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

Salman Ahmed*, Ya Xiao†, Kevin Snow‡, Gang Tan‡, Fabian Monrose§, Danfeng (Daphne) Yao∗
∗Computer Science, Virginia Tech, †Zeropoint Dynamics, LLC, ‡Computer Science and Engineering, Penn State University, §Computer Science, UNC at Chapel Hill
{ahmedms,yax99,danfeng}@vt.edu, kevin@zeropointdynamics.com, gtan@cse.psu.edu, fabian@cs.unc.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.

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 diversification or randomization, reorders and relocates program elements. Fine-grained randomization would defeat conventional RO 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 RO gadget purposes.

JIT-ROP attacks have the ability to 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 an application from all control-oriented attacks. The recently proposed 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/pointer leakage. Thus, a measurement mechanism to design secure systems is necessary for protecting pointers from leakage. Besides, from a defense-in-depth perspective, it is important for a critical system 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 determining 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 does 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, and register levels). Our key experimental findings and technical contributions are summarized as follows.

- We provide a methodology to compute the upper bound $T$ for re-randomization intervals. If the re-randomization interval is less than $T$, 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 $T$ 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 derandomizing an address space layout, suggesting that an attacker’s 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. The time 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 and 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 the 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 of the module remains unchanged. The Position Independent Executable (PIE) option allows the main executable to be run as position independent code, i.e., PIE relocates the code and data segments. 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, attempts to relocate all the segments of the main executable of a process, including shared libraries, heap, stack, and memory-mapped regions and restructures the internal layouts of these 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.

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., CFI enforcement with spatial and temporal guarantees, CFI implementations with various granularities like [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, 70, 78, 107, 108] and CPI [5, 26, 37, 61, 62, 67, 69]. For similar reasons, we also omit memory permission protections (e.g., XinR [4], NEAR [104], Readactor [27] and Heisenbyte [96]) for this paper. Execute-only-Memory (XOM)$^5$ [64] and Execute-no-Read (XnR) [4] style defenses are also powerful. But, attacks such as AOCR [82] and code inference [92] are still possible with these defenses. We also discuss the need for

---

$^1$https://github.com/securesystemslab/multicompiler
$^2$https://github.com/immunant/selfrando
$^3$https://github.com/kevinKoo/CCR
$^4$https://github.com/securesystemslab/multicompiler
$^5$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.
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 engines and does not use any flaws of JIT compilers to per-form 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 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 of 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., system SYS gadgets) \[81\].

**Definition 2.** The upper bound $T_{\mathcal{A}}^P$ 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 $T_{\mathcal{A}}^P < T_{\mathcal{A}}^P$, the set of gadgets obtained under $T_{\mathcal{A}}^P$ does not converge to any of the four gadget sets.

**Extended footprint (EX-FP) gadgets**: A gadget is an extended footprint gadget if it is an instance of the Turing-complete gadget set or an instance of attack-specific gadgets. An EX-FP gadget may contain additional instructions that may cause side effects in an attack payload. EX-FP gadgets include longer memory addressing expressions.

**Minimum footprint (MIN-FP) gadgets**: A minimum footprint gadget is an instance of the Turing-complete gadget set or attack-specific gadgets 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 MOVs 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.

![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.](image)

To overcome both coarse- and fine-grained ASLR and conduct an attack to gain privileged operations, an attacker needs to perform the 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 2-3 for JIT-ROP and steps 2 and 3 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 3). 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.** Fine-grained ASLR randomizes the base addresses, as well as the internal structures of various segments and modules of a process. Thus, simply adjusting the addresses of pre-computed gadgets (as in the basic ROP) no longer works. An attacker needs to find gadgets dynamically at the time of an exploit. Scanning a process’ address space linearly for gadgets by starting 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 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 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 (0x11F9000-0x11FF9FF), disassembles the data, searches for call and jmp instructions to find other pointers (0x11FB410 and 0x11FCCF4) 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.

**Gadget identification.** In step 3 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 [107] and bin-CFI [108]) 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 5 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 3 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 functions or system gadgets. An attacker generates a payload dynamically at step 3 in the presence of fine-grained code randomization or manually at step 5 in the presence of 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 W0X. An attacker may utilize the same vulnerability as in step 2 or a different vulnerability to hijack a program’s control flow at step 2 to redirect the flow to the stored payload. The target of a payload is to achieve an attack goal, e.g., memory leak or launching a malicious application/root shell.

