Measuring Attack Surface Reduction in the Presence of Code (Re-)Randomization

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

***Abstract*—Just-in-time return-oriented programming (JIT-ROP) technique 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 do starting pointers in JIT-ROP impact gadget availability?*; *how would one compute the re-randomization interval effectively to defeat JIT-ROP attacks? what impact do fine-grained randomization and re-randomization have on the Turing completeness of JIT-ROP payloads?* We conduct a comprehensive measurement study on the effectiveness of fine-grained code randomization and re-randomization, with 5 tools, 13 applications, and 19 dynamic libraries. We provide methodologies to measure JIT-ROP gadget availability, quality, and their Turing completeness, as well as to empirically determine the upper bound of re-randomization intervals in re-randomization schemes. Experiments show that instruction reordering is the only fine-grained single-round randomization approach that thwarts current gadget finding techniques under the JIT-ROP threat model. Our results also show that the locations of leaked pointers used in JIT-ROP attacks have no impacts on gadget availability, suggesting high pointer-based connectivity among code pages.**

I. Introduction

Just-in-time return-oriented programming (JIT-ROP) (e.g., [75]) 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 ROP code reuse attacks [72], as the attacker no longer has direct access to the code pages of the victim program and its libraries. In other words, a leaked pointer only unlocks a small portion of the code region under fine-grained code randomization, seriously limiting the attack's ability to harvest code for ROP gadget purposes.

JIT-ROP attacks have the ability to discover new code pages dynamically [75], 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 [8], Shuffler [86], Remix [20], CodeArmor [18], RuntimeASLR [50], Stabilizer [26], etc.) have the potential to defeat JIT-ROP attacks. 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 and Turing completeness property of gadgets has not been reported. In ROP literature, Turing completeness refers to a set of gadgets that cover the Turing-complete operations including memory, assignment, arithmetic, logic, control flow, function call, and system call [65].

Some (re-)randomization techniques can 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, Turing completeness property, 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 performance or 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 re-randomization interval choices.

We report our experimental findings on code pointer leakage, gadget availability, Turing completeness property, and gadget chain formation, under fine-grained ASLR and re-randomization tools including Zipr [38] and Shuffler [86]. Our evaluation involves up to 13 applications and 19 dynamic libraries. We aim to experimentally answer the following research questions (RQs).

RQ #1: How does fine-grained code (re-)randomization quantitatively impact on the availability of various kinds of gadgets that are essential for the completeness of JIT-ROP payloads?

RQ #2: How does the location of a code pointer leak impact on the availability of gadgets under fine-grained code randomization?

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

RQ #3: How does fine-grained code randomization impact the quality of a gadget chain (i.e., payload)?

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 for re-randomization intervals. We compute the upper bound $\mathcal{T}$ by measuring the minimum time for an attacker to find a Turing-complete set of JIT-ROP gadgets. In other words, if the re-randomization interval is less than $\mathcal{T}$, then a JIT-ROP attacker is unable to obtain a Turing complete gadget set. Our experiments show that this upper bound ranges from 4 seconds to 17 seconds for various applications (with dynamic libraries).
  In addition, we present a general methodology for quantifying the number of JIT-ROP gadgets. For fine-grained single-round randomization, our results show that an instruction-level solution (namely Zipr [38]) limits the availability of gadgets up to 90% and successfully breaks the Turing completeness of JIT-ROP payloads. We also observe that fine-grained randomization slightly degrades the gadget quality, in terms of register-level corruption.
- Our experiments show that locations 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 address space layout. A pointer leakage in any location allows attackers to obtain a basic set of gadgets.
  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. This finding indicates the necessity of randomizing stack over heap or global variables.
- From the perspective of defense, our findings suggest that code isolation and reducing semantic code connectivity are important for defending against JIT-ROP attacks. As instruction-level reordering breaks up gadgets, current gadget finding and chaining techniques are no longer effective. Thus, from the perspective of understanding attack capabilities, redefining traditional ROP gadgets into smaller (i.e., one line) building blocks and demonstrating new gadget chain compilers would be interesting. Our results suggest that gadgets quantity and quality vary with runs. Thus, how to make attacks more reliable and robust in practice is also an interesting direction.

Besides the comprehensive measurement work, we distill common attack operations in existing ASLR-bypassing ROP attacks (e.g., [10], [15], [28], [75]) and present a generalized attack workflow that captures the tasks and goals. This workflow is useful beyond the specific measurement study.

II. Threat Model and Definitions

Coarse-grained ASLR (or traditionally known as only ASLR [81]) 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 **P**osition **I**ndependent **E**xecutable (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 [22], [35], [45], basic blocks [20], [46], [84], instructions [40], or machine registers [41]. We evaluated Zipr[1] [38], Selfrando[2] (SR) [22], Compiler-assisted Code Randomization[3] (CCR) [46], and Multicompiler[4] (MCR) [41]. We also evaluated Shuffler [86], 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$\oplus$X and RELRO are enabled. W$\oplus$X specifies that no address in a process' address space 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 a compiler flag and full RELRO is now by default deployed on mainstream Linux distributions.

In addition, our experimental evaluation is conducted under the following assumptions. Attackers do not have any prior knowledge of the target application's memory layout, i.e., attackers have to derandomize the layout through an attack. Fine-grained code randomization is applied in every executable and associated library in a target system (unless specified otherwise). Similar to JIT-ROP [75], we assume that no code pointer protection [7], [23], [31], [48], [49], [51], [53] exists in the target application. We discuss the need for measuring code pointer protection solutions under the JIT-ROP model in Section VI. We assume that memory permission-related protections such as XnR [6], NEAR [85], Readactor [24] and destructive read-related protections such as Heisenbyte [80], etc. are not present in the victim machine[5]. We assume attackers have already obtained a leaked code pointer (e.g., a function pointer or a virtual table pointer) through remote exploitation of an application/library vulnerability. Such an assumption is standard in existing attack demonstrations.

Next, we define the terms of Turing completeness, upper bound of re-randomization intervals, minimum footprint gadgets, and extended footprint gadgets.

[1] https://git.zephyr-software.com/opensrc/irdb-cookbook-examples
[2] https://github.com/immunant/selfrando
[3] https://github.com/kevinkoo001/CCR
[4] https://github.com/securesystemslab/multicompiler
[5] Attacks (e.g, AOCR [66] and code inference [76]) are still possible with those defenses.

Definition 1: *Turing completeness refers to the availability of 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) [65].*

Definition 2: *The upper bound $\mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$ of a re-randomization scheme $\mathcal{P}$ under a JIT-ROP attacker $\mathcal{A}$ is the maximum amount of time between two consecutive randomization rounds that prevent $\mathcal{A}$ from obtaining a set of Turing-complete gadgets, i.e., for any interval $\mathcal{T}'^{\mathcal{A}}_{\mathcal{P}} < \mathcal{T}_{\mathcal{P}}^{\mathcal{A}}$, the set of gadgets obtained under $\mathcal{T}'^{\mathcal{A}}_{\mathcal{P}}$ cannot cover all the Turing-complete gadgets.*

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 III shows, without gadgets that information may not be sufficient for launching exploits.

Extended footprint (EX-FP) gadgets: Turing-complete gadgets (i.e., load, store, assignment, etc.) and attack-specific gadgets (e.g., reflector gadget, call site gadget, etc.) are useful for arbitrary computation and building an attack payload. A gadget is an extended footprint gadget if it is an instance of the set of Turing-complete or attack-specific gadgets. An EX-FP gadget may contain additional instructions that may cause side effects in an attack payload.

Minimum footprint (MIN-FP) gadgets: A minimum footprint gadget is also an instance of the set of Turing-complete or attack-specific gadget, but it does not cause any side effect in an attack payload.

III. Comparison of JIT-ROP and 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 IV and findings in Section V. We analyze various attack demonstrations with a focus on attacks (e.g., [10], [15], [28], [75]) in our threat model.

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***. We describe each component in Sections III-A, III-B, and III-C, respectively.

- *Memory layout derandomization*: Due to the W⊕X defense, attackers must reuse code (via gadgets) in their attacks. Attackers need to derandomize the memory layout to discover gadgets (steps ②-④ for JIT-ROP and steps ②′ and ④′ for basic ROP in Figure 1). Usually, attackers leverage memory corruption vulnerabilities to leak memory content [78] and start the derandomization process utilizing the leaked memory.
- *System access*: Attackers access privileged system operations by issuing system APIs or gadgets. However, useful attacker actions may also include reading user data (such as email) or stealing authentication tokens (like cookies). These actions do not involve system calls. In this paper, we only consider those attacks that utilize system calls. Attackers invoke system calls through `syscall` gadgets. One can find `syscall` gadgets through step ④ in JIT-ROP along with other gadgets. In basic ROP, one must find the `syscall` gadgets through pointer leakage in system libraries (like `libc`) or application binaries (step ⑨).
- *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. Steps ⑤ and ⑤′ in Figure 1 correspond to the payload generation operation for JIT-ROP and basic-ROP, respectively. The target of a payload is to achieve an attack goal, e.g., memory leak, launching a malicious application, or launching a root shell.

