Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP

Salman Ahmed
Virginia Tech
ahmedms@vt.edu

Ya Xiao
Virginia Tech
yax99@vt.edu

Kevin Z. Snow
Zeropoint Dynamics, LLC
kevin@zeropointdynamics.com

Gang Tan
Penn State University
gtan@cse.psu.edu

Fabian Monrose
UNC at Chapel Hill
fabian@cs.unc.edu

Danfeng (Daphne) Yao
Virginia Tech
danfeng@vt.edu

Abstract

Just-in-time return-oriented programming (JIT-ROP) allows one to dynamically discover instruction pages and launch code reuse attacks, effectively bypassing most fine-grained address space layout randomization (ASLR) protection. However, in-depth questions regarding the impact of code (re-)randomization on code reuse attacks have not been studied. For example, *how would one compute the re-randomization interval effectively by considering the speed of gadget convergence to defeat JIT-ROP attacks?*; *how do starting pointers in JIT-ROP impact gadget availability and gadget convergence time?*; *what impact do fine-grained code randomizations have on the Turing-complete expressive power of JIT-ROP payloads?* We conduct a comprehensive measurement study on the effectiveness of fine-grained code randomization schemes, with 5 tools, 20 applications including 6 browsers, 1 browser engine, and 25 dynamic libraries. We provide methodologies to measure JIT-ROP gadget availability, quality, and their Turing-complete expressiveness, as well as to empirically determine the upper bound of re-randomization intervals in re-randomization schemes using the Turing-complete (TC), priority, MOV TC, and payload gadget sets. Experiments show that the upper bound ranges from 1.5 to 3.5 seconds in our tested applications. Besides, our results show that locations of leaked pointers used in JIT-ROP attacks have no impacts on gadget availability but have an impact on how fast attackers find gadgets. Our results also show that instruction-level single-round randomization thwarts current gadget finding techniques under the JIT-ROP threat model.

CCS Concepts

• **Security and privacy → Systems security**; **Software and application security**.

Keywords

ASLR measurement; security metrics; attack surface quantification; Re-randomization interval; measurement methodology; address/-code pointer impact analysis; JITROP.

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417248

ACM Reference Format:
Salman Ahmed, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng (Daphne) Yao. 2020. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3372297.3417248

1 Introduction

Just-in-time return-oriented programming (JIT-ROP) (e.g., [91]) is a powerful attack technique that enables one to reuse code even under fine-grained address space layout randomization (ASLR). Fine-grained ASLR, also known as fine-grained code randomization or code diversification, reorders and relocates program elements. Fine-grained randomization would defeat conventional ROP code reuse attacks [88], as the attacker no longer has direct access to the code pages of the victim program and its libraries. In other words, a leaked pointer only unlocks a small portion of the code region under fine-grained code randomization, seriously limiting the attack's ability to harvest code for ROP gadget purposes.

JIT-ROP attacks can discover new code pages dynamically [91], by leveraging control-flow transfer instructions, such as *call* and *jmp*. Under fine-grained code randomization, the execution of a JIT-ROP attack is complex, as code page discovery has to be performed at runtime. From the defense perspective, re-randomization techniques (TASR [6], Shuffler [105], Remix [21], CodeArmor [19], RuntimeASLR [66], and Stabilizer [29]) have the potential to defeat JIT-ROP attacks. Besides, protections related to memory permission such as XnR [4], NEAR [104], Readactor [27], destructive read such as Heisenbyte [96], and pointer indirection such as Oxymoron [5] specifically aim to thwart JIT-ROP attacks. Precise implementation of Control-Flow Integrity (CFI) can protect applications from all control-oriented attacks. The recent Multi-Layer Type Analysis (MLTA) [65] technique improves CFI precision greatly by improving the accuracy in identifying indirect call targets.

Even though the great promise of CFI for protecting control-oriented attacks, attackers may find ways to launch new exploits such as control-oriented [24, 38, 45, 84] and non-control-oriented [14, 55, 56] exploits as demonstrated before, where the exploits conform with CFI. A prime requirement of many of these exploits is information or pointer leakage. Thus, a measurement mechanism aiding the design of risk heuristics-based pointer selection and prioritization techniques is necessary for protecting pointers from leakage. Besides, from a **defense-in-depth** perspective, a critical system

requires to deploy multiple complementary security defenses in practice. A single defense may fail due to deployment issues such as implementation flaws or configuration issues. Thus, despite the strong security guarantees of CFI, our ASLR investigation is still extremely necessary.

Re-randomization techniques continuously shuffle the address space at runtime. This continuous shuffling breaks the runtime code discovery process by making the already discovered code pages obsolete. However, the interval between two consecutive randomizations must satisfy both performance and security guarantees.

Quantitative evaluation of how code (re-)randomization impacts code reuse attacks, e.g., in terms of interval choices, gadget availability, gadget convergence, and speed of convergence has not been reported. We define *gadget convergence* as the attack stage where an attacker has collected all the necessary gadgets. For example, if an attacker has found at least one gadget for each type of Turing-complete (TC) operations, then the gadget set is TC convergence. TC operations include memory, assignment, arithmetic, logic, control flow, function call, and system call [81].

(Re-)randomization techniques make it difficult for current gadget finding techniques to discover all gadgets. Thus, in-depth and systematic measurement is necessary, which can provide new insights on the impact of code (re-)randomization on various attack elements, such as code pointer leakage, various gadget sets, and gadget chain formation. It is also important to investigate how to systematically compute an effective re-randomization interval. Current re-randomization literature does not provide a concrete methodology for experimentally computing an upper bound of re-randomization intervals. Shorter intervals (e.g., millisecond-level) incur runtime overhead whereas longer intervals (e.g., second-level) give attackers more time to launch exploits. An upper bound would help guide defenders to make informed interval choices.

We report our experimental findings on re-randomization interval choices considering the speed of gadget convergence, code pointer leakage, gadget availability, and gadget chain formation, under fine-grained ASLR and re-randomization schemes.

Launching exploits is not a feasible measurement methodology to evaluate ASLR's effectiveness, due to *i)* low scalability – exploit payload is not platform or application portable, *ii)* failure to exploit may not necessarily mean security, and *iii)* low reproducibility. Our evaluation involves up to 20 applications, including 6 browsers, 1 browser engine, and 25 dynamic libraries.

We designed a measurement mechanism that allows us to perform JIT-ROP's code page discovery in a scalable fashion. This mechanism enables us to compare results from a number of programs and libraries under multiple ASLR conditions (coarse-grained, fine-grained function level, fine-grained basic block level, fine-grained instruction level, and register level). Our key experimental findings and technical contributions are summarized as follows.

- We provide a methodology to compute the upper bound $\mathcal{T}$ for re-randomization intervals. If the re-randomization interval is less than $\mathcal{T}$, then a JIT-ROP attacker is unable to obtain various gadget sets such as the Turing complete gadget set, priority gadget set, MOV TC gadget set, and gadgets from real-world payloads (see the definitions of gadget sets in Section 2). We compute the upper bound $\mathcal{T}$ by measuring the minimum time for an attacker to find a specific gadget set, i.e., the shortest time to reach gadget convergence for the gadget set. The upper bound ranges from 1.5 to 3.5 seconds in our tested applications such as *nginx*, *proftpd*, *firefox*, etc.
- Our findings show that starting code pointers do not have any impact (i.e., zero standard deviations) on the reachability from one code page to another. Every code pointer leak is equally viable for revealing an address space layout, suggesting that attackers' discovered gadgets eventually converge to a gadget set no matter where the starting pointer is.
- Our findings also show that the starting code pointers have an impact on the speed of convergence. That means the time needed for a JIT-ROP attacker to discover a gadget set varies with the locations of starting code pointers. In our experiments, the time for obtaining the Turing-complete gadget set ranges from 2.2 to 5.8 seconds.
- We also present a general methodology for quantifying the number of JIT-ROP gadgets. Our results show that a single-round instruction-level randomization scheme can limit the availability of gadgets up to 90% and break the Turing-complete operations of JIT-ROP payloads. Also, fine-grained randomization slightly degrades the gadget quality, in terms of register-level corruption. A stack has a higher risk of revealing dynamic libraries than a heap or data segment because our experiments show that stacks contain 16 more *libc* pointers than heaps or data segments on average.

Besides, we distill common attack operations in existing ASLR-bypassing ROP attacks (e.g., [8, 15, 32, 91]) and present a generalized attack workflow that captures the tasks and goals. This workflow is useful beyond this specific measurement study.

2 Threat Model and Definitions

Coarse-grained ASLR (or traditionally known as only ASLR [98]) randomly relocates shared libraries, stack, and heap, but does not effectively relocate the main executable of a process. This defense only ensures the relocation of the base address of a segment or module. The internal layout of a segment or module remains unchanged. The **P**osition **I**ndependent **E**xecutable (PIE) option allows to relocate the main executable in random locations in each run. For comparison purposes, we performed experiments on coarse-grained ASLR with PIE enabled on a 64-bit Linux system.

Fine-grained ASLR, aka fine-grained code randomization or code diversification, relocates all the segments and dependencies of the main executable of a process and restructures the internal layouts of the segments. The granularity of the randomization varies, e.g., at the level of functions [25, 43, 58], basic blocks [21, 59, 103], instructions [52], or machine registers [27, 53]. We evaluated randomization schemes at various levels of granularities using Zipr[1] [50], Selfrando[2] (SR) [25], Compiler-assisted Code Randomization[3] (CCR) [59], and Multicompiler[4] (MCR) [53]. We also evaluated Shuffler [105], a re-randomization tool. We are unable to test other tools due to various robustness and availability issues.

[1] https://git.zephyr-software.com/opensrc/irdb-cookbook-examples
[2] https://github.com/immunant/selfrando
[3] https://github.com/kevinkoo001/CCR
[4] https://github.com/securesystemslab/multicompiler

We assume standard defenses such as W⊕X and RELRO are enabled. W⊕X specifies that no address is writable and executable at the same time. RELRO stands for Relocation Read Only. It ensures that the Global Offset Table (GOT) entries are read-only. RELRO is now by default deployed on mainstream Linux distributions.

Layered defenses. CFI and Code Pointer Integrity (CPI) solutions are very powerful techniques. Yet, it is still necessary for one to experimentally measure the effectiveness of various defense implementations in practice (e.g., CPI enforcement with spatial and temporal guarantees and CFI effectiveness in different granularities [12]). From a measurement perspective, it is useful and necessary to isolate various defense factors. Decoupling them helps one better understand the individual factor's security impact. Otherwise, it might be too complicated to interpret the experimental results. This is the reason we chose to focus on ASLR defenses in this work and omit other defenses (e.g., CFI [1, 28, 42, 69, 70, 72, 78, 108, 109] and CPI [5, 26, 37, 61, 62, 67]. For similar reasons, we also omit memory permission protections (e.g., XnR [4], NEAR [104], Readactor [27], Heisenbyte [96], and Execute-only-Memory (XOM[5]) [64]) for this paper. We also discuss the need for measuring code pointer protection solutions under the JIT-ROP model in Section 6.

We assume attackers have already obtained a leaked code pointer (e.g., a function or a virtual table pointer) through remote exploitation of a vulnerability. Such an assumption is standard in existing attack demonstrations. Also, fine-grained code randomization is applied in every executable and associated library in a target system (unless specified otherwise). A JIT-ROP attacker knows nothing about the applied fine-grained randomization.

