Aalto University
School of Science

Master's Programme in Computer, Communication and Information Sciences

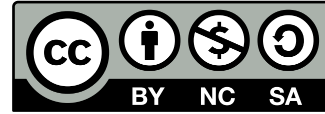
Preshuf: Pre-Shuffling Binaries in Secure Hardware

Armand-Alexandru Balint

Master's Thesis
2025

© 2025 Armand-Alexandru Balint

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 4.0 International" license.



Aalto University
School of Science

Author Armand-Alexandru Balint

Title Preshuf: Pre-Shuffling Binaries in Secure Hardware

Degree programme Computer, Communication and Information Sciences

Major Computer Science

Supervisor Dr. Lachlan Gunn

Advisor Arto Niemi (MSc)

Collaborative partner Huawei Technologies

Date 24 November 2025 **Number of pages** 93 **Language** English

Abstract

Fine-grained randomization of code is an effective countermeasure to code-reuse attacks, but its practical deployment is hindered by the high performance overhead of the randomization itself. This thesis presents Preshuf, an architecture that is trying to mitigate this performance-security trade-off by separating the computationally costly randomization from the startup sequence of the application. Preshuf implements an asynchronous model where a background daemon makes use of a hardware-isolated Trusted Execution Environment on ARM64 to continuously pre-shuffle binaries at a function-granular level and encrypt them using AEAD cryptography. At runtime, a lightweight preloader only manages a fast and secure decryption before executing the program, therefore making the shift of overhead from the permutation logic to a much more lightweight set of cryptographic operations. The system was implemented and evaluated on both emulated and physical platforms, through QEMU and a Raspberry Pi 3. The results demonstrate that this approach introduces a minimal load-time latency that can be less than a fourth of the required overhead for the randomization itself for larger binaries on representative hardware. Additionally, the ongoing re-randomization produces a “refreshing defense” that makes leaked runtime information ephemeral, thereby forcing the attacker into Just-In-Time exploit development within a set time window. The work shows that the asynchronous pre-shuffling model does make high-entropy moving target defenses more practical without the need of compromising security for performance.

Keywords ASLR , OS Hardening , Memory protection , TEE , Cryptography

Preface

I would like to begin by expressing my sincere gratitude to everyone who has supported and believed in me throughout this journey, especially considering the choice of pursuing this thesis during my first year of studies.

That being said, the very first bit of gratitude goes to Huawei Technologies, particularly, my advisor, Arto Niemi, and Sampo Sovio. Despite my initial lack of expertise in ASLR and with the ARM architecture, they still decided to give me a chance, selected me for the topic, and provided great guidance throughout.

Moreover, I would like to extend a further thank you to my supervisor, Lachlan J. Gunn, who accepted to be my supervisor without much concern for what is to follow, provided clear advice and detailed comments on my work, and was always able to provide advice, even at the most "unholy" hours.

Lastly, I extend my profound thanks to my parents, my dedicated colleagues, and my friends, Tori, Dalia, Bahsoun, Bernardo, and Parsa, for their unwavering support and affirmation throughout this endeavor. I am truly grateful for everything!

Otaniemi, 24 November 2025 Armand-Alexandru Balint

Contents

Abstract 3

Preface 4

Contents 5

Activities & Terminology 7

1 Introduction 8
- 1.1 Thesis objective . . . 9
- 1.2 Research questions . . . 9
- 1.3 Contributions . . . 10

2 Background 11
- 2.1 Memory safety and memory vulnerabilities: history & landscape . . 11
- 2.2 Code Randomization . . . 14
  - 2.2.1 Base address randomization solutions . . . 14
  - 2.2.2 The need for fine-grained randomization . . . 16
  - 2.2.3 Leakage-resistant randomization . . . 18
- 2.3 Process . . . 19
- 2.4 Isolation . . . 21

3 Problem Statement 24
- 3.1 Threat Model . . . 24
- 3.2 Requirements . . . 25
- 3.3 Threat of Runtime Interception . . . 26