A. Memory Layout Derandomization

Derandomizing the fine-grained address space layout is the key for mounting code-reuse attacks with gadgets. This step requires overcoming several obstacles.

Memory disclosure. The most common way of derandomizing memory layout is through a memory disclosure vulnerability in an application. 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 [5], [39], Heap Feng Shui [77], Flip Feng Shui [64]) to leak memory contents (Steps ② and ②′). Details on memory corruption model can be found in [34], [79] and an example in [78].

Code reuse. Due to W⊕X defense, adversaries cannot inject code in their payload. Return-oriented programming (ROP) [72] and its variants jump-oriented programming (JOP) [12] and call-oriented programming (COP) [36] 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 [27], [75]. The difference between basic ROP [72] and JIT-ROP [75] 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 [2] 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 adversary needs to find gadgets dynamically at the time of an exploit. She may attempt to scan the process address space

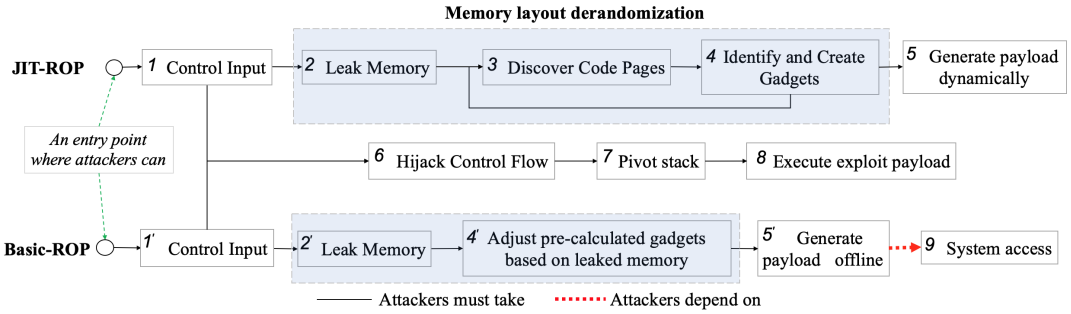

Fig. 1: An illustration of the commonalities and differences between a conventional (or basic) ROP attack (bottom) and a JIT-ROP attack (top). The top light-gray-box highlights the key steps in JIT-ROP to overcome fine-grained ASLR.

in order to search for gadgets. However, this linear scanning may lead to a segmentation fault and crash the process. A powerful technique introduced by JIT-ROP [75] is code page harvest, which is explained next.

JIT-ROP attacks exploit the connectivity of code in memory to derandomize and locate instructions. The **code harvest process** in JIT-ROP identifies gadgets at runtime by reading and disassembling the text segment of a process. This process starts with computing the page number of a disclosed code pointer at runtime. A 64-bit system uses the first 52 bits for page numbers if the page size is 4K. Once the page number is computed, the process reads the entire 4K data of that page. A light-weight disassembler converts the page data into instructions. The code harvest process 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 (0x11F9000-0x11F9FFF), disassembles the data, searches for `call` and `jmp` instructions to find other pointers (0x11FB410 and 0x11FCFF4) to jump to those code pages. This process is recursive and stops when all the reachable code pages are discovered. We implemented this code harvesting method for our evaluation. It is important to mention that indirect calls to library functions can also be resolved to jump to the library code pages.

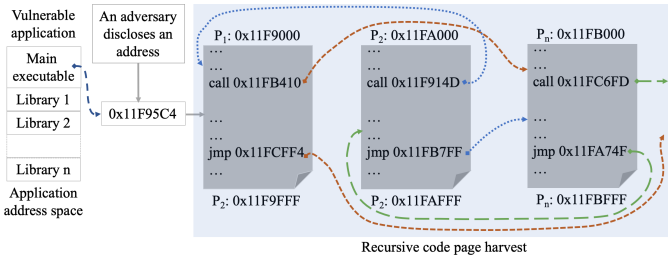

Fig. 2: An illustration of the recursive code harvest process of JIT-ROP [75]. An adversary discloses an address from the main executable or libraries (in this case from the main executable) of an application through a vulnerability.

Code page harvest and gadget identification. In step ③, an adversary utilizes the leaked memory address to read code pages of the process' code segments. In step ④, she can identify and create gadgets by disassembling the read code pages. She can scan for byte values corresponding to `ret` opcodes (e.g., 0xC2, 0xC3) and perform a narrow-scoped backward disassembly from there. She also can use an improved disassembler (e.g., Capstone [13]) to perform gadget identification. The adversary performs step ③ and ④ repeatedly until she finds enough gadgets for her exploit.

B. System Access

If the execute-only defenses (e.g., XnR [6], NEAR [85], Readactor [24] and Heisenbyte [80]) and CFI[6] (e.g., CCFIR [88] and bin-CFI [89]) 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 [75], the approach is slightly different, as adversaries commonly invoke a system API instead of invoking a system call directly. `Syscall` gadgets can be found in an application's code or dynamic library. For basic ROP attacks, attackers can adjust pre-computed system gadgets from dynamic libraries, given that she manages to obtain a code pointer from a dynamic library (e.g., libc). Step ⑨ in Figure 1 is for this task. This task is performed manually and offline. The attacker may obtain the library code pointer from an application's stack or heap or data segment.

C. Payload Generation

Once an adversary derandomizes the memory layout of a process and gets access to enough gadgets, she glues different parts (e.g., gadgets, functions, strings, constants, etc.) together to build a payload or attack chain. The adversary may generate the payload dynamically at step ⑤ 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. Because a payload is primarily a set of addresses that point to some existing code in an application's address space, attacks do not execute anything stored in a stack/heap, which is protected by W⊕X. The adversary may utilize the same vulnerability as in step ② or a different vulnerability to hijack a program's control flow at step ⑥ to redirect the flow to the stored payload.

[6]which restricts "gadgets" to only legitimately called functions

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 II.

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 VII in the Appendix were designed by Carlini and Wagner [15] to bypass specific CFI implementations (namely, kBouncer [60] and ROPecker [21]).

IV. Experimental Design

We describe our measurement methodologies for evaluating fine-grained ASLR's impact on the memory layout derandomization, system API 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 ROP gadgets that are available to attackers under the JIT-ROP code harvesting mechanism. Another challenge is how to quantify *i)* the difficulty of accessing internal system functions and *ii)* the quality of gadget chains. For the former, our approach is to compute the number of system gadgets and 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.

A. Measurement Methodologies

Measurement Methodology for Memory Layout Derandomization. This methodology is for evaluating RQs #1, #2, and #3. We manually extract 19 types of gadgets from various attacks [10], [14], [15], [36], [75]. 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 VII in the Appendix shows those gadget types in more details.

These 19 types of gadgets include the Turing-complete set of gadgets (see Definition 1). The Turing-complete gadgets and 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` and `FS` are very application-specific. They do not have any concrete forms or concrete attack goals. These gadgets are used to trick defense mechanisms. We also attempt to evaluate the block-oriented gadgets used for Block-Oriented Programming (BOP) [43].

In our experiments, we measure the occurrences of these gadgets under fine-grained code randomization and re-randomization. For code re-randomization, we attempted to use six re-randomization tools. However, some of the tools are unavailable and some have runtime and compile-time issues;[7] in the end, we were able to obtain only Shuffler [86]. For code randomization, we used four relatively new code randomization tools: Zipr [38], SR [22], CCR [46], and MCR [41], because of their reliability. Table I shows the key differences between these randomization schemes.

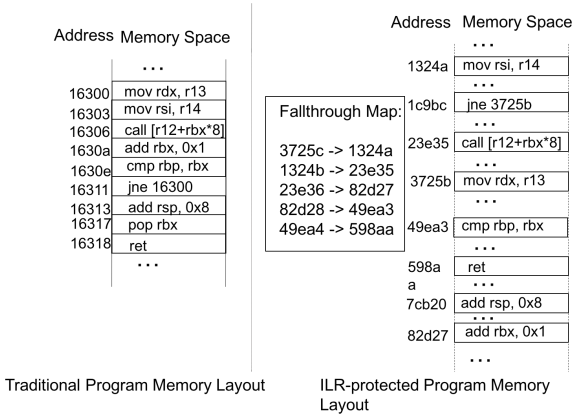

Fig. 3: Instruction location randomization. This figure is adopted from ILR [40].

We compile and build a coarse-grained 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. For Shuffler [86], we do not need to prepare special binaries. We use ropper [67], an offline gadget finder tool, under coarse-grained ASLR. Under fine-grained ASLR, we recreate the JIT-ROP [75] exploitation process, including code page discovery and gadget mining. We use the capstone [13] disassembler for disassembling a code page.

We write a tool to search for gadgets of a specific type. We scan the opcodes of `ret` (0xC3) and `ret xxx` (0xC2) and perform a narrow-scoped backward disassembly from those locations to collect ROP gadgets. Similarly, we scan the opcodes of `int 0x80` (0xCD 0x80), `syscall` (0x0F 0x05), `sysenter` (0x0F 0x34) and `call gs:[10]` (0x65 xFF 0x15 0x10 0x00 0x00 0x00) for system gadgets.