Native vs. WebAsm vs. JavaScript version of JIT-ROP. While the original JIT-ROP attack was demonstrated in a browser using JavaScript, the attack approach has general applicability in both native and scripting environments. Our experiments are focused on the native execution of JIT-ROP attacks. We conducted the experiments for measuring the re-randomization upper bound using the native JIT-ROP code module. The execution time of WebAssembly is within 2x of native code execution [49]; JavaScript is on average 34% slower than WebAssembly [49]. Thus, our re-randomization intervals measured using the native execution would be conservatively applicable for the scripting environments as well. Besides, JIT-ROP is not related to the JIT compilers of JavaScript (JS) engines and does not use any flaws of JIT compilers to perform a code-reuse attack, though some work [3] uses such flaws. JIT-ROP harvests gadgets from a target binary's static code, which is finely randomized; it does not harvest gadgets from dynamically generated code (e.g., scripts). Thus, JS or WebAsm versions do not make substantial differences in gadget availability.

Next, we discuss the terms Turing-complete gadget set, priority gadget set, MOV TC gadget set, re-randomization upper bound, minimum footprint gadgets, and extended footprint gadgets.

Definition 1. Turing-complete gadget set refers to a set of gadgets that covers the Turing-complete operations including memory operations (i.e., load memory LM and store memory SM gadgets), assignments (i.e., load register LR and move register MR gadgets), arithmetic operations (i.e., arithmetic AM, arithmetic load AM-LD, and arithmetic store AM-ST gadgets), logical operations (i.e., logical gadgets), control flow (i.e., jump JMP gadgets), function calls (i.e., CALL gadgets), and system calls (i.e., syscall SYS gadgets) [81].

Definition 2. The upper bound $\mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$ of a re-randomization scheme $\mathcal{P}$ under a JIT-ROP attacker $\mathcal{A}$ is the maximum amount of time between two consecutive randomization rounds that prevents $\mathcal{A}$ from obtaining a Turing-complete, priority, MOV TC, or payload gadget set, i.e., for any interval $\mathcal{T}'^{\mathcal{A}}_{\mathcal{P}} < \mathcal{T}^{\mathcal{A}}_{\mathcal{P}}$, the gadgets obtained under $\mathcal{T}'^{\mathcal{A}}_{\mathcal{P}}$ does not converge to any of the four gadget sets.

Extended and Minimum footprint gadgets: A gadget is an extended footprint (EX-FP) gadget if it is an instance of the four gadget sets. An EX-FP gadget may contain additional instructions that may cause side effects in an attack payload. EX-FP gadgets include the longer memory addressing expressions. A minimum footprint (MIN-FP) gadget is also an instance of the four gadget sets without causing any side effects.

Our definition of the Turing-complete gadget set represents our best efforts, by no means the only way. For example, a pair of load (LM) and store (SM) gadgets may potentially replace a move (MR) gadget. However, they may not be directly equivalent due to possibly mismatching memory offsets of EX-FP load gadgets or the scarcity of MIN-FP load gadgets. Excluding load-n-store from the Turing-complete gadget set might underestimate attackers' capabilities, while including them might overestimate attackers' capabilities. We perform our measurements considering the Turing-complete gadget set that enables the highest expressiveness of ROP attacks. However, under this condition, our results might underestimate the attackers' capabilities. To balance an attacker's capabilities, we further break down the Turing-complete gadget set into two smaller gadget sets: *i)* priority gadget set and *ii)* MOV TC gadget set. The *priority* gadget set includes 10 most frequently used gadgets in 15 real-world ROP chains from Metasploit. The *MOV Turing-complete* gadget set [35] requires six MOV gadgets and four unique registers. Besides, we also include three real-world ROP payloads from Metasploit in our measurement.

New metrics proposed by Brown and Pande's [11] work – functional gadget set expressivity and special-purpose gadget availability – are new leads that will help relax the expressiveness condition of the Turing-complete gadget set in the future.

Our security definition of the upper bound in Definition 2 is specific to the JIT-ROP threat, and is not applicable to other threats (e.g., side-channel threats). A shorter interval may still allow attackers to gain information. However, as our Section 3 shows, without gadgets that information may not be sufficient for launching exploits.

3 JIT-ROP vs. Basic ROP Attacks

We manually analyze a number of advanced attacks to extract common attack elements and identify unique requirements. We illustrate the key technical differences between JIT-ROP and conventional (or basic) ROP attacks. This section helps one understand our experimental design in Section 4 and findings in Section 5. We analyze various attack demonstrations with a focus on attacks (e.g., [8, 15, 32, 91]) in our threat model.

To overcome both coarse- and fine-grained ASLR and conduct an attack using privileged operations, an attacker needs to perform the

[5]XoM is now supported natively at the hardware level on x86 systems with memory protection keys (MPK) support and Armv7-M or Armv8-M processors.

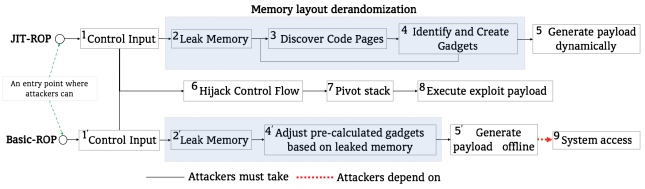

Figure 1: An illustration of the commonalities and differences between a conventional (or basic) ROP attack (bottom) and a JIT-ROP attack (top). The top gray-box highlights the key steps in JIT-ROP to overcome fine-grained ASLR.

tasks presented in Figure 1. The attack workflow has three major components: ***memory layout derandomization***, ***system access***, and ***payload generation***.

3.1 Memory Layout Derandomization

Derandomizing an address space layout is the key for mounting code-reuse attacks. Due to the W⊕X defense, attackers need to derandomize the memory layout to discover gadgets (steps ②-④ for JIT-ROP and steps ②′ and ④′ for basic ROP in Figure 1). Usually, attackers leverage memory corruption vulnerabilities to leak memory [94] and derandomize an address space layout using the leaked memory. This step requires overcoming several obstacles.

Memory disclosure. The most common way of derandomizing memory layout is through a memory disclosure vulnerability. Attackers use vulnerabilities in an application's memory (e.g., heap overflows, use-after-free, type confusion, etc.) and weaknesses in system internals (e.g., vulnerabilities in the glibc malloc implementation or its variants [2, 51], Heap Feng Shui [93], and Flip Feng Shui [80]) to leak memory contents (Steps ② and ②′). Details on memory corruption can be found in [41, 95] and an example in [94].

Code reuse. Due to W⊕X defense, adversaries cannot inject code in their payload. ROP [88] and its variants Jump-Oriented Programming (JOP) [10] and Call-Oriented Programming (COP) [45] can defeat this defense. These techniques use short instruction sequences (i.e., gadget) from the code segments of a process' address space and allow an adversary to perform arbitrary computations. ROP tutorials can be found in [34, 91]. The difference between basic ROP [88] and JIT-ROP [91] is described next.

Basic ROP. Coarse-grained ASLR only randomizes the base addresses of various segments and modules of a process. The content of the segments and modules remains unchanged. Thus, it is feasible for an adversary to launch a basic ROP attack [97] using gadgets given a leaked address from the code segment of interest. The adversary only needs to adjust the addresses of pre-computed gadgets w.r.t. the leaked address. Step ④′ in Figure 1 is about this task.

Just-in-time ROP. Since adjusting the addresses of pre-computed gadgets (as in the basic ROP) no longer works under fine-grained ASLR, an attacker needs to find gadgets dynamically at the time of an exploit. Scanning a process' address space linearly for gadgets from a disclosed code pointer may not be effective because this linear scanning may lead to crash the process due to reading an unmapped memory. A powerful technique introduced in JIT-ROP [91] is the recursive code page harvest, which is explained next.

The **recursive code harvest** technique exploits the connectivity of code in memory to derandomize and locate instructions (step ③ in Figure 1). The technique identifies gadgets at runtime by reading and disassembling the text segment of a process. The technique computes the page number from a disclosed code pointer and reads the entire 4K data of that page. A light-weight disassembler converts the page data into instructions. The code harvest technique searches for chain instructions, such as *call* or *jmp* instructions to find code pointers to other code pages.

An illustration is shown in Figure 2. The code harvest process starts from the disclosed pointer (0x11F95C4), reads 4K page data (0x11F9000-0x11F9FFF), disassembles the data, searches for *call* and *jmp* instructions to find other pointers (0x11FB410 and 0x11FCFF4) to jump to those code pages. This process is recursive and stops when all the reachable code pages are discovered.

Snow *et al.* demonstrated the JIT-ROP attack in a browser. Since exploiting a memory corruption bug remotely covers a wide variety of exploits, a browser is an ideal interface for JIT-ROP attacks. The scripting environment of a browser enables easy interfacing of a JavaScript-based JIT-ROP attack payload. Similarly, JIT-ROP attack payload can be embedded into a PDF reader that supports JavaScript (e.g., Adobe Reader). However, an attacker must convert any non-scripting attack code to script for the scripting environment. For example, the original JIT-ROP framework was written in C/C++ and was transpiled to JavaScript to demonstrate on Internet Explorer.

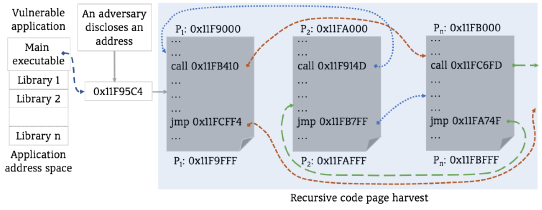

Figure 2: An illustration of the recursive code harvest process of JIT-ROP [91]. An adversary discloses an address from the main executable or libraries (in this case from the main executable) of an application through a vulnerability.

Gadget identification. In step ④ of Figure 1, attackers identify gadgets by scanning for byte values corresponding to *ret* opcodes (e.g., 0xC2, 0xC3) from the read code pages and perform a narrow-scoped backward disassembly. The adversary performs step ③ and ④ repeatedly to find required gadgets for the target exploit.

3.2 System Access

Attackers need to issue system APIs or gadgets to perform privileged operations. If the CFI defenses (e.g., BCFT [42], CCFIR [108] and bin-CFI [109]) are not enforced, adversaries do not need to invoke the entire functions to ensure legitimate control flow. An adversary can just chain together enough gadgets for setting up the arguments of a system call and invoking it. This observation is particularly true for Linux, which is the focus of this paper. In Windows exploits [91], the approach can be slightly different, as adversaries commonly invoke a system API instead of invoking a system call directly. *Syscall* gadgets can be found in an application's code or dynamic library. For basic ROP attacks, attackers can adjust

pre-computed system gadgets from dynamic libraries, given that she manages to obtain a code pointer from a dynamic library (e.g., *libc*). Step ⑨ in Figure 1 is for this task. This task is performed manually and offline. The attacker may obtain the library code pointer from an application's stack or heap or data segment. One can find system gadgets through step ④ in JIT-ROP.

3.3 Payload Generation

Attackers generate payloads by putting many pieces (e.g., gadgets, functions, constants, strings, etc.) together. This process must ensure a setup for calling system APIs or system gadgets. An attacker generates a payload dynamically at step ⑤ under fine-grained code randomization or manually at step ⑤′ under coarse-grained code randomization and stores the payload in a stack/heap. Because a payload is primarily a set of addresses that point to some existing code in an application's address space, attacks do not execute anything stored in a stack/heap, which is protected by W⊕X. An attacker may utilize the same vulnerability as in step ② or a different vulnerability to hijack a program's control flow at step ⑥ to redirect the flow to the stored payload. A payload usually targets to achieve an attack goal, e.g., memory leak or launching a malicious application/root shell.

Attack chains with minimal side effects are desirable for attackers, i.e., having a payload that fulfills attack goals without generating any unnecessary computations. However, this property may not be guaranteed if code randomization limits gadget availability. We refer to the side effect of gadgets as *footprints*. We defined the *minimum* and *extended* footprint gadgets in Section 2.