4 Design 27
- 4.1 OP-TEE . . . 27
- 4.2 Randomization and Cryptography . . . 30
  - 4.2.1 Performing the randomization . . . 30
  - 4.2.2 Using the Trusted Execution Environment . . . 32
  - 4.2.3 File encryption . . . 32
- 4.3 Bottlenecks and implications . . . 34

5 Implementation 36
- 5.1 Implementation of the Code Shuffling Tool in the TA . . . 37
- 5.2 Implementation of the Encryption Tool in the TA . . . 38
- 5.3 Preloader-based Startup Interception . . . 40
- 5.4 Architectural Synthesis and Operational Flow . . . 41
- 5.5 Necessary OP-TEE modifications . . . 42

6 Analysis of results 44
6.1 Testing Criteria . . . 44
6.2 Performance Evaluation . . . 44
6.3 Performance measurement . . . 47
6.4 Security Evaluation . . . 47
6.5 Analysis of Randomization Effectiveness and Attack Complexity . . . 49
6.6 Formal Adversarial Models and Notation . . . 53

7 Related Work 57
7.1 Benchmarks of related approaches . . . 62

8 Discussion 65
8.1 Limitations of the Preshuf architecture . . . 65
8.2 Enhancing system integrity through attestation . . . 65
8.3 Architectural tradeoffs in decryption timings . . . 66
8.4 Mitigating metadata storage overhead . . . 67
8.5 Final Remarks and the Broader Impact of Preshuf . . . 68

9 Conclusions 70
9.1 Future Work . . . 71
9.1.1 OS-Level Monitoring through the Linux Audit Framework . . . 71
9.1.2 Hardware-Enforced Monitoring through Hypervisor-Based Introspection . . . 72

A Benchmarked code for time data 88

B Benchmarked code for entropy graphs 89
B.1 Code used for entropy benchmarking . . . 89
B.2 Code used to extract the function addresses . . . 90

C System Binaries and PRR Injection Overhead 91

Activities & Terminology

Code randomization	making code pointer values unpredictable
Relocation	fixing up code and data pointers
Asynchronous Re-randomization	background process of continuously shuffling binaries
Startup Interception	intercepting program launch to perform a preparatory action
At-Rest Protection	securing a binary on disk via encryption and integrity checks
Process Load	mapping code and data from binary file to memory

Terminology

AEAD	Authenticated Encryption with Associated Data
ASLR	Address Space Layout Randomization
CA	Client Application (Normal World component)
Daemon	a background process that runs without direct user control
ELF	Executable and Linkable Format
Fine-grained randomization	function-granular or smaller randomization
Gadget	existing instruction sequence ending in a return
Preloader	a program that executes before the main application
ROP	Return-Oriented Programming
RU	Randomization Unit
Secure Storage	storage protected by a TEE/TPM
TA	Trusted Application (Secure World component)
TEE	Trusted Execution Environment
W⊕X	exclusive capability of writing or executing

1 Introduction

Securing software systems constitutes a persistent challenge, primarily because the attacker holds a distinct advantage over the defender: An attacker can accomplish their goal by finding just a few flaws in a system of millions of lines of code, while the defender must achieve near-perfection. This lopsided challenge is further worsened by fundamental characteristics of software engineering. More precisely, a compiler deterministically transforms source code into a single executable artifact, which is then distributed, byte-for-byte, to a vast user base through package managers and application stores. This practice creates a uniform attack surface where the internal structure of the program is identical for every user, which permits a single vulnerability to be exploited universally.