Measurement 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 call gadgets under the coarse- and fine-grained code 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 try to exploit vulnerabilities to leak libc pointers from the stack/heap/data-segment, rather we assume that we know the address mapping of the libc library 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 V-D.

[7] Remix [20] and CodeArmor [18] are not available. TASR [8] is not accessible due to policy issues. Runtime ASLR [50] and Stabilizer [26] have runtime and compile time issues, respectively.

TABLE I: The key differences in the various randomization and re-randomization schemes evaluated.

Tools	Randomization Scheme(s)	Randomization Time	Compiler Assistance Required?	Techniques	Performance Overhead
Shuffler [86]	Function-level re-randomization	Runtime	No	- Loads itself as a userspace program - Contains a separate thread for shuffling the functions continuously - Represents code pointers as indices for flexibility	14.9% [86]
Zipr [38]	Instruction-level randomization	Static rewriting	No	- Reorders all instructions and generates ILR static rewrite rules - Executes randomly scatter instructions using a process-level virtual machine (PVM) utilizing static rewrite rules or a fallthrough map - Keeps the same layout unless rewrite again	<5% [38]
SR [22]	Function-level randomization	Load time reorder	No	- Adds a linker wrapper that intercepts calls to the linker and asks the selfrando library to extract the necessary information to reorder functions - Reorders functions once a binary loaded into memory - Reorders on each load	<1% [22]
MCR [41]	Function- and register-level randomization	Link time reorder	Yes	- Reorders functions and machine registers during link time optimization - Keeps the same layout unless compiled and built again	1% [41]
CCR [46]	Function and block-level randomization	Installation time	Yes	- Extracts metadata during compilation - Reorders functions and basic-block based on the metadata - Keeps the same layout unless rerandomized again	0.28% [46]

Measurement Methodology for Payload Generation. We evaluate our RQ #3 (in Section V-C) using this methodology. 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.

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. 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 [86] runs itself alongside the userspace 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 [38] reorders the location of each instruction in an executable or library (Figure 3). Zipr works directly on binaries or libraries with no compiler supports. Zipr [38] 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) [22] applies code diversification at the load time. This tool collects *Translation and Protection (TRaP)* information, a minimal set of metadata for function boundaries during the linking phase. This tool also inserts a dynamic library called *libselfrando*. At the load time, this library 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. SR can use either GCC or Clang as its compilation engine.

Multicompiler (MCR) [41] 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 padding such as NOP insertion, global padding, and insertion of 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) [46] 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 boundaries, functional block boundaries, and object block boundaries), fixup, and jump table of an executable during compilation and linking phases. The tool embeds this metadata into the executable by adding a new section called `.rand`. A Python script then rewrites the executable binary by reordering the positions of basic blocks and functional blocks. 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 [35], ASLP [45], Remix [20], and STIR [84], are not publicly available. Some available tools (e.g., MCR [41], CCR [46] and SR [22]) operate on the source code level. They require the recompilation of source code including dynamic libraries. We experienced multiple linking issues while using CCR and SR to compile Glibc code. The tool authors confirmed the limitations (discussed in Section VI). ORP [61] was the randomization tool used in Snow *et al.*'s JIT-ROP demonstration [75]. It operates on Windows binaries, incompatible with our setup.

V. Evaluation Results and Insights

Experimental setup. All experiments are performed on a Linux machine with Ubuntu 16.04 LTS 64-bit operating system.

TABLE II: Numbers of the experimental applications and dynamic libraries for each tool.

Tool	Applications (13 Total)	Libraries (19 Total)
Shuffler [86]	12	15
Zipr [38]	9	9
SR [22]	11	15
MCR [41]	8	8
CCR [46]	9	9

We write several Python and bash scripts for automating our analysis and measurement process. The scripts have around 3,500 lines of code including around 2,000 lines of Python code on top of GDB-Python-Utils [4]. We use the Python regular expression library *re* for finding semantically different gadgets, e.g., the Turing-complete gadgets and attack-specific gadgets. To overcome the issues involved with JIT-ROP [75] for searching gadgets on the fly at runtime, we run/load each application/library and attach the application/library to GDB. The Python scripts are also loadable in GDB. This setup allows us to avoid a process' memory mapping related complexities. We also plan to make our analysis tool and data available to the public.

We perform our experiments on the latest and stable versions of *bzip2*, *cherokee*, *hiawatha*, *httpd*, *lighttpd*, *mupdf*, *nginx*, *openssl*, *proftpd*, *sqlite*, *openssh*, *thttpd*, *tor*, and *xpdf*. We also perform our experiments on dynamic libraries. Dynamic libraries include *libcrypto*, *libgmp*, *libhogweed*, *libxcb*, *libpcre*, *libgcrypt*, *libgnutls*, *libgpg-error*, *libtasn1*, *libz*, *libnettle*, *libopenjp2*, *libopenlibm*, *libpng16*, *libtomcrypt*, *libunistring*, and *libxml2*. 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 server area, PDF reader, cryptography, networking utility, database, browser, math library, image library, and system library. Table II shows the numbers of applications/libraries used for evaluating Shuffler [86], Zipr [38], SR [22], CCR [46], and MCR [41]. Each tool evaluates a different set of applications and libraries because no tool is capable of (re-)randomizing all of our selected applications (13 in total) and libraries (19 in total). However, we also evaluate these tools using the common set of applications and libraries that these tools can randomize.

To evaluate Shuffler [86], we take 100 consecutive address space snapshots from an application/library re-randomized by Shuffler [86]. Then, we manually analyze the address space snapshots. We manually compile (or rewrite the executables of) these programs to enforce fine-grained code randomization up to function level using SR [22], basic block level using CCR [46], both functional and register levels using MCR [41], and instruction level using Zipr [38]. 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 (as required). We run, load or rewrite each application or dynamic library 100 times to reduce the impact of variability on the number of gadgets in each run or load.

A. Impact on the Availability of Gadgets

RQ #1: How does fine-grained code (re-)randomization quantitatively impact on the availability of gadgets? Does it break the Turing completeness?

We measure a total of 13 types of gadgets, including 11 gadgets for Turing-complete operations and 2 attack specific gadgets from various attack demonstrations [10], [15], [33], [36], [65], [75]. We measure the set of gadgets that are required for Turing-complete operations defined in Definition 1. For partial Turing-complete gadgets, we measure the percentage of Turing-complete gadgets. Partial Turing-complete gadgets have gadgets to perform some Turing-complete operations, but not all. For example, 50% of Turing-complete gadgets means that the gadgets can perform 50% of the Turing-complete operations. We identify 11 types of gadgets necessary for seven Turing-complete operations. Any type of gadgets missing from the 11 types of gadgets makes the Turing-complete gadgets partial. We check the existence of these 11 types of gadgets and calculate the percentage of Turing-complete gadgets.

1) Impact of Re-randomization: We assess the impact of different re-randomization intervals on the availability of Turing-complete gadgets. We run the Nginx server and re-randomize the Nginx's address space with a re-randomization interval of 30 seconds. We take 100 snapshots of Nginx's address space for 100 consecutive re-randomizations. We identify the number of minimum and extended footprint gadgets from each snapshot. We calculate the average number of minimum and extended footprint gadgets for each snapshot.

To measure the upper bound for re-randomization intervals and assess the impact of different re-randomization intervals on the availability of Turing-complete gadgets, we run dynamic code harvesting process for 12 applications including 15 libraries such as *libc*, *libcrypt*, *libpcre*, *libz*, *libcrypto*, *libgnutls*, etc. For each application, we record the time at the start of the code page harvesting process, after each code page harvest, and the completion of the code page harvesting process. We also measure the number of Turing-complete gadget types that the code harvesting process covered so far while recording the time. Figure 4 shows the percentages of discovered Turing-complete gadgets for different re-randomization intervals.

Figure 4 shows that as the re-randomization interval increases, the percentages of the Turing-complete gadgets also increase. Longer intervals help attackers reveal more Turing-complete gadgets than shorter intervals. However, longer intervals may not immediately help attackers reveal more Turing-complete gadgets. For example, we notice some flat lines of several seconds for some applications such as mupdf ($\sim$4s), hiawatha ($\sim$4s), proftpd ($\sim$7s), thttpd ($\sim$9s), and cherokee ($\sim$11s). The flat lines indicate that the percentage of Turing-complete gadgets does not increase with the increase of the intervals within the duration of those flat lines. Most flat lines also indicate that one or two missing gadget types prevent the set from reaching Turing completeness. That is, one or two types of gadgets are very scarce. The most scarce gadgets are Load-Memory (LR) and Arithmetic-Load (AM-LD). For example, the gadget discovery process searches for the Load-Memory (LR) gadgets for around 7 seconds for proftpd. The fundamental reason for the scarcity is that some applications (including libraries) do not have pure register-based memory access. The memory accesses are made by adding some offsets to the registers.

