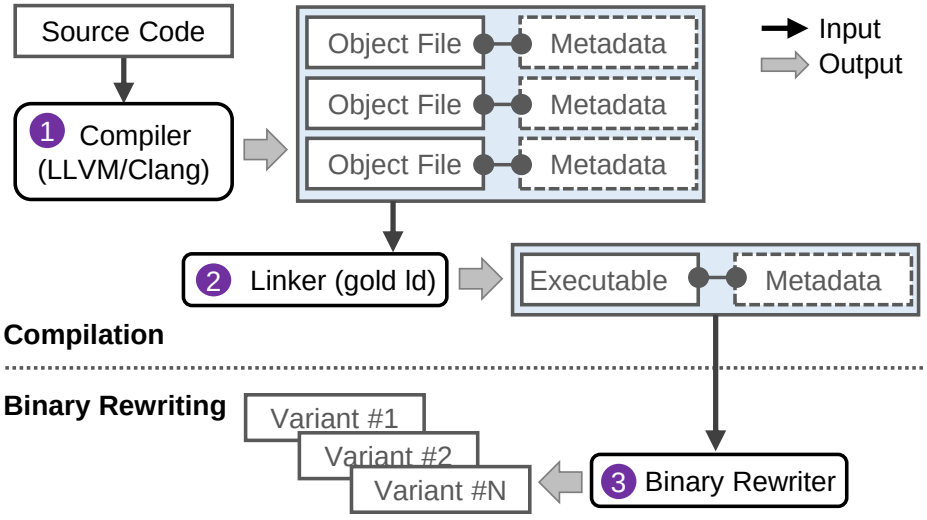
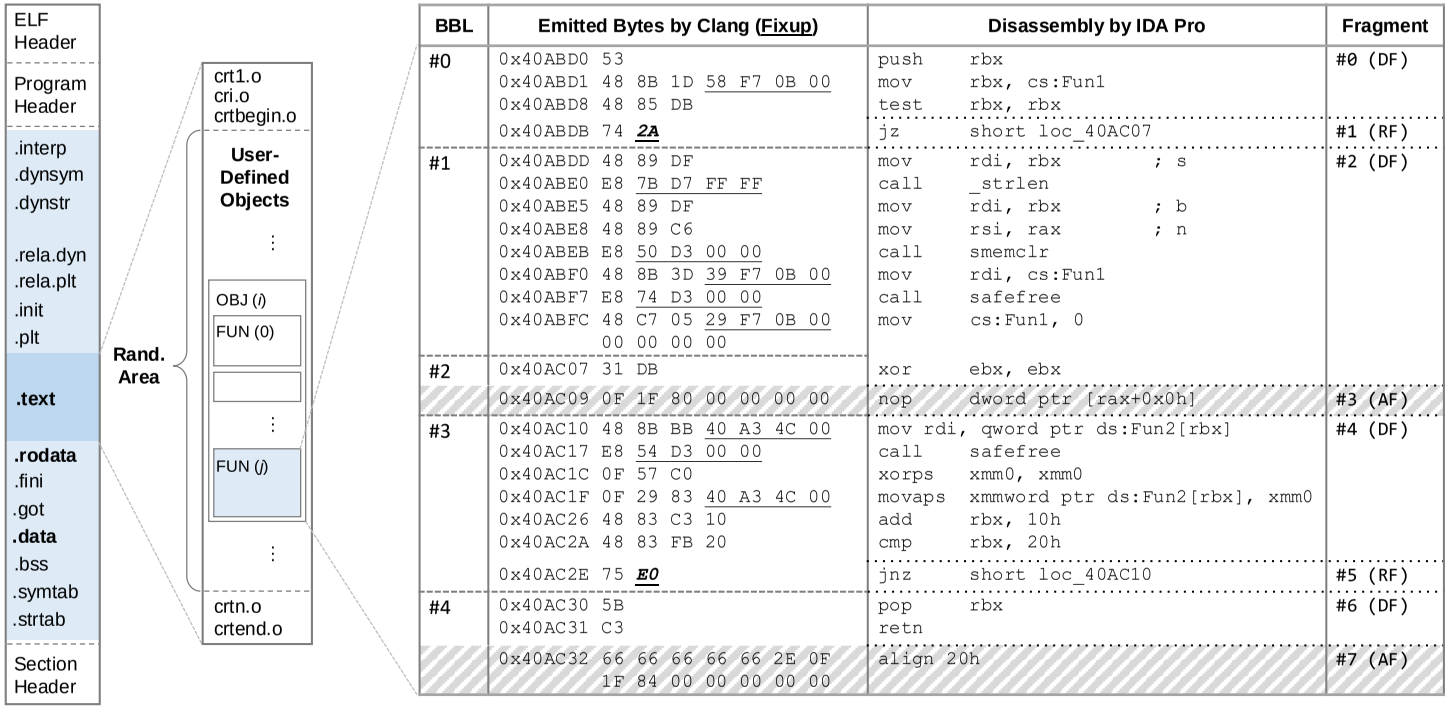


Figure 15. Overview of the proposed approach. A modified compiler collects metadata for each object file ①, which is further updated and consolidated at link time into a single extra section in the final executable ②. At the client side, a binary rewriter leverages the embedded metadata to rapidly generate randomized variants of the executable ③.

static executables, shared objects, PIC, partial/full RELRO [201], exception handling, LTO, and even CFI.

5.3.2 Compiler-level Metadata

Our work is based on LLVM [202], which is widely used in both academia and industry, and we picked the ELF format and the x86-64 architecture as our initial target platform. Figure 16 illustrates an example of the ELF layout generated by Clang (LLVM's native C/C++/Objective-C compiler).



BBL	Emitted Bytes by Clang (Fixup)	Disassembly by IDA Pro	Fragment
#0	0x40ABD0 53 0x40ABD1 48 8B 1D 58 F7 0B 00 0x40ABD8 48 85 DB	push rbx mov rbx, cs:Fun1 test rbx, rbx	#0 (DF)
	0x40ABDB 74 ***2A***	jz short loc_40AC07	#1 (RF)

#1	0x40ABDD 48 89 DF
	0x40ABE0 E8 7B D7 FF FF
	0x40ABE5 48 89 DF
	0x40ABE8 48 89 C6
	0x40ABEB E8 50 D3 00 00
	0x40ABF0 48 8B 3D 39 F7 0B 00
	0x40ABF7 E8 74 D3 00 00
	0x40ABFC 48 C7 05 29 F7 0B 00
	00 00 00 00
		mov rdi, rbx ; s
		call _strlen
		mov rdi, rbx ; b
		mov rsi, rax ; n
		call smemclr
		mov rdi, cs:Fun1
		call safefree
		mov cs:Fun1, 0
			#2 (DF)

#2	0x40AC07 31 DB	xor ebx, ebx	
	0x40AC09 0F 1F 80 00 00 00 00	nop dword ptr [rax+0x0h]	#3 (AF)

#3	0x40AC10 48 8B BB 40 A3 4C 00
	0x40AC17 E8 54 D3 00 00
	0x40AC1C 0F 57 C0
	0x40AC1F 0F 29 83 40 A3 4C 00
	0x40AC26 48 83 C3 10
	0x40AC2A 48 83 FB 20
		mov rdi, qword ptr ds:Fun2[rbx]
		call safefree
		xorps xmm0, xmm0
		movaps xmmword ptr ds:Fun2[rbx], xmm0
		add rbx, 10h
		cmp rbx, 20h
			#4 (DF)

	0x40AC2E 75 ***E0***	jnz short loc_40AC10	#5 (RF)
#4	0x40AC30 5B 0x40AC31 C3	pop rbx retn	#6 (DF)
	0x40AC32 66 66 66 66 66 2E 0F 1F 84 00 00 00 00 00	align 20h	#7 (AF)

Figure 16. An example of the ELF layout generated by Clang (left), with the code of a particular function expanded (center and right). The leftmost and rightmost columns in the code listing ("BBL" and "Fragment") illustrate the relationships between basic blocks and LLVM's various kinds of fragments: data (DF), relaxable (RF), and alignment (AF). Data fragments are emitted by default, and may span consecutive basic blocks (e.g., BBL #1 and #2). The relaxable fragment #1 is required for the branch instruction, as it may be expanded during the relaxation phase. The padding bytes at the bottom correspond to a separate fragment, although they do not belong to any basic block.

Layout Information Initially, the range of the transformable area is identified, as shown in the left side of Figure 16. This area begins at the offset of the first object in the `.text` section and comprises all user-defined objects that can be shuffled. We modified LLVM to append a new section named `.rand` in every compiled object file so that the linker can be aware of which objects have embedded metadata. In our current prototype, we assume that all user-defined code is consecutive. Although it is possible to have intermixed code and data in the same section, we have ignored this case for now, as by default LLVM does not mix code and data when emitting `x86` code. This is the case for other modern compilers too—Andriesse et al. [139] could identify 100% of the instructions when disassembling GCC and Clang binaries (but CFG reconstruction still remains challenging).

When loading a program, a sequence of startup routines assist in bootstrap operations, such as setting up environment variables and reaching the first user-defined function (e.g., `main()`). As shown in Figure 16, the linker appends several object files from `libc` into the executable for this purpose (`crt1.o`, `cri.o`, `crtbegin.o`). Additional object files include process termination operations (`crtn.o`, `crtend.o`). Currently, these automatically-inserted objects are out of transformation—this is an implementation issue that can be easily addressed by ensuring that a set of augmented versions of these objects is made available to the compiler. At program startup, the function `_start()` in `crt1.o` passes five parameters to `__libc_start_main()`, which in turn invokes the program's `main()` function. One of the parameters corresponds to a pointer to `main()`, which we need to adjust after `main()` has been displaced.

The metadata we have discussed so far are updated at link time, according to the final layout of all objects. The upper part of Table 2 summarizes the collected layout-related metadata.

Basic Block Information The bulk of the collected metadata is related to the size and location of objects, functions, basic blocks (BBL), and fixups, as well as their relationships. For example, a fixup inherently belongs to a basic block, a basic block is a member of a function, and a function is included in an object. The LLVM backend goes through a very complex code generation process which involves all scheduled module and function passes for emitting globals, alignments, symbols, constant pools, jump tables, and so on. This

process is performed according to an internal hierarchical structure of *machine functions*, *machine basic blocks*, and *machine instructions*. The machine code (MC) framework of the LLVM backend operates on these structures and converts machine instructions into the corresponding target-specific binary code. This involves the `EmitInstruction()` routine, which creates a new chunk of code at a time, called a *fragment*.

As a final step, the assembler (`MCAssembler`) assembles those fragments in a target-specific manner, decoupled from any logically hierarchical structure—that is, the unit of the assembly process is the fragment. We internally label each instruction with the corresponding parent basic block and function. The collection process continues until instruction relaxation has completed, to capture the emitted bytes that will be written into the final binary. As part of the final metadata, however, these labels are not essential, and can be discarded. As shown in Table 2, we only keep information about the lower boundary of each basic block, which can be the end of an object (`OBJ`), the end of a function (`FUN`), or the beginning of the next basic block (`BBL`).

Going back to the example of Figure 16, we identify three types of *data*, *relaxable*, and *alignment* fragments, shown at the right side of the figure. The center of the figure shows the emitted bytes as generated by Clang, and their corresponding code as extracted by the IDA Pro disassembler, for the j-th function of the i-th object in the code section. The function consists of five basic blocks, eight fragments, and contains eleven fixups (underlined bytes).

As discussed in Section 5.2.2, relaxable fragments are generated only for branch instructions and contain just a single instruction. Alignment fragments correspond to padding bytes. In this example, there are two alignment fragments (#3 and #7): one between basic blocks #2 and #3, and one between function j and the following function. For metadata compactness, alignment fragments are recorded as part of the metadata for their preceding basic blocks. The rest of the instructions are emitted as part of data fragments.

Another consideration is *fall-through* basic blocks. A basic block terminated with a conditional branch implicitly falls through its successor depending on the evaluation of the condition. In Figure 16, the last instruction of BBL #0 jumps to BBL #2 when the zero flag is set, or control falls through to BBL #1. Such fall-through basic blocks must be marked so that they can be treated appropriately during reordering, as discussed in Section 5.3.4.