For ROP attacks (e.g., [15]) that bypass control-flow integrity (CFI) defenses, the attackers also need to prepare specialized payloads in addition to the previous tasks. For example, the Flashing (FS) and Terminal (TM) gadgets in Table 5 in the Appendix were designed by Carlini and Wagner [15] to bypass specific CFI implementations (namely, kBouncer [76] and ROPecker [23]).

4 Measurement Methodologies

We describe our measurement methodologies for evaluating fine-grained ASLR's impact on the memory layout derandomization, system access, and payload generation of JIT-ROP. One major challenge is how to **quantify** the impact of fine-grained code randomization. Our approach is to count the number of available gadgets under the JIT-ROP code harvest mechanism. Other challenges are how to quantify *i)* the difficulty of accessing privileged operations and *ii)* the quality of gadget chains. For the former, our approach is to measure the number of system gadgets and count *libc* pointers in a stack or heap or data-segment of an application. To quantify the quality of gadget chains, we design a register-level measurement heuristic by computing the register corruption rate.

4.1 Methodology for Derandomization

Gadget selection. We manually extracted 21 types of gadgets from various attacks [8, 14, 15, 45, 91]. These gadget types include load memory (LM), store memory (SM), load register (LR), move register (MR), arithmetic (AM), arithmetic load (AM-LD), arithmetic store (AM-ST), LOGIC, jump (JMP), call (CALL), system call (SYS), and stack pivoting (SP) gadgets. In addition to these, the gadget types also include some attack-specific gadgets such as call preceding (CP), reflect (RF), call site (CS2) and entry point (EP) gadgets. Table 5 in the Appendix shows those gadget types in more details.

These 21 types of gadgets include the Turing-complete gadget set (see Definition 1). These gadgets also include the priority and MOV TC gadget sets (Table 6 in the Appendix). We use the Turing-complete, priority, and MOV TC gadget sets for our evaluation because we can precisely identify those gadgets. We also include gadgets from three real-world ROP payloads from Metasploit [30, 31] and Exploit-Database [13]. We leave the attack-specific gadgets out of our evaluation due to the lack of their concrete forms and attack goals. Attackers used the attack-specific gadgets to trick defense mechanisms. We also discuss the evaluation of the block-oriented gadgets used for Block-Oriented Programming (BOP) [56].

Methodology for single-round randomization experiments. In our experiments, we measure the occurrences of gadgets from the Turing-complete gadget set under fine-grained code randomization schemes. To enforce the code randomization schemes, we used four relatively new code randomization tools: Zipr [50] (instruction-level randomization), SR [25] (function-level randomization), CCR [59] (block-level randomization), and MCR [53] (function + register-level randomization), because of their reliability. Table 7 in Appendix shows the key differences between these schemes. We compile and build a coarse- and a fine-grained version of each application or dynamic library for each run using each of the four randomization tools, i.e., each run has a different randomized code. We use LLVM Clang 3.9, Clang 3.8 and GCC 5.4 as the compilers for CCR, MCR and SR, respectively. We run, load or rewrite each application or library 100 times to reduce the impact of variability on the number of gadgets in each run or load.

We use ropper [83], an offline gadget finder tool, under coarse-grained ASLR. Under fine-grained ASLR, we write a tool to recreate the JIT-ROP [91] exploitation process, including code page discovery and gadget mining. Our tool can search for gadgets of a specific type. We scan the opcodes of *ret* (0xC3) and *ret xxx* (0xC2) and perform a narrow-scoped backward disassembly from those locations to collect ROP gadgets. Similarly, we scan the opcodes of *int 0x80* (0xCD 0x80), *syscall* (0x0F 0x05), *sysenter* (0x0F 0x34) and *call gs:[10]* (0x65 xFF 0x15 0x10 0x00 0x00 0x00) for system gadgets. We consider the gadgets only from the legitimate instructions, not from instructions within overlapping instruction bytes.

Methodology for re-randomization experiments. For code re-randomization schemes, we attempted to use six re-randomization tools. However, some of the tools are unavailable and some have runtime and compile-time issues[6]; in the end, we were able to obtain only Shuffler [105]. To evaluate the impact of re-randomization, we take 100 consecutive address space snapshots from an application/library re-randomized by Shuffler [105]. Then, we manually analyze the address space snapshots.

The choice of re-randomization intervals is important for a re-randomization scheme. An effective re-randomization interval should hinder attackers' capabilities while ensuring performance guarantees. Our measurement methodology determines the upper bound (see definition 2) of effective re-randomization intervals by considering the fastest speed of gadget convergence, i.e., the

[6]Remix [21] & CodeArmor [19] are not available. TASR [6] is not accessible for policy issues. Runtime ASLR [66] & Stabilizer [29] have run & compile time issues, respectively.

minimum time for convergence. To measure the time of gadget convergence, we run the recursive code harvest process for an application and record the times it takes to converge to different gadget sets such as Turing-complete, priority, MOV TC, and payload gadget sets. We record the number of leaked gadget types that the code harvest process covered so far, while recording the convergence time. The code harvest terminates upon gadget convergence. We record multiple convergence times by starting the code harvesting process from multiple pointer locations to capture the variability. To select multiple starting pointers, we choose a random code pointer from each code page of an application. Choosing a single random code pointer from each code page allows us to identify all instructions and pointers on that code page.

4.2 Methodology for System Access

We measure the difficulty of accessing privileged operations through the availability of system gadgets and vulnerable library pointers in a stack, heap or data-segment. For system gadgets, we compare the number of system gadgets under the coarse- and fine-grained randomization and compute the reduction in the gadget quantity. For the measurement of vulnerable pointers in a stack/heap/data-segment, we examine the overall risk associated with a stack/heap/data-segment by identifying the number of unique *libc* pointers in that stack/heap/data-segment. For the evaluation purpose, we do not exploit vulnerabilities to leak *libc* pointers from the stack/heap/data-segment. Rather, we assume that we know the address mapping of *libc* and can find the *libc* pointers through a linear scanning of the stack/heap/data-segment. We discuss the existence of *libc* pointers in popular applications in Section 5.6.

4.3 Methodology for Payload Generation

We focus on measuring the quality of individual gadgets to approximate the quality of a gadget chain. The quality of a set of gadgets for generating payloads is essential, as attackers need to use gadgets to set up and prepare register states. To measure the quality of individual gadgets, we perform a register corruption analysis for each gadget, which is briefly described next. The detail description of our register corruption analysis is in Appendix A.1.

Typically, a gadget contains one core instruction that serves the purpose of that gadget. For example, an MR gadget may contain *mov eax, edx* as the core instruction and some additional instructions before/after the core instruction. We measure the register corruption rate by analyzing how the core instruction of a gadget can get modified by those additional instructions. In the case of multiple core instructions of a gadget type, we consider the core instruction that is closest to the ret instruction. A core instruction may be modified by *i)* the instruction(s) before the core instruction, *ii)* the instruction(s) after the core instruction, and *iii)* both the instruction(s) before/after the core instruction. For each gadget, we consider these three scenarios and determine whether the gadget is corrupted or not. Next, in the following paragraphs, we discuss the code randomization and re-randomization tools briefly.

Shuffler [105] runs itself alongside the user space program that it aims to protect. It has a separate asynchronous thread that continuously permutes all the functions to make any memory leaks unusable as fast as possible.

Zipr [50] reorders the location of each instruction in an executable or library (an example in Figure 9 in the Appendix). Zipr works directly on binaries or libraries with no compiler supports. Zipr [50] is based on the Intermediate Representation Database (IRDB) code. Zipr shuffles code during the rewriting process, which is called block-level instruction layout randomization.

Selfrando (SR) [25] is compiler-agnostic and applies code diversification at the load time using function boundary-metadata called *Translation and Protection (TRaP)* and inserting a dynamic library called *libselfrando*. At the load time, *libselfrando* takes control of the execution, reorders the position of each function in an executable utilizing the TRaP information, and relinquishes the control to the original entry point of the executable.

Multicompiler (MCR) [27, 53] applies the code diversification at the link time. This tool randomizes functions, machine registers, stack-layout, global symbols, VTable, PLT entries, and contents of the data section. The tool also supports insertion of NOP, global padding, and padding between stack frames. We choose the function and machine register level randomization for our evaluation. MCR uses the Clang-3.8 LLVM compiler.

Compiler-Assisted Code Randomization (CCR) [59] applies the code diversification at the installation time, i.e., rewrites an executable binary by reordering the functions and basic blocks of the executable. This tool collects metadata for code layout, block boundaries (i.e., the basic block, functional block, and object block boundaries), fixup, and jump table of an executable during compilation and linking phases. A Python script rewrites the executable binary utilizing the collected metadata. In our experiments, CCR uses the clang-3.9 LLVM compiler.

Availability and robustness of fine-grained ASLR tools. We found that the majority of code diversification implementations, including ASR [43], ASLP [58], Remix [21], and STIR [103], are not publicly available. Some available tools (e.g., MCR [27, 53], CCR [59] and SR [25]) operate on the source code level that requires recompilation. We experienced multiple linking issues while using CCR and SR to compile Glibc code. The tool authors confirmed the limitations (discussed in Section 6). ORP [77] was the randomization tool used in Snow *et al.*'s JIT-ROP demonstration [91]. It operates on Windows binaries, incompatible with our setup.

5 Evaluation Results and Insights

Experimental setup. We implemented a JIT-ROP native code module. All experiments are performed on a Linux machine with Ubuntu 16.04 LTS 64-bit operating system. We write Python and bash scripts for automating our measurement process. Our code and data are available at https://github.com/salmanyam/jitrop-native.

We perform our experiments on the latest and stable versions of applications including *bzip2*, *cherokee*, *hiawatha*, *httpd*, *lighttpd*, *mupdf*, *nginx*, *openssl*, *proftpd*, *sqlite*, *openssh*, *thttpd*, *xpdf*, and *mupdf*, browsers including *firefox*, *chromium*[7], *tor*, *midori*, *netsurf*, and *rekonq* and browser engines such as *webkit*. We also perform our experiments on dynamic libraries. Dynamic libraries include

[7]Due to the incompatibility of the LLVM compiler version and the use of custom linkers with custom linking flags, we are unable to randomize the Chromium browser using SR, CCR, and MCR. Zipr also fails to randomize chromium possibly due to the large size of the executable (~944MB). However, we include a non-randomized version of the chromium browser in our re-randomization experiments.

libcrypto, *libgmp*, *libhogweed*, *libxcb*, *libpcre*, *libgcrypt*, *libgnutls*, *libgpg-error*, *libtasn1*, *libz*, *libnettle*, *libopenjp2*, *libopenlibm*, *libpng16*, *libtomcrypt*, *libunistring*, *libxml2*, *libmozgtk*, *libmozsandbox*, *libxul*, *libmozsqlite3*, *liblgpllibs*, *libwebkit2gtk-3.0*, and *musl*. We select these applications or dynamic libraries based on their popularity for attack demonstrations. Also, these applications or libraries are from diverse areas such as Web Server, Browser, PDF reader, networking, database, and cryptography, math, image, and system.

Table 1: Numbers of the applications and dynamic libraries for experiments.

Experiment	Applications (20 Total)	Libraries (25 Total)
Re-randomization interval	17	15
Instruction-level rand.	15	14
Function-level rand.	17	21
Function + register-level rand.	12	13
Basic block-level rand.	15	15

Table 1 shows the numbers of applications/libraries used for measuring the upper bound for re-randomization intervals and evaluating instruction-level [50], functional-level [25], function+register-level [27, 53], and basic block-level [59] randomizations. Each experiment evaluates a different set of applications and libraries because no (re-)randomization tool is capable of (re-)randomizing all of our selected applications (20 in total) and libraries (25 in total). However, we also conduct our experiments and report results using the common set of applications and libraries.

We measure a total of 11 types of gadgets for the Turing-complete set, 10 types for the priority set, and 7 types for the MOV TC set. Different payloads have different types and numbers of gadgets.

