PtrProxy: Efficient Code Re-Randomization on AArch64 Platform

Luo Chenke[1], Fu Jianming[1,*], Ming Jiang[2], Xie Mengfei[1], Peng Guojun[1]

[1] Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan, Hubei 430072 China

[2] Department of Computer Science, Tulane University, New Orleans, LA 70118 USA

[*] The corresponding author, email: jmfu@whu.edu.cn

Cite as: C. Luo, J. Fu, *et al*., "Ptrproxy: Efficient code re-randomization on aarch64 platform," *China Communications*, 2025, vol. 22, no. 6, pp. 63-81. **DOI**: 10.23919/JCC.ja.2024-0077

Abstract: Memory-unsafe programming languages, such as C/C++, are often used to develop system programs, rendering the programs susceptible to a variety of memory corruption attacks. Among these threats, just-in-time return-oriented programming (JIT-ROP) stands out as an advanced method for conducting code-reuse attacks, effectively circumventing code randomization safeguards. JIT-ROP leverages memory disclosure vulnerabilities to obtain reusable code fragments dynamically and assemble malicious payloads dynamically. In response to JIT-ROP attacks, several re-randomization implementations have been developed to prevent the use of disclosed code. However, existing re-randomization methods require recurrent re-randomization during program runtime according to fixed time windows or specific events such as system calls, incurring significant runtime overhead.

In this paper, we present the design and implementation of PtrProxy, an efficient re-randomization approach on the AArch64 platform. Unlike previous methods that necessitate frequent runtime re-randomization or reply on unreliable triggering conditions, this approach triggers the re-randomization process by detecting the code page harvest operation, which is a fundamental operation of the JIT-ROP attacks, making our method more efficient and reliable than previous approaches. We evaluate PtrProxy on benchmarks and real-world applications. The evaluation results show that our approach can effectively protect programs from JIT-ROP attacks while introducing marginal runtime overhead.

Keywords: code reuse attacks; re-randomization; return-oriented programming; security and privacy; software security

Received: Feb. 01, 2024
Revised: Mar. 10, 2024
Editor: Fang He

I. INTRODUCTION

The battle between cyber attackers and defenders over memory corruption vulnerabilities has intensified, resulting in an ongoing struggle [1–9]. With modern operating systems deploying W⊕X protection (memory cannot be writable and executable at the same time), attackers have resorted to reusing code snippets from vulnerable programs to construct their attacks [10]. These code snippets, also known as "gadgets," are identified by examining the disassembled binary code. Attackers then connect these gadgets in a precise sequence to create harmful payloads and redirect the control flow to the gadgets to launch the attack. Control Flow Integrity (CFI) [11–20] and Shadow Stacks (SS) [21–26] have been proposed to prevent the hijacking of control flow. However, they do not always guarantee adequate protection [27–36]. To mitigate the ROP threat in the case of CFI and SS failure,

researchers have proposed various code randomization techniques [37–49] to mitigate the threat of ROP attacks, which reorganize the code layout in memory to impede the construction of gadgets. However, code randomization is susceptible to memory disclosure, making the randomized code layout evident to attackers and undermining the fundamental memory secrecy assumption of code randomization [50]. To circumvent code randomization protection, attackers have developed the JIT-ROP technique [51]. It leverages repeated exploitation of memory disclosure vulnerabilities to collect code gadgets on the fly, utilizing the disclosed code pointers present on memory pages. Hence, the technique of JIT-ROP can bypass the protection mechanism of code randomization, rendering even the finest-grained randomization strategies ineffective.

Execute-only Memory (XoM) [52–63] protects programs from JIT-ROP attacks by preventing attackers from disclosing executable memory. However, even though modern compilers refrain from embedding data in code areas, the embedded data is still present in real-world binaries [64, 65, 54, 66, 56]. Moreover, precisely separating data and code in binaries is still an undecidable problem [67, 66, 65], which makes it difficult to implement XoM protection on precompiled legacy programs. These limitations render the deployment of XoM impractical in real-world environments. To overcome the restrictions imposed by XoM, researchers have introduced the concept of Destructive Code Reads (DCR) [66, 68–70]. DCR does not prohibit attackers from disclosing code; instead, it destructively erases the disclosed code to prevent the code from being executed as gadgets. Although researchers have taken a step forward in attempting to defend against JIT-ROP attacks with DCR, the DCR has been shown to be unable to protect programs effectively from the impact of some advanced JIT-ROP attacks [71]. Similar to DCR, re-randomization [72–81] does not restrict attackers from disclosing code. It continuously randomizes the code layout during program runtime to prevent attackers from leveraging disclosed gadgets to mount JIT-ROP attacks. Re-randomization does not impose restrictions on embedded data, and in comparison to DCR, it offers enhanced security protection, making it an excellent choice for countering JIT-ROP attacks. However, existing re-randomization methods either randomize the code layout by frequent randomization at specific time intervals, which incurs a high overhead [82], or rely on conditions such as I/O or program crashes that are not closely tied to the characteristics of JIT-ROP attacks, leaving attackers with opportunities to complete JIT-ROP attacks without triggering the randomization.

In this paper, we propose a novel re-randomization method called *PtrProxy* to effectively protect against JIT-ROP attacks with marginal overhead. PtrProxy utilizes the *code page harvest*, a necessary step for JIT-ROP attacks, as the trigger condition for re-randomization. During the JIT-ROP attacks, attackers need to disclose numerous mapped code pages to search for gadgets. The mapped code pages are collected by referring to code pointers gathered from known code pages. This process is known as code page harvest. The code page harvest must be undertaken to avoid attempting to disclose unmapped memory regions, which could disrupt the attack [66, 82]. PtrProxy detects the code page harvest by redirecting code pointers from code pages to an unreadable trampoline. Upon detecting the occurrence of code page harvest, PtrProxy triggers a code layout randomization to thwart the ongoing JIT-ROP attack. PtrProxy can work on stripped binaries with marginal overhead, which makes PtrProxy an optimal choice for defending JIT-ROP attacks.

We implemented the prototype of PtrProxy and deployed it on a development board with the ARMv8 architecture to evaluate its performance and security. We employ microbenchmarks to assess the additional system overhead introduced by PtrProxy's modifications to the system kernel. We also use macrobenchmarks to assess the additional performance overhead introduced by PtrProxy on computationally intensive programs. In addition, we use Phoronix Test Suite [83] to assess the performance impact of PtrProxy on real-world applications. The evaluation results indicate that PtrProxy exhibits excellent performance, causing marginal runtime overhead on benchmarks and real-world applications. We use a gadget search experiment and a real-world case study to demonstrate the protective capabilities of PtrProxy. The experimental results demonstrate that PtrProxy can effectively protect programs from the JIT-ROP attacks.

In a nutshell, we make the following key contributions:

- We proposed a novel re-randomization ap-

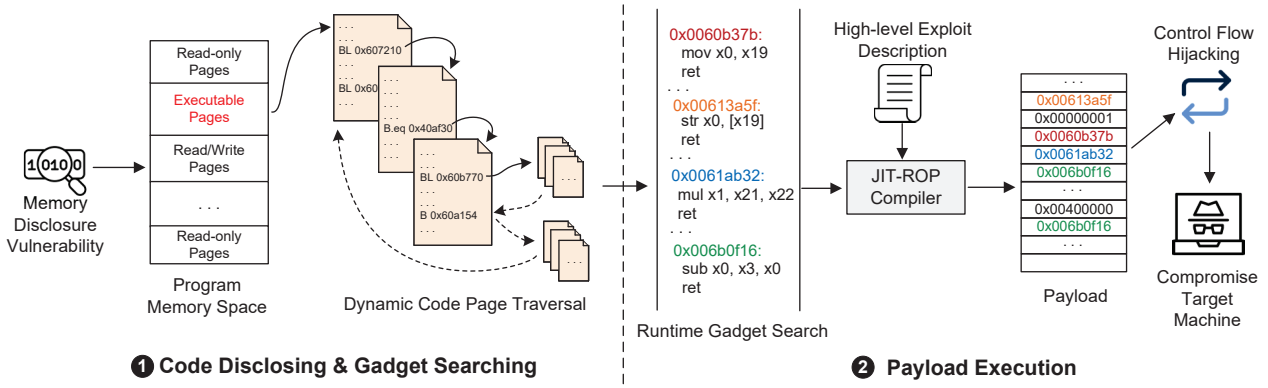

Figure 1. *Overview of a typical JIT-ROP attack.*

proach, PtrProxy, which efficiently implements re-randomization to defend against JIT-ROP attacks.

- We innovatively utilized pointer forwarding to detect code page harvest.
- To the best of our knowledge, PtrProxy reveals minimal runtime overhead compared to existing re-randomization tools. Our extensive evaluation demonstrates that PtrProxy is a practical solution for real-world adoption.

II. BACKGROUND

In this section, we provide background information on JIT-ROP attacks and the importance of addressing memory disclosure vulnerabilities. We also review existing JIT-ROP defense approaches and identify their limitations (e.g., lack of practicality), which have prompted our research. Finally, we introduce how the limitations of existing re-randomization approaches motivate our research.