A simple programming mistake, such as accessing a dangling pointer through a use-after-free bug or corrupting adjacent data structures via an off-by-one error, can be all an attacker needs to gain an exploitable primitive [116] and achieve a full system compromise [10]. This is because such errors corrupt the memory state in predictable ways, allowing an attacker to overwrite adjacent data structures, such as a vtable pointer or saved frame pointer on the stack. The history of cybersecurity is replete with such cases [150] and examples where these seemingly small flaws led to catastrophic financial and operational damage. For instance in 2003 with the "Slammer" worm, which exploited a buffer overflow in Microsoft's SQL Server to execute arbitrary code, causing about $1.2 billion in damages by shutting down networks worldwide [65]. Similarly, the 2014 "Heartbleed" vulnerability, an out-of-bounds read in the OpenSSL library, did not grant code execution but allowed attackers to exfiltrate private cryptographic keys and other sensitive data from millions of servers [51]. The volume and complexity of code written in C and C++ makes manually locating and patching every such flaw an economically and logistically intractable task [117]. The problem is compounded by the fact that even with ideal application code, vulnerabilities in the application's dependencies or libraries can be inherited. Reactive defenses like antivirus software often struggle against more modern threats and attack vectors [70], and so, while other proactive mitigations exist, they present their own set of limitations. Stack canaries, for instance, are a common defense but are ineffective against heap-based corruption or attacks that do not directly overwrite the stack frame [39]. Stronger defenses, such as comprehensive Control-Flow Integrity or software-based memory safety solutions like bounds checking, can offer stronger protection, but tend to introduce significant performance overheads, making them be considered unacceptable for many production environments. This necessitates a different approach that can stop these vulnerabilities at their core without degrading the user experience.

The goal of code randomization is to directly counter the primary advantage of a memory-focused attacker [128], namely their foreknowledge of the memory layout of a target process. The core principle is to introduce artificial diversity into a program's structure, making it a unique, unpredictable instance of itself. In ideal circumstances, this transformation guarantees that the addresses of functions and data are no longer at the predictable locations. An exploit written for a standard, unmodified binary will

fail due to its hardcoded nature and so, the offsets will point to data or code unrelated to the exploit's logic or will trigger a crash. The operating system and processor do not require code to be at a specific location to function correctly [76]; they only require that all internal pointers and references are consistent [71]. Consequently, on operating systems like Linux, an application's executable file on disk can be replaced or deleted after it has been loaded. The running process remains unaffected because its code and data segments have already been mapped into independent memory pages, and the process will not reference the on-disk file again during its execution [23]. This flexibility is what randomization schemes exploit: as long as the program's logic remains intact, its internal arrangement can be drastically altered without any functional impact, thereby invalidating an attacker's reconnaissance.

1.1 Thesis objective

The viability of a code randomization scheme is based on both its security and its performance. An implementation done purely through software is not only vulnerable to inspection by a compromised OS [33], but can also be slowed by the overhead of kernel interactions. Moreover, a defense mechanism that cannot protect itself is fundamentally flawed. In a Normal World context, securing the randomization process would need constant mediation by the OS kernel to isolate its memory and control its execution, a process that is inherently slow due to frequent context switches and system call overhead. This work proposes a system that performs fine-grained code randomization within a hardware-isolated TEE. This approach aims to provide a blend of strong root of trust and ideal performances advantages, as the entire permutation and relocation process can execute in a single, protected context without costly interruptions or transitions. The upcoming proposed model breaks the dependency on the main OS for protection, thereby removing the associated performance cost.

1.2 Research questions

On top of the enhanced security, code randomization techniques provide a benefit that is largely invisible to the user, and this is a significant advantage over defenses that require user interaction or cause functional changes. However, this invisibility can be compromised by performance degradation. Every computational cycle spent on shuffling functions or relocating pointers is a cycle not spent on the application's main task, and thus some suboptimal designs of randomization schemes can lead to significant load-time delays and runtime slowdown, therefore negatively impacting the user experience. One of the main challenges, therefore, is of efficiency, especially as more fine grained variants of code randomization require a lot of processing and computation, thus introducing great overhead. Moreover, additional challenges arise when considering secrecy, predictability and integrity. All of the above, therefore, motivate the following research questions:

1. **RQ1:** To what extent can an asynchronous, refreshing defense model mitigate the performance overhead of fine-grained code randomization at application

launch time?

2. **RQ2:** Can fine-grained code randomization be done as a transparent, automated background process completely invisible to the user?

3. **RQ3:** Is the TEE capable enough to execute our randomization logic and guarantee confidentiality and integrity for the processes?

4. **RQ4:** What is the combined impact on security and performance when integrating AEAD encryption to protect randomized binaries at rest against offline analysis?