5.1 Re-randomization Upper Bound

We determine the upper bound of re-randomization intervals by measuring the fastest speed of gadget convergence across the Turing-complete, priority, MOV TC, and payload gadget sets, i.e., measuring the minimum time that an attacker needs to collect all gadget types from any of the four gadget sets. Table 2 shows the minimum (i.e., fastest speed) and the average time to leak all gadget types in a set. The minimum and the average time is calculated over 17 applications/browsers. From the table, we notice that the re-randomization upper bounds, i.e., the minimum time, range from 1.5 to 3.5 seconds. We observe some variability ($\sigma = 0.8$) in the minimum time, having the priority and MOV TC gadget sets the lowest (1.5s) and highest (3.5s) time, respectively. Intuitively, the reason for this variability could be related to the number of gadget types necessary for each gadget set. However, we observe that the minimum time for the MOV TC gadget set is larger than the TC or priority gadget set even though the MOV TC has fewer gadget types. To understand more about this variability, we analyze how gadget types are leaked over time for individual applications/browsers across the four types of gadget sets.

Figure 3 shows the minimum time to obtain the Turing-complete gadget set from individual application or browser along with a timeline for new gadget type leaks. Each gray ● mark with a number n on top of it represents the time to leak n gadget types. The bold ● mark represents the time to leak 11 gadget types from the Turing-complete gadget set. For example, it takes roughly 1 and 4.3 seconds to leak 6 and 11 gadget types, respectively from *cherokee*.

Table 2: Minimum and average time to leak all gadget types from TC, priority, MOV TC, and payload gadget sets. The percentage (%) of time is spent for leaking gadgets versus analyzing gadgets. The minimum, average, and percentages for each set are calculated using 17 applications including browsers. Payload* ⟶ average of three payload sets.

	Time to leak all gadget types		Gadget analysis	
Gadget set	**Minimum (s)**	**Average (s)**	**Leak (%)**	**Analysis (%)**
TC	2.2	4.3	17	83
Priority	1.5	3.5	13	87
MOV TC	3.5	5.3	16	84
Payload*	2.1	4.8	12	88
Average	**2.3s**	**4.5s**	**14.5%**	**85.5%**

The number of leaks increases as time increases. However, the effect of the increase may not be immediate. For example, in Figure 3, the code harvest process takes roughly 0.7 seconds to leak 8 distinct gadget types from *netsurf*. If the time increases to 1 or 2 seconds, the number of leaked gadgets is still the same, i.e., 8 distinct gadget types. However, if the time is more than 3 seconds, the number of leaked gadgets starts to increase. We call the time between 0.7 to 3 seconds as non-reactive.

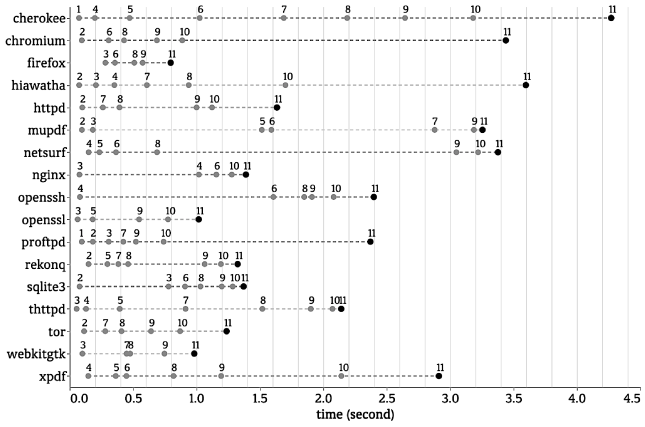

Figure 3: Minimum time to obtain the Turing-complete gadget set with a timeline for new gadget type leaks. Each gray filled circle (●) with a number n on top of it represents the time to leak n gadget types. The bold filled circle (●) indicates the time to leak all gadget types. Applications and browsers are randomized with a function-level scheme [25].

We observe a number of long non-reactive times for some other applications such as chromium (0.89–3.44s), hiawatha (1.7–3.6s), mupdf (0.18–1.52s and 1.6–2.88s), openssh (0.08–1.61s), proftpd (0.74–2.37s), and xpdf (1.19–2.14s). Most of these non-reactive times are towards the end of their timelines. These non-reactive times indicate that a few missing gadget types prevent the discovered set from being Turing-complete quickly. That is, a few types of gadgets are very scarce. The scarcest gadgets are Load-Memory (LM), Arithmetic-Load (AM-LD), and System Call (SYS) gadgets. The fundamental reason for the scarcity is that some applications (including libraries) have a few register-based memory accesses. Besides, the main executable of an application does not have SYS gadgets in most cases.

We also observe similar non-reactive times for obtaining the priority and MOV TC gadget sets. The variability in the minimum time of the four gadget sets is due to the Arithmetic-Load (AM-LD) gadget type. Since the priority gadget set does not include AM-LD, its code page harvest process is the fastest. The time for the MOV TC gadget set is relatively longer than the TC and priority gadget set, even though MOV TC does not include AM-LD. The reason for this long time is that the MOV TC set includes several specialized Load-Memory (LM) and Store-Memory (ST) gadget types.

The MOV TC gadget set is powerful since it takes only a few *mov* instructions with four register pairs to perform the Turing-complete operations. To observe to what extent MOV TC gadgets are prevalent in applications, we count the numbers of six MOV gadgets (MR, ST, STCONSTEX, STCONST, LM, and LMEX described Table 6 in the Appendix) and the System Call (SYS) gadget while measuring the minimum time to find these gadgets. STCONSTEX, STCONST, and LMEX gadgets are variants of ST and LM gadgets. The average number of gadgets for MR is 51, ST is 14, STCONSTEX is 35, STCONST is 2, LM is 3, LMEX is 15, and SYS is 23. As expected, the number of Load-Memory (LM) gadgets is low, which indicates the scarcity of this gadgets. Besides, we observe the number of Store-Constant (STCONST) is also low, which is necessary for performing comparison and conditional operations.

Our re-randomization upper bound calculation includes the gadget analysis overhead. Thus, we perform additional analyses to investigate the time spent to leak address space versus gadget analysis. We find that on average around **15%** of the time is spent on leaking address space, while the rest for gadget searching (Table 2). This result indicates that a JIT-ROP attacker spends a significant amount of time searching for gadget types. Thus, the upper bound of re-randomization intervals is subject to change based upon an optimized gadget search strategy.

Clearly, the upper bound for the re-randomization intervals also depends on the machine (e.g., CPUs, cache size, memory, etc.) where the measurement is conducted. Using our methodology, defenders can perform the measurement on their machines to determine what intervals are appropriate for their applications, while satisfying overhead constraints. In Section 6, we discuss the implications of re-randomization intervals in real-world operations.

We call the upper bound of re-randomization intervals as the "best-case" re-randomization interval from a defender's perspective because the defender has to rerandomize by the time of the interval, if not sooner. This raises the question regarding the effectiveness of "best-case" intervals over "worst-case" intervals. The "worst-case" interval indicates the time required to build a useful gadget chain using a minimal set of gadgets. In reality, attackers' goals vary. It is difficult to determine a minimum set of gadgets common and necessary across all attack chains. Besides, our "best-case" interval includes the time for discovering SYS gadgets that are scarce. Some attack scenarios may not require the SYS gadgets, but the necessity of SYS gadgets or system APIs in attack chains have been shown by previous work [8, 10, 15, 32, 91].

5.2 Impact of the Location of Pointer Leakage

We measure the impact of pointer locations on JIT-ROP attack capabilities, by comparing the number of gadgets harvested and the time of harvest under different *starting* pointer locations. We aim to find out whether or not the number of gadgets and the time depend on the location of a pointer leakage when a fine-grained randomization scheme is applied.

Impact of pointer locations on gadget availability. To measure the impact of pointer locations on gadget availability, we collect the number of minimum and extended footprint gadgets by leaking a random code pointer from **each** code page of *hiawatha*, *httpd*, *lighttpd*, *nginx*, *proftpd*, and *thttpd* and starting the code harvesting process from that leaked code pointer. Then we calculate the average number of gadgets for each leaked pointer. We leak a single code pointer from a single code page randomly because choosing any single random code pointer from a code page allows us to identify all instructions and all code pointers on that code page. Table 3 shows the number of leak code pointers or addresses and the numbers of minimum and extended footprint gadgets. We restrict the code harvest process to harvest gadgets from the main executable of an application to find how well the code of that application is connected. We exclude the dynamic libraries for this experiment because many applications use a common set of libraries and the gadgets from this common set of libraries (if not excluded) would dominate the total number of gadgets.

For all applications, we observe that the pointer's location does not have any impact on the total number of minimum and extended footprint gadgets. For example, regardless of the location of starting point in *nginx*, we observe 26 minimum and 788 extended gadgets when randomized by the instruction-level randomization scheme; 222 minimum and 5277 extended footprint gadgets when randomized by the function-level scheme; 111 minimum and 1731 extended footprint gadgets when randomized by function + register-level scheme; and 204 minimum and 4822 extended footprint gadgets when randomized by block-level scheme. **These findings indicate that an application's code segment is very well-connected, making JIT-ROP attacks easier**.

The numbers of leaked addresses in Table 3 are different for different randomization schemes because we use different backends (i.e., compilers) to enforce the schemes. Different backends optimize the same application differently. This increases/decreases the number of code pages. Since we leak a random address from each code page, the number of leaked addresses varies with tools.

Impact of pointer locations on code harvest time. To measure the impact of code pointer locations on the time, we measure the time required to leak all gadget types from the Turing-complete gadget set. We start the code harvest process from a random code pointer leaked from each code page of an application or browser and record the time to collect all gadget types. Figure 4 shows the minimum, maximum, and average time to leak all gadgets for different applications and browsers. For a few code pointers from several applications/browsers (e.g., 3 out of 111 code pointers for *nginx* or 8 out of 40 code pointers for *openssl* or 2 out of 41 for *tor*), the code harvest process takes significantly shorter time than the average. We analyze the reason for this phenomenon.

We find that most applications/browsers have some code pages that contain a diverse set of gadgets. For example, *nginx* contains 9 code pages that have at least 5 distinct gadget types from the Turing-complete gadget set. Whenever the code harvest process accesses those code pages sooner, the discovered gadgets quickly converge to Turing-complete.

Table 3: Impact of locations of pointer leaks on gadget availability. The same application has different numbers of address leaks for different schemes due to different backends (i.e., compilers) that produce different sized executables of the same program. The size of an executable is proportional to the number of code pages. Also, the numbers of gadgets from the function-level scheme [25] and function + register-level scheme [27, 53] are not comparable due to their different backends.

Program	Instruction-level scheme [50]			Function-level scheme [25]			Function + register-level scheme [27, 53]			Block-level scheme [59]		
	# of leaked addresses	# of MIN-FP	# of EX-FP	# of leaked addresses	# of MIN-FP	# of EX-FP	# of leaked addresses	# of MIN-FP	# of EX-FP	# of leaked addresses	# of MIN-FP	# of EX-FP
hiawatha	41	9	223	42	41	1259	47	44	1042	39	31	793
httpd	91	16	634	91	141	4453	MCR produces linking error for httpd			86	176	4764
lighttpd	53	8	235	53	103	2512	68	118	2544	45	74	1783
nginx	114	26	788	121	222	5277	49	111	1731	114	204	4822
proftpd	131	17	523	187	96	7395	131	115	4466	131	125	3986
thttpd	10	8	172	17	22	583	16	31	535	15	24	428

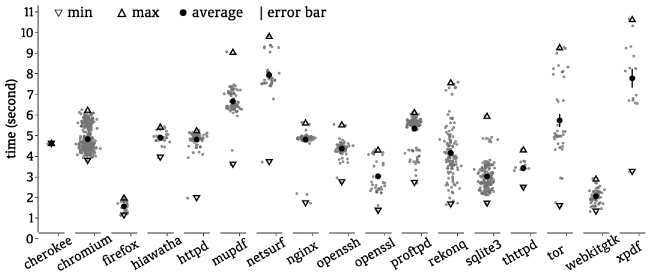

Figure 4: Impact of starting code pointer locations on gadget harvesting time. Each ● indicates the time for harvesting the Turing-complete gadget set. The minimum, maximum, and average time is calculated by starting code harvest process from multiple code pointer locations. A small amount of jitter has been added to x-axis for each application/browser for better visibility of times along the y-axis.