2.1 Just-In-Time ROP

W⊕X prevents attackers from executing shellcode injected into the stack, thereby shifting the memory corruption attacks towards ROP attacks that reuse existing code snippets within the program. The development of fine-grained randomization defenses [84, 85, 41, 40, 47, 86] has increased the difficulty of implementing ROP attacks, leading traditional ROP attacks to evolve into more complex JIT-ROP attacks [51]. The main characteristic of JIT-ROP attacks is their ability to search for gadgets in real time and construct attack payloads by exploiting dynamically disclosed program code. Figure 1 illustrates a typical JIT-ROP attack process. The entire JIT-ROP process can be divided into two stages: code disclosing & gadget searching (❶ in Figure 1) and payload execution (❷).

In stage ❶, attackers exploit memory disclosure vulnerabilities to dynamically disclose the code of the target program and search for usable gadgets. To avoid crashing the program by searching unmapped code pages and interrupting the attack process, instead of linearly searching through the entire process memory space, attackers collect code pointers from known code pages to identify other mapped code pages. Subsequently, they traverse other mapped code pages using specific strategies. Common traversal strategies include Depth-first Traversal Strategy and Breadth-first Traversal Strategy [82]. During the traversal of code pages, attackers conduct gadget searches to find gadgets that can be used to construct attack payloads, such as reading/writing memory, executing arithmetic operations, etc. Subsequently, in stage ❷, attackers employ a JIT-ROP compiler to dynamically construct the attack payload at runtime based on a high-level exploit description and the collected gadgets. They then hijack the control flow to execute the attack payload. As a result, attackers get control of the targeted devices by executing the attack payload.

The whole JIT-ROP attack process is carried out on the fly, demanding that attackers access a significant number of code pages within a short timeframe during stage ❶ and rapidly construct the attack payload. XoM disrupts JIT-ROP attacks by limiting the attackers' capability of code disclosure in stage ❶. Unlike XoM, both DCR and re-randomization defend

against JIT-ROP attacks in stage ❷ by restricting attackers' ability to execute the disclosed code. The difference between DCR and re-randomization is that DCR destroys disclosed code to prevent gadget execution, while re-randomization shuffles the code layout continuously to prevent gadget execution.

2.2 Memory Disclosure Prevention

2.2.1 Execute-Only Memory

(XoM) works on stage ❶ in Figure 1 to defend JIT-ROP attacks by removing the read permission of code pages. However, in the traditional memory permission control model, the executable and read permissions of code pages are inseparable. This implies that memory pages with executable permissions must also have read permissions. XnR [52] introduced the prototype of XoM for the first time by prohibiting the reading of all executable memory through software simulation. However, XnR assumes that code does not include data, and this assumption is often not valid [65]. After XnR, there are two types of subsequent work on XoM, attempting to make this assumption valid. The first type of subsequent XoM work is *compile-time transformation* [57, 53, 55, 58, 59]. This type of work attempts to relocate embedded data by re-compiling source code using custom compilers. The drawback of this type of method is evident – they are incapable of protecting a vast number of pre-compiled legacy programs. Moreover, these methods often struggle to handle embedded data in hand-written assembly code. The second type of subsequent XoM work is *binary hardening* [56, 54, 60]. This type of method attempts to use binary rewriting to separate embedded data in binaries. However, currently, differentiating code and data in binaries remains an undecidable problem [67, 66, 65]. Therefore, the separation of embedded data in binaries through binary rewriting often encounters challenges, leading to incomplete separation of embedded data or erroneous separation of code as data. These problems render XoM currently impractical.

2.2.2 Destructive Code Reads

(DCR) [66, 68] work on stage ❷ to defend JIT-ROP attacks while still allowing legitimate data reads within code pages. DCR allows attackers to disclose code, but destroys disclosed code to prevent their execution as gadgets. It does not prevent attackers from reading code pages, thus breaking the assumption of XoM that embedded data must not exist. Embedded data in the program can still be read as usual, but after reading, it cannot be executed as code. While DCR seemed to take a significant step toward practical deployment of JIT-ROP defense, unfortunately, four attack methods proposed by Snow et al. [71] completely broke DCR's security guarantees, rendering DCR an unsuccessful attempt at the JIT-ROP defenses.

2.3 Re-Randomization

Re-randomization also works in stage ❷ to defend against JIT-ROP attacks. In contrast to XoM and DCR, re-randomization is agnostic to code disclosure. Instead, it continuously randomizes the code layout during program runtime based on various trigger conditions. This ensures that the positions of gadgets are constantly changing and attackers are unable to utilize previously identified gadgets for conducting attacks. After each randomization, attackers are unable to utilize previously identified gadgets for conducting attacks. Thus, re-randomization provides an effective defense mechanism against attacks. Based on differences in trigger conditions, we classify re-randomization into two types: **time-driven** and **event-driven**.

2.3.1 Time-Driven Re-Randomization

This type of re-randomization blindly conducts randomization at specific intervals, regardless of whether an attack is in progress. Shuffler [72], as a representative of this category of methods, implemented re-randomization on the x86-64 platform. It supports randomization intervals of 50ms, 100ms, and 200ms. When randomization is performed every 50ms, Shuffler incurs an additional runtime overhead of 14.9%. However, such a runtime overhead is deemed unacceptable for a security feature. HARM [80] designed a re-randomization scheme for microcontrollers. It employs the same trigger strategy as Shuffler, but on U-Disk FreeRTOS, it incurs an overhead of 21%. CodeArmor [75] and Stabilizer [77] incur overheads of 6.9% and 6.7%, respectively. Time-driven re-randomization provides better security guarantees with shorter randomization in-

tervals. However, shorter intervals inevitably lead to higher overhead, and excessive performance overhead in the pursuit of security protection is unacceptable [87].

2.3.2 Event-Driven Re-Randomization

To reduce the additional overhead introduced by frequent re-randomization, researchers have proposed event-driven re-randomization. This type of method does not blindly randomize at fixed time intervals but instead triggers randomization by detecting the occurrence of specific events. TASR [73] performs randomization by detecting the occurrence of I/O events, randomizing the program's code layout every time an I/O request happens. On SPEC 2006, TASR's overhead is reduced to 2.1%, but on I/O-intensive programs, the overhead increases to 10.1%. MARDU [79] randomizes the program's code layout every time the program crashes. This means that the code layout will be different each time the program restarts after a crash. On SPEC 2006, MARDU has an average overhead of 5.5%. ReRanz [78] triggers re-randomization by monitoring the syscall calling pattern. It incurs an overhead of 5.33%. SafeHidden [81] triggers re-randomization by detecting attackers' memory layout probing. It results in a performance overhead of 5.78% on SPEC CPU 2006. RuntimeASLR [76] can only protect programs using the worker model (e.g., web servers). It achieves re-randomization of worker processes by dynamically instrumenting the parent process. While it incurs relatively low overhead in worker processes, it leads to an unacceptable overhead of several orders of magnitude in the parent process.

2.4 Motivation

Time-driven re-randomization blindly conducts randomization at frequent intervals during program runtime, incurring substantial overhead. Event-driven re-randomization, when employing reasonable trigger conditions, can provide effective protection with lower overhead. However, existing event-driven re-randomization methods do not tightly integrate the characteristics of JIT-ROP attacks when selecting trigger conditions. This gives attackers opportunities to bypass defense and employ JIT-ROP attacks. For example, the protection of TASR, MARDU, and ReRanz can be bypassed by attackers by carefully constructing the payload without triggering the I/O event, program crash, or specific syscall patterns. For SafeHidden, attackers can detect the memory layout by referring to the code pointers without probing. In such cases, SafeHidden becomes ineffective. The drawback of RuntimeASLR is more evident as it can only protect programs using the worker model, making it unable to protect other, more common types of programs. Moreover, existing methods trigger randomization under many benign behaviors, which leads to unnecessary additional overhead.

The drawbacks of existing event-driven re-randomization methods motivated our research. Our approach closely integrates the characteristics of JIT-ROP attacks. We detect the JIT-ROP attacks in stage ❶ and defend the attacks in stage ❷. By detecting the code page harvesting process, we can monitor JIT-ROP attacks and trigger re-randomization. Our method does not trigger the code randomization during normal program execution, but only initiates randomization when an attack is detected. Hence, our approach exhibits the best performance when compared to all previous re-randomization techniques.

III. OVERVIEW OF PTRPROXY

3.1 Threat Model

PtrProxy aims to defend against JIT-ROP by providing an event-triggered re-randomization approach, based on a well-defined adversary model. The model includes the following assumptions:

- W⊕X: The target system has a mechanism in place to prevent the coexistence of executable and writable permissions on the same memory page. This is a critical assumption that serves as the foundation of ROP defenses. If this mechanism were not in place, attackers could directly execute injected shellcode, rendering ROP techniques unnecessary.
- Control-Flow Hijacking: The program is vulnerable to memory corruption, allowing hijacking of control flow.
- Transparent Configuration: The attacker possesses knowledge of the target system's configuration, as well as access to the source code of the target program.