1.3 Contributions

The implementation of our model, Preshuf, is the result of a collaborative effort with Huawei's HSSL team, who provided a foundational randomization framework that included the core shuffling algorithm and an initial implementation of the ELF partitioning and relocation logic. Building upon this, the primary contributions of this thesis are:

1. The design and implementation of a complete, TEE-based security architecture that moves the entire randomization and encryption process into a hardware-isolated environment, protecting it from a compromised host.

2. The creation of a transparent, refreshing defense model consisting of an asynchronous daemon for continuous re-randomization and a preloader for on-demand, secure decryption and execution.

3. The system-wide integration of these components, including the necessary modifications to the build system and porting to a physical hardware platform (Raspberry Pi 3) for a realistic evaluation.

4. A performance and security analysis of the final system that aims to validate its low overhead, and effectiveness against the defined threat model.

2 Background

The programming languages C and C++ have long been foundational in software development, particularly for systems and performance-demanding applications due to their efficiency and precise control over memory allocation. Despite their prevalence and capabilities, these languages, by design, do not enforce memory safety. They do not automatically prevent common programming errors related to memory access, such as writing beyond their allocated buffer boundaries or accessing memory after it has been deallocated. The absence of built-in safeguards against these memory errors leaves systems vulnerable to malicious exploitation [142].

In addition, recent studies [117] have shown that around 70% of vulnerabilities come from memory safety, and this has given rise to a progression of possible attacks. Theorized since the 1970s [132], initial exploits commonly took the form of buffer overflows, where an attacker-controlled input would exceed its designated buffer's capacity, overwriting adjacent stack frames [96]. This could corrupt control data, most notably the function return address, to divert execution flow to injected shellcode. The adoption of memory protection mechanisms like W⊕X, which prevents the execution of data pages, rendered simple code injection ineffective. In response, attackers developed code-reuse attacks, beginning with Return-to-Libc in 1997. This method repurposes entire functions from linked libraries, such as `system()`, by overwriting a return address to point to the function's entry point and arranging for its arguments to be on the stack [148]. Return-Oriented Programming (or ROP for short) generalizes this concept, chaining together short, pre-existing instruction sequences (otherwise called gadgets, in the context of ROP), each terminating in a `ret` instruction. By carefully constructing a chain of gadget addresses on the stack, an attacker can cause significant damage to the system without injecting any new code [127].

Fixing such vulnerabilities from pre-compiled dependencies could theoretically be done through a complete source code audit followed by the patching of vulnerable code. Such an undertaking, however, is insurmountable and logistically infeasible across the plethora of already-compiled software [31]. As such, in an attempt to prevent the exploitation of these vulnerabilities, multiple lines of defense were developed. Mitigations like stack canaries, shadow stacks, and comprehensive bounds-checking were introduced to enforce data integrity and memory safety. Concurrently, Control-Flow Integrity schemes were created to constrain indirect branches to legitimate targets. In parallel with these efforts, code randomization methods appeared as a probabilistic defense. This section will examine these methods, their evolution, and how they can remediate these threats.

2.1 Memory safety and memory vulnerabilities: history & landscape

System programming often relies on low-level languages, most of which does not provide inherent memory safety. As a result, oversights during the development process, such as inadequate memory management or accessing deallocated memory,

can render a program's memory vulnerable to attacks. The lack of in-built memory boundaries considerably contributes to these security flaws. [142, 165, 7]

Although not formally documented widely at the time, informal accounts suggest that programmers familiar with the `IBM OS/390` environment have been aware of these vulnerabilities since the 1960s [133], whereas the first mention of a buffer overflow attack, together with still relevant memory safety proposals, was in a 1972 report by J. Anderson [37]. Following that, one of the first cases of memory exploitation dates back to the late 1980s, with the Morris Worm being one of the first instances of this [53]. The worm exploited several vulnerabilities in the targeted systems, including a buffer overflow vulnerability within the *finger* daemon. The center was tasked with coordinating the technical response to these major security incidents; the urgency being underscored by the data that by the early 2000s, more than half of the advisories published by CERT were dedicated to vulnerabilities related to memory corruption [164].