Future directions. Our findings imply that any valid code pointer leak is equally viable with regards to the coverage of gadgets. This observation reasserts that disrupting the code connectivity is an effective defense strategy, utilized in Oxymoron [5], Readactor [27], XnR [4], NEAR [104], Heisenbyte [96], and ASLR-Guard [67] tools. Thus, a large-scale quantitative assessment on the effectiveness of these security tools is necessary to find out the practicality and feasibility for deployment. Also, the design of risk heuristics-based pointer selection and prioritization for protecting pointers from leakage would be an interesting direction. The idea is to prioritize code pointers based on the convergence time and data pointers based on their sensitivity (e.g., data pointers used in loops).

5.3 Impact on the Availability of Gadgets

Impact of Single-round Randomization Schemes. Table 4 summarizes the impact of fine-grained code randomization schemes on the availability of gadgets in various applications (i.e., the main executables) and dynamic libraries. We measure the numbers of the various gadgets (as mentioned in Section 4.1) for each application and library before and after enforcing the four fine-grained randomization schemes. We run each application or library 100 times after randomizing each time when necessary[8]. The numbers of gadgets are averaged over 100 runs for each application or library. Then the numbers of gadgets are averaged over all applications and libraries for each randomization scheme. Table 4 shows the overall gadget reductions in application and library categories for each randomization scheme.

On average, the number of gadgets is reduced (by 18%–28% for minimum footprint and 37%–45% for extended footprint gadgets) when applications are randomized using function-, block-, and function+register-level schemes. For dynamic libraries, the reductions range from around 21%–47% for minimum footprint gadgets and around 37%–44% for extended footprint gadgets. However, instruction-level randomization reduces the overall gadget amount significantly by around 80%–90% for both minimum and extended footprint gadgets. Table 4 also shows the reduction of gadgets in seven Turing-complete (TC) operations and indicates whether the Turing-complete expressiveness is preserved after applying the code randomization. The numbers before and after a vertical bar (|) indicate the reduction of minimum and extended footprint gadgets for a TC operation. Since the numbers of applications/libraries are different for different randomization schemes, we validate the schemes using a common set of applications and libraries where the result (Figure 7 in the Appendix) shows a consistent reduction.

The Turing-complete expressiveness of ROP gadgets is preserved in the randomized applications or libraries when the schemes are function, block, and function+register-level randomizations. However, instruction-level randomization scheme [50] does not retain the Turing-complete expressiveness for minimum footprint gadgets. The Turing-complete expressiveness is hampered when there is no gadget in one of the Turing-complete operations. For example, in Table 4, the reduction of minimum footprint gadgets in memory and arithmetic operations is almost 100% for applications. That means there is no gadget to do memory and arithmetic operations, which are required for reliable attacks. The reductions for libraries in the two categories (i.e., memory and arithmetic) are 93.7% and 91.8%, respectively. For both application and library cases, the reductions are not exactly 100%, because some applications/libraries contain a few gadgets. When the numbers of gadgets are averaged over all applications or libraries, the average is close to zero.

Most of the applications and libraries do not contain any *syscall* gadgets (as expected), as applications and libraries usually make syscalls through *libc*. This is why the number of *syscall* gadgets is low (2-3) and one gadget loss leads to around 33% reduction.

[8]One compilation with 100 runs, 100 times randomization, 100 times compilation, and 100 times rewriting are required for SR, CCR, MCR, and Zipr, respectively.

Table 4: Impact of fine-grained single-round randomization on the availability of gadgets in various applications and dynamic libraries. Instruction-level randomization scheme [50] is applied on 15 applications and 14 dynamic libraries, function-level scheme [25] on 17 applications and 21 dynamic libraries, function + register-level scheme [27, 53] on 12 applications and 13 dynamic libraries, and basic block-level scheme [59] on 15 applications and 15 dynamic libraries. The data of each application or library is the average result of 100 runs/loads/rewrites. The standard deviations vary between 0.3∼3.4 for minimum footprint and 5.04∼22.85 for extended footprint gadgets. ⇓ indicates reduction.

				Reduction (%) of Turing-complete (TC) gadgets in 7 TC operations (MIN-FP | EX-FP)							
Randomization schemes	Granularity	⇓ (%) MIN-FP	⇓ (%) EX-FP	Memory	Assignment	Arithmetic	Logical	Control Flow	Function Call	System Call	TC Preserved?
Applications											
Inst. level rando. [50]	Inst.	**79.7**	**82.5**	**97.4** | 82.7	58.8 | 81.7	**95.9** | 64.9	85.8 | 85.4	49.4 | 80.1	67.4 | 83.9	83.3 | 0	✗*
Func. level rando. [25]	FB	27.63	36.55	0.8 | 29.2	10.6 | 43.5	19.3 | 15.1	35.1 | 35.9	21.1 | 29.1	18.2 | 46.9	0 | 0	✓
Func.+Reg. level rando. [53]	FB & Reg.	17.62	42.37	-8.3 | 35.0	-5.1 | 35.2	26.1 | 44.9	21.3 | 38.1	34.0 | 60.2	11.8 | 64.9	80.0 | 0	✓
Block level rand. [59]	BB	19.58	44.64	5.5 | 40.9	6.1 | 47	26.1 | 33.7	20.4 | 37.4	41.2 | 63.1	23.3 | 56.3	0.0 | 0	✓
Libraries											
Inst. level rando. [50]	Inst.	**81.3**	**92.2**	**93.7** | 96.1	60.7 | 93	**91.8** | 84.9	84.5 | 90.4	59.8 | 93.5	51.8 | 92.9	66.7 | 0	✗*
Func. level rando. [25]	FB	46.5	43.8	24.2 | 71.1	15.9 | 31	41.2 | 65.4	56.9 | 25	34.5 | 78.7	23 | 75.8	3.5 | 14.5	✓
Func.+Reg. level rando. [53]	FB & Reg.	44.2	43.9	35.5 | 44.8	35.3 | 43.4	63.2 | 61.8	44.8 | 49.0	36.4 | 52.1	43.1 | 35.3	66.7 | 0	✓
Block level rand. [59]	BB	20.98	37.0	7.3 | 36.3	8.1 | 32.1	13.9 | 55.9	24.8 | 31.6	22.2 | 52.1	18.1 | 44.6	50.0 | 0	✓

* For instruction-level randomization scheme [50], TC is not preserved for minimum footprint gadgets, but TC is preserved for extended footprint gadgets.

We also assess the gadget availability under a *single* randomization pass of Shuffler [105] by analyzing 100 consecutive address space snapshots from *nginx* after each re-randomization with an interval of 30 seconds. On average, we observe a 24% and 3% reduction in gadget availability for minimum and extended footprint gadgets compared to a non-randomized *nginx*, respectively. The low reductions are expected, as Shuffler's security relies on continuous randomization, not a single randomization pass.

Ideally, function-level randomization does not break gadgets, only shifts the gadgets from one location to another. Basic-block or machine-register-level randomization may break some gadgets due to the memory layout perturbation and register allocation randomization. It is not surprising that the function, block, or register-level randomizations to have low gadget reduction. However, instruction-level randomization perturbs the memory layout significantly as we observe a large gadget reduction by Zipr.

Future directions. Redefining traditional ROP gadgets into smaller (e.g., one line) building blocks and demonstrating new gadget chain compilers (e.g., two-level construction) by tackling the instruction-level perturbations are interesting new attack directions.

5.4 Impact on Performance Overhead

We measure the performance overhead of the five (re-)randomization tools to evaluate the overhead in our measurement environment. To measure the performance overhead, we use 8 applications in domains such as web servers, FTP servers, browsers, security protocols, and file compression tools. The applications are *nginx*, *httpd*, *proftpd*, *hiawatha*, *lighttpd*, *openssl*, *firefox*, and *bzip*. Applications are randomized using the five (re-)randomization tools. We use criteria such as HTTP request latency, FTP upload speed, browser page-load time, compression time, and effectiveness of cryptographic algorithms to measure the performance overhead.

We measure HTTP request latency by running an HTTP benchmark using *wrk* [44] for 30 seconds to read an HTML page from a server. The benchmark includes 12 threads and 400 HTTP open connections. To measure FTP upload speed, we run a benchmark using *ftpbench* [102]. The benchmark runs 10 concurrent operations for 10 seconds. We use OpenSSL *speed* to test the performance of aes-128-gcm, aes-256-gcm, aes-128-cbc, and aes-256-cbc algorithms. We use the Linux *time* command to measure compression time. Finally, we use a website speed test tool [99] to measure a browser's page load time. For Shuffler, we measure the overhead for three different re-randomization intervals: 10ms, 100ms, and 1s.

We run each measurement for five times and calculate the average for each application. Then, we average the overheads over the 8 applications. For Shuffler, we observe 3% overhead with 1s re-randomization interval, 5% for 100ms, and 12% for the 10ms interval consistent with the reported result [105]. We observe 23% overhead for Zipr, 10% for SR, 3% for CCR, and 10% for MCR which are comparable to or higher than what's reported. The reported overheads for Zipr, SR, CCR, and MCR are around 5% [50], 1% [25], 0.28% [59], and 1% [53], respectively.

5.5 Impact on the Quality of a Gadget Chain

The purpose of this analysis is to estimate the quality of a gadget chain. We measure the quality of a gadget through the register corruption analysis for individual gadgets, following the procedure described in Section 4.3. We measure the register corruption rate for `MV`, `LR`, `AM`, `LM`, `AM-LD`, `SM`, `AM-ST`, `SP`, and `CALL` gadgets. Some gadgets such as `CP`, `RF`, and `EP` (described in Table 5 in the Appendix) are special purpose gadgets that are used to trick defense mechanisms, such as CFI [1], kBouncer [76], and ropecker [23]. Thus, we omit these gadgets from the quality analysis.

We found that the overall register corruption rate is slightly higher (∼6%) in the presence of fine-grained randomization. This slightly higher register corruption rate indicates that the formation of gadget chain is slightly harder in fine-grained randomization compare to the coarse-grained randomization.

We present the detailed results in Table 8 in the Appendix, including the average number of unique registers used in each gadget. We observe the number of unique registers used in each gadget ranges from 1 to 4 in our register corruption measurement.

Sometimes, fine-grained randomization decreases the register corruption rate. For example, for Nginx, the corruption rate of the load memory (LM) gadgets is reduced from 44% to 15%, when fine-grained randomization is in place. This reduction is likely due to the relatively smaller number of gadgets in the presence of the fine-grained randomization.

Future directions. Designing randomization solutions to increase the register corruption rate in gadgets would be interesting as a high register corruption rate would make attacks unreliable.