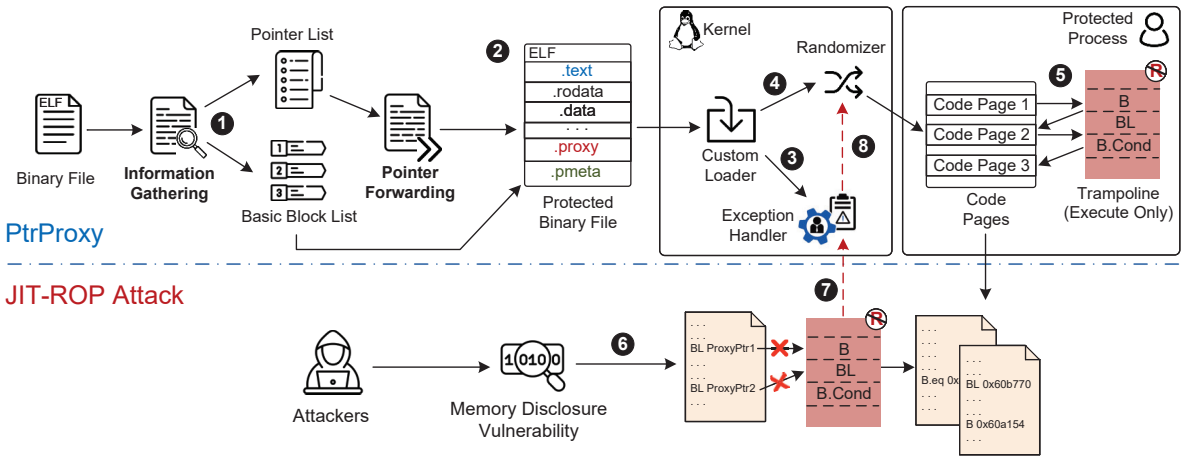

Figure 2. *Overview of PtrProxy.*

This adversary model is consistent with previous offensive and defensive papers [51–53, 56]. It specifically aligns with the robust model introduced in JIT-ROP attacks [51].

The architecture of PtrProxy is demonstrated in Figure 2. We demonstrate how PtrProxy protects programs in the upper part, and how PtrProxy detects the occurrence of harvest and triggers code randomization in the lower part.

3.2 Userspace Components

We design our re-randomization method as the basic block granularity. To collect basic block boundaries in the binary for further randomization use, and gather code pointers that need to be forwarded and fixed after randomization, we first perform static binary analysis on the binary to collect information (❶ in Figure 2). Next, we generate trampolines based on the source jump instruction and target addresses of the cross-page branch instructions. We then use binary rewriting to redirect the pointers in the code page to the trampoline. Finally, we attach the trampoline and basic blocks boundary information to the end of the ELF file (.proxy and .pmeta, ❷), and mark it as a PtrProxy-protected binary in the ELF header. This ensures that the loader in the kernel can correctly identify and load the protected binary.

3.3 Kernel Components

The kernel components are responsible for loading the protected binary, initializing metadata for randomization, monitoring the occurrence of code page harvest, and implementing randomization. Firstly, the custom loader recognizes the ELF header when loading the binary file. If it is a non-PtrProxy-protected binary, it will be handed over to the regular loader processing. If it is a PtrProxy-protected binary, the loader initializes the metadata required for randomization, and notifies the exception handler to monitor code page harvest and trigger randomization (❸). After the initializing, the custom loader loads the program code and trampoline, and triggers randomization to ensure the code layout is different between each run (❹). The trampoline must be loaded with execute-only permissions so that any disclosure of the trampoline triggers detection and initiates randomization. Since we can ensure that the trampoline does not contain any data, it can safely be set to execute only. In the protected user process, any jump between code pages necessarily goes through the trampoline (❺).

3.4 Randomization Triggering

When a JIT-ROP attack occurs, the attackers will attempt to perform code page harvest (❻). Code page harvesting is an essential process in JIT-ROP attacks [51, 82], as without this process, attackers will inevitably try to access unmapped memory areas when disclosing code, leading to program crashes and attack interruptions. When the attackers try to find other mapped code pages through pointers within a code page, it triggers the code page harvest detection (❼). Then, when the exception handler in the kernel detects that code page harvest is occurring, it triggers the ran-

domization process (❽). Please note that the order of trampoline items is also randomized to prevent attackers from guessing the trampoline layout through multiple leaks of the same memory page.

IV. DESIGN & IMPLEMENTATION

Programs on the AArch64 architecture often contain embedded data due to constant pools [54]. The prototype we implement can be deployed on AArch64 architecture-based smartphones, servers, and IoT devices. Due to the high efficiency of our approach, it is particularly well-suited for IoT devices, as IoT devices are particularly sensitive to overhead. Please note that PtrProxy can be adapted to other platforms with minimal modifications. In this section, we follow the workflow of PtrProxy to introduce its design and implementation.

4.1 Information Gathering

We implement the code re-randomization at the basic block granularity. Therefore, before performing randomization, it is necessary to collect basic block boundaries. Additionally, we need to collect all pointers to the code areas because, during randomization, we must fix all code pointers and embedded data pointers that reference code areas. We utilize Egalito [88] on stripped binaries to completely extract the pointers and basic block boundaries. Egalito is a binary recompilation technique that focuses on precise disassembly and restructuring of binaries using an advanced intermediate representation, enabling detailed analysis and manipulation of executable code. Egalito has provided a detailed explanation of the disassembly strategy, so we won't discuss it further here.

4.1.1 New ELF Format

To store the collected information for subsequent pointer forwarding and re-randomization, we define a new ELF file format, as depicted in Figure 3. Based on the original ELF file format, we choose the first undefined byte within the EI_PAD array in the e_ident field of the ELF header as the flag for PtrProxy. We name this flag *PROXY_ENABLED*. When the custom loader in the kernel loads the protected ELF file, it determines the loading process by checking this flag. We

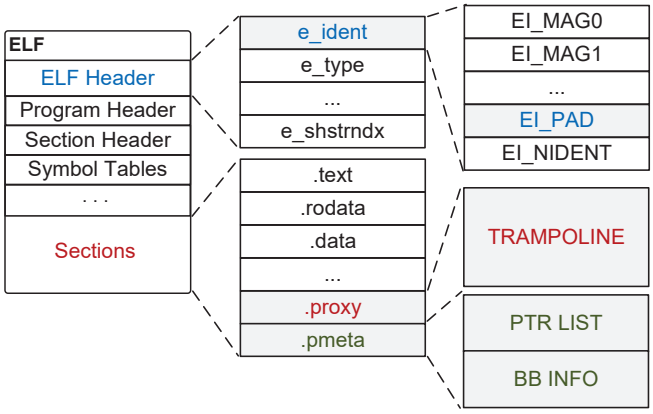

Figure 3. *The new ELF file format.*

attach the collected pointers and basic block boundaries to the end of the ELF file as a separate section (as the *".pmeta"* in Figure 3). This section will be loaded into kernel space to prevent potential disclosure and tampering by attackers. The *.proxy* section in Figure 3 is used to store the generated trampoline. We will provide further details on this part in the subsequent discussion.

4.2 Code Pointer Forwarding

Next, we will forward the cross-page jumps and generate trampolines based on the jump target address and the source jump instruction type. The trampoline is designed to execute a series of instructions that jump through the source instruction and the target instruction. We only forward cross-page jumps because forwarding pointers to the same code page is meaningless, as attackers cannot rely on these pointers to discover other code pages. Moreover, it introduces unnecessary runtime overhead. When generating the trampolines, we refer to the instructions in the program responsible for cross-page jumps as "source instructions," and the final target instruction as "target instructions." The instructions within the trampoline that execute the target cross-page jumps are termed "jump forward instructions," while the instructions within the trampoline that simulate return instructions, taking the control flow back to the next instruction after the source instruction, are referred to as "jump backward instructions."

4.2.1 Pointer Forwarding Rules

For B and BL instructions, we begin by generating the jump forward instruction in the trampoline that jumps to the final target. The jump type is consistent with the original instruction. Since the BL instruction expects the control flow to eventually return from the target instruction, for BL instruction, we generate an additional B instruction to jump back to the next instruction after the source instruction. This emulates the behavior of a return instruction. For conditional branch instructions like B.eq (branch if equal), we set the target to another intermediate B instruction within the trampoline, and then this intermediate instruction, in turn, jumps to the target instruction. Thus, for B.cond instructions, we generate a total of three instructions: the B.cond instruction that is responsible for the conditional jump, the jump backward instruction to return to the next instruction after the source instruction, and the jump forward instruction to navigate to the target instruction. The reason for this approach is that B.cond instructions have limited addressing distance, so we use intermediate B instructions within the trampoline to facilitate the transition. AArch64 is a fixed-length instruction set, where all instructions are only 32 bits in length. The B.cond instruction consists of an 8-bit opcode, 4 bits for condition information, and one fixed bit set to 0. Hence, the B.cond instruction has a 19-bit operand, limiting its addressing range to +/-1MB. In contrast, the B instruction allocates 8 bits to the opcode, leaving 26 bits for the operand, enabling a wider addressing range of +/-128MB. Most programs have code sizes below 100MB, making the use of B instructions an effective means of transitioning between the code and the trampoline. For B.cond instructions within the trampoline, if the jump condition is not met, the jump backward instruction is responsible for returning to the next instruction after the source instruction. If the jump condition is met, the jump forward instruction is responsible for jumping to the target address.