Clearly, the value of the upper bound for the re-

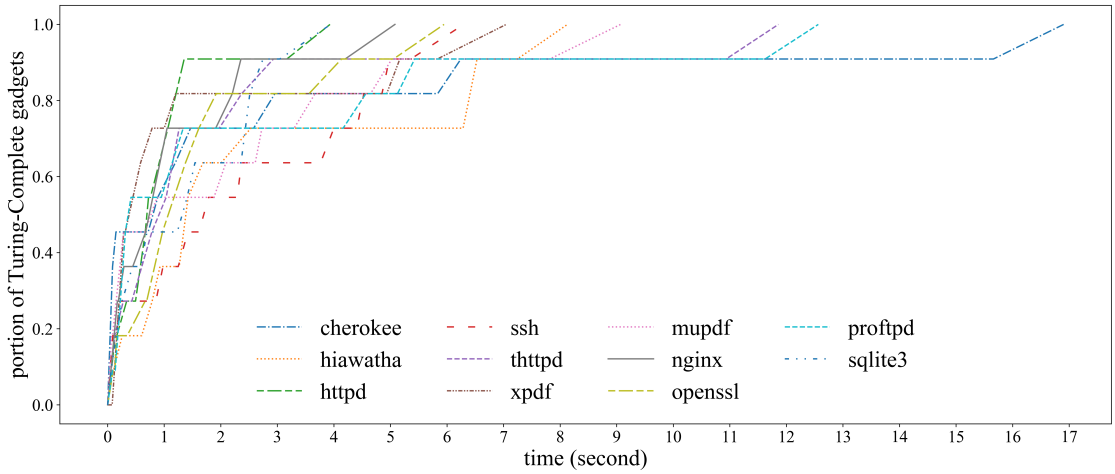

Fig. 4: Changes of the percentage of Turing-complete gadgets with re-randomization intervals. The percentage of Turing-complete gadgets (Y-axis) of the an application is computed based on the number of distinct Turing-complete gadgets across the 11 types of Turing-complete gadgets described in Definition 1. Turing-complete gadgets include the gadgets from the associated dynamic libraries as well as the application's main executable.

randomization interval depends on the machine (e.g., CPUs, cache size, memory, etc.) where the measurement is conducted. Figure 4 shows that the lowest re-randomization interval to achieve 100% of Turing-complete gadgets is around 4 seconds in our experiments. The number is computed by taking the lowest value that is observed across the 11 applications (with the associated dynamic libraries) in Figure 4 to achieve the Turing-complete gadget set. We choose the lowest value to ensure security guarantees. Using our methodology, defenders can perform the measurement on their machines to help determine what interval is appropriate, while satisfying overhead constraints.

We also notice that 8 applications achieve ∼60% of Turing-complete gadgets within 1.0∼1.5 seconds. This observation indicates the necessity of the relevance analysis of partial Turing-complete gadgets in the context of attack goals. For example, partial Turing-complete gadget set could be useful for some attack scenarios where attackers do not need the system call gadgets. Thus, defenders need to determine to what extent partial gadgets are useful and set the re-randomization interval accordingly. In our experiments, we conservatively consider only the 100% of Turing-complete gadgets to compute the upper bound of re-randomization intervals. We leave the relevance analysis of partial-gadgets in the context of attack goals as a future research direction. In Section VI, we discuss the implications of these results in real-world operations, e.g., how to choose re-randomization intervals for performance- and security-critical applications.

2) Impact of Single-round Randomization: Table III summarizes the impact of fine-grained code randomization 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 above) for each application and library before and after randomizing with the four randomization tools (i.e., Zipr [38], SR [22], MCR [41], and CCR [46]). Each application or library is run/loaded in memory for 100 times after randomizing 100 times when necessary[8]. 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 tool. Table III shows the overall gadget reductions in application and library categories for Zipr [38], SR [22], MCR [41], and CCR [46].

On average, the amount of gadgets is reduced (by 27%∼39% for minimum footprint and 34%∼38% for extended footprint gadgets) when applications are randomized using SR, CCR, and MCR. For dynamic libraries, the reductions range from around 6% to 24% for minimum footprint gadgets and around 33% to 38% for extended footprint gadgets. However, Zipr [38] reduces the overall gadget amount significantly by around 80%∼90% for both minimum and extended footprint gadgets. Table III also shows the reduction of gadgets in seven Turing-complete (TC) categories and indicates whether the Turing completeness 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 category. Since the number of applications or libraries is different for each randomization tool in Table III, we also use the common set of applications and libraries to validate the results shown in Table III. Figure 8 in the Appendix shows the validation results. According to the results, all the tools exhibit similar reduction while we evaluate them using the common set of applications and libraries.

The Turing completeness is preserved in the randomized versions of applications or libraries when randomized by SR, MCR, and CCR. The Turing completeness is also preserved for both minimum and extended footprint gadgets. However, Zipr destroys the Turing completeness property for minimum footprint gadgets. Turing completeness is broken when there is no gadget in one of the Turing-complete categories. For example, in Table III, the reduction of minimum footprint

[8]Compiling once and running 100 times is enough for SR. 100 times diversification, 100 times compilation, and 100 times rewriting are required for CCR, MCR, and Zipr, respectively.

TABLE III: Impact of fine-grained single-round randomization on the availability of gadgets in various applications and dynamic libraries. The data in each row of SR [22] is generated by averaging the data of 11 applications and 15 dynamic libraries. The data in each row of Zipr [38] and CCR [46] is generated by averaging 9 applications and 9 libraries for each tool. 8 applications and 8 dynamic libraries are used for MCR [41]. The data of each application or library is the average result of 100 runs/loads/rewrites. The standard deviations vary between 0.3∼3.4 for minimum footprint and 5.04∼22.85 for extended footprint gadgets. ⇓ indicates reduction.

				Reduction (%) of Turing-complete (TC) gadgets in 7 TC categories (MIN-FP | EX-FP)							
Tools	Granularity	⇓ (%) MIN-FP	⇓ (%) EX-FP	Memory	Assignment	Arithmetic	Logical	Control Flow	Function Call	System Call	TC Preserved?
Applications											
Zipr	Inst.	**80.91**	**88.45**	**100** | 93.5	61.9 | 91.5	**100** | 86.3	57.5 | 82.1	66.0 | 88.7	73.1 | 92.5	83.33 | 0	✗*
SR	FB	40.28	36.53	3.6 | 21.0	10.8 | 42.9	14.7 | 9.8	35.5 | 36.2	23.4 | 29.3	25.0 | 48.4	0 | 0	✓
MCR	FB & Reg.	37.19	34.81	-16.7 | 25.6	-4.4 | 23.0	22.0 | 38.8	2.4 | 28.8	40.5 | 59.2	14.0 | 63.7	80.0 | 0	✓
CCR	BB	27.02	38.77	2.3 | 31.7	4.4 | 41.2	12.2 | 24.4	4.8 | 26.4	56.0 | 71.2	30.9 | 61.4	0.0 | 0	✓
Libraries											
Zipr	Inst.	**91.63**	**85.42**	**94.4** | 91.4	67.2 | 89.1	**96.8** | 88.1	83.5 | 89.0	65.4 | 89.1	62.5 | 86.7	66.67 | 0	✗*
SR	FB	23.54	37.91	23.5 | 29.3	19.2 | 40.4	31.5 | 43.1	48.9 | 43.2	47.7 | 56.1	36.6 | 39.9	22.91 | 0	✓
MCR	FB & Reg.	6.34	37.77	24.1 | 37.5	30.2 | 39.6	56.3 | 55.9	45.7 | 45.4	37.0 | 54.1	43.4 | 42.3	66.67 | 0	✓
CCR	BB	10.89	33.66	9.7 | 26.5	11.1 | 46.4	22.6 | 35.9	21.9 | 39.8	25.9 | 45.6	23.2 | 44.6	50.0 | 0	✓

* For Zipr, TC is not preserved for minimum footprint gadgets, but TC is preserved for extended footprint gadgets.

gadgets in memory and arithmetic categories is 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 94.4% and 96.8%, respectively. These two reductions are not 100% because some libraries contain a few gadgets (a total of 4 for memory and 3 for arithmetic). When the numbers of gadgets are averaged over the number of libraries, the average value becomes close to zero.

Most of the applications and libraries do not contain any `syscall` gadgets (as expected) because applications and libraries usually make syscalls through `libc`. However, applications or libraries may have occasional use of the `syscall` function. For example, the `log_tid()` function is the only one of `httpd` that invokes `syscall` function. Similarly, other applications or libraries occasionally invoke the `syscall` function. This is why the number of `syscall` gadgets is low in most cases. Since SR is only able to randomize a light-weight version of libc (`musl`), we see slightly high values for system gadgets in Table III for SR.

We also assess the availability of Turing-complete gadgets under a *single* randomization pass of Shuffler [86]. On average, we observe a 24% reduction in minimum footprint gadgets and 3% reduction in extended footprint gadgets compared to the non-randomized version of Nginx. Reduction means that the re-randomization technique prevents the current gadget finding tools from obtaining gadgets from an application's address space. The low reductions are expected, as Shuffler's security relies on the capability of continuously shuffling code locations, not a single randomization pass.