Table 2. Collected randomizaton-assisting metadata

Metadata	Collected Information	Collection time
Layout	Section offset to first object	Linking
	Section offset to `main()`	Linking
	Total code size for randomization	Linking
Basic Block (BBL)	BBL size (in bytes)	Linking
	BBL boundary type (BBL, FUN, OBJ)	Compilation
	Fall-through or not	Compilation
	Section name that BBL belongs to	Compilation
Fixup	Offset from section base	Linking
	Dereference size	Compilation
	Absolute or relative	Compilation
	Type (c2c, c2d, d2c, d2d)	Linking
	Section name that fixup belongs to	Compilation
Jump Table	Size of each jump table entry	Compilation
	Number of jump table entries	Compilation

Fixup Information Evaluating fixups and generating relocation entries are part of the last processing stage during layout finalization, right before emitting the actual code bytes. Note that this phase is orthogonal to the optimization level used, as it takes place after all LLVM optimizations and passes are done. Each fixup is represented by its offset from the section's base address, the size of the target (1, 2, 4, or 8 bytes), and whether it represents a relative or absolute value.

As shown in Table 2, we categorize fixups into four groups, similar to the scheme proposed by Wang et al. [203], depending on their location (source) and the location of their target (destination): code-to-code (c2c), code-to-data (c2d), data-to-code (d2c), and data-to-data (d2d). We define data as a universal region that includes all other sections except the `.text` section. This classification helps in increasing the speed of binary rewriting when patching fixups after randomization, as discussed in Section 5.3.4.

Jump Table Information Due to the complexity of some jump table code fragments, extra metadata needs to be kept for their correct handling during randomization. For non-PIC/PIE (position independent code/executable) binaries, the compiler generates jump table entries that point to targets using

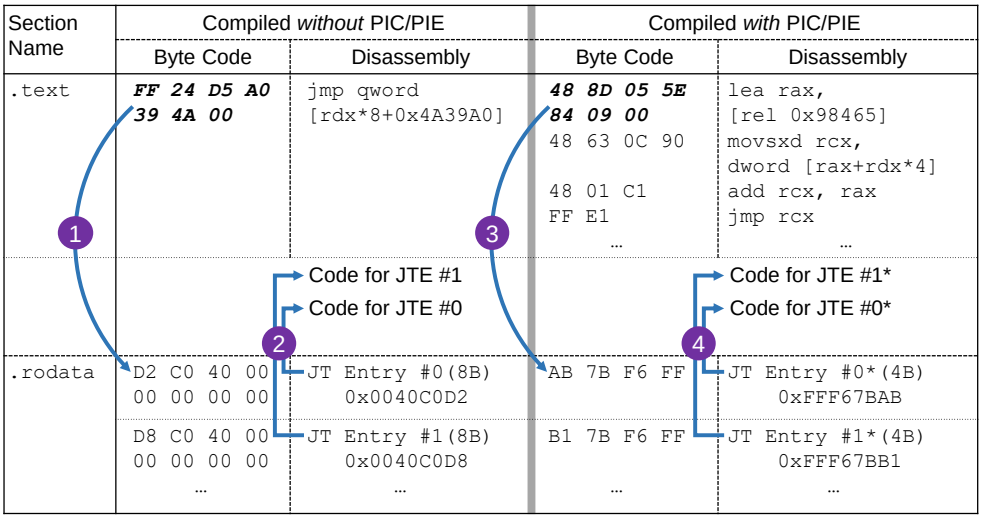


Figure 17. Example of jump table code generated for non-PIC and PIC binaries.

their absolute address. In such cases, it is trivial to update these destination addresses based on their corresponding fixups that already exist in the data section.

In PIC executables, however, jump table entries correspond to *relative* offsets, which remain the same irrespectively of the executable's load address. Figure 17 shows the code generated for a jump table when compiled without and with the PIC/PIE option. In the non-PIC case, the `jmp` instruction directly jumps to the target location ① by dereferencing the value of an 8-byte absolute address ② according to the index register `rdx`, as the address of the jump table is known at link time (`0x4A39A0`). On the other hand, the PIC-enabled code needs to compute the target with a series of arithmetic instructions. It first loads the base address of the jump table into `rax` ③, then reads from the table the target's relative offset and stores it in `rcx`, and finally computes the target's absolute address ④ by adding to the relative offset the table's base address.

To appropriately patch such jump table constructs, for which no additional information is emitted by the compiler, the only extra information we must keep is the number of entries in the table, and the size of each entry. This information is kept along with the rest of the fixup metadata, as shown in Table 2, because the relative offsets in the jump table entries should be updated

after randomization according to the new locations of the corresponding targets.

5.3.3 Link-time Metadata Consolidation

The main task of the linker is to merge multiple object files into a single executable. The linking process consists of three main tasks: constructing the final layout, resolving symbols, and updating relocation information. First, the linker maps the sections of each object into their corresponding locations in the final sections of the executable. During this process, alignments are adjusted and the size of extra padding for each section is decided. Then, the linker populates the symbol table with the final location of each symbol after the layout is finalized. Finally, it updates all relocations created by the assembler according to the final locations of those resolved symbols. These operations influence the final layout, and consequently affect the metadata that has already been collected at this point. It is thus crucial to update the metadata according to the final layout that is decided at link time.

Our CCR prototype is based on the GNU `gold` ELF linker that is part of `binutils`. It aims to achieve faster linking times compared to the GNU linker (`ld`), as it does not rely on the standard binary file descriptor (BFD) library. Additional advantages include lower memory requirements and parallel processing of multiple object files [204].

Figure 18 provides an overview of the linking process and the corresponding necessary updates to the collected metadata. Initially, the individual sections of each object are merged into a single one, according to the naming convention ①. For example, the two code sections `.text.obj1` and `.text.obj2` of the two object files are combined into a single `.text` section. Similarly, the metadata from each object is extracted and incorporated into a single section, and all addresses are updated according to the final layout ②.

As part of the section merging process, the linker introduces padding bytes between objects in the same section ③. At this point, the size of the basic block at the end of each object file has to be adjusted by increasing it according to the padding size. This is similar to the treatment of alignment bytes within an object file, which is considered as part of the preceding basic block (as discussed in Section 5.3.2). Note that we do not need to update anything related to whole functions or objects, as our representation of the

layout relies solely on basic blocks. Updating the size of the basic blocks that are adjacent to padding bytes is enough for deriving the final size of functions and objects.

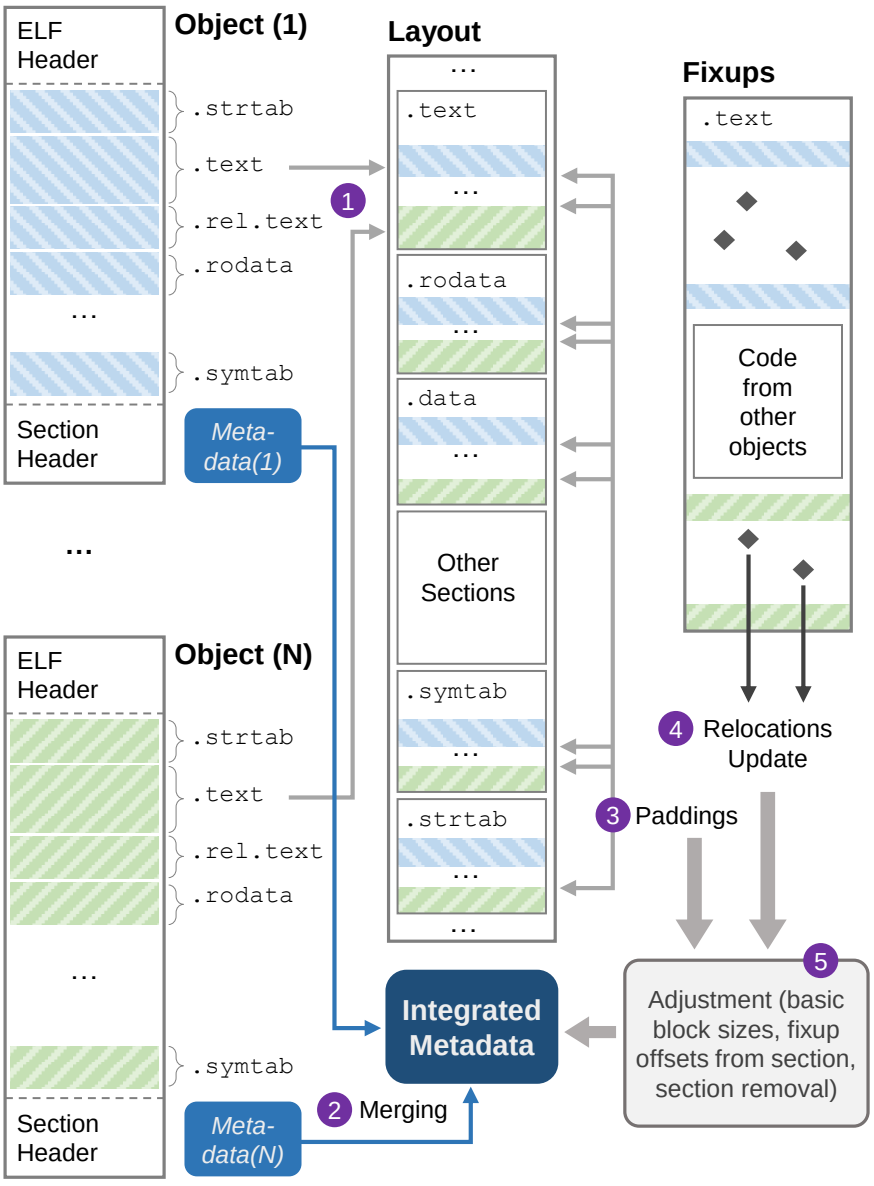


Figure 18. Overview of the linking process. Per-object metadata is consolidated into a single section.

Once the layout is finalized and symbols are resolved, the linker updates the relocations recorded by the assembler ④. Any fixups that were already resolved at compilation time are not available in this phase, and thus the corresponding metadata remains unchanged, while the rest is updated accordingly. Finally, the aggregation of metadata is completed ⑤ by updating the binary-level metadata discussed in Section 5.3.2, including the offset to the first object, the total code size for transformation, and the offset to the main function (if any).

A special case that must be considered is that a single object file may contain multiple `.text`, `.rodata`, `.data` or `.data.rel.ro` sections. For instance, C++ binaries often have several code and data sections according to a name mangling scheme, which enables the use of the same identifier in different namespaces. The compiler blindly constructs these sections without considering any possible redundancy, as it can only process the code of a single object file at a time. In turn, when the linker observes redundant sections, it nondeterministically keeps one of them and discards the rest [205]. This deduplication process can cause discrepancies in the layout and fixup information kept as part of our metadata, and thus the corresponding information about all removed sections is discarded at this stage. This process is facilitated by the section name information that is kept for basic blocks and fixups during compilation. Note that section names are optional attributes required only at link time. Consequently, after deduplication has completed, any remaining section name information about basic blocks and fixups is discarded, further reducing the size of the final metadata.

5.3.4 Code Randomization

To strike a balance between performance and randomization entropy, we have opted to maintain some of the constraints imposed by the code layout decided at link time, due to short fixup sizes and fall-through basic blocks. As mentioned earlier, these constraints can be relaxed by modifying the width of short branches and adding new branches when needed. However, our current choice has the simplicity and performance benefit of keeping the total size of code the same, which helps in maintaining caching characteristics due to spatial locality. To this end, we prioritize basic block reordering at intra-function level, and then proceed with function-level reordering.

Distance constraints due to fixup size may occur in both function and basic block reordering. For instance, it is typical for functions to contain a short fixup that refers to a *different* function, as part of a jump instruction used for tail-call optimization. At the rewriting phase, basic block reordering proceeds without any constraints if: (a) the parent function of a basic block does not have any distance-limiting fixup, or (b) the size of the function allows reaching all targets of any contained short fixups. Note that the case of multiple functions sharing basic blocks, which is a common compiler optimization, is fully supported.

From an implementation perspective, the simplest solution for fall-through basic blocks is to assume that both child blocks will be displaced away, in which case an *additional* jump instruction must be inserted for the previously fall-through block. From a performance perspective, however, a better solution is to avoid adding any extra instructions and keep *either* of the two child basic blocks adjacent to its parent—this can be safely done by inverting the condition of the branch when needed. In our current implementation we have opted for this second approach, but have left branch inversion as part of our future work. As shown in Section 5.5.5, this decision does not impact the achieved randomization entropy.

After the new layout is available, it is essential to ensure fixups are updated accordingly. As discussed in Section 5.3.2, we have classified fixups into four categories: c2c, c2d, d2c and d2d. In case of d2d fixups, no update is needed because we diversify only the code region, but we still include them as part of the metadata in case they are needed in the future. The dynamic linking process relies on c2d (relative) fixups to adjust pointers to shared libraries at runtime.