4.2.2 Forwarding Instruction Selection

For different source instructions with the same jump type and target, we generate a single trampoline instruction sequence to minimize memory overhead. After generating the trampoline, we attach it as an execute-only section (referred to as the ".proxy" section in Figure 2) within the ELF file. When the custom loader loads this section, it will load it into program memory with execute-only permission. We will describe our method for enforcing execute-only permissions on the trampoline in the following sections. After generating the trampoline, we need to perform a rewrite of the original instructions. Regardless of the source instruction type, we replace it with a B instruction to redirect to the intermediate instruction sequence in the trampoline. We use B instructions instead of BL instructions because B instructions do not modify the LR register, which is used to store the return address. After these two steps, all the original cross-page jump instructions have been transformed into instructions that first jump to the trampoline and then proceed to the actual target address. When returning from the target address, the process involves first returning from the target address to the trampoline and then returning to the next instruction after the source instruction. This approach effectively prevents attackers from disclosing the return addresses stored in the stack. Once pointer forwarding is completed, we can detect whether a code page harvest is occurring by monitoring any disclosure to the trampoline. Next, we will discuss how to implement code page harvest detection.

4.2.3 Pointer Forwarding Example

Figure 4 illustrates an example of the process of pointer forwarding, and the source code corresponding to the instructions. In the source code, two functions, *foo* and *bar*, are defined. The *foo* calls the *bar* function, and their binary code resides on different code pages after compilation. The *bar* function is exceptionally large, spanning multiple code pages in the binary code. Within the *bar*, there is a switch structure, and one of the cases jumps to a handler that is located on another code page. Finally, the handler performs another cross-page jump, directing to the "next_label" location. The upper part of Figure 4 represents the unprotected code pages generated after compiling the source code (❶ in Figure 4). These three pages are interconnected through BL and B.cond instructions, with the third page further linked to other pages using a B instruction. During a JIT-ROP attack, attackers can locate these three pages individually through BL and B.cond instructions and, via the B instruction

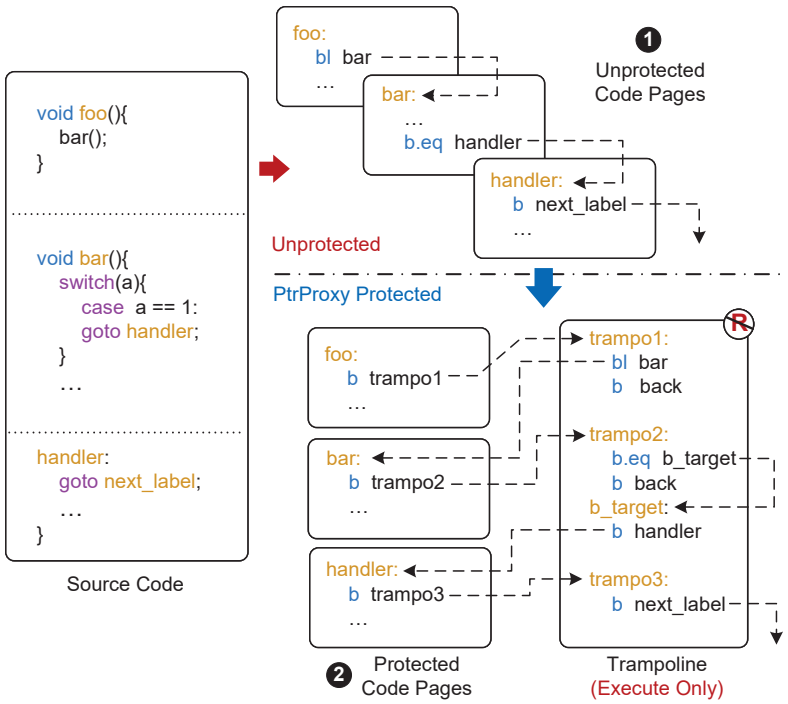

Figure 4. *The concept of PtrProxy pointer forwarding.*

within the third page, disclose other additional pages. The code pages after pointer forwarding are depicted in the lower part of Figure 4 (❷). For the first BL instruction, we have generated a jump forward instruction *bl bar* within the trampoline to achieve the call to the *bar* function. Subsequently, we have generated a B instruction to facilitate the jump back to the next instruction after the source instruction when the *bar* function returns. Finally, we replace the original *bl bar* instruction with *b trampo1*, completing the forwarding process for the first BL instruction. For the B.eq instruction, we generate the corresponding B.eq instruction *B.eq b_target* within the trampoline, followed by the generation of the jump backward instruction *b back* and the jump forward instruction *b handler*. When *a == 1*, which satisfies the jump condition of B.eq, the execution will jump to *b_target* with *B.eq b_target*, then proceed to *b handler*, ultimately leading to the *handler*. When *a != 1* (which means that the jump condition of B.eq is not met), the execution will jump backward to the next instruction after the source instruction, as directed by the *b back* instruction. For the final B instruction, we only need to generate a corresponding B instruction to complete the forwarding, as B instructions do not expect the control flow to return.

4.3 Code Page Harvest Detection

Since we can guarantee that our generated trampoline contains no data, we only need to set it as execute-only to detect the code page harvest. When an attacker is disclosing (reading) the trampoline, our exception handler in the kernel can capture the disclosure behavior and trigger the randomization process. Although Android versions above 10 have discontinued support for XoM, we can still utilize the AArch64 AP/XN feature to enforce execute-only permissions on the trampoline. In the AArch64 architecture, the hardware mechanisms that affect page read, write, and execute permissions are the AP (Access Permission), UXN (Unprivileged eXecute Never), and PXN (Privileged eXecute Never) flags. The AP flags consist of 2 bits and can affect the read/write permissions for user programs running in EL0 and kernel programs running in EL1. UXN and PXN are used to manage the execution permissions for EL0 and EL1 programs, respectively. When UXN is set to 1, EL0 programs do not have execution permission. When PXN is set to 1, EL1 programs do not have execution permission.

All possible combinations of AP, UXN, and PXN, along with their corresponding permissions for EL0 and EL1, are summarized in Table 1. The “R, W, X” in

EL0 and EL1 columns in Table 1 represent the "Read, Write, Execute" permissions, respectively. We need to disallow read and write operations on the trampoline while retaining execution rights, enabling us to detect any leakage attempts by attackers. Hence, for EL0, we require execute (X) permissions, as indicated by the blue entries in the EL0 column of Table 1. When we randomize the code layout, we will randomize the trampoline at the same time to prevent attackers from probing the trampoline layout. So, we need the read and write permissions (RW) for the trampoline in EL1, without the execute permission, as indicated by the red entries in the EL1 column of Table 1. In Table 1, the combination that satisfies both requirements is AP as 00, UXN as 0, and PXN as 1. So, when loading the trampoline into memory, we choose this combination to achieve disclosure detection for the trampoline. When an attacker attempts to disclose the trampoline, the exception handler in the kernel will capture this operation and randomize the code layout.

Table 1. *AP, UXN, PXN combinations and their corresponding permissions at EL0 and EL1.*

AP	UXN	PXN	EL0	EL1
00	0	0	X	RWX
00	0	1	X	RW
00	1	0	None	RWX
00	1	1	None	RW
01	0	0	RWX	RWX
01	0	1	RWX	RW
01	1	0	RW	RWX
01	1	1	RW	RW
10	0	0	X	RX
10	0	1	X	R
10	1	0	None	RX
10	1	1	None	R
11	0	0	RX	RX
11	0	1	RX	R
11	1	0	R	RX
11	1	1	R	R

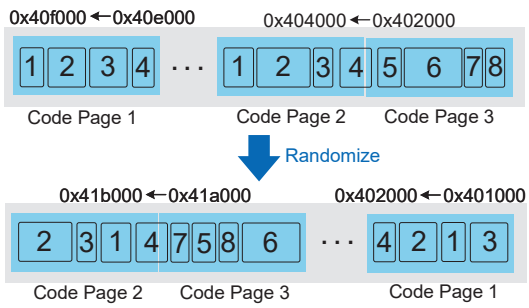

Figure 5. *An example of our randomization policy.*

4.4 Code Layout Randomization

Next, we introduce our randomization strategy. We initially apply "intra-page randomization" to randomize the layout of basic blocks within the same page; then, we proceed to "inter-page randomization," which shuffles the positions of code pages. For cross-page basic blocks, we choose not to randomize them, and consider these two pages as fall through pages, treating them as a single entity during inter-page randomization. This approach can prevent attackers from disclosing the entire program's code by continuously triggering randomization and disclosing the same code page. Additionally, it avoids generating excessive cross-page jumps, thereby preventing unnecessary overhead. At the same time, after each round of code layout randomization, we update the new positions of basic blocks and the list of pointers pointing to these basic blocks.