5.6 Availability of Libc Pointers

This experiment measures the risks associated with an application's heap, stack, or data-segment for revealing a library location. For simplicity, we consider only the risk associated with revealing the *libc* library w.r.t. the basic ROP attacks. We count the number of unique *libc* pointers in a target application's stack, heap, and data-segment when the application reaches a certain execution point. We define the execution points for the target applications. For example, the execution point for *proftpd* is when *proftpd* is ready to accept connections. We assume that *i)* coarse-grained randomization is enforced, and *ii)* adversaries cannot perform recursive code harvest to find gadgets. This experiment targets a weak attack model where an adversary leaks a (known) library pointer and adjusts pre-computed gadgets based on the leaked pointer. We regard a library pointer (e.g., *libc* pointer) as known if the pointer is loaded in the same location in the stack of an application for multiple runs. A pointer in a stack, heap, or data-segment may point to a non-library function, which in turn points to a library (e.g., *libc*).

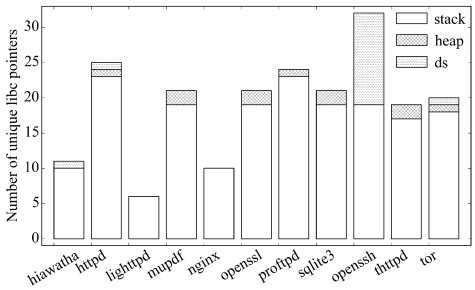

Figure 5: Libc pointers in the stack, heap and data segment of a program. Stacks contain more pointers, carrying higher risks of pointer leakage.

Figure 5 shows the number of unique *libc* pointers in the stack, heap, and data-segment of 11 applications including web servers, PDF reader, cryptography library, database, and browser. According to the observations in Figure 5, heap or data-segment contains only one *libc* code pointer (on average) while stack contains 17 *libc* code pointers. This finding indicates that high risk is associated with stack than heap or data-segment. It also suggests that the safeguard and (re-)randomization of stack is more important than protecting/randomizing heap or global variables.

Future directions. Protecting a stack/heap/data-segment from leaking data pointers that contain code pointers is an interesting research direction. A similar risk assessment for C++ binaries may indicate the importance of protecting the read-only sections that include many pointers to virtual methods.

6 Discussion

Metrics for evaluating fine-grained randomization. Traditionally, both coarse-grained (e.g., PaX ASLR [98]) and fine-grained (e.g., SR [25], CCR [59], Remix [21], Binary stirring [103], ILR [52] and ASLP [58]) randomizations use entropy to measure the effectiveness of hindering code-reuse attacks. However, such an entropy measure is not useful under the JIT-ROP threat model, as chunks of code are still available. Inclusion of distances between permuted functions or basic blocks for computing entropy would not work either, because the code's semantic connectivity (e.g., through *call* and *jmp*) is still not captured. Code connectivity is what JIT-ROP attacks leverage to discover code pages. In comparison, our measurement methodology more accurately reflects JIT-ROP capabilities and is more meaningful under the JIT-ROP model. How to design an entropy-like metric to capture the degree of code isolation or the **semantic connectivity** in code is an interesting open problem.

Availability of Block-Oriented Programming (BOP) gadgets. We measure the numbers of BOP functional blocks for register assignments/modifications, memory reads/writes, system/library calls, and conditional jumps using the BOP compiler (BOPC) [56]. We observed almost no change in the numbers of BOP functional blocks in randomized versions compared to the non-randomized versions for CCR [59] and MCR [27, 53]. As BOPC operates on static binary, we could not use Shuffler [105] and SR [25] because they randomize a memory layout at runtime. BOPC does not seem to run on binaries rewritten by Zipr [50].

Impact of the compiler optimizations on gadget availability. We assess the impact of code transformations and optimizations (-O0, -O1, -O2, -O3, -Ofast, -Os) on the availability of gadgets. We compare the unoptimized and optimized versions of *nginx*, *apache*, *proftpd*, *openssh*, and *sqlite3* to assess the impact. We find a smaller number of LM, SM, and MR gadgets in unoptimized code than optimized code (Figure 8 in the Appendix). The reason is the tendency of *mov* and *ret* instructions staying together in optimized code, but not in unoptimized code. Besides, compilers sometimes emit extra instructions for optimizations that may increase gadgets.

Reachability of gadgets. We design our experiments based on the availability of various kinds of gadgets. However, in reality, attackers need to conduct a series of operations including finding a vulnerability or leaking memory for the actual invocations of gadgets. In Section 2, we assume that an attacker has already overcome the initial obstacles, especially finding a memory leak. Our experiments are focused on comparing gadget availability of various code (re-)randomization schemes using the leaked memory.

Operational re-randomization intervals. Our methodology helps guide software owners (e.g., server owners) to set the appropriate re-randomization intervals. For example, if the owners prioritize performance over security, they can set an interval just below $\mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$ (Definition 2). If the owners prioritize security over performance, they can set an interval much shorter than $\mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$.

Need for randomizing Glibc. Unfortunately, SR, CCR, MCR, and Zipr were all unable to randomize Glibc. For CCR and MCR, the LLVM Clang compiler (backend of CCR and MCR) does not have support for compiling some Glibc's GCC specific extensions such as ASM GOTO. SR also cannot randomize some parts of Glibc. That is why we evaluate a lightweight standard C library `musl-libc` [63]),

but only SR can randomize `musl-libc`. Shuffler can reorder Glibc by disabling manual jump table construction.

Limitations. Both CFI and XoM defenses are powerful and have capabilities to prevent JIT-ROP attacks. These two defenses with continuous re-randomization would be even more powerful. However, we did not enforce CFI and XoM in this work to isolate an individual defense's security impact. In this work, we addressed many important questions related to fine-grained (re-)randomization, not yet answered by the literature. We leave the analysis and measurement of CFI and XoM as future research.

Our current work does not measure zombie gadgets [92] and microgadgets [54]. The gadgets that are available after applying destructive read defenses (e.g., XnR [4], NEAR [104], Readactor [27], and Heisenbyte [96]) are called *zombie gadgets* [92]. Destructive read defenses only allow code execution, no read after execution. In this way, destructive reads can limit gadget availability, but cannot eliminate all gadgets. We plan to assess the availability of zombie and microgadgets in our future work.

Another limitation is that we assume the code pointer obfuscation is not enforced. If enforced, code pointer obfuscation (e.g., CPI [61, 62], Oxymoron [5]) could make JIT-ROP code page discovery less effective, reducing the gadget availability. Understanding how code pointer obfuscation impacts JIT-ROP and measuring the effectiveness of this defense under various attack conditions (e.g., Isomeron [32] and COOP [84]) are interesting problems.

One limitation of time-based re-randomization schemes is that the intervals need recalculation with the evolution of hardware or a program itself. Event-based re-randomization schemes can be effective in this case. However, event-based schemes may trigger unnecessary re-randomization if events are frequent, e.g., re-randomizing every time a program outputs [6].

Key Takeaways

❶ *Effective re-randomization upper bound.* Our methodology for measuring various gadget sets systematically by considering the gadget convergence time helps compute the effective upper bound for re-randomization intervals of a re-randomization scheme. Our results show that this upper bound ranges from 1.5 to 3.5 seconds. Applying our methodology on their machines will help re-randomization adopters to make informed configuration decisions.

❷ *All leaked pointers are created equal for gadget convergence, but not for the speed of gadget convergence.* Regardless of the location of pointer leakage, we obtained the same number of minimum and extended footprint gadgets via JIT-ROP. This observation indicates that **any** pointer leak from an application's code segment is equally useful for attackers. However, the time for obtaining the gadgets varies for different leaked pointers.

❸ *Turing-complete operations.* Function, basic-block, or machine register level fine-grained randomization preserves Turing-complete expressive power of ROP gadgets, however, instruction-level randomization does not.

❹ *Connectivity.* Code connectivity is the main enabler of JIT-ROP. As the conventional entropy metric does not capture code connectivity, it should not be used to measure ASLR security under the JIT-ROP threat model.

❺ *Gadget quality.* Our findings suggest that current fine-grained randomizations do not impose significant gadget corruption.

7 Related Work

The related research has two themes: 1) demonstrating attacks and 2) discovering countermeasures. Attack demonstrations range from stack smashing [75], return-to-libc [60, 79, 106], to ROP [15, 17, 57], JOP [10], DOP [55], ASLR bypasses [8, 32, 40, 47, 55, 91], and CFI bypasses [7, 14, 15, 45, 56].

Researchers have also proposed a range of defenses for ROP attacks [1, 9, 18, 23, 27, 28, 33, 34, 36, 39, 46, 72, 74, 76–78, 85, 100, 108, 109], CFI bypass [108], and ASLR bypass [4–6, 21, 27, 32, 43, 52, 58, 59, 67, 68, 77, 96, 103–105]. A categorical representation of these defenses is given in our attack-path diagram (Figure 10 in the Appendix). Binary analysis tools are also available to understand [90] and mitigate [101] these ROP or code-reuse attacks.

Most of the above-mentioned defenses are variants of W⊕X (e.g., NEAR [104] and Heisenbyte [96]), memory safety (e.g., HardScope [73], Memcheck [71], AddressSanitizer [87], and StackArmor [20]), ASLR (e.g., fine-grained randomization [6, 21, 59, 86, 103, 105]), and CFI (e.g., CCFIR [108] and bin-CFI [109]). These defenses are capable of preventing most code-reuse attacks [8, 32, 40, 91] except a few cases such as inference attacks that are performed using zombie gadgets [92] or relative address space layout [48, 82]. The latest advancement in control-flow transfers such as MLTA [65] significantly advances CFI that can prevent most control-oriented attacks. Recent attention on non-control-oriented or data-only attacks [55, 56] motivated researchers to develop practical Data-Flow-Integrity (DFI) [16] solutions (details of non-control attacks in [22]). Currently, it is challenging to implement a practical DFI solution considering the overhead of data-flow tracking.

From the defense-in-depth perspective, it is desirable to have some degree of redundancy (e.g., CFI, ASLR, or complementary solutions like anomaly detection [107]) in system protection. A single deployed defense may be compromised due to unknown implementation flaws or configuration issues. Thus, investigations in multiple directions [12, 54, 89, 100] is necessary for gauging the feasibility of existing defenses. Our work investigates various aspects of ASLR – including timing – by evaluating security metrics such as various gadget sets, interval choices, and code pointer leakages. We also assess how security tools in the ASLR domain impact on these security metrics, quantitatively.

8 Conclusions

We presented multiple general methodologies for quantitatively measuring the ASLR security under the JIT-ROP threat model and conducted a comprehensive measurement study. One method is for computing the number of various types of gadgets and their quality. Another method is for experimentally determining the upper bound of re-randomization intervals. The upper bound helps guide re-randomization adopters to make more informed configuration decisions.

Acknowledgments

We thank our shepherd, Georgios Portokalidis, for his support and valuable feedback for this work. We also thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by the NSF under grant No. CNS-1838271.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] Patroklos Argyroudis and Chariton Karamitas. 2012. Exploiting the jemalloc memory allocator: Owning Firefox's heap. *Blackhat USA* (2012).

[3] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *NDSS*.

[4] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1342–1353.

[5] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *USENIX Security Symposium*. 433–447.

[6] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 268–279.

[7] Andrea Biondo, Mauro Conti, and Daniele Lain. 2018. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. NDSS.

[8] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[9] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 353–362.

[10] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 30–40.

[11] Michael D. Brown and Santosh Pande. 2019. Is less really more? Towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*.

[12] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.

[13] Exploit Database by Offensive Security. 2012. HT Editor 2.0.20 - Local Buffer Overflow (ROP). https://www.exploit-db.com/exploits/22683. Last accessed 05 May 2020.

[14] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*. 161–176.

[15] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium*. 385–399.

[16] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 147–160.

[17] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 559–572.

[18] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. 2009. DROP: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security*. Springer, 163–177.

[19] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 514–529.

[20] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*.

[21] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 50–61.

[22] Long Cheng, Hans Liljestrand, Md Salman Ahmed, Thomas Nyman, Trent Jaeger, N Asokan, and Danfeng Daphne Yao. 2019. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Secure Development, SecDev 2019*. Institute of Electrical and Electronics Engineers Inc., 114–128.