5.4 Implementation

Our CCR prototype supports ELF executables for the Linux x86-64 platform. To augment binaries, we modified LLVM/Clang v3.9.0 [202] and the `gold` linker v2.27 of GNU Binutils [206]. At the user side, binary executable randomization is performed by a custom binary rewriter that leverages the embedded metadata. In this section, we discuss the main modifications that were required in the compiler and linker, and the design of our binary rewriter. We encountered many challenges and pitfalls in our attempt to maintain

compatibility with advanced features such as inline assembly, lazy binding, exception handling, link-time optimization, and additional protections like control flow integrity.

5.4.1 Compiler

In our attempt to modify the right spots in LLVM for collecting the necessary metadata, we encountered several challenges. First, as explained in Section 5.3.2, the assembler operates on an entirely separate view based on fragments and sections, compared to the logical view of basic blocks and functions. For this reason, we had to modify the LLVM backend itself, rather than writing an LLVM pass, which would be more convenient, as LLVM offers a flexible interface for implementing optimizations and transformations.

Second, recall that fine-grained randomization necessitates absolute accuracy when it comes to basic block sizes. A single misattributed byte can result in the whole code layout being incorrect. In this regard, obtaining the exact size of each instruction is important for deriving the right sizes of both its parent basic block and function. In our implementation, extracting this information relies on labeling the parents of each and every instruction. However, we encountered several cases of instructions not belonging to any basic block. For example, sequences like `cld; rep stos;` may appear without any parent label. These are handled by including the instructions as part of the basic block of the previous instruction.

5.4.2 Linker

The linker performs several operations that influence considerably the final binary layout, and many of them required special consideration. First, there are cases of object files with zero size, e.g., when a source code file contains just a definition of a structure, without any actual code. Interestingly, such objects result in padding bytes that must be carefully accounted for when randomizing the last basic block of an object. Besides removing the metadata for redundant sections due to the deduplication process (discussed in Section 5.3.3), there are other sections that require special handling. These include `.text.unlikely`, `.text.exit`, `.text.startup`, and `.text.hot`, which the GNU linker handles differently for compatibility purposes. The special sections have unique features including independent positions (ahead of all

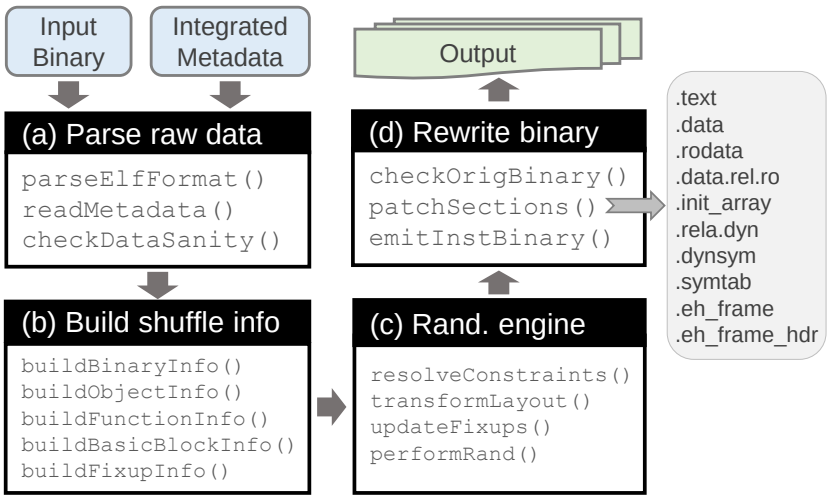


Figure 19. Overview of the rewriting process. The rewriter parses the augmented ELF binary (a) and organizes all information required for randomization in a tree data structure (b). Randomization is performed based on this structure (c), and the new layout is then written into the final binary (d).

other code) and redundant section names within a single object file (i.e., multiple `.text.startup` sections), resulting in non-consecutive user-defined code in the `.text` section that must be precisely captured as part of our metadata for randomization to function properly.

5.4.3 Binary Rewriter

We developed our custom binary rewriter in Python, and used the `pyelftools` library for parsing ELF files [207]. The rewriter takes the augmented ELF executable to be randomized as its sole input. The core randomization engine is written in ∼2KLOC. The simple nature of the rewriter makes it easy to be integrated as part of existing software installation workflows. In our prototype, we have integrated it with Linux's `apt` package management system through `apt`'s wrapper script functionality.

As illustrated in Figure 19, binary rewriting comprises four phases. Initially, the ELF binary is parsed and some sanity checks are performed on the extracted metadata. We employ Protocol Buffers for metadata serialization, as they provide a clean, efficient, and portable interface for structured data

streams [208]. To minimize the overall size of the metadata, we use a compact representation by keeping only the minimum amount of information required. For example, as discussed in Section 5.3.2, basic block records denote whether they belong to the end of a function or the end of an object (or both), without keeping any extra function or object information per block. The final metadata is stored in a special `.rand` section, which is further compressed using `zlib`. Next, all information regarding the relationships between objects, functions, basic blocks, and fixups is organized in an optimized data structure, which the randomization engine uses to transform the layout, resolve any constraints, and update target locations.

5.4.4 Exception Handling

Our prototype supports the exception handling mechanism that the `x86_64 ABI` [209] has adopted, which includes stack unwinding information contained in the `.eh_frame` section. This section follows the same format as the `.debug_frame` section of DWARF [210], which contains metadata for restoring previous call frames through certain registers. It consists of one or more subsections, with each forming a single CIE (Common Information Entry) followed by multiple FDEs (Frame Descriptor Entry). Every FDE corresponds to a function in a compilation unit. One of the FDE fields describes the `initial_loc` of the function that holds the relative address of the function's entry instruction, which requires to be patched during the rewriting phase.

As shown in Figure 20, the range an FDE corresponds to is determined by both the `inital_loc` and `address_range` fields. Additionally, `.eh_frame_hdr` contains a table of tuples (`inital_loc`, `fde_pointer`) for quickly resolving frames. Because these tuples are sorted according to each function's location, the table must be updated to factor in our transformations. Note that our rewriter parses the exception handling sections directly, with no additional information.

Our current CCR prototype does not support randomization with custom exception handling at the basic block level (custom exception handling is fully supported for function-level randomization). As mentioned above, the `.eh_frame` section contains a compact table with entries corresponding to possible instruction addresses in the program. The exception handling mecha-

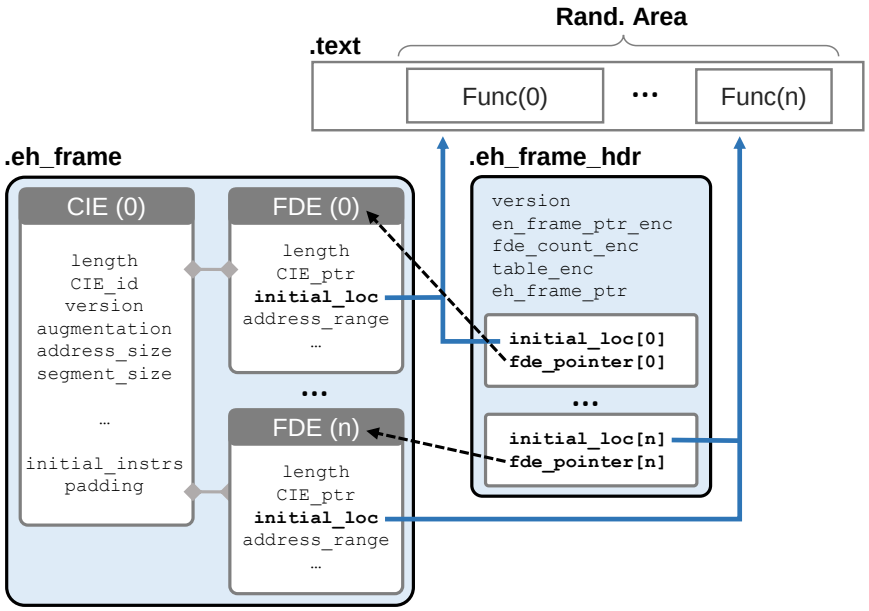


Figure 20. Structure of an `.eh_frame` section for exception handling. Bold fields must be updated after transformation according to the encoding type specified in the `.eh_frame_hdr` section.

nism triggers a pre-defined instruction sequence, written in a domain-specific (debugger) language. For example, `DW_CFA_set_loc N` means that the next instructions apply to the first N bytes of the respective function (based on its location). Each FDE may trigger a series of instructions, including the ones in a language-specific data area (LSDA), such as `.gcc_except_table` (if defined), for properly unwinding the stack. To fully support this mechanism, the LSDA instructions should be updated according to the new locations of a functions' basic blocks. We plan to support this feature in future releases of our framework.

5.4.5 Link-Time Optimization (LTO)

Starting with v3.9, LLVM supports link-time optimization [211] to allow for inter-module optimizations at link time.[3] Enabling LTO generates a

[3]Either `ld.bfd` or `gold` is needed, configured with plugin support [212]. LLVM's LTO library (`libLTO`) implements the plugin interface to interact with the linker, and is invoked

non-native object file (i.e., an LLVM bitcode file), which prompts the linker to perform optimization passes on the merged LLVM IR. Our toolchain interposes at LTO's instruction lowering and linking stage to collect the appropriate metadata of the final optimized code.

5.4.6 Control Flow Integrity (CFI)

LLVM's CFI protection [106] offers six different integrity check levels, which are available only when LTO is enabled.[4] The first five levels are implemented by inserting sanitization routines, while the sixth (`cfi-icall`) relies, among other mechanisms, on function trampolines. Our current CCR prototype supports the first five modes, but not the sixth one, because the generated trampolines at call sites are internally created by LLVM using a special intrinsic,[5] rendering their boundaries unknown.

5.4.7 Inline assembly

The LLVM backend has an integrated assembler (`MCAssembler`) that emits the final instruction format, which is internally represented by an `MCInst` instance. In general, the instruction lowering process includes the generation of `MCInst` instances. Fortunately, the LLVM assembly parser (`AsmParser`) independently takes care of emitting `MCInst` information also in case of inline assembly, which allows us to tag the parents of all embedded instructions generated from the parser. Moreover, the assembler processes instruction relaxation for inline assembly as needed.

5.5 Experimental Evaluation

We evaluated our CCR prototype in terms of runtime overhead, file size increase, randomization entropy, and other characteristics. Our experiments

by `clang` with the `-flto` option.

[4]Applying CFI requires the `-flto` option at all times. Additionally, both `-fsanitize=cfi-{vcall,nvcall,cast-strict,derived-cast, unrelated-cast, icall}` and `-fvisibility={default,hidden}` flags should be provided to `clang`.

[5]LLVM's `llvm.type.test` intrinsic tests if the given pointer and type identifier are associated.

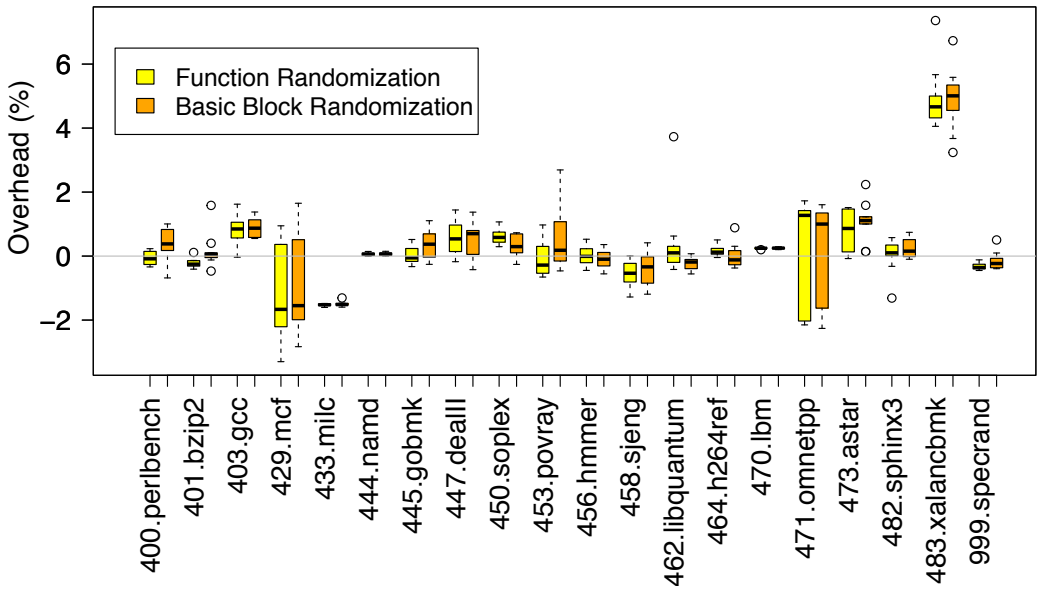


Figure 21. Performance overhead of fine-grained (function vs. basic block reordering) randomization for the SPEC CPU2006 benchmark tests.

were performed on a system equipped with an Intel i7-7700 3.6GHz CPU, 32GB RAM, running the 64-bit version of Ubuntu 16.04.