Figure 5 is an example illustrating the randomization of three code pages. In Figure 5, there are three code pages, and within each code page, the numbered blocks represent basic blocks. Prior to randomization, all basic blocks are arranged in order. The address of code page 1 is 0x40e000∼0x40f000, while code pages 2 and 3 are adjacent, covering addresses 0x402000∼0x404000. Firstly, the order of basic blocks within all pages has been randomized, except for basic block 4, which makes code pages 2 and 3 fall through pages. The lower part of Figure 5 depicts the memory layout after randomization of the three pages. Firstly, the order of basic blocks within all pages has been randomized, except for the basic block 4, which makes code pages 2 and 3 fall through pages. Following that, the order between pages has been shuffled. Code page 1 now spans addresses 0x401000∼0x402000, while code pages 2 and 3 cover addresses 0x41a000∼0x41b000. This completes one round of randomization.

During program runtime, we maintain two code variants to minimize the time required for randomization. Figure 6 illustrates the timeline of the kernel creating code variants and the program triggering randomization. After loading the process, the kernel synchronously starts creating code variants. After creating the code variant, the kernel waits for the process to trigger code randomization. When the process triggers randomization, the kernel switches the currently

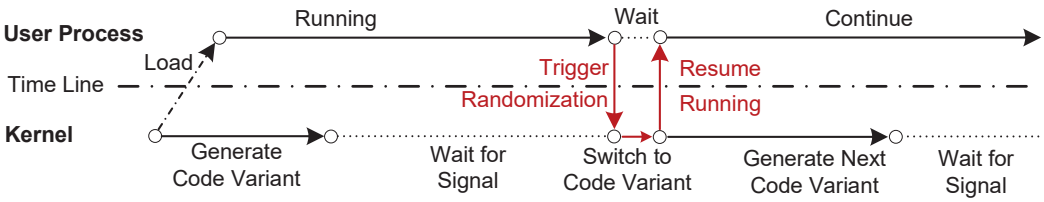

Figure 6. *The timeline of creating code variant and triggering randomization.*

running code to the code variant and resumes program execution. After the switching, the kernel immediately starts synchronously generating the next code variant and waits for the process to trigger the next randomization.

When the randomization is triggered, we can simply flush the page table to switch the currently running code version to the code variant. This is a very short time and imperceptible process for users. When randomization occurs, users do not experience program pauses or sudden slowdowns. After generating each code variant, the pointer list and basic block boundary information are updated on the corresponding variant. After that, we perform a stack unwind, scanning the code pointers on the stack and updating them to their new positions after randomization. Simultaneously, with each randomization, we also shuffle the order of the trampoline to prevent attackers from probing its layout. This strategy minimizes the time needed to trigger the randomization, as the most time-consuming process of code variant generation has already been distributed across program runtime.

4.5 System Kernel Components

The PtrProxy kernel components consist of three main parts: the custom loader, the exception handler, and the Randomizer. Next, we introduce each component in detail.

4.5.1 Custom Loader

We implement a custom loader to load the new ELF format binary (Figure 3). When the custom loader loads the program, it initially checks the PROXY_ENABLED flag in the ELF header. If this flag is set to 0, the system's standard binary loader takes over the binary loading process. If the flag is set to 1, the custom loader proceeds with its loading procedure. Next, the custom loader loads the pointer lists and basic block boundary information stored in the *.pmeta* section into memory. These contents are loaded into kernel space to prevent them from being disclosed or tampered with by attackers from the user level. Then, the custom loader loads the trampolines stored in the *.proxy* section with execute-only permissions into memory, and initializes the exception handler to detect disclosure. Furthermore, the custom loader proceeds to load the rest of the binary, and invokes the Randomizer to perform an initial code layout randomization, ensuring that the code layout varies each time the program is executed. To protect the trampolines from being probed by attackers, the trampoline order is also shuffled during the initial randomization process.

4.5.2 Exception Handler

To detect the disclosure of the trampoline and trigger the randomization process, we implement a page fault exception handler. When an attacker attempts to disclose the trampoline, the MMU triggers a page access permission fault due to the missing read permissions on the trampoline, and this page fault triggers our page fault handler. In the page fault handler, we notify the Randomizer to initiate a randomization process.

4.5.3 Randomizer

The primary tasks of the Randomizer include 1) concurrently generating code variants and 2) flushing the user process page table upon receiving a signal from the exception handler. During program execution, the Randomizer concurrently generates a code variant following the randomization strategy depicted in Figure 5. When generating the code variant, the Randomizer establishes a memory page address mapping table according to the page addresses before and after randomization. Upon receiving a signal from the exception handler, the Randomizer switches the currently running code variant by flushing the page table based

on the memory page mapping table. After the page table flushes, it updates the PC (Program Counter) register to the corresponding location in the new code variant. Thus, the program is able to continue from the instruction where the randomization was triggered. This enables rapid code variant switching for code randomization, ensuring that the user remains unaware of the ongoing code randomization process.

V. PERFORMANCE EVALUATION

We evaluate the performance of PtrProxy from four aspects: 1) performance on microbenchmarks; 2) performance on macrobenchmarks; 3) performance on real-world applications; 4) performance comparison with existing work.

Our evaluations were conducted on an RK3588S 8-core ARMv8 chip with 16GB of RAM, running Ubuntu 20.04 with Linux kernel 5.10. Our evaluation results indicate that PtrProxy incurs marginal additional overhead, with an average of 1.52% on SPEC CPU 2017 and 1.22% on real-world applications. The memory overhead incurred by PtrProxy is also minimal, with an average of only 1.21%. In the following subsections, we will focus on the measurement of runtime slowdown, where we ran both the standard version and the PtrProxy-protected version for each binary.

5.1 Microbenchmarks

In the system kernel, we made changes to the binary loader, page fault exception handler, and process context structure. To evaluate the performance impact of these kernel modifications, we conducted a comparison between the standard Linux kernel and the modified version by running lmbench.

Table 2 displays the running time for kernel operations related to process creation and page fault handling. The two process creation operations, Fork Proc and Exec Proc, use fork and exec to create processes, respectively. These operations resulted in an overhead of 0.55% and 2.21%, respectively, due to the extra steps required for loading the trampoline and pointer list. The Page Fault operation shows the overall overhead for page fault handling, with a 1.18% overhead. The last operation, Prot Fault, displays the protection fault handling overhead. The page protection fault handler needs to check if the trampoline is being disclosed, resulting in a 2.21% overhead.

Table 2. *Time for kernel operations related to process creation and page fault handling (in μs). Smaller is better.*

Kernel	Fork Proc	Exec Proc	Page Fault	Prot Fault
Standard	594	1644	0.931	0.770
PtrProxy	600	1653	0.942	0.787
Overhead	1.01%	0.55%	1.18%	2.21%

We modified the process context structure to store some extra information in the process context, such as the trampoline address and the pointer list. This may cause additional overhead during context switches. Table 3 shows the context switch time for both the standard kernel and the modified kernel. The upper half of the first row shows the number of processes involved in context switches, while the lower half shows the memory usage of each process. For example, "2p/64k" means a context switch between two processes, each using 64 KB of memory. All entries in the table show overhead values that are mostly zero, indicating that PtrProxy does not significantly affect the performance of kernel context switches.

5.2 Macrobenchmarks

We ran both the standard version and the PtrProxy-protected version of SPEC CPU 2017 with the *ref* workload to evaluate the performance impact of PtrProxy on compute-intensive programs. We take the running time of the standard version as the baseline to measure the additional overhead incurred by PtrProxy's protection. Additionally, we evaluated the performance overhead of PtrProxy on SPEC CPU 2006 to compare its performance with previous XoM approaches.