[23] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. In *NDSS*.

[24] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 952–963.

[25] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. 2016. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies* 2016, 4 (2016), 454–469.

[26] Stanley Crispin Cowan, Seth Richard Arnold, Steven Michael Beattie, and Perry Michael Wagle. 2010. Pointguard: method and system for protecting programs against pointer corruption attacks. US Patent 7,752,459.

[27] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 763–780.

[28] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 292–307.

[29] Charlie Curtsinger and Emery D. Berger. 2013. Stabilizer: Statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 219–228.

[30] Rapid7 Vulnerability & Exploit Database. 2018. Firebird Relational Database CNCT Group Number Buffer Overflow. https://www.rapid7.com/db/modules/exploit/windows/misc/fb_cnct_group. Last accessed 05 May 2020.

[31] Rapid7 Vulnerability & Exploit Database. 2018. ProFTPD 1.3.2rc3 - 1.3.3b Telnet IAC Buffer Overflow (Linux). https://www.rapid7.com/db/modules/exploit/linux/ftp/proftp_telnet_iac. Last accessed 05 May 2020.

[32] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS*.

[33] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2009. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*. ACM, 49–54.

[34] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 40–51.

[35] Stephen Dolan. 2013. mov is Turing-complete. *Cl. Cam. Ac. Uk* (2013), 1–4.

[36] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 75–88.

[37] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 781–796.

[38] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 28–39.

[39] Ivan Fratrić. 2012. ROPGuard: Runtime prevention of return-oriented programming attacks. In *Technical report*.

[40] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *NDSS*.

[41] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *NDSS*.

[42] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1009–1022.

[43] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *USENIX Security Symposium*. 475–490.

[44] Will Glozer. 2018. wrk-a HTTP benchmarking tool. https://github.com/wg/wrk. Last accessed 03 May 2020.

[45] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 575–589.

[46] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*. 417–432.

[47] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)*. 105–119.

[48] Enes Göktaş, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. 2018. Position-independent code reuse: On the effectiveness of ASLR in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy*

(EuroS&P). IEEE, 227–242.

[49] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[50] William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. 2017. Zipr: Efficient Static Binary Rewriting for Security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 559–566.

[51] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic heap layout manipulation for exploitation. *arXiv preprint arXiv:1804.08470* (2018).

[52] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd my gadgets go?. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 571–585.

[53] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 1–11.

[54] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. 2012. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 7–7.

[55] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.

[56] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1868–1882.

[57] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Branch regulation: Low-overhead protection from code reuse attacks. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 94–105.

[58] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 339–348.

[59] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted Code Randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 461–477.

[60] Sebastian Krahmer. 2005. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. https://users.suse.com/~krahmer/no-nx.pdf. Last accessed 10 May 2020.

[61] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Vol. 14.

[62] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2018. Code-Pointer Integrity. In *The Continuing Arms Race*. Association for Computing Machinery and Morgan & Claypool, 81–116.

[63] Musl libc. 2011. A lightweight standard C library. https://www.musl-libc.org. Last accessed 09 May 2020.

[64] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *Acm SIGPLAN Notices* 35, 11 (2000), 168–177.

[65] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1867–1881.

[66] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *NDSS*.

[67] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 280–291.

[68] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *USENIX Security Symposium*. 139–156.

[69] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 941–951.

[70] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *NDSS*, Vol. 26. 27–30.

[71] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM SIGPLAN notices* 42, 6 (2007), 89–100.

[72] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 577–587.

[73] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, and Ahmad-Reza Sadeghi. 2019. HardScope: Hardening Embedded Systems Against Data-Oriented Attacks. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[74] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 49–58.

[75] Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7, 49 (November 1996). http://www.phrack.com/issues.html?issue=49&id=14

[76] Vasilis Pappas, Michalis Polychronakis, and Angelos Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security Symposium*. 447–462.

[77] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 601–615.

[78] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 144–164.

[79] Alexander Peslyak. 1997. "return-to-libc" attack. *Bugtraq, Aug* (1997).

[80] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*. 1–18.

[81] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012), 2.

[82] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *NDSS*.

[83] Sascha Schirra. 2014. Ropper tool. https://github.com/sashs/Ropper. Last accessed 4 July 2018.

[84] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 745–762.

[85] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. 2014. Evaluating the effectiveness of current anti-ROP defenses. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 88–108.

[86] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *NDSS*.

[87] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.

[88] Hovav Shacham. 2007. 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*. ACM, 552–561.

[89] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security*. ACM, 298–307.

[90] Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, et al. 2016. Sok: (State of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[91] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 574–588.

[92] Kevin Z. Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. 2016. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 954–968.

[93] Alexander Sotirov. 2007. Heap feng shui in JavaScript. *Black Hat Europe* (2007).

[94] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*. ACM, 1–8.

[95] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[96] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 256–267.

[97] Corelan Team. 2018. Universal DEP/ASLR bypass with msvcr71.dll and mona.py. https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/. Last accessed 10 February 2018.
[98] PaX Team. 2003. PaX address space layout randomization (ASLR). (2003).
[99] Uptrends. 2020. Website Speed Test. https://www.uptrends.com/tools/website-speed-test. Last accessed 03 May 2020.
[100] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1675–1689.
[101] Victor Van Der Veen, Enes Göktaş, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.
[102] Konstantin vz'One Enchant. 2014. ftpbench-benchmark for load testing FTP servers. https://github.com/selectel/ftpbench. Last accessed 03 May 2020.
[103] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*. ACM, 157–168.
[104] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 35–46.
[105] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*. 367–382.
[106] Rafal Wojtczuk. 2001. The advanced return-into-lib (c) exploits: PaX case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e* (2001).
[107] Danfeng Yao, Xiaokui Shu, Long Cheng, and Salvatore J Stolfo. 2017. Anomaly detection as a service: challenges, advances, and opportunities. *Synthesis Lectures on Information Security, Privacy, and Trust* 9, 3 (2017), 1–173.
[108] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.
[109] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*. 337–352.

A Appendix

A.1 Register corruption analysis

Typically, a gadget contains a core instruction (other than *ret*) that serves the purpose of that gadget. For example, the core instruction of the gadget in Listing 1 is *mov eax, edx* and the gadget serves as a move register (MR) gadget. The core instruction is the instruction that an attacker needs. All the instructions (except *ret*) before or after the core instruction are usually unnecessary. However, these extra instructions may modify the source or destination register of a core instruction. If these extra instructions modify the registers of a core instruction, we treat the gadget as a corrupted gadget. In Listing 1, the instruction (*mov edx, dword ptr [rdi]*) before the core instruction modifies the source register (*edx*) of the core instruction and the instructions (*shr eax, 0x10; xor eax, edx*) after the core instruction modify the destination register (*eax*). We identify three scenarios when core instructions get corrupted as follows:

(1) ***Scenario 1***: A core instruction is only affected by the instruction(s) before the core instruction,
(2) ***Scenario 2***: A core instruction is only affected by the instruction(s) after the core instruction, and
(3) ***Scenario 3***: A core instruction is affected by both the instruction(s) before/after the core instruction.

We identify three types of gadgets considering the three scenarios above where the core instructions get corrupted. Figure 6 shows the three type of gadgets. Each gadget has one or more instructions before or after the core instruction. For example, Type 1 gadget in Figure 6 has a core instruction in the middle and one or more instructions before or after the core instruction. The core instruction has two registers for this kind. One or more instruction(s) before the core instruction may modify the source register (*rdx*) in Figure 6a. Similarly, one or more instruction(s) after the core instruction may modify the destination register (*rax*) in the figure.

Listing 1: An example gadget where the core instruction is "mov eax, edx;".

```
mov edx, dword ptr [rdi]; | mov eax, edx; | shr eax, 0x10; xor eax, edx; ret;
```

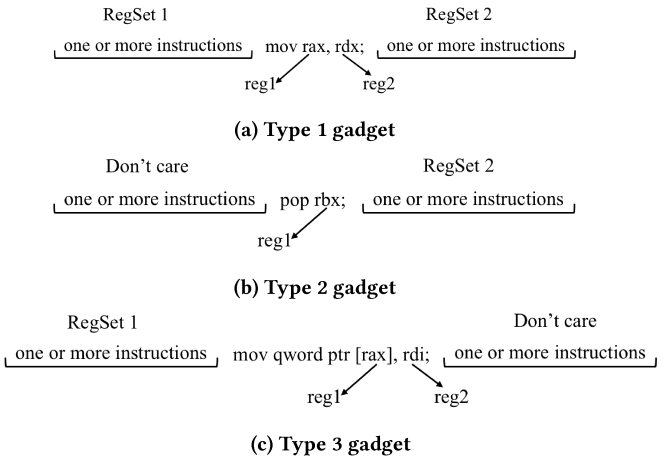

Figure 6: A set of gadget types for measuring the quality of individual gadget through the register corruption analysis

However, for Type 2 gadget in Figure 6b, the core instruction has just one register. That means that the additional instructions before the core instruction cannot affect the register of the core instruction. Thus, we do not care the instructions before the core instruction. For Type 3 gadget in Figure 6c, the core instruction writes the value of *rdi* to a memory location pointed by *rax*. That is why we do not care if the register (*rax*, *rdi*) values get modified by the instructions after the core instructions.

A gadget is corrupted if registers in the core instruction get modified. We perform our register corruption analysis by identifying the corrupted registers in the core instructions of a gadget as follows.

First, we identify the set of instructions (before or after the core instruction) that can modify the source or destination register of the core instruction. We find that 17 instructions (*mov*, *lea*, *add*, *sub*, *imul*, *idiv*, *pop*, *inc*, *dec*, *xchg*, *and*, *or*, *xor*, *not*, *neg*, *shl*, and *shr*) can modify a register value of a core instruction. That means that these instructions use the source register of a core instruction as its destination register or the destination register of a core instruction as its source register. We treat the registers of such instructions as conflicting registers.

Second, we extract the conflicting registers (`RegSet1`) for Types 1 and 3 gadgets and `RegSet2` for Types 1 and 2.

Third, if the `RegSet1` and/or `RegSet2` contain more than one conflicting registers, we treat the core instruction of that gadget as corrupted, i.e., the gadget itself is corrupted.

Table 5: Gadgets used in advanced ROP attacks [8, 14, 15, 45, 91] . △ indicates an addition/subtraction/multiply/division. ϕ indicates logical operations such as and, or, left-shift, and right-shift. ▽ indicates any operation that modifies stack pointer (SP). SN → Short name. TC? indicates whether a gadget is included in the Turing-complete gadget set or not.