5.5.1 Randomization Overhead

We started by compiling the entire SPEC CPU2006 benchmark suite (20 C and C++ programs) with our modified LLVM and `gold` linker, using the `-O2` optimization level and without the PIC option. Next, we generated 20 different variants of each program, 10 using function reordering and 10 more using function and basic block reordering. Each run was performed 10 times for the original programs, and a single time for each of the 20 variants.

Figure 21 shows a boxplot of the runtime overhead for function reordering and basic block reordering. The dark horizontal line in each box corresponds to the median overhead value, which mostly ranges between zero and one across all programs. The top and bottom of each box correspond to the upper and lower quartile, while the whiskers to the highest and lowest value, excluding outliers, which are denoted by small circles (there were 14 such cases out of the total 400 variants, exhibiting an up to 7% overhead). Overall,

the average performance overhead is negligible at 0.28%, with a 1.37 standard deviation. The average overhead per benchmark is reported in Table 4, which also includes further information about the layout and fixups of each program.

Interesting cases are `mcf` and `milc`, the variants of which consistently exhibit a slight performance improvement, presumably due to better cache locality (we performed an extra round of experiments to verify it). In contrast, `xalancbmk` exhibited a distinguishably high average overhead of 4.9%. Upon further investigation, we observed a significant increase in the number of L1 instruction cache misses for its randomized instances. Given that `xalancbmk` is one of the most complex benchmarks, with a large number of functions and heavy use of indirect control transfers, it seems that the disruption of cache locality due to randomization has a much more pronounced effect. For such cases, it may be worth exploring profile-guided randomization approaches that will preserve the code locality characteristics of the application.

5.5.2 ELF File Size Increase

Augmenting binaries with additional metadata entails the risk of increasing their size at levels that may become problematic. As discussed earlier, this was an issue that we took into consideration when deciding what information to keep, and optimized the final metadata to include only the minimum amount of information necessary for code diversification.

As shown in Table 4, file size increase ranges from 1.68% to 20.86%, with an average of 11.46% (13.3% for the SPEC benchmarks only). We consider this a rather modest increase, and do not expect it to have any substantial impact to existing software distribution workflows. The Layout columns (Objs, Funcs, BBLs) show the number of object files, functions, and basic blocks in each program. As expected, the metadata size is proportional to the size of the original code. Note that the generated randomized variants do not include any of the metadata, so their size is the same as the original binary.

5.5.3 Binary Rewriting Time

We measured the rewriting time of our CCR prototype by generating 100 variants of each program and reporting the average processing time. We repeated the experiment twice, using function and basic block reordering,

Table 3. Applications used for correctness testing

Application	Tested Functionality
ctags-5.8	Index a large corpus of source code
gzip-1.8	Compress and decompress a large file
oggenc-1.0.1	Encode a WAV file to OGG format
putty-0.67	Connect to a remote server through the terminal
lighttpd-1.4.45	Start the server and connect to the main page
miniweb	Start the server and connect to the main page
opensshd-7.5	Start an SSH server and accept a connection
vsftpd-3.0.3	Start an FTP server and download a file
libcapstone-3.0.5	Test a disassembly in various platforms
dosbox-0.74	Run an old DOS game within the emulator

respectively. As shown in Table 4 (Rewriting columns) the rewriting process is very quick for small binaries, and the processing time increases linearly with the size of the binary. The longest processing time was observed for `xalancbmk`, which is the largest and most complex (in terms of number of basic blocks and fixups) among the tested binaries. All but four programs were randomized in under 9s, and more than half of them in under 1s.

The reported numbers include the process of updating the debug symbols present in the `.symtab` section. As this is not needed for production (stripped) binaries, the rewriting time in practice will be shorter—indicatively, for `xalancbmk`, it is 30% faster when compiled without symbols. Note that our rewriter is just a proof of concept, and further optimizations are possible. Currently, the rewriting process involves parsing the raw metadata, building it into a tree representation, resolving any constraints in the randomized layout, and generating the final binary. We believe that the rewriting speed can be further optimized by improving the logic of our rewriter's randomization engine. Moving from Python to C/C++ is also expected to increase speed even further.

5.5.4 Correctness

To ensure that our code transformations do not affect in any way the correctness of the resulting executable, in addition to the SPEC benchmarks, we

compiled and tested the augmented versions of ten real-world applications. For example, we parsed the entire LLVM source code tree with a randomized version of `ctags` using the `-R` (recursive) option. The MD5 hash of the resulting index file, which was 54MB in size, was identical to the one generated using the original executable. Another experiment involved the command-line audio encoding tool `oggenc`—a large and quite complex program (58,413 lines of code) written in C [213]—to convert a 44MB WAV file to the OGG format, which we then verified that was correctly processed. Furthermore, we successfully compiled popular server applications (web, FTP, and SSH daemons), confirming that their variants did not malfunction when using their default configurations. Application versions and the exact type of activity used for functionality testing are provided in Table 3.

Table 4. Experimental evaluation dataset and results (* indicates programs written in C++)

Program	Layout			Fixups				Size (KB)			Rewriting (sec)		Overhead		Entropy (log_{10})	
	Objs	Funcs	BBLs	.text	.rodata	.data	.init_ar.	Orig.	Augm.	Increase	Func	BBL	Func	BBL	Func	BBL
400.perlbench	50	1,660	46,732	70,653	7,872	1,765	0	1,198	1,447	20.86%	7.69	8.05	-0.07%	0.32%	4,530	5,011
401.bzip2	7	71	2,407	2,421	75	0	0	90	101	12.80%	0.19	0.21	-0.23%	0.16%	100	157
403.gcc	143	4,326	118,397	189,543	84,357	367	0	3,735	4,465	19.54%	52.30	53.89	0.82%	0.91%	13,657	16,483
429.mcf	11	24	375	410	0	0	0	22	25	12.02%	0.08	0.09	-1.27%	-0.98%	23	44
433.milc	68	235	2,613	5,980	50	36	0	148	170	14.94%	0.48	0.50	-1.53%	-1.50%	456	600
444.namd*	23	95	7,480	8,170	24	0	0	312	345	10.49%	0.50	0.56	0.06%	0.07%	148	187
445.gobmk	62	2,476	25,069	44,136	1,377	21,400	0	3,949	4,116	4.23%	21.28	20.43	0.05%	0.35%	7,272	8,271
447.dealII*	6,295	6,788	100,185	103,641	7,954	1	45	4,217	4,581	8.65%	38.08	39.18	0.60%	0.52%	23,064	25,601
450.soplex*	299	889	13,741	15,586	1,561	0	61	467	531	13.76%	1.90	1.99	0.60%	0.28%	2,234	2,983
453.povray*	110	1,537	28,378	47,694	10,398	617	1	1,223	1,406	14.92%	5.67	5.88	-0.08%	0.50%	4,130	4,939
456.hmmer	56	470	10,247	14,265	798	156	0	343	400	16.53%	1.14	1.19	0.00%	-0.11%	1,042	1,313
458.sjeng	119	132	4,469	8,978	431	0	0	155	186	19.93%	0.50	0.53	-0.55%	-0.38%	221	334
462.libquantum	16	95	1,023	1,373	319	0	0	55	62	13.57%	0.19	0.19	0.40%	-0.24%	148	207
464.h264ref	42	518	14,476	23,180	320	321	0	698	782	12.01%	1.97	2.06	0.17%	0.00%	1,180	1,468
470.lbm	2	17	133	227	0	0	0	22	24	8.15%	0.06	0.06	0.25%	0.25%	14	24
471.omnetpp*	366	1,963	22,118	34,212	3,411	240	75	843	952	12.95%	4.73	4.94	0.03%	0.25%	5,560	6,983
473.astar*	14	88	1,116	1,369	6	1	0	56	62	12.03%	0.17	0.17	0.78%	1.08%	134	169
482.sphinx3	44	318	5,557	9,046	26	207	0	213	249	16.54%	0.68	0.72	0.02%	0.23%	656	815
483.xalancbmk*	3,710	13,295	130,691	142,128	19,936	323	0	6,217	6,836	9.95%	88.09	89.94	4.92%	4.89%	48,863	61,045
999.specrand	2	3	11	32	0	0	0	8	9	11.07%	0.03	0.03	-0.32%	-0.15%	0.8	1.6
ctags	50	423	8,550	13,618	3,733	507	0	795	851	7.03%	1.17	1.21	-	-	915	1,095
gzip	34	103	2,895	5,466	466	21	0	267	289	8.13%	0.40	0.41	-	-	164	194
lighttpd	50	351	5,817	9,169	818	98	0	866	903	4.23%	0.96	0.99	-	-	732	891
miniweb	7	67	1,322	1,681	65	74	0	56	64	14.54%	0.19	0.19	-	-	94	113
oggenc	1	428	7,035	7,746	183	3,869	0	2,120	2,156	1.68%	2.79	2.74	-	-	942	2,285
openssh	122	1,135	18,262	29,815	2,442	90	0	2,144	2,248	4.83%	4.04	4.17	-	-	3,398	3,856
putty	79	1,288	20,796	31,423	3,126	118	0	1,069	1,184	10.78%	3.71	3.82	-	-	2,927	3,610
vsftpd	39	516	3,793	7,148	74	0	0	138	163	18.48%	0.65	0.67	-	-	1,147	1,227
libcapstone	42	402	21,454	47,299	13,002	5	0	2,777	2,931	5.69%	10.64	11.31	-	-	863	1,040
dosbox*	630	3,127	66,522	124,814	14,906	2,585	18	11,729	12,145	3.54%	37.59	38.12	-	-	9,503	10,941

5.5.5 Randomization Entropy

We briefly explore the randomization entropy that can be achieved using function and basic block reordering, when considering the current constraints of our implementation. Let F_{ij} be the jth function in the ith object, f_i the number of functions in that object, and b_{ij} the number of basic blocks in the function F_{ij}. Suppose there are p object files comprising a given binary executable. The total number of functions q and basic blocks r in the binary can be written as $q = \sum_{i=0}^{p-1} f_i$ and $r = \sum_{i=0}^{p-1} \sum_{j=0}^{f_i-1} b_{ij}$. Then, the number of possible variants with function reordering is $q!$ and with basic block reordering is $r!$. Due to the large number of variants, let the randomization entropy E be the base 10 logarithm of the number of variants. In our case, we perform basic block randomization at intra-function level first, followed by function reordering. Therefore, the entropy can be computed as follows:

$$E = \log_{10}(\prod_{i=0}^{p-1}(\prod_{j=0}^{f_i-1} b_{ij}!) \cdot (\sum_{i=0}^{p-1} f_i)!)$$

However, as discussed in Section 5.3.4, our current implementation has some constraints regarding the placement of functions and basic blocks. Let the number of such function constraints in the ith object be y_i. Likewise, fall-through blocks are currently displaced together with their previous block. Similarly to functions, in some cases the size of a fixup also constrains the maximum distance to the referred basic block. Let the number of such basic block constraints in function F_{ij} be x_{ij}. Given the above, the entropy in our case can be calculated as:

$$E = \log_{10}(\prod_{i=0}^{p-1}(\prod_{j=0}^{f_i-1} (b_{ij} - x_{ij})!) \cdot (\sum_{i=0}^{p-1} (f_i - y_i))!)$$

Using the above formula, we report the randomization entropy for function and basic block level randomization in Table 4. We observe that even for small executables like `lbm`, the number of variants exceeds 300 trillion. Consequently, our current prototype achieves more than enough entropy, which can be further improved by relaxing the above constraints (e.g., by separating fall-through basic blocks from their parent blocks, and adding a relaxation-like phase in the rewriter to alleviate existing fixup size constraints).

5.6 Limitations

Our prototype implementation demonstrates the feasibility of CCR by enabling practical fine-grained code randomization (basic block reordering) on a popular platform (x86-64 Linux). There are, of course, several limitations, which can be addressed with additional engineering effort and are part of our future work.

First, individual assembly source code files (`.s`) are currently not supported. Note that assembly code files differ from inline assembly (which is fully supported), in that their processing by LLVM is not part of the standard abstract syntax tree and intermediate representation workflow, and thus corresponding function and basic block boundaries are missing during compilation. Still, symbols for functions contained in `.s` files are available, and we plan to include this information as part of the collected metadata.