Later guides, such as "Smashing the stack for fun and profit" [95], proceeded to attract additional attention to stack-overflow attack methodologies. These attacks typically involved injection of malicious code into the memory of a process and redirecting execution to that code. As a way of counteracting this, defenses were introduced to prevent memory pages from being simultaneously writable and executable (W⊕X) [50]. This started with efforts like kernel modifications for non-executable stacks [103] and progressed to system-wide protections [121, 122]

The introduction of W⊕X forced a change in the means attackers use to achieve their goals. Instead of code injection, attackers began to repurpose instruction sequences already present in the target program that would eventually yield their desired outcome. `Return-to-usr` [38] and `return-to-libc` attacks were an early example of this, diverting execution to library functions to execute commands [123, 125]. This evolved into the now-known Return Oriented Programming, where attackers chain short sequences of existing instructions ending in a return-to-construct arbitrary computations without injecting new code, thus bypassing W⊕X. [130]. To showcase such a vulnerability, below we can see a snippet of vulnerable code and a sample ROP attack.

```
1 void vulnerable_function() {
2     char b[128];
3     memset(b, 0, sizeof(b));
4     printf("Enter your
         payload:\n");
5     fflush(stdout);
6     gets(b);
7 }
```

```
8  void gadget() {
9      __asm__("pop %rdi; ret");
10 }
11 int main(int argc, char **
       argv) {
12     vulnerable_function();
13     return 0;
14 }
```

Figure 1: Code vulnerable to buffer overflows, Ret2libc, and ROP

Here, the attacker's goal is to execute `system("/bin/sh")` to gain a shell. Using ROP, they can achieve this without injecting any code. Instead, they can craft a payload

consisting of a chain of addresses. This chain, when placed on the stack by the buffer overflow, will execute their desired command, and in a non-randomized environment, the payload on the stack would be structured as follows:

Table 1: Stack Frame of `vulnerable_function` Before and After Overflow

Stack Address (Relative to RBP)	Size (bytes)	Contents BEFORE Overflow	Contents AFTER Overflow	Description
		— High Memory Addresses —		
`[RBP + 8]`	8	`0x0000000000401214`	`0x000000004011f2`	**Return Address.** Hijacked to point to our `pop rdi; ret` gadget.
`[RBP + 0]`	8	(Address of `main`'s RBP)	`0x4141414141414141`	**Saved Base Pointer.** Overwritten by the last 8 bytes of padding.
`[RBP - 1]` `[RBP - 128]`	128	`\x00 \x00 ... \x00`	`0x414141...41414141`	**Buffer `b`.** Completely filled with padding characters ('A').
		— ROP Chain Written Below Original Stack Frame —		
`[RBP - 136]`	8	(Undefined stack space)	`Address of "/bin/sh"`	**ROP Payload.** Argument for `system`, popped into the RDI register.
`[RBP - 144]`	8	(Undefined stack space)	`Address of system`	**ROP Payload.** Target of the gadget's `ret` instruction; gets executed.
		— Low Memory Addresses —		

Because of this, to this day, memory-related security flaws remain a substantial issue. The vast amount of software already compiled from C and C++ sources makes comprehensive fixing of vulnerable code or rewriting into safer languages impractical [31]. To combat this, code randomization strategies emerged as a pragmatic defense [128]. These methods aim to obscure the memory layout of a program, making it considerably more difficult for an attacker to reliably locate and exploit known vulnerabilities using preexisting code segments [82]. Below can be found an instance of an ROP attack nullified by code randomization:

Table 2: Visualizing the failure of a static ROP Attack in a Randomized Environment

Stack Address (Relative to RBP)	Size (bytes)	Actual Memory Contents (Post-Randomization)	Attacker's Static Payload (Now Invalid)	Result of Mismatch
		— High Memory Addresses —		
`[RBP + 8]`	8	`0x00000000000040????`	`0x000000004011f2`	**Failed Jump.** The `ret` instruction follows the attacker's pointer but finds random data, causing an immediate crash. `...4011f2` → **[SIGSEGV]**
`[RBP + 0]`	8	(Address of `main`'s RBP)	`0x4141414141414141`	**Saved RBP Overwritten.**
`[RBP - 1]` `[RBP - 128]`	128	`\x00 \x00 ...`	`0x414141...41414141`	**Buffer Filled.**
		— The ROP Chain is Now Just Inert Data —		
`[RBP - 136]`	8	(Undefined stack space)	`Static Addr of "/bin/sh"`	**Meaningless Pointer.**
`[RBP - 144]`	8	(Undefined stack space)	`Static Addr of system`	**Meaningless Pointer.**
		— Low Memory Addresses —		