3) Reasons Behind Gadget Changes: Ideally, function or basic block randomization should not destroy gadgets because the gadget elements within a basic or function block are relocated with the relocation of the basic or function block. That means we should not observe any gadget reduction for SR and CCR. However, we see reasonable (up to 40%) gadget reduction for SR and CCR. Primarily, this reduction occurs due to the dynamic code page harvesting process. Code page coverage of dynamic code harvesting process depends on how well a program or library is connected by function calls. Less connectivity of a program or library results in less code page coverage, which in turns results in less number of gadgets than the offline versions.

Some randomization tools (e.g., SR [22]) utilize some compiler optimization techniques (e.g., using `leave` instruction before `ret` instruction). A leave instruction before a `ret` instruction breaks a gadget (details on how leave impacts on a gadget chain in Appendix C). Through manual code inspection, we find that a substantial number of `ret` instructions are preceded by `leave` instructions in the SR's randomized code.

Future directions. We conservatively define the upper bound of a re-randomization scheme by the time required for an attacker to achieve 100% of the Turing-complete gadget set. However, in reality, partial Turing-complete gadget set could be sufficient for an attacker in some scenarios. Thus, the relevance analysis of partial Turing-complete gadget set could be an interesting research direction.

Our findings also show that instruction-level reordering destroys almost all (∼90%) gadgets. Current gadget finding and chaining techniques no longer work when instruction reordering is enabled. Thus, redefining traditional ROP gadgets into smaller (e.g., one line) building blocks and demonstrating new gadget chain compilers (e.g., two-level construction) are interesting new attack directions. In addition, understanding the capabilities of attackers who possess a partial set of Turing-complete gadgets would be useful.

B. Impact of the Location of Pointer Leakage

RQ #2: How does the location of a code pointer leak impact the availability of gadgets in the presence of fine-grained code randomization? We measure the impact of pointer locations

on JIT-ROP attack capabilities, by comparing the number of gadgets harvested under different *starting* pointer locations. We aim to find out whether or not the number of gadgets depends on the location of a pointer leakage when a fine-grained diversification is applied. We collect the total numbers of minimum and extended footprint gadgets by leaking a random code pointer from **each** code page of hiawatha, httpd, lighttpd, nginx, proftpd, and thttpd.

Table IV shows the number of leak code pointers or addresses and the numbers of minimum and extended footprint gadgets that can be harvested by starting from the leaked pointers. We restrict the code harvesting process to harvest gadgets only within the text segment of an application to find how well the code of a program is connected. 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 Zipr; 222 minimum and 5277 extended footprint gadgets when randomized by SR; 111 minimum and 1731 extended footprint gadgets when randomized by MCR; and 204 minimum and 4822 extended footprint gadgets when randomized by CCR. These findings indicate that an application's code segment is very well-connected, making JIT-ROP attacks easier.

The numbers of leaked addresses in Table IV are different for different backends. 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 for different tools.

For Zipr, we cannot pick a random code pointer from the application's code segment, because Zipr makes the code segment sparse by transforming the segment. The sparse code segment contains many bad values. Thus, for Zipr, we randomly leak the addresses of 10 functions from each application and use those addresses as the starting points.

Future directions. Our findings imply that **any** valid code pointer leak from an application's code segment is equally viable. Regardless of the randomization, a pointer leakage in any location allows attackers to access a set of minimum and extended footprint gadgets. These observations suggest that disrupting the connectivity of code segment would be an effective defense strategy. Although this kind of disruption solutions (e.g., Oxymoron [7]) exist, they increase the runtime overhead and cannot protect from the variants of JIT-ROP (e.g., Isomeron [28]). Thus, a randomization time solution that disrupts the connectivity of code while keeping the execution order intact would be an interesting research direction.

C. Impact on the Quality of a Gadget Chain

RQ #3 How does fine-grained code randomization impact the quality of a gadget chain (i.e., payload)? 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 IV-A.

We measure the register corruption rate for `MV`, `LR`, `AM`, `LM`, `AM-LD`, `SM`, `AM-ST`, `SP`, and `CALL` gadgets. Some gadgets (e.g., `CP`, `RF`, `EP`, etc.) described in Table VII (in Appendix) are special purpose gadgets that are used to trick defense mechanisms, such as CFI [3], kBouncer [60], and ropecker [21]. Thus, we omit these gadgets from the quality analysis.

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

We present the detailed results in Appendix (Table VIII). Table VIII also reports the average number of unique registers used in each gadget. This number reflects how many registers (ranging from 1 to 4) are involved in a gadget on average.

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

Future directions. Most randomization solutions reorder functions, basic-blocks, and instructions. MCR [41] goes one step deeper, reorders machine registers, and replaces `mov reg1, reg2` instructions with equivalent `lea` instruction. Besides these, designing randomization solutions that increase the register corruption rate in gadgets would be interesting as high register corruption rate would make attacks unreliable.

D. Availability of Libc Pointers

This experiment measures the risks associated with a heap, stack or data segment of an application for revealing a library

TABLE IV: The impact of the location of a pointer leak. The same application has different numbers of leaked addresses for different tools because each tool uses different backends (i.e., compilers). Different backends produce different sized executable of the same program. Size of an executable is proportional to the number of code pages.

Program	Zipr [38]			SR [22]			MCR [41]			CCR [46]		
	# of leaked addresses	# of MIN-FP	# of EX-FP	# of leaked addresses	# of MIN-FP	# of EX-FP	# of leaked addresses	# of MIN-FP	# of EX-FP	# of leaked addresses	# of MIN-FP	# of EX-FP
hiawatha	10	9	223	42	41	1259	47	44	1042	39	31	793
httpd	10	16	634	91	141	4453	MCR produces linking error for httpd			86	176	4764
lighttpd	10	8	235	53	103	2512	68	118	2544	45	74	1783
nginx	10	26	788	121	222	5277	49	111	1731	114	204	4822
proftpd	10	17	523	187	96	7395	131	115	4466	131	125	3986
thttpd	10	8	172	17	22	583	16	31	535	15	24	428

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

	Ptrs in stack that point to libc code		Ptrs in heap that point to libc code		Ptrs in data segment that point to libc code	
Program name	unique ptrs	# of occurrences	unique ptrs	# of occurrences	unique ptrs	# of occurrences
hiawatha	10	14	0	0	1	1
httpd	23	37	1	1	1	1
lighttpd	6	6	0	0	0	0
mupdf	19	40	2	4	0	0
nginx	10	12	0	0	0	0
openssl	19	41	2	4	0	0
proftpd	23	36	1	1	0	0
sqlite3	19	41	2	4	0	0
ssh	19	26	0	0	**13**	**13**
thttpd	17	20	2	2	0	0
tor	18	66	1	15	1	1
Average	17	31	1	3	1	1

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 `proftpd` is when `proftpd` 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 refer a library pointer (e.g., libc pointer) 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).

Table V shows the number of unique libc pointers and the number of times those pointers appear 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 Table V, 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. Three (i.e., Zipr, SR, and CCR) out of the four randomization solutions are designed to randomize only code so that information leakage becomes useless. Only MCR protects stack by inserting padding and randomizing stack frames. These protections are important as they stop memory leakage or harden memory leakage processes in the first place. Our results show that stack is associated with high risk than other segments. Thus, designing randomization solutions that stop leaking address from memory, specially from stacks, would be an interesting research direction.

E. Impact of the Compiler Optimizations

To assess the impact of code transformations and optimizations, we attempt to discover what kind of code optimizations and transformations can impact the availability of various types of gadgets. To do this, we prepare normal and randomized binaries with different optimization levels (-O0, -O1, -O2, -O3, -Ofast, -Os). We select three server applications (nginx, apache, proftpd), one secure networking utility application (openssh), and one lightweight database (sqlite3) to assess the impact of code optimizations.

The interesting finding is that the unoptimized code seems to be more secure than optimized code. Figure 5 shows the number of Turing-complete gadgets in different optimization levels. On average, the optimization levels greater than or equal -O1 have 72 more gadgets than -O0 for GCC and 75 for Clang (not shown in the Figure 5 for brevity). We analyze and find the following reasons.

Reason 1: The main reason for the unoptimized code (-O0) to have such a low number of gadgets is that the unoptimized code has zero (0) LM, SM, and MR gadgets for all applications. All three gadgets (LM, SM, and MR) are involved with the `mov` instruction. In unoptimized code, other instructions such as `leave`, `pop rbp`, `add rsp, 0x30`, etc. are present between the `mov` and `ret` instructions. However, optimized code ($\geq$ -O1) may remove these additional instructions between `mov` and `ret`. Listing 2 shows portion of assembly code in three optimization levels (-O0, -O1, and -O2) for the C function demonstrated in Listing 1. Listing 2 shows that there is no `mov` and `ret` instructions stay together in optimization level -O0 or -O1, but in optimization level -O2. The tendency of `mov` and `ret` instructions staying together in optimized code primarily contributes to the presence of LM, SM, and MR gadgets in optimized code.