It is desirable for attackers to obtain attack chains that have minimal side effects, i.e., having a payload that fulfills attack goals without generating any unnecessary computation. However, this property may not be guaranteed if the gadget availability is limited by code randomization. We refer to the side effect of gadgets as footprints. We defined the minimum footprint gadget and extended footprint gadget 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 or re-randomization. Our approach is to count the number of gadgets that are available to attackers under the JIT-ROP code harvest mechanism. Other challenges are how to quantify the difficulty of accessing internal system functions and 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. In order to quantify the quality of gadget chains, we design a register-level measurement heuristic to compute 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). The Turing-complete, priority, and MOV TC gadget sets with some attack-specific gadgets (e.g., CP, RF, CS2, and EP) are appropriate for our evaluation because we can precisely identify those gadgets. Some attack-specific gadgets such as CS1, FS, etc. are very application-specific and do not have concrete forms or attack goals. These gadgets are used to trick defense mechanisms. We leave these gadgets out of our evaluation. We also include gadgets from three real-world ROP payloads from Metasploit [30, 31] and Exploit-Database [13]. We 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 version 3.9.0, version 3.8.0 and GCC version 5.4.0 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 (0x8F 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:


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 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, we discuss the code randomization and re-randomization tools briefly in the following paragraphs.

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.

Multicomplier (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 as its compilation engine.

Compiler-Assisted Code Randomization (CCR) [59] applies the code diversification at the installation time, i.e., rewrites an executable binary after 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 as its compilation engine.

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 analysis and measurement process. We will publish our JIT-ROP native implementation, analysis tool and data.

We perform our experiments on the latest and stable versions of applications including bzip2, cherokee, hiawatha, http, lighttpd, mupdf, nginx, openssl, proftpd, sqlite, openssh, httptpd, 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 libcrypto, libgmp, libhogweed, libxml2, libpcre, libgcrypt, libgnutls, libpq-error, libtasn1, libz, libnettle, libopenjp2, libpng16, libtomcrypt, libunistring, libxml2, libmozjs1k, libmozsmdb, libxml, libmozsqlite3, libgplibs, libwebkit2gtk-3.0, and musl. We select these applications or dynamic libraries by considering the fact that many attackers demonstrate their attacks on most of these applications or libraries. Besides, these applications/libraries include a diverse set of areas such as the web server, browser, PDF reader, networking, database, and libraries in cryptography, math, image, and system.

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

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Applications (20 Total)</th>
<th>Libraries (25 Total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-randomization interval</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Instruction-level rand</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>Function-level rand</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>Function + register-level rand</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>Basic block-level rand</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

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 (σ = 0.8) in the minimum time with the priority and MOV TC gadget sets having 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.

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/browsers. Payload* → average of three payload sets.

<table>
<thead>
<tr>
<th>Gadget set</th>
<th>Minimum (s)</th>
<th>Average (s)</th>
<th>Leak (%)</th>
<th>Analysis (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>2.2</td>
<td>4.3</td>
<td>17</td>
<td>85</td>
</tr>
<tr>
<td>Priority</td>
<td>1.5</td>
<td>3.5</td>
<td>13</td>
<td>87</td>
</tr>
<tr>
<td>MOV TC</td>
<td>3.5</td>
<td>5.3</td>
<td>16</td>
<td>84</td>
</tr>
<tr>
<td>Payload*</td>
<td>2.1</td>
<td>4.8</td>
<td>12</td>
<td>88</td>
</tr>
<tr>
<td>Average</td>
<td>2.3s</td>
<td>4.6s</td>
<td>14.5</td>
<td>85.5%</td>
</tr>
</tbody>
</table>

Figure 3 shows the minimum time to obtain the Turing-complete gadget set from individual application/browser along 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 (●) 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.