Second, any use of self-modifying code is not supported, as the self-modification logic should be changed to account for the applied randomization. In such cases, compatibility can still be maintained by excluding (i.e., "pinning" down) certain code sections or object files from randomization, assuming all their external dependencies are included.

A slightly more important issue is fully updating all symbols contained in the debug sections according to the new layout after rewriting. Our current CCR prototype does update symbol table entries contained in the `.symtab` section, but it does not fully support the ones in the `.debug_*` sections. Although in practice the lack of full debug symbols is not a problem, as these are typically stripped off production binaries, this is certainly a useful feature to have. In fact, we were prompted to start working on resolving this issue because the lack of correct debug symbols for newly generated variants hindered our debugging efforts during the development of our prototype.

Finally, our prototype does not support programs with custom exception handling when randomization at the basic block level is used (this is not an issue for function-level randomization). No additional metadata is required to support this feature (just additional engineering effort). Further details about exception handling are provided in the Appendix.

5.7 Discussion

Other types of code hardening Basic block reordering is an impactful code randomization technique that ensures that no ROP gadgets remain in their original locations, even *relatively* to the entry point of the function that includes them—an important aspect for defending against (indirect) JIT-ROP attacks that rely on code pointer leakage [121–123]. For a function that consists of just a single basic block, however, the relative distance of any gadgets from its entry point still remains the same. This issue can be trivially addressed by modifying our rewriter to insert a (varying) number of NOPs or junk instructions at the beginning of the function [51]. Other more narrow-scope transformations, such as instruction substitution, intra basic block instruction reordering, and register reassignment [53, 62] can also be supported effortlessly, since our metadata provides precise knowledge about the boundaries of basic blocks. In fact, we have started leveraging such metadata for augmenting our rewriter with *agile* hardening capabilities: that is, strip (or not) hardening instrumentation (e.g., CFI [106], XOM [51]) based on where the target application is going to be deployed, thereby enabling precise and targeted protection.

Defending against more sophisticated attacks that rely on whole-function reuse [31,32,123,128,129,214] requires more aggressive transformations, such as code pointer indirection [48,125,215,216] or function argument randomization. We leave the exploration of how our metadata could be extended to facilitate such advanced protections as part of future research.

Error reporting, whitelisting, and patching One of the main benefits of code randomization based on compiler–rewriter cooperation is that it allows for maintaining compatibility with operations that rely on software uniformity, which currently is a major roadblock for its practical deployment. By performing the actual diversification on endpoints, any side-effects that hinder existing norms can be reversed.

For instance, a crash dump of a diversified process can be post-processed right after it is generated so that code addresses are changed to refer to the original code locations of the master binary that was initially distributed (otherwise, it will be of no use to its developers). Similarly, code integrity checking and whitelisting mechanisms can be modified to de-randomize the

in-memory or on-disk code before actually verifying it. This randomization reversal process can be supported by including a randomization seed within each variant (which in conjunction with the original metadata will provide all the necessary information for the task) [53]. The seed can be kept as part of the on-disk binary (i.e., it does not need to exist in memory), to prevent attackers from getting any extra information about the randomized layout, e.g., through a memory disclosure vulnerability.

Code *signing* does not require any modification, since master binaries can continue to be signed normally before distribution. At the client side, the binary rewriter can proceed only after verifying the signature. Binary-level software patching is also not significantly affected. Patches can continue to be released in the same way as before, based on the master binary. At the client side, the patch can be applied on the master binary, and then a new (updated) variant can be generated.

Intellectual property As an outcome of the compilation process, most of the high-level programming language structure and semantics are lost from the resulting binary code. Especially for proprietary software, the inherent complexity of code disassembly combined with the lack of symbolic information (and the potential use of code obfuscation) can hinder significantly any attempts of reconstructing the original code semantics through binary code disassembly, control flow graph extraction, and decompilation.

The metadata needed to facilitate code randomization can certainly aid in extracting a more accurate view of the assembly code and the control flow graph of a binary, but does not convey any new symbolic information that would help in extracting higher-level program semantics (function and variable names aid reverse engineering significantly). We do not consider this issue a major concern, as vendors who care about protecting their intellectual property against reverse engineering rely on more aggressive code obfuscation techniques (e.g., software packing or instruction virtualization). Alternatively, parts of code or whole modules for which such concerns apply can be excluded so that no additional metadata is kept for them.

Availability

Our prototype open-source implementation is available at:
`https://github.com/kevinkoo001/CCR`.

6 Configuration-Driven Software Debloating

6.1 Background

Applications often allow users to specify initial settings, options, parameters, and other features by editing a separate configuration file (typically in ASCII format). The types of configuration directives vary across different programs. In general, a directive consists of a variable and a single or multiple values that can be assigned to it. Of particular interest for our purposes are directives associated with specific functionality that is carried out by a standalone library—when such a directive is disabled, then the corresponding library could be completely removed.

Listing 1 shows a complete instance of an Nginx configuration. Although Nginx supports 724 different configuration options from 85 components, a simple configuration like this suffices for a basic web service. The comments highlight directives that are associated with certain libraries. For example, the `gzip` directive at line 19 is associated with the `libz.so` library. As it is specified under the `server` structure, the `gzip` directive applies to all `location` sub–structures. Similarly, the `image_filter` and `rewrite` directives in lines 26 and 27 result in the loading of a graphic library (`libgd.so`) and regular expression library (`libpcre.so`), respectively. In some cases, multiple directives have to be defined to enable a certain capability, such as the SSL-related directives in lines 18, 20, and 21.

6.2 Configuration-Driven Code Debloating

In this section, we describe a novel technique to find unused functionalities at the module level.

Removing the code that will remain unused according to a given configuration without breaking the functionality of the program requires addressing two main requirements. First, the code that is related to a particular configuration directive needs to be precisely identified. Second, the rest of the code must be analyzed to ensure that it does not depend on the code that will be removed if that particular directive is disabled. Both of the above requirements are quite challenging to address in an automated and exhaustive way. An ideal

```
# /etc/nginx/nginx.conf
worker_processes  1;
error_log  /var/log/nginx/error.log;

events { worker_connections  1024; }

http {
  include  mime.types;
  index    default.html default.htm;
  default_type application/octet-stream;

  access_log    /usr/local/nginx/logs/nginx.pid;
  geoip_country /usr/local/nginx/conf/GeoIP.dat; #libGeoIP.so
  charset UTF-8;
  keepalive_timeout 65;

  server {
    listen 443 ssl;                    #libssl.so
    gzip   on;                         #libz.so
    ssl_certificate  cert.pem;         #libssl.so
    ssl_certificate_key cert.key;      #libssl.so

    location / {
      root   /var/www/hexlab;
      index  default.php;
      image_filter resize 150 100;     #libgd.so
      rewrite ^(.*)$ /msie/$1 break; #libpcre.so
    }

    location /test {
      xml_entities    /var/www/hexlab/entities.dtd; #libxml2.
          so
      xslt_stylesheet /var/www/hexlab/one.xslt;     #libxslt.
          so
    }
  }
}
```

Listing 1: Example of an Nginx configuration file.

approach would take a configuration directive as input, automatically identify all associated code, and extract the subset of that code that is not needed by the rest of the program when this particular functionality is disabled. This may be feasible using a combination of control and data flow analysis, but before investigating such a complex solution, our goal in this work is to derive a first estimate of the attack surface reduction potential that such a configuration-driven debloating scheme would offer.

Unneeded code removal can be performed at different levels of granularity, e.g., at the instruction, function, or library level. For instance, prior works remove the functions that are not imported (and thus not used) from the libraries linked to the main executable [79, 80]. Given the complexity of identifying all functions that are exclusively needed by a given configuration directive, in this work we decided instead to aim for deriving a lower bound, and perform code debloating at the *library* level. The intuition behind this decision is that many types of functionality that can be enabled or disabled through configuration directives are often carried out by third-party libraries. For instance, as shown in the Nginx configuration example of Section 6.1, if content compression is needed, then this will be performed by the `libz.so` library.

To pinpoint the libraries that are exclusively associated with a given configuration directive, we perform differential testing using a combination of static and dynamic analysis. The directives to be analyzed, as well as appropriate test inputs for driving the execution of the application during dynamic analysis, are manually selected after studying the configuration documentation of the application in conjunction with observing which libraries are loaded. For this work, we focused on server applications (Nginx, VSFTPD, and OpenSSH), as they typically require at least some minimal configuration specification for proper operation.

Figure 22 shows an overview of our approach. First, we compile the program by enabling code coverage profiling, and run it twice, with a given directive enabled and disabled. By comparing the two code coverage reports, we then pinpoint any extra library code that was exercised only when the directive was enabled. This allows us to derive an initial mapping between directives and libraries, which is then refined in a second dependency analysis step, which builds the dependency graph across all libraries in the program and identifies any dependencies that are exclusive to a given library. Finally, a

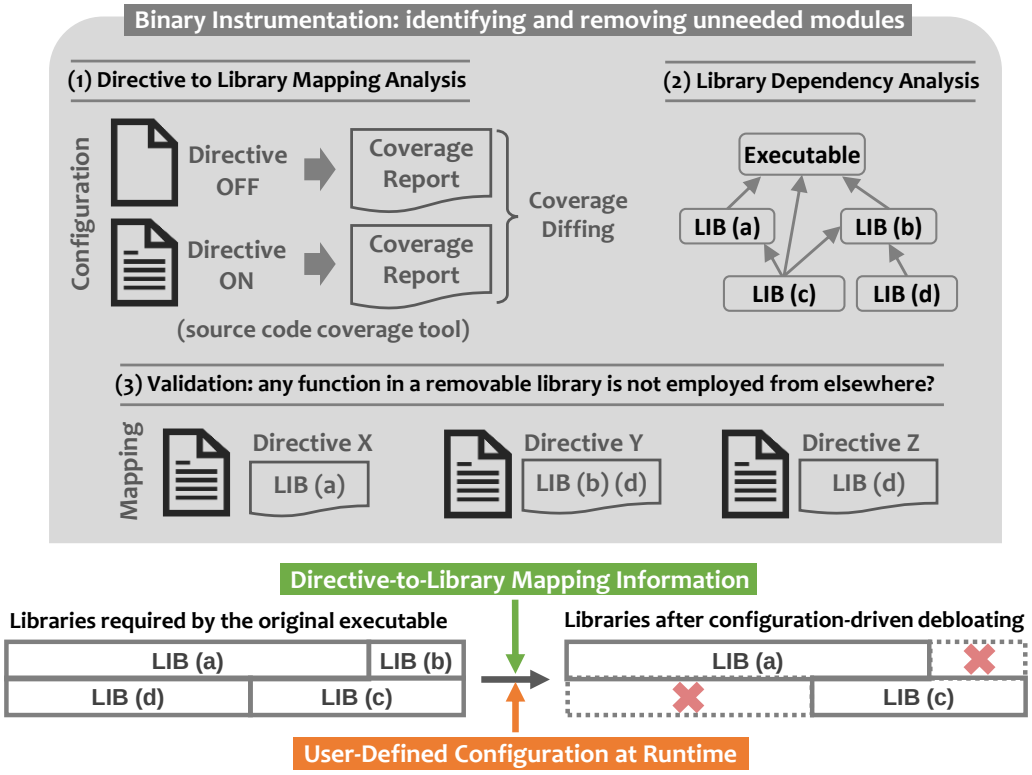


Figure 22. Overview of the configuration-driven code debloating process.

static analysis step analyzes the whole program and verifies that the identified directive-dependent libraries are not used by any other part of the program. The whole analysis process is performed only once per configuration directive. The resulting directive-to-library mapping information can then be used as input for instrumenting the main executable to avoid loading any unneeded libraries for which the corresponding directive is disabled.

6.2.1 Mapping Directives to Libraries

To identify the libraries that are exclusively used by certain configuration directives, we perform differential testing by comparing the source code coverage during execution with and without a given directive. Our technique relies on the LLVM source code coverage tool (`llvm-cov` [217]), to identify the exercised code for a given combination of configuration directives and

test inputs.

Before testing a given directive, we have to manually prepare i) the specially crafted configuration file that enables or disables the functionality of interest, and ii) a set of appropriate program inputs to ensure that the feature of interest will be invoked. We initially specify the simplest configuration with all directives to be tested disabled (i.e., commented out) as our base configuration. We then generate one configuration per directive (or set of directives) that enables a particular feature or functionality that is likely to be carried out by a specific library (or set of libraries).