Figure 7 shows the runtime slowdown of PtrProxy on SPEC CPU 2017. The rightmost two stripped bars (green and blue bars) show the average and geometric mean value of overhead, respectively. As we can see from Figure 7, twenty overhead values are very close to zero, and the highest overhead value (8.62%) is observed in povray. Our further investigation revealed that povray has many dispatch loop structures that call multiple small handler functions. According to our pointer forwarding strategy, all these handler

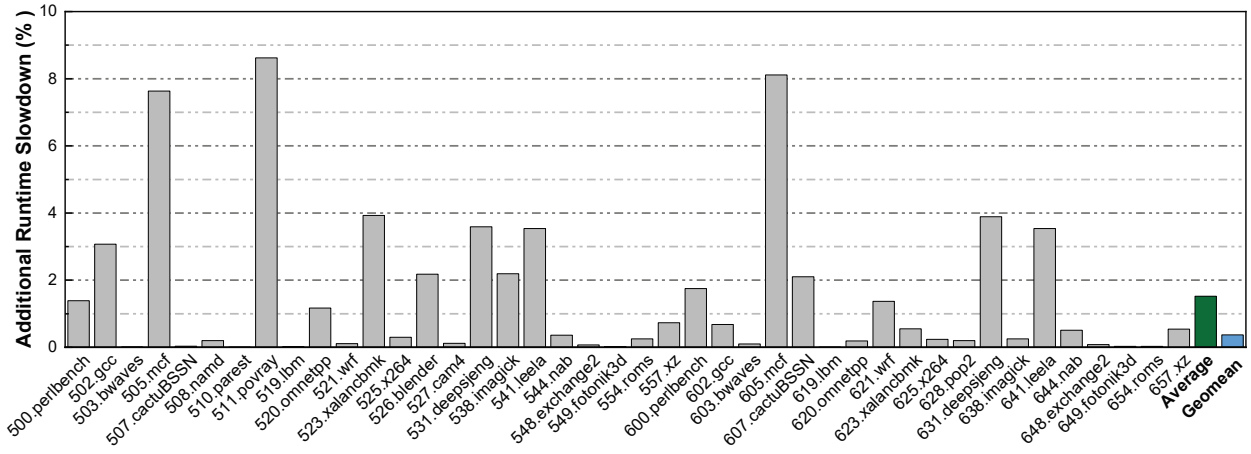

Figure 7. *Additional runtime slowdown (%) of PtrProxy on SPEC CPU 2017 (ref workload).*

Table 3. *Context switch time (in μs). Smaller is better.*

Kernel	2p	2p	2p	8p	8p	16p	16P
	0K	16K	64K	16K	64K	16K	64K
Standard	3.30	3.42	3.72	19.06	20.18	33.44	35.91
PtrProxy	3.36	3.41	3.70	19.33	20.01	33.69	36.03
Overhead	1.82%	-0.29%	-0.54%	1.42%	-0.85%	0.75%	0.33%

function calls are forwarded to the trampoline, resulting in two additional jumps for each handler function call, one for jumping to the actual handler function address and another for jumping back to the callee. Since povray spends most of its running time on executing handler functions, it accumulates a relatively high overhead. 505.mcf and 605.mcf are both derived from MCF, which is utilized for single-depot vehicle scheduling in public mass transportation. Similar to povray, MCF spends most of its running time executing three utility functions iteratively (getArcPosition, arc_compare, and spec_qsort), which results in a relatively high overhead. In general, the average overhead of all SPEC benchmarks is 1.52%, and the geometric mean is 0.37%, indicating a limited performance impact on CPU-intensive programs.

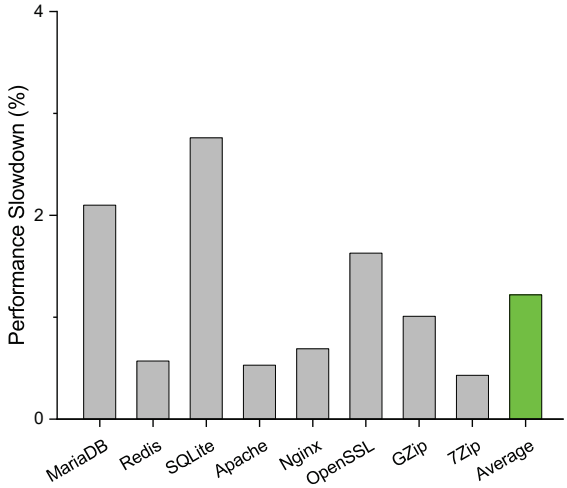

Figure 8. *Runtime overhead on real-world applications.*

5.3 Real-World Applications

We use eight real-world applications to test the performance impact of PtrProxy on real-world applications. We use the Phoronix Test Suite [83] to run these real-world applications and take the results of their standard versions as the baseline. Then, we run the PtrProxy-protected version of these binaries under the same benchmarks in Phoronix, and compare their results with the baseline version, to assess the additional overhead introduced by PtrProxy. Figure 8 illustrates PtrProxy's additional overhead on these real-world applications. The highest overhead was observed in SQLite, reaching 2.76%. The 7Zip has the lowest overhead, at only 0.43%. The rightmost bar in Figure 8 is the average overhead of PtrProxy on these real-world applications, which is only 1.22%. The

experimental results indicate that PtrProxy does not introduce significant overhead on real-world applications.

5.4 Performance Comparison

We also conducted another experiment to compare the performance of PtrProxy with other prominent peer tools, including Shuffler [72], Remix [74], CodeArmor [75], Stabilizer [77], TASR [73], MARDU [79], ReRanz [78], and SafeHidden [81]. We compared their SPEC CPU 2006 performance data reported in their papers with ours. To compare the performance on the same benchmarks, we also evaluated the additional overhead of PtrProxy on SPEC CPU 2006, which is 1.03% on average. Figure 9 shows the performance comparison result. Shuffler has a significantly higher overhead (13.5%) compared to other approaches. CodeArmor, Stabilizer, MARDU, ReRanz, and SafeHidden exhibit similar overheads, which are 6.9%, 6.7%, 5.5%, 5.3%, and 5.8%. Remix and TASR incur relatively low overhead (2.9% and 2.1%, respectively). In contrast, among all approaches in Figure 9, PtrProxy exhibits the lowest overhead (1.03%).

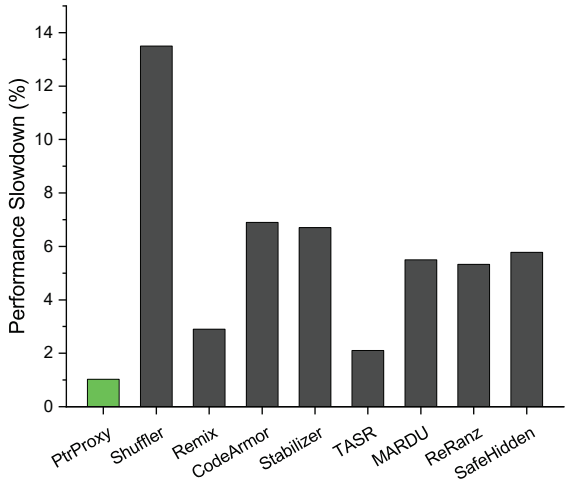

Figure 9. *Performance comparison with previous re-randomization implementations.*

VI. SECURITY EVALUATION

In this section, we conduct a gadget search experiment on the programs we used in our performance evaluation to evaluate how well PtrProxy can defend against memory disclosure attacks. We also use a case study to demonstrate the ability of PtrProxy to defend against real-world attacks.

6.1 Gadget Search Experiment

Under PtrProxy's protection, the window for attackers to conduct an attack lies between two randomizations. If an attacker fails to collect enough gadgets to complete the attack between these two randomizations, the code layout will be disrupted, rendering all collected gadgets ineffective. We designed an experiment to assess how many bytes of code attackers can disclose and how many useful gadgets they can find between two randomization triggers. In this experiment, we randomly select locations in the program's code region and use both breadth-first search (BFS) and depth-first search (DFS) to disclose code until triggering randomization. When randomization is triggered, we employ ROPGadget [89] with a scan depth set to 100 bytes to search for useful gadgets within the disclosed code. For each program, we start the disclosure from 1000 different randomly chosen locations and take the average result as the final outcome.

Table 4 shows the gadget search result. Column 1 contains the programs on which we conducted our experiment. Columns 2 and 3 are the average code bytes attackers can disclose for each program (in bytes) with breadth-first search (BFS) and depth-first search (DFS). In our experiment, most of the attempts failed to find any usable gadget; only some of the attempts can find only **one** usable gadget. The last column shows how many attempts can find **one** usable gadget for each program. From Table 4, we can observe that, on average, using BFS and DFS strategies can disclose 1294 and 1031 bytes of code, respectively. This is proved insufficient to build a ROP chain by the previous ROP gadget search papers [90, 91]. The offline verification technique proposed by Schwartz et al. [90] tries to harvest ROP gadgets with small amounts of code, but it still requires at least 20KB of code to construct a complete payload chain. The "microgadgets" technique [91] attempts to use the gadgets restricted to 2 or 3 bytes in length to construct the ROP chain. However, it needs to scan at least 3MB of code to find enough microgadgets. We found that the disclosing bytes using DFS are less than those using BFS. This is because DFS prioritizes tracking pointers found in code pages more aggressively than BFS, making it

more likely to trigger randomization earlier than BFS. The average number of attempts that can find **one** gadget is 19. This encouraging result indicates that in the majority of disclosure attempts, no useful gadgets can be found. Even if a small number of attempts successfully found one gadget, it is not feasible to complete a JIT-ROP attack with just one single gadget.