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), mopdf (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
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 depends 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td># of leaked addresses</td>
<td># of leaked addresses</td>
<td># of leaked addresses</td>
</tr>
<tr>
<td>hiawatha</td>
<td>41</td>
<td>9</td>
<td>223</td>
</tr>
<tr>
<td>lighttpd</td>
<td>91</td>
<td>16</td>
<td>634</td>
</tr>
<tr>
<td>nginx</td>
<td>53</td>
<td>8</td>
<td>235</td>
</tr>
<tr>
<td>proftpd</td>
<td>114</td>
<td>26</td>
<td>788</td>
</tr>
<tr>
<td>thttpd</td>
<td>131</td>
<td>17</td>
<td>523</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>172</td>
</tr>
</tbody>
</table>

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. STCONST, 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, and LMEX is 15. As expected, the number of Load-Memory (LM) gadgets is low, which indicates the scarcity of these gadgets. Besides, we observe the number of Store-Constant (STCONST) is also low, which is necessary for performing comparison and conditional operations. The average number of (SYS) gadgets is adequate (23).

Our re-randomization upper bound calculation includes the overhead of analyzing different gadget types. Thus, we perform additional analyses to investigate how far the address space is leaked and how much time is spent on gadget analysis. According to our findings (Table 2), we find that on average around 15% of the time is spent on leaking address space, while the rest for gadget searching. 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 value of 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. 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].

Table 3: Impact of locations of pointer leaks on gadget availability. The same application has different numbers of leaked addresses for different tools because each tool uses a different backend (i.e., compiler). Different backends produce different sized executables of the same program. 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.
The numbers of leaked addresses in Table 3 are different for different randomization schemes because we use different tools to enforce these randomization schemes. Different tools use different backends and 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 run the code harvest process starting 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 opensfl 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.

![Figure 4: Impact of starting pointer locations on gadget harvesting time. Each ㅇ indicates the time for harvesting the Turing-complete gadget set. The minimum, maximum, and average time is calculated by running code harvest process from multiple starting 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.](image)

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.

**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 used in Oxymoron [5], Readaecor [27], XnR [4], NEAR [104], Heisenbyte [96], and ASLR-Guard [67] tools. A large-scale quantitative assessment on the effectiveness of these security tools is necessary to find out the practicality and feasibility of these tools 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 randomizing with the four fine-grained randomization schemes. Each application or library is run/loaded in memory for 100 times after randomizing 100 times when necessary. The numbers of gadgets are averaged over 100 runs/loads of an application or library. Then the numbers of gadgets are averaged over the number of 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 scheme 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 Turing-complete operation. Since the numbers of applications/libraries are different for randomization schemes, we also evaluate these schemes and validate the results (Figure 7 in the Appendix) with a common set of applications and libraries. The results show a consistent reduction in all the schemes.

The Turing-complete expressiveness of ROP gadgets is preserved in the randomized versions of 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 the number of applications/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 syscall through libc. This is why the number of `syscall` gadgets is low (2–3) and one gadget loss leads to around 33% reduction. Since SR is only able to randomize a light-weight version of libc (`musl`),

---

[1] One compilation with 100 runs, 100 times randomization, 100 times compilation, and 100 times rewriting are required for SR, CCR, MCR, and Zipt, respectively.
we observe slightly high values (84) and low reduction for system gadgets in Table 4 for the function-level scheme.

We also assess the gadget availability under a single randomization pass of Shuffler [105]. We take 100 consecutive address space snapshots from nginx after each re-randomization with an interval of 30 seconds and manually find gadgets from the snapshots. On average, we observe a 24% and 3% reduction in gadget availability for minimum and extended footprint gadgets compared to a non-randomized version of nginx, respectively. The low reductions are expected, as Shuffler’s security relies on the capability of continuously shuffling code locations, 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 randomization schemes have low gadget reduction. On the other hand, instruction-level randomization perturbs the memory layout significantly. That is why we observe a large reduction in gadget availability 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 ftplibbench [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 compared to the coarse-grained randomization.