We have implemented an analysis tool that automatically enables and disables the directive(s) for a given feature, runs the application with the appropriate test cases, and maps the directive(s) into one or more libraries. Comparing the two code coverage reports allows us to pinpoint the extra code that is associated with the tested functionality when the corresponding directive is enabled, and thus the associated libraries. For example, consider `libgd.so`, which is used by Nginx for image manipulation—a functionality that is supported by Nginx, but is *disabled* by default. When Nginx runs without this feature (the default case), we can safely exclude `libgd.so` from being loaded, as no other part of the code relies on it.

6.2.2 Library Dependence and Validation

The list of candidate libraries for removal from the mapping phase must be validated by checking whether i) other libraries have any dependencies from the candidate directive-related libraries, and ii) the rest of the program still uses any functions from the candidate directive-related libraries.

The first case can be easily handled by statically analyzing the imports of the main executable and all dynamic libraries. By building a library dependence graph, we can identify additional libraries that may be needed solely by a directive-dependent library, which can then be removed as well. For example, going back to Figure 22, if `libB` is directive-dependent, then `libD` can also be removed when the directive is disabled, as it is not needed by any other module. On the other hand, `libC` cannot be removed because it is still needed by other libraries. The second case is handled by identifying all exported functions of a directive-related library, and checking whether any of them are used by other parts of the source code. For instance, although

differential analysis shows that the `gzip` directive depends on `libz.so`, we cannot simply remove it because a function from `libz.so` is used by other parts of the code.

6.3 Evaluation

We evaluated the potential of configuration-driven code debloating by experimenting with three widely used server applications (Nginx, VSFTPD, and OpenSSH) running on Ubuntu 16.04. Our aim is to explore the impact of various features *that are disabled by default* on code bloat, i.e., how much code could be removed when a given feature is disabled in a certain configuration. In addition, we also compare configuration-driven code debloating with the alternative (and orthogonal) approach of code debloating based on removing any non-imported functions from libraries [79, 80].

6.3.1 Identifying Non-default Functionality

One way to identify promising features that are disabled by default and which may be associated with libraries that are loaded but remain unused would be to go through the various configuration directives and pinpoint the ones that seem promising. However, as discussed in Section 6.1, the large amount of configuration directives for some applications would make this approach quite time consuming (testing *all* directives would be even more so). Instead, we followed the opposite approach and went through the loaded libraries of each application, trying to identify those that seem specific to a certain functionality, and then looking for related configuration directives in the documentation. Once the directives corresponding to a given library are identified, a special configuration can be specified in our analysis tool to run the application with pre-defined test cases (Section 6.2.1).

One of the libraries Nginx loads (in its default installation, e.g., when installed by a package manager like `apt-get`) is `libGeoIP.so`, which provides code for mapping IP addresses to their geographic locations. We can easily assume that there may be a configuration directive related to geolocation, and we indeed identified three related directives (`geoip_city`, `geoip_country`, and `geo_org`). Similarly, the presence of `libxslt.so` is related to the `xslt_stylesheet` directive, as shown in Listing 1.

Table 5. Libraries exclusively used by certain (disabled by default) features, and their corresponding footprints in terms of code size and ROP gadgets, for three server applications.

Program	Functionality	Libraries	# Functions	Code (bytes)	# Gadgets
Nginx	(Whole)	libdl.so.2, libpthread.so.0, libcrypt.so.1, libpcre.so.3*, libz.so.1*, libc.so.6, libssl.so.1.0.0, libcrypto.so.1.0.0, and all libraries from other features	38,712	13,853,649	150,914
	GeoIP	libGeoIP.so.1*	386	137,663	460
	XSLT	libxml2.so.2*, libxslt.so.1*, libexslt.so.0*, libm.so.6, libicui18n.so.55*, libicuuc.so.55*, libicudata.so.55, libstdc++.so.6	20,066	5,766,222	55,595
	Image filtering	libgcc_s.so.1, libgd.so.3*, libjpeg.so.8, libpng12.so.0*, libfreetype.so.6*, libxcb.so.1*, libfontconfig.so.1*, libXpm.so.4*, libX11.so.6, libvpx.so.3, libtiff.so.5, libexpat.so.1*, liblzma.so.5*, libjbig.so.0*, libXau.so.6*, libXdmcp.so.6*	9,240	4,800,102	49,286
VSFTPD	(Whole)	libcrypt.so.1, libc.so.6, libcap.so.0*, and all libraries from other features	9,005	2,993,778	44,221
	SSL	libssl.so.1.0.0, libcrypto.so.1.0.0	5,427	1,457,041	21,668
	PAM	libpam.so.0, libaudit.so.1*, libdl.so.2	211	65,294	1,008
	TCP wrapper	libwrap.so.0*, libnsl.so.1	230	69,169	1,175
OpenSSH	(Whole)	libcrypt.so.1, libdl.so.2, libcrypto.so.1.0.0, libutil.so.1, libresolv.so.2, libz.so.1*, libc.so.6, libgcrypt.so.20, libselinux.so.1, libsystemd.so.0, libgpg-error.so.0*, librt.so.1, liblzma.so.5*, libpcre.so.3*, libaudit.so.1*, and all libraries from other features	15,563	5,341,425	68,914
	Kerberos	libgssapi_krb5.so.2*, libkrb5.so.3*, libk5crypto.so.3*, libkrb5support.so.0*, libcom_err.so.2*, libpthread.so.0	3,823	1,043,336	10,740
	PAM	libpam.so.0	75	29,920	429

Although we begin with a single library per directive (or set of directives), it is often the case that a directive-dependent library exclusively relies on other libraries that are not used by other parts of the program, which our analysis identifies as well. For example, `libgd.so` for image filtering in Nginx subsequently loads 16 more libraries, such as `libpng12.so`, `libtiff.so`, and `libjpeg.so`. By following this approach, we identified three main features (GeoIP, XSLT, and image filtering) which are disabled by default, and account for the *vast majority* of loaded libraries of a default Nginx installation. In particular, among the 33 libraries loaded by default, 25 are solely required for the above three features—*just eight libraries are really needed* when none of those features are enabled. The left part of Table 5 (first three columns) summarizes the types of functionality that depend on certain directives, and the corresponding libraries that are exclusively required by them, for the three applications we tested, as a result of our directive-to-mapping analysis described in Section 5.3.1.

For Nginx, we could identify all libraries for the different directives without any particular test traffic, i.e., by simply starting and stopping the web server with each configuration. For the other two applications, we had to generate realistic traffic, including a complete authentication and log in process, because features related to authentication (e.e., PAM, TCP wrapper, Kerberos) require an actual login attempt to generate a meaningful code coverage report.

6.3.2 Attack Surface Reduction

To get a better insight on the degree of the achieved attack surface reduction, given that the code size of libraries varies widely, we provide more detailed information about the amount of code, number of functions, and number of ROP gadgets that are removed for each feature, as well as for the original program (last three columns in Table 5). We used ROPGadget [92] with its default options to discover the available ROP gadgets in each module. Note that two common libraries, the virtual dynamic shared object (`linux-vdso.so`) and the dynamic loader (`ld-linux-x86_64.so`), are omitted from the table due to their small size.

The rows denoted as "whole" in the Functionality column correspond to the original (non-debloated) binary that is typically distributed by the various Linux distributions, i.e., which contains the whole functionality that

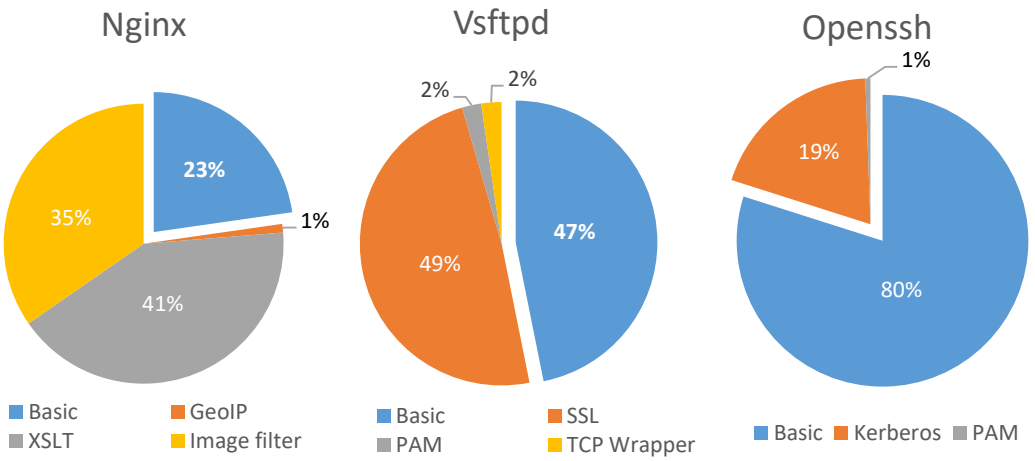


Figure 23. Breakdown of code size according to different configuration directives. "Basic" corresponds to the remaining code after configuration-driven debloating when all directives are disabled, which is the default in all cases.

can potentially be needed by all supported configurations. For example, a default Nginx process comprises 38,712 functions across 33 libraries, which correspond to approximately 14MB of code containing around 150,914 ROP gadgets.

The rest of the rows for each application correspond only to the libraries exclusively needed for a given functionality—all listed functionalities are disabled by default. Notably, the XSLT feature of Nginx alone requires 5.7MB of code—when the whole code base of Nginx is 13.8MB—while image filtering requires 4.8MB of code. As shown in the pie chart of Figure 23, XSLT and image filtering correspond to 41% and 35% of the code. When all three features are disabled (which is very likely to be the case in many configurations), configuration-driven debloating can reduce Nginx's code to just 23% of the original. The reduced code for VSFTPD and OpenSSH with their default configurations is 47% and 80% of the original, respectively. The reduction for OpenSSH is not that significant, as just Kerberos corresponds to a significant fraction of the code (about one fifth).

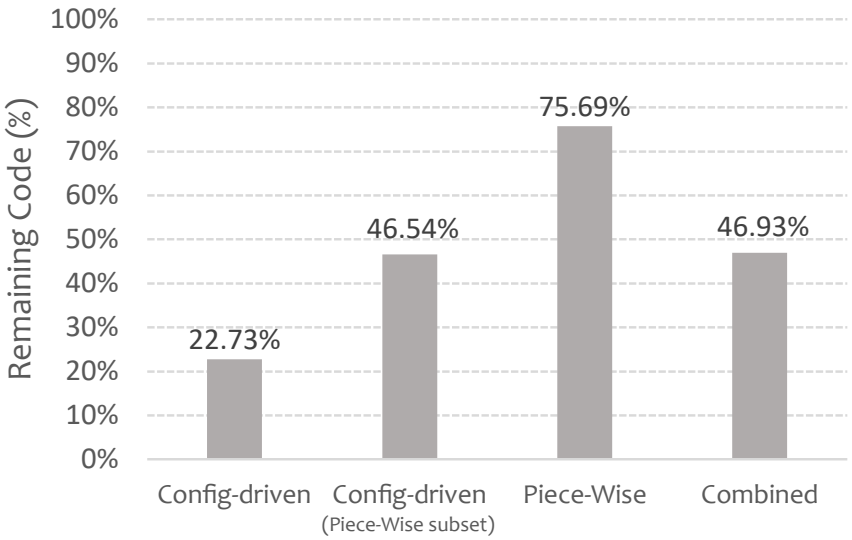


Figure 24. Remaining code for Nginx for different debloating approaches (configuration-driven, Piece-Wise [79], and their combination).

6.3.3 Comparison with Library Customization

In this section, we compare configuration-driven debloating with the alternative—and orthogonal—debloating approach of library customization [78–80]. Library customization works by statically analyzing the code of the application to identify which functions are imported (i.e., actually used) from shared libraries, and then remove the rest. We use the Piece-Wise Compilation implementation [79] as a representative library customization technique. For this set of experiments, we exclude `libc`, the libraries defined under `libc` (i.e., `libcrypt`, `libpthread`, `libm`, `libdl`, and `libcrypt`) and several cryptographic libraries (i.e., `libcrypto`, `libssl`, and `libgcrypt`) because Piece-Wise's modified LLVM compiler could not successfully compile them. The remaining libraries that were successfully processed (33 out of a total of 67 libraries for all three applications) are marked with an asterisk (*) in Table 5.

Figure 24 shows the remaining code for Nginx using its default configuration for i) configuration-driven debloating, ii) the same when considering only the libraries that can be successfully handled by the Piece-Wise compiler, iii) Piece-Wise compilation, and iv) the combination of the two approaches

(i.e., Piece-Wise applied after configuration-driven debloating has removed the non-needed libraries). The second case is provided for a more fair comparison with Piece-Wise debloating, which shows that for Nginx, library specialization alone cannot reach the level of reduction achieved by configuration-driven debloating. Although the combination of both approaches in this case offers only a small benefit (<1%), it may be beneficial for other applications. We could not meaningfully perform the same comparison for VSFTPD and OpenSSH because Piece-Wise could only process less than half of the libraries (mostly the very small ones), which collectively do not represent a substantial amount of the whole code.