Listing 1: Sample C function.

```
long f(int i){
    if (i==0)return 1;
    else return i * f(i-1);
}
```

Listing 2: Portion of assembly code of the C function above in different optimization levels.

```
    -O0                 |    -O1              |    -O2
<clipped>               | <clipped>           | <clipped>
mov    edi,eax          | mov    ebx,edi      | mov    eax,0x1
<clipped>               | <clipped>           | ret
add    rsp,0x18         | pop    rbx          |
pop    rbx              | ret                 |
pop    rbp              |                     |
ret                     |                     |
```

Reason 2: For optimizations, compilers sometimes emit extra instructions that increase gadgets. Additionally, many `mov` instructions get replaced by `xor` instructions. This replacement increases logical gadgets while decreases MR gadgets.

VI. 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 [22], [46], [81], [84]. Randomization tools such as

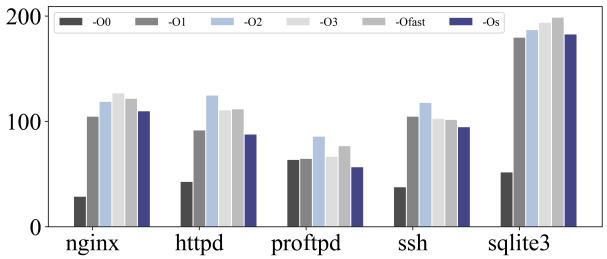

Fig. 5: The number of Turing-complete minimum footprint gadgets in different optimization levels for GCC.

PaX ASLR [81], SR [22], CCR [46], Remix [20], Binary stirring [84], ILR [40] and ASLP [45] use the entropy values as the security metrics to evaluate the security of their randomization schemes. Some tools such as SR [22], CCR [46], Remix [20], and ASLP [45] calculate the entropy value as a function of the number of functions and basic blocks. That means these tools permute the order of functions or basic blocks of program code and place it different places in the address of that program. These tools also consider some constraints such as executable size, and fall-through 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 jump and call) 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. A BOP attack chains blocks of code (e.g., basic blocks) to achieve malicious goals [43]. We attempted to evaluate the impact of code randomization on the availability of BOP gadgets [43]. We used the BOP compiler (BOPC) to find the BOP gadgets. For CCR [46] and MCR [41], we observed almost an identical number of BOP functional blocks for register assignments, register modifications, memory reads, memory writes, system/library calls and conditional jumps in the randomized versions compared to the non-randomized versions[9]. The identical number is expected as CCR and MCR reorder the basic or functional blocks without reordering the content of a basic block or a functional block. This implies that randomization solutions up to basic block-level have no impact on the availability of BOP 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 II, 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 the performance over security, then they can set re-randomization time as the time just before when the gadget discovery process achieves 100% of the Turing-complete gadgets. If the owners prioritize security over performance, they can consider setting the interval as the time when the gadget discovery process achieves around 60-70% of the Turing-complete gadgets.

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 in Glibc. Glibc contains GCC specific non-standard extensions (e.g., ASM GOTO), which Clang does not cover. SR cannot randomize some parts of Glibc. Therefore, we evaluate a lightweight version of the standard C library `musl-libc` [1]) instead of Glibc. Selfrando works on `musl-libc`. However, we are unable to diversify it with the other tools. On the other hand, shuffler can reorder Glibc code by making a few modifications such as disabling manual jump table construction.

Limitations. Our current work does not measure zombie gadgets [76]. The gadgets that are available after applying destructive read defenses are called *zombie gadgets* [76]. Destructive read defenses (e.g., XnR [6], NEAR [85], Readactor [24], and Heisenbyte [80]) are known to protect programs from JIT-ROP attacks. Destructive read defenses only allow code execution. Any attempt to read code pages terminate a process. In this way, destructive reads destroy the availability of gadgets to attackers. However, destructive read defenses cannot completely eliminate all gadgets. For example, runtime code generation capability of JIT compilers allows the creation of multiple copies of the same code (e.g., two native code regions can be created from the same JavaScript code, one copy is used for disclosing layout, and another copy is used for mounting attacks). In addition, loading and unloading features of dynamic libraries allow attackers to load, disclose, destroy, and unload code pages. A fresh loading of the destroyed code pages can be used in attacks utilizing the layout information of the disclosed code pages. Similarly, attackers can infer code layout by creating new processes (e.g., creating new tabs in browsers using JavaScript) and making an informed guess about neighboring bytes after disclosing a few bytes (i.e., implicit reads [76]). Thus, JIT compilers, load/unload features, new process creation, and implicit reads allow attackers to get gadgets even in the presence of destructive read defenses.

In our future work, we plan to assess the availability of zombie gadgets after applying many destructive read defenses. In particular, we plan to assess the entity (i.e., JIT compilers or load/unload feature or new process creation or implicit reads) that can facilitate attacks by providing most gadgets. We will categorize the available zombie gadgets in seven Turing-complete (TC) categories and measure the availability of zombie gadgets in different TC categories.

[9]BOPC requires an executable binary to search for BOP gadgets. Thus, we could not use Shuffler [86] and SR [22] because they randomize the layout at runtime. BOPC does not seem to run on binaries produced by Zipr [38].

Another limitation is that our threat model assumes that code-pointer-obfuscation based defense is not deployed. If used, code pointer obfuscation could make JIT-ROP code page discovery less effective, reducing the gadget availability. For example, Oxymoron [7] showed some effectiveness for obstructing a JIT-ROP attack by making code pointer transformation or redirection through a randomization-agnostic translation table. Code Pointer Integrity (CPI) proposed by Kuznetsov *et al.* [48] is shown to be effective for mitigating JIT-ROP and COOP attacks. Understanding how code pointer obfuscation impacts JIT-ROP and measuring the effectiveness of these defenses under various attack conditions (e.g., Isomeron [28] and COOP [68]) are interesting problems.

Key Takeaways

Effective re-randomization interval. A methodology for the systematic measurement of a Turing-complete gadget set can help compute the effective upper bound for re-randomization intervals of a re-randomization scheme. Our experiments show that this upper bound ranges from 4 seconds to 17 seconds for various applications with dynamic libraries. The upper bound indicates the maximum amount of time between two consecutive randomization rounds that prevent an attacker from obtaining a Turing-complete gadget set. Applying our methodology on their own machines will help re-randomization adopters to make more informed configuration decisions.

Turing-complete operations. Function, basic-block, or machine register level fine-grained randomization preserves Turing completeness, however, instruction-level randomization does not. Besides, unoptimized code seems to limit more Turing-complete operations than optimized code.

All leaked pointers are created equal. Regardless of the location of pointer leakage, we are able to 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. Any leaked pointers would enable attackers to harvest a large number of code pages and gadgets.

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. Approaches for obfuscating code connectivity are promising in building JIT-ROP defenses.

Gadget quality. Our findings suggest that the current fine-grained randomization solutions do not impose significant gadget corruption.

VII. 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 [59], return-to-libc [47], [55], [63], [87], to ROP [15], [16], [44], Jump Oriented Programming (JOP) [12], DOP [42], ASLR bypasses [10], [28], [33], [42], [75], and CFI bypasses [9], [14], [15], [36], [43]. In the meantime, researchers have proposed a range of defenses for ROP attacks [3], [11], [17], [21], [24], [25], [27], [29], [30], [32], [37], [56], [58], [60], [61], [62], [69], [82], [88], [89], CFI bypass [88], and ASLR bypass [6], [7], [8], [20], [24], [28], [35], [40], [45], [46], [51], [52], [61], [80], [84], [85], [86]. A categorical representation of these defenses is given in our attack-path diagram (Figure 6 in the Appendix). Binary analysis tools are also available to understand [74] and mitigate [83] these ROP or code-reuse attacks.

TABLE VI: Conditions and capabilities of offensive techniques. PIE stands for **P**osition **I**ndependent **E**xecutables. ✓ indicates a successful bypass by some other mechanisms and — means that an attack assumes a bypass. ● and ◐ indicate precise and coarse-grained CFI-bypasses, respectively. CG → coarse-grained and FG → fine-grained. * Not explicitly mentioned. We assume FG ASLR must use PIE.