---

**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.

<table>
<thead>
<tr>
<th>Randomization schemes</th>
<th>Granularity</th>
<th>▼ (%)</th>
<th>▲ (%)</th>
<th>Memory</th>
<th>Assignment</th>
<th>Arithmetic</th>
<th>Logical</th>
<th>Control</th>
<th>Function</th>
<th>System</th>
<th>TC Preserved?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIN-PP</td>
<td>EX-PP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst. level rando. [50]</td>
<td>Inst.</td>
<td>79.7</td>
<td>82.5</td>
<td>97.4</td>
<td>92.7</td>
<td>38.8</td>
<td>35.7</td>
<td>92.8</td>
<td>64.9</td>
<td>85.8</td>
<td>85.4</td>
</tr>
<tr>
<td>Func. level rando. [25]</td>
<td>FB</td>
<td>27.63</td>
<td>36.55</td>
<td>0.8</td>
<td>29.2</td>
<td>10.6</td>
<td>43.5</td>
<td>19.9</td>
<td>35.1</td>
<td>35.9</td>
<td>21.1</td>
</tr>
<tr>
<td>Func. + Reg. level rando. [53]</td>
<td>FB &amp; Reg.</td>
<td>17.62</td>
<td>42.37</td>
<td>-8.3</td>
<td>35.0</td>
<td>-5.1</td>
<td>35.2</td>
<td>26.1</td>
<td>44.9</td>
<td>21.3</td>
<td>38.1</td>
</tr>
<tr>
<td>Block level rando. [59]</td>
<td>BB</td>
<td>19.58</td>
<td>44.64</td>
<td>5.5</td>
<td>40.9</td>
<td>6.1</td>
<td>47</td>
<td>26.1</td>
<td>33.7</td>
<td>20.4</td>
<td>37.4</td>
</tr>
<tr>
<td></td>
<td>MIN-PP</td>
<td>EX-PP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst. level rando. [50]</td>
<td>Inst.</td>
<td>81.3</td>
<td>92.2</td>
<td>93.7</td>
<td>96.1</td>
<td>60.7</td>
<td>93</td>
<td>91.8</td>
<td>84.9</td>
<td>84.5</td>
<td>90.4</td>
</tr>
<tr>
<td>Func. level rando. [25]</td>
<td>FB</td>
<td>46.5</td>
<td>43.8</td>
<td>24.2</td>
<td>71.1</td>
<td>15.9</td>
<td>31</td>
<td>41.2</td>
<td>65.4</td>
<td>56.9</td>
<td>25</td>
</tr>
<tr>
<td>Func. + Reg. level rando. [53]</td>
<td>FB &amp; Reg.</td>
<td>44.2</td>
<td>43.9</td>
<td>35.5</td>
<td>64.8</td>
<td>35.3</td>
<td>43.4</td>
<td>63.2</td>
<td>61.8</td>
<td>44.8</td>
<td>49.0</td>
</tr>
<tr>
<td>Block level rando. [59]</td>
<td>BB</td>
<td>20.98</td>
<td>37.0</td>
<td>7.3</td>
<td>36.3</td>
<td>8.1</td>
<td>32.1</td>
<td>13.9</td>
<td>55.9</td>
<td>24.8</td>
<td>31.6</td>
</tr>
</tbody>
</table>

* For instruction-level randomization scheme [50], TC is not preserved for minimum footprint gadgets, but TC is preserved for extended footprint gadgets.
We observe the number of unique registers used in each gadget. The execution point is defined differently for different types of gadgets. This experiment measures the risks associated with a heap, stack, or data segment. It also suggests that the safeguard library pointer and adjusts pre-computed gadgets based on the 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. The execution point is defined differently for different types of applications. For example, the execution point for profiplt is when profiplt is ready to accept connections. We assume that i) coarse-grained randomization is enforced, and ii) adversaries are not able to 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 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 randomization/re-randomization of stack is more important than protecting/randomizing heap or global variables.

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 a heap, stack or data segment of an application 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. The execution point is defined differently for different types of applications. For example, the execution point for profiplt is when profiplt is ready to accept connections. We assume that i) coarse-grained randomization is enforced, and ii) adversaries are not able to 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 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 randomization/re-randomization of stack is more important than protecting/randomizing heap or global variables.

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.

6 Discussion

Metrics for evaluating fine-grained randomization. Traditionally, both coarse- and fine-grained randomization solutions use entropy to measure the effectiveness of hindering code-reuse attacks [25, 59, 98, 103]. Randomization tools such as PaX ASLR [98], SR [25], CCR [59], Remix [21], Binary stirring [103], ILR [52] and ASLP [58] use entropy values as the security metrics to evaluate the security of their randomization schemes. Some tools such as SR [25], CCR [59], Remix [21], and ASLP [58] calculate entropy as a function of the number of function- or basic-blocks. That means these tools permute the order of functions or basic blocks.