6.4 Discussion and Limitations

Our current implementation requires the source code of the application to collect code coverage information during the profiling phase. The reliance on source code means that the technique is not applicable on close-source software, while the profiling phase requires appropriate inputs to exercise the corresponding code paths, which may result in missed functionality, and entails a fair amount of manual preparation. For example, exercising the code for the PAM functionality in OpenSSH required realistic interaction with the server, including proper user authentication.

Our library dependence analysis and validation steps mitigate this issue, but a more principled approach may be possible by combining code and data flow analysis techniques, which we leave as part of our future work. Another aspect that currently involves manual analysis is the identification of particular configuration directives that seem promising enough to analyze. A fully automated approach would be capable of exhaustively analyzing all directives, and even certain directive combinations.

A drawback of relying on source code coverage is that its information may not be entirely accurate. Based on our experience with the LLVM source coverage tool, the coverage report is not generated properly in cases of some forking applications. In particular, when the code uses `_exit()` instead of `exit()`, the tool fails to catch the termination of the process. Therefore, we had to modify the source code as a workaround for both VSFTPD and OpenSSH. In addition, the environment variable, `LLVM_PROFILE_FILE` was not propagated to forked processes in OpenSSH, which resulted in empty

report files. We resorted to running OpenSSH in debug mode, which disables forking, to extract proper coverage information.

It is worth mentioning that we excluded Apache from our evaluation because its modular design is directly exposed to the configuration file. Enabling a certain feature is performed by actually specifying the precise path to the corresponding shared library implementing that feature in the configuration.

7 Conclusion and Future Work

7.1 Summary

Memory corruption vulnerabilities that allow an adversary to change the intended control flow of an original program are a long–standing problem in the security community. The root cause of such vulnerabilities is a lack of memory safety checks. Today, modern operating systems incorporate several mitigation techniques including non-executable memory and ASLR by default, which has reduced the amount of classic code injection attacks. Meanwhile, the emergence of ROP attacks has received much attention from both academia and industry. In particular, disclosure-aided code reuse attacks, which enable adversaries to achieve arbitrary code execution, have become a standard form of modern exploitation. Moreover, recent adversarial advancement of code reuse attacks has seen the introduction of a JIT-ROP that is able to generate a functional payload from collecting gadgets at runtime. In response to this attack, adaptive defenders suggested the concept of XOM. One important lesson learned from the past number of decades is that a bullet–proof defense solution against code reuse attacks does not exist because adversarial threat models keep evolving, so the arms race continues. It is also noteworthy that, with regard to leveraging existing code snippets to form an exploitable payload, the attack surface constantly grows because the volume of operating systems and applications tends to increase with the addition of new features.

Software diversity is indisputably one of the most promising protection mechanisms, and its effectiveness is evident in the vast body of work that has emerged on it over the last number of decades. XOM protection also requires fine–grained code diversification to be effective either as a standalone defense or as a prerequisite for execute-only hardware and software memory protections. However, only ASLR has actually seen widespread adoption, which is surprising given the effectiveness of fine–grained code randomization against ROP. Our findings show that there are three main reasons why prior studies in this area have remained academic exercises: i) the lack of a transparent and streamlined model for diversified binaries; ii) the unaffordable cost for end users of creating the mutations; and iii) incompatibility with well-established software builds and other mechanisms that rely on software uniformity and distribution norms. The key factors for successful deployment of code randomization are transparency, reliability, compatibility, and cost.

In this dissertation, we present a practical software specialization against code reuse attacks to tackle the difficulties of generating and deploying diversified code in practice.

First, we propose instruction displacement, a practical code diversification technique for stripped binary executables without source code, which is applicable even with partial code disassembly coverage. The main idea is to displace any non-randomized gadgets into random locations, thereby improving the entropy and coverage of existing code diversification techniques. Our experimental evaluation exhibits that instruction displacement reduces the number of remnant gadgets from 15.0% to 2.8% with a negligible average runtime overhead (0.36%).

Second, we explore code inference attacks that undermine the state-of-the-art defense mitigation of destructive code reads and propose a practical defense against the deduction of the precise structure of code. The main idea behind the defense is to adopt re-randomization at runtime by detecting code disclosure and replacing the part of the code disclosed with a different randomized version.

Third, we present compiler–assisted code randomization (CCR), a compiler–rewriter cooperation model for a practical, generic, robust, and fast fine–grained code transformation. The model fills a gap in prior code randomization works as it constitutes the first hybrid approach in which both end users and software vendors cooperate to generate a unique mutation. Unlike the previous works, CCR does not rely on recompilation, disassembly, or binary analysis for variant generation. Rather, we identify a minimal set of supplementary information that is extracted from the compilation toolchain, augmenting final binaries with transformation–assisting metadata for on-demand rewriting on endpoints. We have implemented a prototype of this approach by extending the LLVM compiler and gold linker and developing a simple binary rewriter that leverages the embedded metadata to create randomized variants using basic block reordering. Our experimental evaluation is promising in terms of feasibility and practicality as it incurs a modest average file size increase (11.5%) and a negligible average runtime overhead (0.28%).

Lastly, we introduce configuration-driven code debloating, an approach that eliminates feature-specific shared libraries that are only activated when the user defines a certain functionality using configuration directives (typically disabled by default). Using a semi-automated approach, our technique iden-

tifies libraries that are needed solely for the implementation of a particular functionality and maps them to certain configuration directives. Based on this mapping, feature-specific libraries are not loaded at all if their corresponding directives are disabled. The results of our experimental evaluation demonstrate that our approach can remove up to 77% of the original code.

7.2 Future Work and Directions

We have already discussed a few limitations of our current implementation and outlined possible solutions as part of future work in Section 3.6, Section 5.6, and Section 6.4. The following research directions capture our vision of achieving practical software protection via actual deployment.

Support a General Compiler–rewriter Approach: Our proposed approach, compiler–assisted code randomization (CCR), is based on the ability to produce transformation–assisting metadata at compilation time. Unlike the LLVM compiler or GCC, a number of other commodity compilation toolchains, such as Microsoft Visual Studio, have a closed-source base. Thus, it is beyond our capability to directly modify COTS development toolchains to make the suggested approach feasible. However, to achieve our goal of implementing software diversity across different architectures and platforms, we plan to introduce certain essential requirements as a standard, including a de-randomization mechanism to assist crash reporting and an efficient way of verifying the integrity of a variant.

Extend Useful Features to the Current CCR Prototype: As part of our future work, we plan to explore more aggressive combinations of other types of randomization techniques. For instance, in-place randomization within a basic block could boost randomization entropy even further. Introducing a minimal number of unconditional jump instructions could relax the constraint of fall-through basic blocks within a function which is a limitation of the current prototype. Another possible extension of the current prototype is to explore how to support hand–written assembly, debugging sections, and entire CFI modes with extra engineering efforts. Besides the prevalent IA-64 architecture, we also plan to support other widely used architectures, such as ARM, PowerPC, and MIPS.

Seek Other Applications for CCR: We believe that combining the benefits of compiler-level and binary-level code randomization techniques can benefit a myriad of other frameworks that require reliable binary rewriting because compiler–assisted binary instrumentation inherently yields absolute precision and full–coverage code extraction. It would be possible to extract the required metadata from the compilation toolchain on demand (i.e., function or basic block boundaries for code debloating). Another interesting application would be to have a variant represent a copyright (because of its uniqueness) if a software vendor generates a seed for mutation.

Explore Fine–grained Configuration–driven Attack Surface Reduction: With the given configuration of an application, we plan to explore a better way of reducing code surface into a finer granularity (i.e., at the function level). We will also investigate an automated method of generating a combination of possible configuration directives. If necessary, it would be worth designing a debloating–friendly configuration as standard.

References

[1] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. Lynn, and T. Santoro, "The Cornell commission: On Morris and the worm," 1989.

[2] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, p. 365, 1996.

[3] D. Moore, C. Shannon, and J. Brown, "Code-Red: a case study on the spread and victims of an Internet worm," in *Proceedings of Internet Measurement Workshop 2002*, Nov 2002.

[4] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "The spread of the sapphire/slammer worm," CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, 2003.

[5] Trendmicro, "Conficker/ downad 9 years after: Examining its impact on legacy systems," https://blog.trendmicro.com/trendlabs-security-intelligence/conficker-downad-9-years-examining-impact-legacy-systems/, 2017.

[6] TrendMicro, "Massive wannacry/wcry ransomware attack hits various countries," 2017, https://blog.trendmicro.com/trendlabs-security-intelligence/massive-wannacrywcry-ransomware-attack-hits-various-countries/.

[7] ——, "Microsoft Patch Tuesday of March 2017: 18 Security Bulletins; 9 Rated Critical, 9 Important," 2017, https://blog.trendmicro.com/trendlabs-security-intelligence/microsoft-patch-tuesday-march-2017-18-security-bulletins-9-critical-9-important/.

[8] Malwarebytes, "All about ransomware," 2018, https://www.malwarebytes.com/ransomware/.

[9] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008, https://bitcoin.org/bitcoin.pdf.

[10] l. Erlingsson, Y. Younan, and F. Piessens, "Handbook of information and communication security," 2010.

[11] C. Details, "The ultimate security vulnerability datasource," https://www.cvedetails.com/vulnerabilities-by-types.php, 2010.

[12] V. V. D. Veen, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Proceedings of the 15th Internation Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2012.

[13] T. W. Laszlo Szekeres, Mathias Payer and D. Song, "SoK : Eternal war in memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[14] R. S. D. B. J. M. J. L. B.C. Ward, S.R. Gomez and H. Okhravi, "Survey of cyber moving targets second edition," *Technical Report, Lincoln Laboratory, Massachusetts Institute of Technology*, 2018.

[15] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, 1998.

[16] Vendicator, "StackShield," http://www.angelfire.com/sk/stackshield/.

[17] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, 2003.

[18] S. Bhatkar and R. Sekar, "Data Space Randomization," in *Proceedings of the 5th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[19] R. Hensing, "Understanding DEP as a mitigation technology," 2009, http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep- as-a-mitigation-technology-part-1.aspx.

[20] S. Designer, "Getting around non-executable stack (and fix)," 1997, http://seclists.org/bugtraq/1997/Aug/63.

[21] Nergal, "The advanced return-into-lib(c) exploits: PaX case study," *Phrack*, vol. 11, no. 58, Dec. 2001.

[22] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, 2007.

[23] Phoronix, "The linux kernel gained 2.5 million lines of code, 71k commits in 2017," https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kernel-Commits-2017.

[24] Openhub, "Chromium total lines," https://www.openhub.net/p/chrome/analyses/latest/languages_summary.

[25] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computer Survey*, vol. 51, no. 3, pp. 50:1–50:39, May 2018.

[26] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering (ISCE)*, 2007.

[27] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers and Security*, vol. 75, pp. 118 – 137, 2018.

[28] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Security Symposium*, Aug 2014.

[29] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[30] Nicholas Carlini and David Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[31] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[32] 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 *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[33] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehman, and Fabian Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[34] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[35] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz, "Evaluating the effectiveness of current anti-rop defenses," in *Proceedings of the 17th Internation Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2014.

[36] P. Team, "Address space layout randomization," 2003, http://pax.grsecurity.net/docs/aslr.txt.

[37] M. Miller, T. Burrell, and M. Howard, "Mitigating software vulnerabilities," Jul. 2011, http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788.

[38] J. Ganz and S. Peisert, "ASLR: How Robust Is the Randomness?" in *IEEE Cybersecurity Development (SecDev)*, 2017.

[39] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

[40] D. A. D. Zovi, "Practical return-oriented programming," 2010.

[41] R. Johnson, "A castle made of sand: Adobe Reader X sandbox," 2011.

[42] Parvez, "Bypassing Microsoft Windows ASLR with a little help by MS-Help," Aug. 2012, http://www.greyhathacker.net/?p=585.

[43] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[44] J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[45] 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 *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[46] J. Gionta, W. Enck, and P. Larsen, "Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, 2016, pp. 189–197.

[47] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[48] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[49] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[50] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[51] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse," in *Proceedings of the 12th European conference on Computer Systems (EuroSys)*, 2017.

[52] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, "NORAX: Enabling execute-only memory for COTS binaries on AArch64," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[53] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[54] F. B. Cohen, "Operating system protection through program evolution," *Computers and Security*, vol. 12, pp. 565–584, Oct. 1993.