Table 4. *Attackers' ability to find potential gadgets when PtrProxy is enabled.*

	Breadth-first Search	Depth-first Search	# of Attempts to Find One Gadget
SPEC CPU 2017	976	871	21
MariaDB	1763	1170	28
Redis	2101	1065	33
SQLite	1399	1297	11
Apache	1107	831	15
Nginx	917	820	24
OpenSSL	1134	1062	20
GZip	1034	1149	5
7Zip	1212	1018	13
Average	1294	1031	19

6.2 Case Study

We use the vulnerable version of Nginx (CVE-2013-2028) [92], which has a stack buffer overflow vulnerability, to evaluate the effectiveness of PtrProxy. This is a fairly powerful stack overflow vulnerability that allows an adversary to perform arbitrary memory reads. We run this vulnerable version of Nginx with PtrProxy protection enabled as a web server. Then, we keep searching for the gadgets in a publicly available exploit for CVE-2013-2028. After 1000 times of disclosing attempts, not a single gadget in the exploit was found. In all attempts, the maximum code disclosure in a single attempt was 2979 bytes. **One** gadget was found out of 103 attempts, but none of these gadgets met the requirements for the CVE-2013-2028 exploit. Our security evaluation suggests that PtrProxy can effectively protect programs from the threat of JIT-ROP attacks.

VII. RELATED WORK & DISCUSSION

7.1 Indirect Code Disclosure

Crane et al. [53] pointed out that in addition to disclosing code directly, there still exists an *indirect memory disclosure attack* that can infer the code layout without directly reading the code pages by harvesting code pointers in stack and heap. To prevent this, they proposed a method that redirects the code pointers to an unreadable trampoline, thus effectively solving the problem of indirect memory disclosure. However, this defense still remains the problem of how to prevent direct code disclosure. Therefore, we focus on addressing the remaining issues in code disclosure prevention.

7.2 Protection Support on Other Platforms

We can port PtrProxy to other platforms like RISC-V and x86-64. It is straightforward to port our work to RISC-V, because RISC-V is also a fixed-length Instruction Set Architecture (ISA). We can forward the jump instructions to the trampoline by simply changing the operand. However, porting to x86-64 presents a different challenge, because x86-64 is a variable-length ISA. When forwarding the jumping instructions, the original instruction length may not be enough for jumping to the target trampoline address. There are already extensive discussions regarding this challenge [93, 94, 88], providing us with an opportunity to potentially port PtrProxy to the x86-64 platform. Porting PtrProxy to other platforms and addressing these related challenges will be our future work. When porting to x86-64, we can utilize Intel MPK hardware mechanism [95] to detect trampoline disclosure. On RISC-V, there is no direct way to detect trampoline detection. However, we can still draw on intra-process isolation ideas (e.g., Donky [96]) to implement trampoline disclosure detection.

7.3 Application Scenarios of PtrProxy

As a code re-randomization approach, PtrProxy is designed to protect different types of devices from JIT-ROP attacks while incurring marginal runtime overhead. There are three main application scenarios for PtrProxy. One application scenario is the server side applications. Due to the need for frequent interaction with users, highly interactive server applications (such as web servers like Nginx, Apache, and DBMS like MySQL) are susceptible to JIT-ROP attacks. PtrProxy can be deployed on servers to protect these server side applications. Another application scenario is the protection of routers. On routers, there are different implementations of various network protocols at different layers for packet forwarding. Although these net-

work protocol implementations do not interact directly with users, attackers may construct malicious packets to trigger and exploit vulnerabilities, making these network protocol implementations vulnerable to JIT-ROP attacks as well. PtrProxy can prevent the exploitation of these vulnerabilities even when the vulnerabilities are triggered. The last scenario is the protectiono of IoT devices, because IoT devices often need to implement specific protocols for interconnectivity, such as Bluetooth Low Energy (BLE) and ZigBee. The implementation of these protocols often inevitably introduces vulnerabilities. PtrProxy can still protect IoT devices from JIT-ROP attacks even when these vulnerabilities exist. Additionally, PtrProxy introduces only marginal overhead, making it particularly suitable for IoT devices that are sensitive to power consumption and performance costs.

VIII. CONCLUSION

Advanced code-reuse attacks like JIT-ROP use memory disclosure vulnerabilities to bypass the load-time code randomization protection. Re-randomization is a retrofit solution to counter this type of attack. However, the previous re-randomization approaches have issues in terms of both performance and security. This paper introduces PtrProxy, a technique that implements re-randomization on the AArch64 platform. Unlike previous work, PtrProxy triggers randomization by detecting traversing code pages, which reduces performance overhead and enhances security. The security and performance evaluations show that PtrProxy is feasible for real-world adoption. The potential of PtrProxy to shift the balance of the memory war in favor of defenders is high.

ACKNOWLEDGEMENT

This work is supported in part by the National Natural Science Foundation of China (62272351, 61972297, 62172308).

References

[1] L. Szekeres, M. Payer, *et al.*, "Sok: Eternal war in memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[2] V. Kuznetzov, L. Szekeres, *et al.*, "Code-pointer integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[3] C. DeLozier, L. Kavya, *et al.*, "Hurdle: Securing jump instructions against code reuse attacks," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[4] V. Duta, C. Giuffrida, *et al.*, "Pibe: Practical kernel control-flow hardening with profile-guided indirect branch elimination," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[5] H. Cho, J. Park, *et al.*, "Vik: Practical mitigation of temporal memory safety violations through object id inspection," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[6] H. Zhang, M. Ren, *et al.*, "One size does not fit all: Security hardening of mips embedded systems via static binary debloating for shared libraries," in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[7] X. Yang, G. Peng, *et al.*, "Powerdetector: Malicious powershell script family classification based on multi-modal semantic fusion and deep learning," *China Communications*, 2023, vol. 20, no. 11, pp. 202–224.

[8] Y. Zhou, G. Peng, *et al.*, "A survey on the evolution of bootkits attack and defense techniques," *China Communications*, 2024, vol. 21, no. 1, pp. 102–130.

[9] S. Zhang, Y. Guo, *et al.*, "Atssc: An attack tolerant system in serverless computing," *China Communications*, 2024, vol. 21, no. 6, pp. 192–205.

[10] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[11] M. Abadi, M. Budiu, *et al.*, "A theory of secure control flow," in *International Conference on Formal Engineering Methods*, 2005, pp. 111–124.

[12] M. Abadi, M. Budiu, *et al.*, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2009, vol. 13, no. 1, pp. 1–40.

[13] J. Corbet, "A new llvm cfi implementation," https://lwn.net/Articles/898040/, [online].

[14] Y. Cheng, Z. Zhou, *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[15] A. Mashtizadeh, A. Bittau, *et al.*, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015, pp. 941–951.

[16] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," *Technical Report*, 2012.

[17] V. Pappas, P. Michalis, *et al.*, "Transparent rop exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*,

2013, pp. 447–462.
[18] C. Zhang, T. Wei, *et al.*, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P '13)*, 2013, pp. 559–573.
[19] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, 2013, pp. 337–352.
[20] N. Burow, S. Carr, *et al.*, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, 2017, vol. 50, no. 1, pp. 1–33.
[21] J. Xu, S. Patel, *et al.*, "Architecture support for defending against buffer overflow attacks," *Proceedings of the Second Workshop Evaluating and Architecting System Dependability*, 2002.
[22] N. Burow, X. Zhang, *et al.*, "Sok: Shining light on shadow stacks," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P '19)*, 2019, pp. 985–999.
[23] L. Davi, A. Sadeghi, *et al.*, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (AsiaCCS '11)*, 2011, pp. 40–51.
[24] T. Chiueh and F. Hsu, "Rad: A compile-time solution to buffer overflow attacks," in *Proceedings 21st International Conference on Distributed Computing Systems*, 2001, pp. 409–417.
[25] Intel, "Complex shadow-stack updates (intel® control-flow enforcement technology)," http://tiny.cc/91ogvz, [online].
[26] T. Dang, P. Maniatis, *et al.*, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (CCS '15)*, 2015, pp. 555–566.
[27] J. Xu, L. Di Bartolomeo, *et al.*, "Warpattack: Bypassing cfi through compiler-introduced double-fetches," in *Proceedings of the 2023 IEEE Symposium on Security and Privacy (S&P '23)*, 2023, pp. 1271–1288.
[28] F. Schuster, T. Tendyck, *et al.*, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P '15)*, 2015, pp. 745–762.
[29] N. Carlini, A. Barresi, *et al.*, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, 2015, pp. 161–176.
[30] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security '14)*, 2014, pp. 385–399.
[31] L. Davi, A. Sadeghi, *et al.*, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, 2014, pp. 401–416.
[32] E. Göktas, A. Elias, *et al.*, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P '14)*, 2014, pp. 575–589.
[33] E. Göktaş, A. Elias, *et al.*, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, 2014, pp. 417–432.
[34] F. Schuster, T. Tendyck, *et al.*, "Evaluating the effectiveness of current anti-rop defenses," in *International Workshop on Recent Advances in Intrusion Detection*, 2014, pp. 88–108.
[35] M. Malvica, "Bypassing intel cet with counterfeit objects," https://www.offsec.com/offsec/bypassing-intel-cet-with-counterfeit-objects/, [online].
[36] Marcoramilli, "From rop to lop bypassing control flow enforcement," https://shorturl.at/jDJM1, [online].
[37] P. Team, "Address space layout randomization (aslr)," https://pax.grsecurity.net/docs/aslr.txt, 2003.
[38] S. Bhatkar, D. DuVarney, *et al.*, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th Conference on USENIX Security Symposium (USENIX Security'03)*, 2003.
[39] S. Bhatkar, D. DuVarney, *et al.*, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th Conference on USENIX Security Symposium (USENIX Security'05)*, 2005.
[40] C. Kil, J. Jun, *et al.*, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006.
[41] C. Giuffrida, K. Anton, *et al.*, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st Conference on USENIX Security Symposium (USENIX Security'12)*, 2012.
[42] J. Hiser, N. Anh, *et al.*, "Ilr: Where'd my gadgets go?" in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.
[43] V. Pappas, M. Polychronakis, *et al.*, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.
[44] L. Davi, A. Dmitrienko, *et al.*, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th Symposium on Information, Computer and Communications Security (Asia CCS'13)*, 2013.
[45] A. Gupta, J. Habibi, *et al.*, "Marlin: Mitigating code reuse attacks using code randomization," *IEEE Transactions on Dependable and Secure Computing*, 2015, vol. 12, no. 3, pp. 326–337.
[46] J. Fu, Y. Lin, *et al.*, "Code reuse attack mitigation based on function randomization without symbol table," in *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 394–401.
[47] H. Koo, Y. Chen, *et al.*, "Compiler-assisted code randomization," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*, 2018.
[48] A. Homescu, S. Neisius, *et al.*, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*, 2013.
[49] R. Wartell, V. Mohan, *et al.*, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, 2012.
[50] R. Strackx, Y. Younan, *et al.*, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, 2009, pp. 1–8.