Gadget types	Purpose	Minimum footprint	Example	TC?	SN	Source
Move register	Sets the value of one register by another	mov reg1, reg2; ret	mov rdi, rax; ret	✓	MR	[91]
Load register	Loads a constant value to a register	pop reg; ret	pop rbx; ret	✓	LR	[14, 91]
Arithmetic	Stores an arithmetic operation's result of two register values to the first	△ reg1, reg2; ret	add rcx, rbx; ret	✓	AM	[91]
Load memory	Loads a memory content to a register	mov reg1, [reg2]; ret	mov rax, [rdx]; ret	✓	LM	[14, 91]
Arithmetic load	△ a memory content to/from/by a register and store in that register	△ reg1, [reg2]; ret	add rsi, [rbp]; ret	✓	AM-LD	[91]
Store memory	Stores the value of a register in memory	mov [reg1], reg2; ret	mov [rdi], rax; ret	✓	SM	[91]
Arithmetic store	△ a register value to/from/by a memory content and stores in that memory	△ [reg1], reg2; ret	sub [ebx], eax; ret	✓	AM-ST	[91]
Logical	Performs logical operations	ϕ reg1, reg2; ret ϕ reg1, const; ret ϕ [reg1], reg2; ret ϕ [reg1], const; ret	shl rax, cl; ret;	✓	LOGIC	[81]
Stack pivot	Sets the stack pointer, SP	▽ sp, reg	xchg rsp, rax	×	SP	[91]
Jump	Sets instruction pointer, EIP.	jmp reg	jmp rdi	✓	JMP	[91]
Call	Jumps to a function through a register or memory indirect call	call reg or call [reg]	call rdi	✓	CALL	[91]
System Call	Invokes system functions	syscall or int 0x80; ret	syscall	✓	SYS	[81]
Call preceded	Bypasses call-ret ROP defense policy	mov [reg1], reg2; call reg3	mov [rsp], rsi; call rdi	×	CP	[14]
Context switch	Allows processes to write to Last Branch Record (LBR) to flash it	long loop.	3dd4: dec, ecx 3dd5: fmul, [BC8h] 3ddb: jne, 3dd4	×	CS1	[14]
Flashing	Clears the history of LBR (Last Branch Record)	Any simple call preceded gadgets with a ret instruction	jmp A ... A: mov rax, 3; ret;	×	FS	[15]
Terminal	Bypasses kBouncer heuristics	Any gadgets that are 20 instructions long	N/A	×	TM	[15]
Reflector	Allows to jump to both call-preceded or non-call-preceded gadgets	mov [reg1], reg2; call reg3; ... ; jmp reg4	mov [rsp], rsi; call rdi; ... ; jmp rax	×	RF	[14]
Call site	This gadget chains the control to go forward when we have the control on the stack and ret	call reg or call [reg]; ... ret;	call rdi; ... ret;	×	CS2	[45]
Entry point	This gadget chains the control to go forward when we have the control of a call instruction	pop rbp; ... call/jmp reg or call/jmp [reg]	pop rbp ... call/jmp reg or call/jmp [reg]	×	EP	[45]
BROP	Restores all saved registers	pop rbx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;	pop rbx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;	×	BROP	[8]
Stop	Halts the program execution	Infinite loop	4a833dd4: inc rax 3ddb: jmp 3dd4	×	STOP	[8]

Table 6: Gadgets with gadget types in the priority and MOV TC gadget sets.

Priority		MOV TC	
Type	**Gadget**	**Type**	**Gadget**
LR	1. pop reg 2. pop reg; pop reg	MR	1. mov reg, reg/const
AM	3. add reg, const	ST	2. mov [reg], reg
LM	4. mov reg, [reg]; ret	STCONSTEX	3. mov [reg+offset], reg/const
JMP	5. jmp reg	STCONST	4. mov [reg], const
ST	6. mov [reg], reg; ret	LM	5. mov reg, [reg]
SP	7. xchg rsp, reg	LMEX	6. mov reg, [reg+offset]
LOGIC	8. xor reg, reg 9. xor reg, const	SYS	7. syscall
MR	10. mov reg, reg 11. mov reg, const		
CALL	12. call reg 13. mov reg, reg, call reg		
SYS	14. syscall		

In this way, we measure the register corruption rate for `MV`, `LR`, `AM`, `LM`, `AM-LD`, `SM`, `AM-ST`, `SP`, and `CALL` gadgets by dividing the number of corrupted gadgets by the number of all gadgets.

A.2 Validation of randomization results

We evaluate the randomization tools, i.e., Zipr [50], SR [25], MCR [53], and CCR [59] using the common set of applications and libraries that the four randomization tools can randomize. Figure 7 shows the reduction of Turing-complete gadgets observed for four (4) randomization tools using the common set of applications and libraries. In most cases, the reduction using a different set of applications and libraries is similar to the reduction using a common set of applications.

Table 7: Key differences in various randomization and re-randomization schemes evaluated.

Tools	Randomization Scheme(s)	Randomization Time	Compiler Assistance Required?	Techniques	Performance Overhead
Shuffler [105]	Function-level re-randomization	Runtime	No	- Loads itself as a user space program - Contains a separate thread for shuffling the functions continuously	14.9% [105]
Zipr [50]	Instruction-level randomization	Static rewriting	No	- Reorders all instructions and generates ILR static rewrite rules - Executes randomly scatter instructions using a process-level virtual machine (PVM) utilizing static rewrite rules or a fall-through map - Keeps the same layout unless rewrite again	<5% [50]
SR [25]	Function-level randomization	Load time reorder	No	- Adds a linker wrapper that intercepts calls to the linker and asks the selfrando library to extract the necessary information to reorder functions - Reorders functions every time when a binary is loaded into memory	<1% [25]
MCR [53]	Function- and register-level randomization	Compile & Link time reorder	Yes	- Reorders functions and machine registers during link time optimization - Implements compile-time randomization but defers compilation until all translation units have been converted to bitcode - Keeps the same layout unless compiled and built again	1% [53]
CCR [59]	Function and block-level randomization	Installation time	Yes	- Extracts metadata during compilation - Reorders functions and basic-block based on the metadata - Keeps the same layout unless re-randomized again	0.28% [59]

Table 8: Register corruption for various gadgets. The numbers before and after the vertical bar (|) represent the average number of unique register usage and register corruption rate in a gadget, respectively. CG → Coarse-grained. FG → Fine-grained. Fine-grained versions prepared using SR [25].

	Program	MV	LR	AM	LM	AM-LD	SM	AM-ST	SP	CALL	Average	
CG	Nginx	4 | 11%	2 | 0.3%	3 | 21%	3 | 44%	3 | 6%	2 | 47%	2 | 13%	2 | 6%	2 | 9%	—	
	Apache	4 | 16%	2 | 0.5%	3 | 37%	2 | 26%	3 | 10%	2 | 24%	2 | 5%	2 | 3%	2 | 7%	—	
	ProFTPD	3 | 69%	2 | 0.6%	3 | 7%	2 | 24%	2 | 20%	2 | 16%	2 | 11%	4 | 1%	1 | 6%	—	
	Average	**4 | 32%**	**2 | 0.5%**	**3 | 21.7%**	**2 | 31.3%**	**3 | 12%**	**2 | 29%**	**2 | 9.7%**	**3 | 3.3%**	**2 | 7.3%**	3 | 16.3	
FG	Nginx	3 | 9%	1 | 0.1%	2 | 0.1%	3 | 15%	2 | 45%	2 | 13%	2 | 47%	1 | 7%	2 | 4%	—	
	Apache	3 | 27%	1 | 1%	3 | 41%	3 | 27%	2 | 19%	2 | 41%	2 | 0%	2 | 2%	3 | 27%	—	
	ProFTPD	3 | 14%	2 | 1%	3 | 4%	2 | 19%	2 | 22%	2 | 35%	2 | 6%	3 | 11%	3 | 28%	—	
	Average	**3 | 16.7%**	**1 | 0.7%**	**3 | 15%**	**3 | 20.3%**	**2 | 28.7%**	**2 | 29.7%**	**2 | 17.7%**	**2 | 6.7%**	**3 | 19.7%**	2 | 17.24	~5.7%⇑

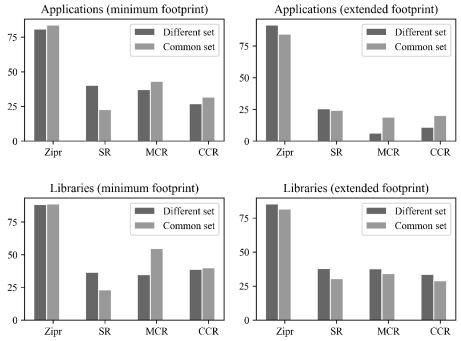

Figure 7: Reduction (%) of TC gadgets observed for four (4) randomization tools using the common set of applications and libraries that the randomization tools can randomize.

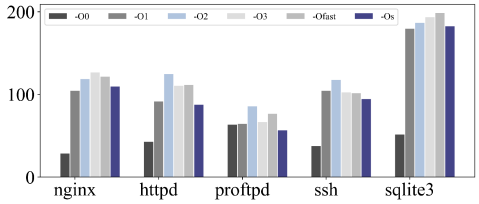

Figure 8: The number of Turing-complete minimum footprint gadgets at different optimization levels for GCC.

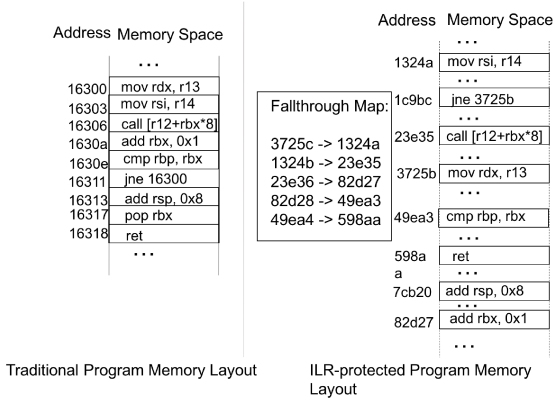

Figure 9: Instruction location randomization. This figure is adopted from ILR [52].

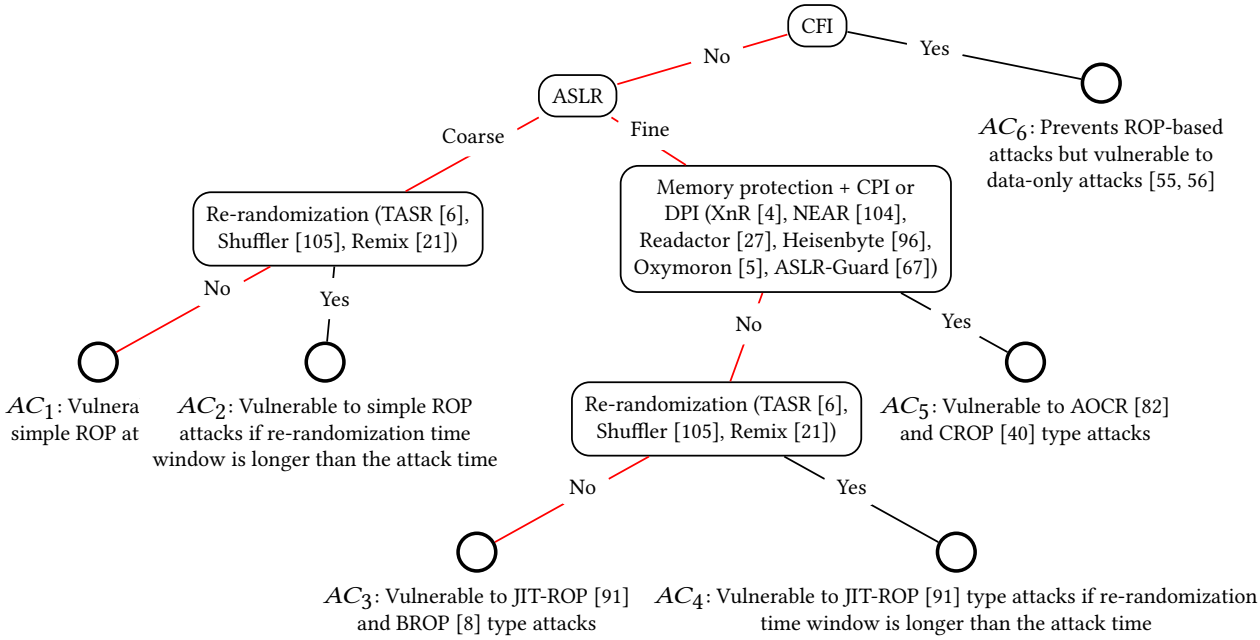

Figure 10: High-level view of the types of ROP attacks and attack-paths based on various security measures. Each rectangle and circle indicate security measures and attack types, respectively. AC stands for attack condition. All the attack conditions have W⊕X, PIE, Canary, and RELRO implicitly.