However, such an entropy measure is not useful under the JIT-ROP threat model, as chunks of code are still available. Including distances between permuted functions or basic blocks in the entropy computation 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 produced 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, profiplt, openssh, and sqlite3 to assess the impact. We find that the unoptimized code contains a smaller number of LM, SM, and MR gadgets than optimized code (Figure 8 in the Appendix). The main reason is due to 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 increase gadgets.

Reachability of gadgets. We design our experiments based on the availability of various kinds of gadgets. However, in reality, it is not an easy task to invoke the available gadgets. 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 the available gadgets utilizing the leaked memory to compare various code (re-)randomization techniques.

**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 $T_p^A$ (Definition 2). If the owners prioritize security over performance, they can set an interval much shorter than $T_p^A$.

**Need for randomizing Glibc.** Unfortunately, SR, CCR, MCR, and Zipr were all unable to completely randomize the Glibc implementation. For CCR and MCR, the LLVM Clang compiler (which CCR and MCR use as their compilers) does not have the support for certain GCC specific extensions (e.g., ASM GOTO) in Glibc. SR also cannot randomize some parts of Glibc. Therefore, we evaluate a lightweight version of the standard C library musl-1.1bc [63] instead of Glibc. Only Selfrando works on musl-1.1bc. Shuffler can reorder Glibc code by making a few modifications such as 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 so that we could isolate an individual defense’s security impact. In this work, we attempt to address 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 a future research direction.

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 destroy the availability of gadgets to attackers. However, destructive read defenses cannot eliminate all gadgets. In our future work, we plan to assess the availability of zombie and micro gadgets after applying destructive read defenses.

Another limitation is that our threat model assumes that code pointer obfuscation-based defense is not deployed. If used, code pointer obfuscation (e.g., CFI [61, 62], Oxymonor [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 these defenses under various attack conditions (e.g., Isomeron [32] and COOP [84]) are interesting problems.

One limitation of the time-based re-randomization schemes is that the time needs 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**

* 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 around 1.5 to 3.5 seconds. Applying our methodology on their machines will help re-randomization adopters to make more informed configuration decisions.

* **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 obtain 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.

* **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.

* **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.

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

### 7 Related Work

The research conducted in the system security area primarily 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, 107, 108], CFI bypass [107], 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 WàX (e.g., NEAR [104] and Heisenbyte [96]), memory safety (e.g., Hard-Scope [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 [107] and bin-CFI [108]). 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 and ASLR) in system protection. A single deployed defense may be compromised due to...
unknown implementation flaws or configuration issues. Thus, in 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.

Acknowledgment

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

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 \texttt{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/after the core instruction is unnecessary. However, these extra instructions may modify the source/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 \texttt{(mov edx, dword ptr [rdi])} before the core instruction modifies the source register (edx) of the core instruction and the instructions \texttt{(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.
3. **Scenario 3**: A core instruction is affected by 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. We also identify three types of gadgets considering the three scenarios above where the core instruction can get corrupted. Figure 6 shows the three types 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 that is 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 also modify the destination register (rax) in the figure.

**Listing 1**: An example gadget where the core instruction is \texttt{"mov eax, edx;"}.

```plaintext
Listing 1: An example gadget
```

**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 \texttt{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/after the core instruction is unnecessary. However, these extra instructions may modify the source/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 \texttt{(mov edx, dword ptr [rdi])} before the core instruction modifies the source register (edx) of the core instruction and the instructions \texttt{(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.
3. **Scenario 3**: A core instruction is affected by 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. We also identify three types of gadgets considering the three scenarios above where the core instruction can get corrupted. Figure 6 shows the three types 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 that is 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 also modify the destination register (rax) in the figure.

**Listing 1**: An example gadget where the core instruction is \texttt{"mov eax, edx;"}.

```plaintext
Listing 1: An example gadget
```

**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

---

**References**


[20] Rafael Wojtczuk. 2001. The advanced return-into-lib (c) exploits: PaX case study. Phrack Magazine, Volume 0x0b, Issue 0x3a, Philer 0x04 of 0x06 (2001).