[51] K. Snow, F. Monrose, *et al.*, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[52] M. Backes, T. Holz, *et al.*, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, 2014.

[53] S. Crane, C. Liebchen, *et al.*, "Readactor: Practical code randomization resilient to memory disclosure," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015, pp. 763–780.

[54] Y. Chen, D. Zhang, *et al.*, "Norax: Enabling execute-only memory for cots binaries on aarch64," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

[55] S. Crane, S. Volckaert, *et al.*, "It's a trap: Table randomization and protection against function-reuse attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, 2015.

[56] J. Gionta, W. Enck, *et al.*, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY'15)*, 2015.

[57] K. Braden, S. Crane, *et al.*, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[58] D. Kwon, J. Shin, *et al.*, "uxom: Efficient execute-only memory on arm cortex-m," in *Proceedings of the 28th Conference on USENIX Security Symposium (USENIX Security'19)*, 2019.

[59] Z. Shen, K. Dharsee, *et al.*, "Fast execute-only memory for embedded systems," in *Proceedings of the 2020 IEEE Secure Development (SecDev '20)*, 2020.

[60] M. Zhang, M. Polychronakis, *et al.*, "Protecting cots binaries from disclosure-guided code reuse attacks," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*, 2017.

[61] M. Pomonis, T. Petsios, *et al.*, "krˆx: Comprehensive kernel protection against just-in-time code reuse," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*, 2017.

[62] J. Gionta, W. Enck, *et al.*, "Preventing kernel code-reuse attacks through disclosure resistant code diversification," in *Proceedings of 2016 IEEE Conference on Communications and Network Security (CNS'16)*, 2016.

[63] S. Gravani, M. Hedayati, *et al.*, "Fast intra-kernel isolation and security with iskios," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21)*, 2021.

[64] X. Meng and B. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016.

[65] C. Pang, R. Yu, *et al.*, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*, 2021.

[66] A. Tang, S. Simha, *et al.*, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, 2015.

[67] R. Wartell, Y. Zhou, *et al.*, "Differentiating code from data in x86 binaries," in *Machine Learning and Knowledge Discovery in Databases*, 2011.

[68] J. Werner, G. Baltas, *et al.*, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS'16)*, 2016.

[69] C. Wilson and L. Arriaga, "Secure commodity software by enforcing memory permission policies," https://www.academia.edu/download/50294358/Transparent_Enforcement_for_Software_Memory_Security.pdf, [online].

[70] J. Pewny, P. Koppe, *et al.*, "Breaking and fixing destructive code read defenses," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*, 2017.

[71] K. Snow, R. Rogowski, *et al.*, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016.

[72] W. David, G. Gobieski, *et al.*, "Shuffler: Fast and deployable continuous code re-randomization," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[73] D. Bigelow, T. Hobson, *et al.*, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'12)*, 2012.

[74] Y. Chen, Z. Wang, *et al.*, "Remix: On-demand live randomization," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*, 2016.

[75] X. Chen, H. Bos, *et al.*, "Codearmor: Virtualizing the code space to counter disclosure attacks," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (Euro S&P'17)*, 2017.

[76] K. Lu, W. Lee, *et al.*, "How to make aslr win the clone wars: Runtime re-randomization," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[77] C. Curtsinger and E. Berger, "Stabilizer: Statistically sound performance evaluation," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[78] Z. Wang, C. Wu, *et al.*, "Reranz: A light-weight virtual machine to mitigate memory disclosure attacks," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'17)*, 2017.

[79] C. Jelesnianski, J. Yom, *et al.*, "Mardu: Efficient and scalable code re-randomization," in *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR'20)*, 2020.

[80] J. Shi, L. Guan, *et al.*, "Harm: Hardware-assisted continuous re-randomization for microcontrollers," in *Proceedings of the 7th European Symposium on Security and Privacy (EuroS&P'22)*, 2022.

[81] Z. Wang, C. Wu, *et al.*, "Safehidden: An efficient and se-

cure information hiding technique using re-randomization," in *Proceedings of the 28th USENIX Security Symposium (USENIX'19)*, 2019.
[82] S. Ahmed, Y. Xiao, *et al.*, "Methodologies for quantifying (re-)randomization security and timing under jit-rop," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20)*, 2020.
[83] P. Suite, "Linux testing & benchmarking platform, automated testing, open-source benchmarking," https://www.phoronix-test-suite.com/, [online].
[84] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security'14)*, 2014.
[85] L. Davi, C. Liebchen, *et al.*, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.
[86] M. Xie, Y. Lin, *et al.*, "Pointerscope: Understanding pointer patching for code randomization," *IEEE Transactions on Dependable and Secure Computing*, 2022.
[87] L. Szekeres, M. Payer, *et al.*, "Sok: Eternal war in memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13)*, 2013.
[88] W. David, H. Kobayashi, *et al.*, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.
[89] J. Salwan, "Ropgadget," https://github.com/JonathanSalwan/ROPgadget, [online].
[90] E. Schwartz, T. Avgerinos, *et al.*, "Q: Exploit hardening made easy," in *Proceedings of the 20th USENIX Conference on Security Symposium (USENIX Security'11)*, 2011.
[91] A. Homescu, M. Stewart, *et al.*, "Microgadgets: Size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT '12)*, 2012.
[92] T. M. Corporation, "Cve-2013-2028 detail," https://www.cve.org/CVERecord?id=CVE-2013-2028, [online].
[93] G. Duck, X. Gao, *et al.*, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, 2020.
[94] Z. Zhang, W. You, *et al.*, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *Proceedings of the 2021 IEEE Symposium on Security and Privacy (S&P '21)*, 2021.
[95] S. Park, S. Lee, *et al.*, "libmpk: Software abstraction for intel memory protection keys (intel mpk)," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, 2019.
[96] D. Schrammel, S. Weiser, *et al.*, "Donky: Domain keys – efficient in-process isolation for risc-v and x86," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21)*, 2021.

Biographies

Luo Chenke received his B.S. degree from Northeastern University in 2018 and earned his Ph.D degree at Wuhan University in 2024. He is currently a Postdoctoral Fellow at Tulane University. His research focuses on system security and software security.

Fu Jianming received his Ph.D degree from Wuhan University, Wuhan, China, in 2000. He is currently a professor at the School of Cyber Science and Engineering, Wuhan University. His research interests include system security, software security, AI security, and mobile security.

Ming Jiang is an Assistant Professor in the Department of Computer Science at Tulane University. He received his Ph.D from The Pennsylvania State University. His research interests span Software and Systems Security. He strives to ground his efforts in practical security problems with an eye towards developing effective solutions to address realistic threats caused by today's emerging technologies.

Xie Mengfei is currently pursuing his Ph.D in the School of Cyber Science and Engineering at Wuhan University under the supervision of Dr. Jianming Fu. His current research focuses on system security and software security.

Peng Guojun received his Ph.D degree from Wuhan University, Wuhan, China, in 2008. He is currently a professor at the School of Cyber Science and Engineering, Wuhan University. His research interests include malware analysis and defense, software security, mobile security, and trusted computing.