[55] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[56] J. Edge, "OpenBSD kernel address randomized link," https://lwn.net/Articles/727697/, 2017.

[57] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *In Proceedings of the 12th USENIX Security Symposium*, 2003.

[58] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[59] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[60] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[61] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[62] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[63] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2013.

[64] 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 *Proceedings of the 8th European conference on Computer Systems (EuroSys)*, 2013.

[65] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[66] S. Crane, A. Homescu, and P. Larsen, "Code randomization: Haven't we solved this problem yet?" in *Proceedings of the IEEE Cybersecurity Development Conference (SecDev)*, 2016.

[67] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[68] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[69] 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[70] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[71] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the New Security Paradigms Workshop (NSPW)*, 2010.

[72] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?" *IEEE Security Privacy*, vol. 12, no. 2, pp. 28–35, Mar 2014.

[73] TechCrunch, "Google says there are now 2 billion active chrome installs," https://techcrunch.com/2016/11/10/google-says-there-are-now-2-billion-active-chrome-installs/, 2016.

[74] Microsoft, "/ORDER (put functions in order)," 2003, http://msdn.microsoft.com/en-us/library/00kh39zz.aspx.

[75] Google, "Syzygy - profile guided, post-link executable reordering," 2009, http://code.google.com/p/syzygy/wiki/SyzygyDesign.

[76] "Profile-guided optimizations," https://docs.microsoft.com/en-us/cpp/build/environment-variables-for-profile-guided-optimizations?view=vs-2017.

[77] 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[78] L. Song and X. Xing, "Fine-grained library customization," in *Proceedings of the ECOOP 1st International Workshop on SoftwAre debLoating And Delayering (SALAD)*, 2018.

[79] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[80] C. Mulliner, "Breaking Payloads with Runtime Code Stripping and Image Freezing," https://www.blackhat.com/docs/us-15/materials/us-15-Mulliner-Breaking-Payloads-With-Runtime-Code-Stripping-And-Image-Freezing.pdf, 2015.

[81] T. L. Yurong Chen and G. Venkataramani, "Damgate: Dynamic adaptive multi-feature gating in program binaries," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, The Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[82] A. G. Hashim Sharif, Muhammad Abubakar and F. Zaffar, "Trimmer: Application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[83] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005, http://www.suse.de/~krahmer/no-nx.pdf.

[84] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[85] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[86] D. A. D. Zovi, "Mac OS X return-oriented exploitation," 2010.

[87] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.

[88] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2011.

[89] A. A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions," in *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 2, 2018, pp. 139–156.

[90] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[91] ropper, "Ropper - rop gadget finder and binary information tool," https://github.com/sashs/Ropper.

[92] ropgadget, "Ropgadget - search your gadgets on your binaries to facilitate your rop exploitation," https://github.com/JonathanSalwan/ROPgadget.

[93] pakt, "Ropc - a turing complete rop compiler," https://github.com/pakt/ropc.

[94] Corelan Team, "Mona," https://github.com/corelan/mona.

[95] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, 2004.

[96] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the 2nd European Workshop on System Security (EuroSec)*, New York, NY, USA, 2009.

[97] J. Bennett, Y. Lin, and T. Haq, "The Number of the Beast," 2013, http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html.

[98] F. J. Serna, "CVE-2012-0769, the case of the perfect info leak," Feb. 2012, http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[99] H. Li, "Understanding and exploiting Flash ActionScript vulnerabilities," 2011.

[100] M. Labs, "MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit," 2013, https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-webkit-exploit/.

[101] V. Kotov, "Dissecting the newest IE10 0-day exploit (CVE-2014-0322)," Feb. 2014, http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/.

[102] B. Antoniewicz, "Analysis of a Malware ROP Chain," Oct. 2013, http://blog.opensecurityresearch.com/2013/10/analysis-of-malware-rop-chain.html.

[103] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005.

[104] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[105] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[106] "Control flow integrity," https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[107] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*, 2011.

[108] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (S&P)*, 2010.

[109] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[110] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proceedings of the 51st Annual Design Automation Conference (DAC)*, 2014.

[111] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[112] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.

[113] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC)*, 2009.

[114] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[115] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2011.

[116] I. Fratric, "Runtime prevention of return-oriented programming attacks," 2012, https://code.google.com/p/ropguard/.

[117] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[118] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[119] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.

[120] Microsoft, "The enhanced mitigation experience toolkit," http://support.microsoft.com/kb/2458544.

[121] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[122] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[123] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[124] J. Pewny, P. Koppe, L. Davi, and T. Holz, "Breaking and fixing destructive code read defenses," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017.

[125] 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 *Proceedings of the 22nd ACM conference on Computer and Communications Security (CCS)*, 2015.

[126] K. Lu, N. Stefan, M. Backes, and W. Lee, "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization," in *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[127] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of*

the 22nd ACM Conference on Computer and Communications Security (CCS), 2015.

[128] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, C. L. Stephen Crane, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity," in *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[129] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[130] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[131] B. Randell, "System Structure for Software Fault Tolerance," in *Proceedings of the International Conference on Reliable Software*, 1975.

[132] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transaction Software Engineering*, vol. 11, no. 12, pp. 1491–1501, Dec. 1985.

[133] K. Pettis, R. C. Hansen, and J. W. Davidson, "Profile guided code positioning," in *Proceedings of the 9th ACM Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[134] R. Lavaee and D. Chen, "ABC Optimizer: Affinity Based Code Layout Optimization," *Technical Report*, 2014.

[135] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 63–68, Dec. 2005.

[136] G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010.

[137] C. Cifuentes and M. V. Emmerik, "Recovery of Jump Table Case Statements from Binary Code," in *Proceedings of the 7th International Workshop on Program Comprehension (IWPC)*, 1999.

[138] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[139] 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 *Proceedings of the 25th USENIX Security Symposium*, 2016.

[140] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[141] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the Tor browser against de-anonymization exploits," *PoPETs*, no. 4, pp. 454–469, 2016.

[142] M. Morton, H. Koo, F. Li, K. Z. Snow, M. Polychronakis, and F. Monrose, "Defeating zombie gadgets by re-randomizing code upon disclosure," in *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2017.

[143] 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 International Conference on Virtual Execution Environments (VEE)*, 2017.

[144] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[145] Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack*, vol. 10, no. 56, Jan. 2000.

[146] T. Durden, "Bypassing PaX ASLR protection," *Phrack*, vol. 11, no. 59, Jul. 2002.

[147] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[148] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, 2010.

[149] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[150] M. P. Shachee Mishra, "Shredder: Breaking Exploits through API Specialization," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[151] X. Z. Zhongshu Gu, Brendan Saltaformaggio and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," in *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[152] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schroder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time os kernel tailoring," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[153] H. M. Mansour Alharthi, Hong Hu and T. Kim, "On the effectiveness of kernel debloating via compile-time configuration," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, The Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2018.

[154] D. W. Yufei Jiang and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.

[155] K. G. Suparna Bhattacharya and M. G. Nanda, "Combining concern input with program analysis for bloat detection," in *Proceedings of*

the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2013.

[156] Y. Jiang, C. Zhang, D. Wu, and P. Liu, "Feature-based software customization: Preliminary analysis, formalization, and methods," in *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2016.

[157] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "Reddroid: Android application redundancy customization based on static analysis," in *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2018.

[158] Apple, "What is app thinning? (ios, tvos, watchos)," https://help.apple.com/xcode/mac/current/#/devbbdc5ce4f, 2015.

[159] V. Rastogi, D. Davidson, L. D. Carli, S. Jha, and P. D. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[160] Q. A. C. David K. Hong and Z. M. Mao, "An initial investigation of protocol customization," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, The Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[161] T. L. Yurong Chen, Shaowen Sun and G. Venkataramani, "Toss: Tailoring online server systems through binary feature customization," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, The Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2018.

[162] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[163] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (KDD)*, 2011.

[164] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[165] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[166] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[167] Hex-Rays, "IDA Pro Disassembler," http://www.hex-rays.com/idapro/.

[168] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*, 2009.

[169] "Orp: in-place binary code randomizer," http://nsl.cs.columbia.edu/projects/orp/.

[170] E. Carrera, "pefile," https://github.com/erocarrera/pefile.

[171] N. A. Quynh, "Capstone: Next-gen disassembly framework," 2014.

[172] Skape, "Locreate: An anagram for relocate," *Uninformed*, vol. 6, 2007.

[173] M. Pietrek, "An in-depth look into the Win32 portable executable file format, part 2," 1994, https://msdn.microsoft.com/en-us/library/ms809762.aspx.

[174] "Wine," http://www.winehq.org.

[175] A. K. Cristiano Giuffrida and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[176] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[177] H. M. Gisbert and I. Ripoll, "On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows," in *Proceedings of the 13th IEEE International Symposium on Network Computing and Applications*, 2014.

[178] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha, "Launching return-oriented programming attacks against randomized relocatable executables," in *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

[179] Fermin J. Serna, "The info leak era on software exploitation," 2012, https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf.

[180] Alexander Sotirov and Mark Dowd, "Bypassing Browser Memory Protections," 2008, https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf.

[181] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[182] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[183] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: transparent code randomization for just-in-time compilers," in *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS)*, 2013.

[184] I. Guilfanov, "Cross-window message broadcast interface," https://github.com/diy/intercom.js.

[185] H. Koo and M. Polychronakis, "Juggling the gadgets: Binary-level code randomization using instruction displacement," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[186] "Polyverse," https://polyverse.io/, 2017.

[187] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *Computer Journal*, vol. 23, no. 3, pp. 223–229, 1980.

[188] G. Ramalingam, "The Undecidability of Aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, September 1994.

[189] M. Ludvig, "CFI support for GNU assembler (GAS)," http://www.logix.cz/michal/devel/gas-cfi/, 2003.

[190] Using the GNU Compiler Collection (GCC), "Common Function Attributes," https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html, 2017.

[191] "Profile guided optimization," https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization.

[192] T. Johnson, "ThinLTO: Scalable and Incremental LTO," http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html, 2016.

[193] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in COTS binaries," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[194] K. ElWazeer, "Deep Analysis of Binary Code to Recover Program Structure," *Dissertation*, 2014.

[195] E. Bendersky, "Assembler relaxation," http://eli.thegreenplace.net/2013/01/03/assembler-relaxation, 2013.

[196] Y. Li, "Target independent code generation," http://people.cs.pitt.edu/~yongli/notes/llvm3/LLVM3.html, 2012.

[197] M. Sun, T. Wei, and J. C. Lui, "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[198] J. Corbet, "SMP alternatives," https://lwn.net/Articles/164121/, 2005.

[199] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Dynamic reconstruction of relocation information for stripped binaries," in *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[200] D. Geneiatakis, G. Portokalidis, V. P. Kemerlis, and A. D. Keromytis, "Adaptive Defenses for Commodity Software Through Virtual Application Partitioning," in *Proceedings of the 19th ACM conference on Computer and communications Security (CCS)*, 2012.

[201] T. Klein, "Relro - a (not so well known) memory corruption mitigation technique," http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html, 2009.

[202] "The LLVM Compiler Infrastructure," http://llvm.org.

[203] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[204] I. L. Taylor, "Introduction to gold," http://www.airs.com/blog/archives/38, 2007.

[205] S. Kell, D. P. Mulligan, and P. Sewell, "The missing link: Explaining ELF static linking, semantically," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[206] "GNU Binutils," https://www.gnu.org/software/binutils/.

[207] E. Bendersky, "Pure-python library for parsing ELF and DWARF," https://github.com/eliben/pyelftools.

[208] "Protocol Buffers," https://developers.google.com/protocol-buffers/.

[209] Intel, "System V application binary interface," https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013.

[210] "The DWARF debugging standard," http://dwarfstd.org/.

[211] "LLVM link time optimization design and implementation," https://llvm.org/docs/LinkTimeOptimization.html.

[212] "The LLVM gold plugin," http://llvm.org/docs/GoldPlugin.html.

[213] S. McCamant, "Large single compilation-unit C programs," http://people.csail.mit.edu/smcc/projects/single-file-programs/, 2006.

[214] E. Bosman and H. Bos, "Framing signals—a return to portable shellcode," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[215] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the code space to counter disclosure attacks," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[216] M. Zhang, M. Polychronakis, and R. Sekar, "Protecting COTS binaries from disclosure-guided code reuse attacks," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017.

[217] LLVM, "Source-based code coverage," https://clang.llvm.org/docs/SourceBasedCodeCoverage.html, 2008.