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 as 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/after the core instruction) that can modify the source/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.

In this way, we measure the register corruption rate for MV, LR, AM, LM, AM-LD, SM, AM-STM, SP, and CALL gadgets.

### Table 5: Gadgets used in advanced ROP attacks

<table>
<thead>
<tr>
<th>Gadget types</th>
<th>Purpose</th>
<th>Minimum footprint</th>
<th>Example</th>
<th>TC?</th>
<th>SN</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move register</td>
<td>Sets the value of one register by another</td>
<td>mov reg1, reg2; ret</td>
<td>mov rdi, rax; ret</td>
<td>✓</td>
<td>MR</td>
<td>[91]</td>
</tr>
<tr>
<td>Load register</td>
<td>Loads a constant value to a register</td>
<td>pop reg; ret</td>
<td>pop rdx; ret</td>
<td>✓</td>
<td>LR</td>
<td>[14, 91]</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>Stores an arithmetic operation’s result of two register values to the first</td>
<td>△ reg1, reg2; ret</td>
<td>add rcx, rbx; ret</td>
<td>✓</td>
<td>AM</td>
<td>[91]</td>
</tr>
<tr>
<td>Load memory</td>
<td>Loads a memory content to a register</td>
<td>mov reg1, [reg2]; ret</td>
<td>mov rax, [rdx]; ret</td>
<td>✓</td>
<td>LM</td>
<td>[14, 91]</td>
</tr>
<tr>
<td>Arithmetic load</td>
<td>△ a memory content to/from/by a register and store in that register</td>
<td>△ reg1, [reg2]; ret</td>
<td>add rsi, [rbp]; ret</td>
<td>✓</td>
<td>AM-LD</td>
<td>[91]</td>
</tr>
<tr>
<td>Store memory</td>
<td>Stores the value of a register in memory</td>
<td>mov [reg1], reg2; ret</td>
<td>mov [rdi], rax; ret</td>
<td>✓</td>
<td>SM</td>
<td>[91]</td>
</tr>
<tr>
<td>Arithmetic store</td>
<td>△ a register value to/from/by a memory content and stores in that memory</td>
<td>△ [reg1], reg2; ret</td>
<td>sub [ebx], eax; ret</td>
<td>✓</td>
<td>AM-ST</td>
<td>[91]</td>
</tr>
<tr>
<td>Logical</td>
<td>Performs logical operations</td>
<td>ϕ reg1, reg2; ret</td>
<td>shl rax, cl; ret</td>
<td>✓</td>
<td>LOGIC</td>
<td>[81]</td>
</tr>
<tr>
<td>Stack pivot</td>
<td>Sets the stack pointer, SP</td>
<td>▽ sp, reg</td>
<td>xchg rsp, rax</td>
<td>×</td>
<td>SP</td>
<td>[91]</td>
</tr>
<tr>
<td>Jump</td>
<td>Sets instruction pointer, EIP</td>
<td>jmp reg</td>
<td>jmp rdi</td>
<td>✓</td>
<td>JMP</td>
<td>[91]</td>
</tr>
<tr>
<td>Call</td>
<td>Jumps to a function through a register or memory indirect call</td>
<td>call reg or call [reg]</td>
<td>call rdi</td>
<td>✓</td>
<td>CALL</td>
<td>[91]</td>
</tr>
<tr>
<td>System Call</td>
<td>Invokes system functions</td>
<td>syscall or int 0x80; ret</td>
<td>syscall</td>
<td>✓</td>
<td>SYS</td>
<td>[81]</td>
</tr>
<tr>
<td>Call preceded</td>
<td>Bypasses call-ret ROP defense policy</td>
<td>mov [reg1], reg2; call reg3</td>
<td>mov [rsp], rsi; call rdi</td>
<td>×</td>
<td>CP</td>
<td>[14]</td>
</tr>
<tr>
<td>Context switch</td>
<td>Allows processes to write to Last Branch Record (LBR) to flash it</td>
<td>long loop; 3dd4: dec ecx; 3dd5: fmul, [BC8h]; 3ddb: jne, 3dd4</td>
<td>×</td>
<td>CS1</td>
<td>[14]</td>
<td></td>
</tr>
<tr>
<td>Flashing</td>
<td>Clears the history of LBR (Last Branch Record)</td>
<td>Any simple call preceded gadgets with a ret instruction</td>
<td>jmp A; A: mov rax, 3; ret; 3dd4: dec ecx; 3dd5: fmul, [BC8h]; 3ddb: jne, 3dd4</td>
<td>×</td>
<td>FS</td>
<td>[15]</td>
</tr>
<tr>
<td>Terminal</td>
<td>Bypasses kBouncer heuristics</td>
<td>Any gadgets that are 20 instructions long</td>
<td>mov [reg1], reg2; call reg3; ... jmp rax</td>
<td>×</td>
<td>TM</td>
<td>[15]</td>
</tr>
<tr>
<td>Reflector</td>
<td>Allows to jump to both call-preceded or non-call-preceded gadgets</td>
<td>mov [reg1], reg2; call reg3; ...; jmp rax</td>
<td>mov [rsp], rsi; call rdi; ...; jmp rax</td>
<td>×</td>
<td>RF</td>
<td>[14]</td>
</tr>
<tr>
<td>Call site</td>
<td>This gadget chains the control to go forward when we have the control on the stack and ret</td>
<td>... mov [reg1], reg2; call reg3; ...; ret;</td>
<td>...</td>
<td>×</td>
<td>CS2</td>
<td>[45]</td>
</tr>
<tr>
<td>Entry point</td>
<td>This gadget chains the control to go forward when we have the control of a call instruction</td>
<td>pop rbp; call/jmp reg or call/jmp [reg]</td>
<td>pop rbp; call/jmp reg or call/jmp [reg]</td>
<td>×</td>
<td>EP</td>
<td>[45]</td>
</tr>
<tr>
<td>BRO</td>
<td>Restores all saved registers</td>
<td>pop rdx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;</td>
<td>pop rdx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;</td>
<td>×</td>
<td>BRO</td>
<td>[8]</td>
</tr>
<tr>
<td>Stop</td>
<td>Halts the program execution</td>
<td>Infinite loop</td>
<td>4a833dd4: inc rax 3ddb: jmp 3dd4</td>
<td>×</td>
<td>STOP</td>
<td>[8]</td>
</tr>
</tbody>
</table>
Table 6: Gadgets with gadget types in the priority and MOV TC gadget sets.