	Ability to bypass					
Attack	Need a leak	CG ASLR w/o PIE	CG ASLR w/ PIE	FG ASLR	CFI	FG Tools
AOCR [66]	No	✓	✓	✓	×	Readactor [24]
JIT-ROP [75]	Yes	✓	✓	✓	×	ORP [61]
BOP [43]	Yes	✓	✓	—	●	—
DOP [42]	Yes	✓	×	×	◐	—
CROP [33]	No	—	—	✓	◐	*
BROP [10]	Yes	—	✓	×	×	—
Isomeron [28]	Yes	—	—	✓	×	Oxymoron [7]
CFB [14]	—	—	—	—	●	—
OOC [36]	Yes	✓	✓	×	◐	—
EHH [15]	—	—	—	—	◐	—

Most of the above-mentioned defenses are variants of W⊕X (e.g., No-Execute-After Read [85] and Heisenbyte [80]), memory safety (e.g., HardScope [57], Memcheck [54], AddressSanitizer [71], and StackArmor [19]), ASLR (e.g., fine-grained code randomization/code diversification [20], [46], [84], code re-randomization [8], [86], and SGX-Shield [70]), and CFI (e.g., CCFIR [88] and bin-CFI [89]). These defenses are capable of preventing most code-reuse or ROP-based attacks [10], [28], [33], [75] except a few cases such as inference attacks that are performed using zombie gadgets [76] or relative address space layout [66]. However, In practice, imprecision in resolving indirect control-flow transfers impacts CFI's security guarantees. In addition, there are trade-offs between using coarse-grained CFI (performance overhead is low when enforced) and precise CFI (performance overhead is high when enforced). Similarly, some attacks (e.g., DOP [42]) fail in the presence of a position independent executable. Table VI shows some attacks that bypass many of the modern defenses. We infer this information from the papers. Some information is not explicitly mentioned in those papers. We indicated these by — and *. Our work, quantitatively measuring to what extent ASLR makes code-reuse attacks difficult, complements these attack demonstrations and defenses. Shacham *et al.* measured the entropy improvement (at most 1 bit compared to PaX ASLR [81]) by doing function-level reordering in their paper [73]. That work did not measure how the function-level reordering impacts ROP gadgets. We evaluate the impact of function-level randomization as well as the basic block level randomization. Besides, Veen *et al.* demonstrated a dynamic gadget discovery tool called NEWTON that assists attackers for crafting code-reuse exploits [82]. However, they do not perform quantitative analysis like ours.

VIII. 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.

References

[1] "Musl libc: A lightweight standard c library," https://www.musl-libc.org, 2018, last accessed 10 February 2018.

[2] "Universal dep/aslr bypass with msvcr71.dll and mona.py," https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/, 2018, last accessed 10 February 2018.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[4] E. Acri, "Gdb-python-utils," https://github.com/crossbowerbt/GDB-Python-Utils, 2018, last accessed 10 September 2018.

[5] P. Argyroudis and C. Karamitas, "Exploiting the jemalloc memory allocator: Owning firefox's heap," *Blackhat USA*, 2012.

[6] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1342–1353.

[7] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing." in *USENIX Security Symposium*, 2014, pp. 433–447.

[8] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 268–279.

[9] A. Biondo, M. Conti, and D. Lain, "Back to the epilogue: Evading control flow guard via unaligned targets." NDSS, 2018.

[10] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 227–242.

[11] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 353–362.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[13] Capstone, "The ultimate disassembler," http://www.capstone-engine.org, 2018, last accessed 26 September 2018.

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

[15] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security Symposium*, 2014, pp. 385–399.

[16] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[17] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *International Conference on Information Systems Security*. Springer, 2009, pp. 163–177.

[18] X. Chen, H. Bos, and C. Giuffrida, "Codearmor: Virtualizing the code space to counter disclosure attacks," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 514–529.

[19] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *NDSS*, 2015.

[20] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 50–61.

[21] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against rop attack." Internet Society, 2014.

[22] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 454–469, 2016.

[23] S. C. Cowan, S. R. Arnold, S. M. Beattie, and P. M. Wagle, "Pointguard: method and system for protecting programs against pointer corruption attacks," Jul. 6 2010, uS Patent 7,752,459.

[24] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 763–780.

[25] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 292–307.

[26] C. Curtsinger and E. D. Berger, "Stabilizer: statistically sound performance evaluation," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 219–228.

[27] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 40–51.

[28] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming." in *NDSS*, 2015.

[29] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*. ACM, 2009, pp. 49–54.

[30] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 75–88.

[31] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 781–796.

[32] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," Technical report, Tech. Rep., 2012.

[33] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding." in *NDSS*, 2016.

[34] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2018.

[35] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization." in *USENIX Security Symposium*, 2012, pp. 475–490.

[36] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 575–589.

[37] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX Association, 2014, pp. 417–432.

[38] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, "Zipr: Efficient static binary rewriting for security," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 559–566.

[39] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," *arXiv preprint arXiv:1804.08470*, 2018.

[40] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 571–585.

[41] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–11.

[42] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 969–986.

[43] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1868–1882.

[44] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 2012, pp. 94–105.

[45] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 339–348.

[46] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 461–477.

[47] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005.

[48] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity." in *OSDI*, vol. 14, 2014, p. 00000.

[49] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race*. Association for Computing Machinery and Morgan & Claypool, 2018, pp. 81–116.

[50] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make aslr win the clone wars: Runtime re-randomization." in *NDSS*, 2016.

[51] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 280–291.

[52] G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses." in *USENIX Security Symposium*, 2016, pp. 139–156.

[53] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 941–951.

[54] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[55] T. Newsham, "Non-exec stack," *Bugtraq mailing list*, 2000.

[56] B. Niu and G. Tan, "Modular control-flow integrity," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 577–587, 2014.

[57] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A.-R. Sadeghi, "Hardscope: Hardening embedded systems against data-oriented attacks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[58] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.

[59] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, November 1996. [Online]. Available: http://www.phrack.com/issues.html?issue=49&id=14

[60] V. Pappas, M. Polychronakis, and A. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing." in *USENIX Security Symposium*, 2013, pp. 447–462.

[61] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[62] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[63] A. Peslyak, ""return-to-libc" attack," *Bugtraq, Aug*, 1997.

[64] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack." in *USENIX Security symposium*, 2016, pp. 1–18.

[65] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[66] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz *et al.*, "Address-oblivious code reuse: On the effectiveness of leakage resilient diversity," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[67] S. Schirra, "Ropper tool," https://github.com/sashs/Ropper, last accessed 4 July 2018.

[68] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[69] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-rop defenses," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 88–108.

[70] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.

[71] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[72] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[73] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[74] Y. Shoshitaishvili, C. Kruegel, G. Vigna, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[75] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[76] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 954–968.

[77] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, vol. 2007, 2007.

[78] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*. ACM, 2009, pp. 1–8.

[79] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

[80] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings*

of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 256–267.

[81] P. Team, "Pax address space layout randomization (aslr)," 2003.

[82] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1675–1689.

[83] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 934–953.

[84] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[85] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 35–46.

[86] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization." in *OSDI*, 2016, pp. 367–382.

[87] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[88] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[89] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *USENIX Security Symposium*, 2013, pp. 337–352.

APPENDIX

A. 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 3 is `mov eax, edx` and the gadget serves as a move register (MR) gadget. The core instruction is the instruction that an attacker needs. All the instructions (except `ret`) before/after the core instruction is unnecessary. However, these additional instructions may modify the source/destination register value of a core instruction. If these additional instructions modify the register values of a core instruction, we treat the gadget as a corrupted gadget. In Listing 3, the instruction (`mov edx, dword ptr [rdi]`) before the core instruction modifies the value of the source register (`edx`) of the core instruction and the instructions (`shr eax, 0x10; xor eax, edx`) after the core instruction modify the destination register (`eax`) value. We identify three scenarios when core instructions get corrupted as follows:

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

For each gadget, we consider these three scenarios and determine whether the gadget is corrupted or not.

Listing 3: An example gadget where the core instruction is "mov eax, edx;".

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

Considering the three scenarios above, we identify three types of gadgets where the core instruction can get corrupted. Figure 7 shows the three type of gadgets. Each gadget has one or more instructions before or after the core instruction. For example, Type 1 gadget in Figure 7 has a core instruction in the middle and one or more instructions before or after the core instruction. The core instruction has two registers for this kind. One or more instruction(s) before the core instruction may modify the source register (`rdx`) in Figure 7a. Similarly, one or more instruction(s) after the core instruction may modify the destination register (`rax`) in the figure.

However, for Type 2 gadget in Figure 7b, the core instruction has just one register. That means that the additional instructions before the core instruction cannot affect the register of the core instruction. Thus, we do not care the instructions before the core instruction. For Type 3 gadget in Figure 7c, the core instruction write 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.

We treat 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 for Type 1 and 3 gadgets. We call this set of registers as `RegSet1`. Similarly, we extract `RegSet2` for the instructions after a core instruction for Type 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-ST`, `SP`, and `CALL` gadgets.

B. Validation of randomization results

We evaluate the randomization tools, i.e., Zipr [38], SR [22], MCR [41], and CCR [46] using the common set of applications and libraries that the four randomization tools can randomize. Figure 8 shows the reduction of Turing-complete gadgets observed for different 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.

C. Impact of leave Instruction

Intuitively, one might think that a `leave` instruction may provide similar functionality to an LR gadget, as it unfolds to `mov rsp, rbp; pop rbp`. Or, a store memory (SM) gadget might still serve its purpose with the addition of a `leave` instruction once unfolded. However, the `leave` instruction modifies the stack pointer (SP), which affects the control flow of the resulting gadget chain. We experimentally prove the impact of `leave` instruction on a gadget chain in the following paragraph.