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

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.

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

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

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].

<table>
<thead>
<tr>
<th>Program</th>
<th>MV</th>
<th>LR</th>
<th>AM</th>
<th>LM</th>
<th>AM-LD</th>
<th>SM</th>
<th>AM-ST</th>
<th>SP</th>
<th>CALL</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>4</td>
<td>11%</td>
<td>2</td>
<td>0.3%</td>
<td>3</td>
<td>21%</td>
<td>3</td>
<td>44%</td>
<td>3</td>
<td>16%</td>
</tr>
<tr>
<td>Apache</td>
<td>4</td>
<td>16%</td>
<td>2</td>
<td>0.5%</td>
<td>3</td>
<td>37%</td>
<td>2</td>
<td>26%</td>
<td>3</td>
<td>10%</td>
</tr>
<tr>
<td>Average</td>
<td>4</td>
<td>32%</td>
<td>2</td>
<td>0.5%</td>
<td>3</td>
<td>21.7%</td>
<td>2</td>
<td>31.3%</td>
<td>3</td>
<td>12%</td>
</tr>
<tr>
<td>Nginx</td>
<td>3</td>
<td>9%</td>
<td>1</td>
<td>0.1%</td>
<td>2</td>
<td>0.1%</td>
<td>3</td>
<td>15%</td>
<td>2</td>
<td>49%</td>
</tr>
<tr>
<td>Apache</td>
<td>3</td>
<td>27%</td>
<td>1</td>
<td>1%</td>
<td>3</td>
<td>41%</td>
<td>3</td>
<td>27%</td>
<td>2</td>
<td>19%</td>
</tr>
<tr>
<td>Average</td>
<td>3</td>
<td>16.7%</td>
<td>1</td>
<td>0.7%</td>
<td>3</td>
<td>15%</td>
<td>3</td>
<td>20.8%</td>
<td>2</td>
<td>28.7%</td>
</tr>
</tbody>
</table>

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.