If we unfold the `leave` instruction, we get two instructions: `mov rsp, rbp` and `pop rbp` in Intel syntax. Every time when a function returns, it resets the stack pointer (`rsp`) by the value of the base pointer (`rbp`) using `mov rsp, rbp` instruction, restores the old `rbp` value from the stack using `pop rbp` instruction, and calls `ret` instruction. Compiler optimization techniques replace the `mov` and `pop` instructions by a single `leave` instruction. This reduces the overall program size. When the `leave` instruction is used, a function simply returns by using the `leave` instruction followed by a `ret` instruction. Since `leave` instruction resets `rsp` to `rbp`, this redirects the control-flow of a gadget chain if the `leave` instruction is used in the gadget chain. To assess the impact, we write two ROP gadget chains that spawn a shell. One of the gadgets of the first chain is `mov rax, 0x1; ret`. We replace the `mov rax, 0x1; ret` gadget by

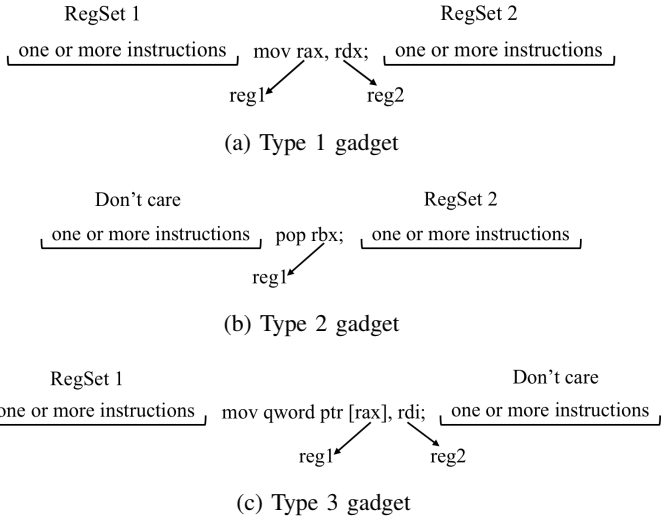

Fig. 7: A set of gadget types for measuring the quality of individual gadget through the register corruption analysis

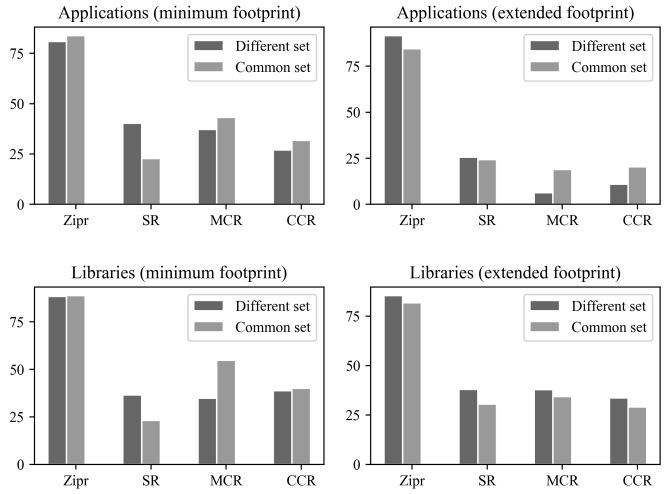

Fig. 8: Reduction (%) of Turing-complete gadgets observed for different randomization tools using the common set of applications and libraries that the randomization tools can randomize.

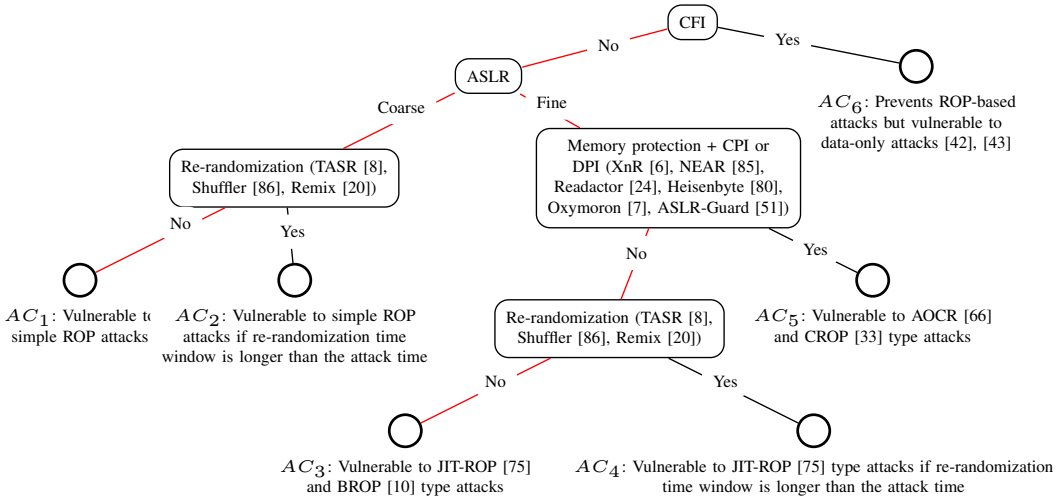

Fig. 6: High-level view of the types of ROP attacks and attack-paths based on various security measures. Each rectangle and circle indicates security measures and attack types, respectively. AC stands for attack condition. All the attack conditions have W⊕X, PIE, Canary, and RELRO implicitly.

TABLE VII: Gadgets used in advanced ROP attacks [10], [14], [15], [36], [75] . △ indicates an addition/subtraction/multiply/division. ▽ indicates any operation that modifies stack pointer (SP). SN → Short name. TC? indicates whether a gadget is included in the Turing-complete gadget set or not.

Gadget types	Purpose	Minimum footprint	Example	TC?	SN	Source
Move register	Sets the value of one register by another	mov reg1, reg2; ret	mov rdi, rax; ret	✓	MR	[75]
Load register	Loads a constant value to a register	pop reg; ret	pop rbx; ret	✓	LR	[14], [75]
Arithmetic	Stores an arithmetic operation's result of two register values to the first	△ reg1, reg2; ret	add rcx, rbx; ret	✓	AM	[75]
Load memory	Loads a memory content to a register	mov reg1, [reg2]; ret	mov rax, [rdx]; ret	✓	LM	[14], [75]
Arithmetic load	△ a memory content to/from/by a register and store in that register	△ reg1, [reg2]; ret	add rsi, [rbp]; ret	✓	AM-LD	[75]
Store memory	Stores the value of a register in memory	mov [reg1], reg2; ret	mov [rdi], rax; ret	✓	SM	[75]
Arithmetic store	△ a register value to/from/by a memory content and stores in that memory	△ [reg1], reg2; ret	sub [ebx], eax; ret	✓	AM-ST	[75]
Stack pivot	Sets the stack pointer, SP	▽ sp, reg	xchg rsp, rax	×	SP	[75]
Jump	Sets instruction pointer, EIP.	jmp reg	jmp rdi	✓	JMP	[75]
Call	Jumps to a function through a register or memory indirect call	call reg or call [reg]	call rdi	✓	CALL	[75]
System Call	Invokes system functions	syscall or int 0x80; ret	syscall	✓	SYS	[65]
Call preceded	Bypasses call-ret ROP defense policy	mov [reg1], reg2; call reg3	mov [rsp], rsi; call rdi	×	CP	[14]
Context switch	Allows processes to write to Last Branch Record (LBR) to flash it	long loop.	3dd4: dec, ecx 3dd5: fmul, [BC8h] 3ddb: jne, 3dd4	×	CS1	[14]
Flashing	Clears the history of LBR (Last Branch Record)	Any simple call preceded gadgets with a ret instruction	jmp A ... A: mov rax, 3; ret;	×	FS	[15]
Terminal	Bypasses kBouncer heuristics	Any gadgets that are 20 instructions long	N/A	×	TM	[15]
Reflector	Allows to jump to both call-preceded or non-call-preceded gadgets	mov [reg1], reg2; call reg3; ... ; jmp reg4	mov [rsp], rsi; call rdi; ... ; jmp rax	×	RF	[14]
Call site	This gadget chains the control to go forward when we have the control on the stack and ret	call reg or call [reg]; ... ret;	call rdi; ... ret;	×	CS2	[36]
Entry point	This gadget chains the control to go forward when we have the control of a call instruction	pop rbp; ... call/jmp reg or call/jmp [reg]	pop rbp ... call/jmp reg or call/jmp [reg]	×	EP	[36]
BROP	Restores all saved registers	pop rbx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;	pop rbx; pop rbp; pop r12; pop r13; pop r14; pop rsi; pop r15; pop rdi; ret;	×	BROP	[10]
Stop	Halts the program execution	Infinite loop	4a833dd4: inc rax 3ddb: jmp 3dd4	×	STOP	[10]

TABLE VIII: 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 [22].

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

`mov rax, 0x1; leave; ret` to generate a second gadget chain. So, the gadget chains are as follows: `<...clipped...>` `mov rax, 0x1; ret` `<...clipped...>` and `<...clipped...>` `mov rax, 0x1; leave; ret` `<...clipped...>`. We successfully get a shell using the first gadget chain but fail to get a shell using the second chain. The reason for the failure is that the `rsp` is reset to `rbp` which results in the redirection of the execution of gadget chain from the address (or value) saved in `rbp`. In this way, `leave` instructions break the gadget chain.