2.2 Code Randomization

Code randomization is a primary technique within the broader field of binary diversification. While diversification can involve numerous transformations, code randomization specifically focuses on altering the memory layout of a program's executable instructions with the ideal of an ever-changing attack surface, and an ultimate goal of confusing the attackers, thus preventing preplanned attacks. This belongs to a wider research field, namely, moving target defense [32]. In our scope, we will define code randomization as the process of obscuring the program's layout, making the absolute and relative addresses of code and data unpredictable. This is also often paired with other memory defense techniques, such as terminator-based buffer overflow detections (implemented via a canary value), `W⊕X`, and pointer encryption [59][1]. The first practical implementation was done in 2001 by the PaX project [120] and was later deployed in Linux, subsequently leading to the birth of the field of code randomization. Throughout the years that followed, the topic of code randomization has gained interest, and methods of different granularity began to be theorized, starting with base address randomization, then fine-grained randomization, and eventually leakage-resistant randomization.

2.2.1 Base address randomization solutions

Early techniques often focused on altering the starting locations of large memory regions [120, 164], while subsequent approaches explored finer degrees of unpredictability [19, 86, 163], reaching down to the level of functions or even smaller randomization units (or RUs for short) [73]. The following paragraphs will detail several of these initial schemes, including the system-level method of randomizing the base addresses of memory segments (ASLR) introduced by the PaX team, and its later refinements on Linux, macOS, and Android.

Linux ASLR First implemented as part of the PaX project's ASLR initiative, this technique departed from a program-specific diversification, and was rather done as an operating system characteristic, embedding randomization directly within the kernel itself [27]. Released initially as a patch in July 2001 and integrated into the Linux kernel, PaX ASLR randomized the base addresses for the kernel and user stacks. The randomization affected a specific number of address bits (for instance, 16 bits in a 32-bit address space) to maintain page alignment and reduce memory fragmentation. It further applied specific deltas (δ_{seg}) to introduce unpredictability. One such offset, *delta_mmap*, affected most memory mappings, including those for dynamic libraries [130]. Another, *delta_exec*, targeted statically linked programs that expected fixed load addresses and lacked internal relocation data, a common format for Linux executables at the time [52].

Binaries that lacked this relocation information would then be subjected to PaX ASLR's dual-mapping strategy, which consisted of one executable with randomized mapping and one non-executable with original address mappings for data access. The

[1] This discussion is based on the currently unpublished survey done in HSSL [115]

kernel would intercept execution attempts to the non-executable mapping and redirect them to the randomized version if security checks passed. That being said, such an approach implies diverse performance considerations, and together with the increasing prevalence of dynamically linked binaries, linking binaries that do not require such complex handling rendered this feature redundant and therefore discontinued in modern Linux distributions [120].

ASLR-NG To address certain weaknesses in the original ASLR, the ASLR-NG initiative was developed. It features a refined allocation and randomization logic with the primary objective of reducing exploitable correlations between disparate memory segments. This approach directly hindered de-randomization attacks where knowledge of one segment's base address and a low entropy delta would allow trivial calculation of another [89]. To accomplish this, ASLR-NG introduced context-aware placement strategies to counter the weaknesses of earlier implementations [110]. For instance, it could position "isolated" memory mappings with very high entropy across a large portion of the process' virtual address mapping range [62]. Conversely, for segments requiring contiguous growth, such as the main thread stack or heap zones managed by glibc's `ptmalloc`, it made use of "specific-zone" strategies. These would apply significant randomization to the base addresses of those zones while preserving their ability to expand.

The practical application of this principle was an objective of ensuring that the offset between the base of `.text` and the base of a frequently used shared library exhibited high entropy [109]. ASLR-NG also improved the maximization of effective randomization bits for all critical mapped regions, including position-independent executables (otherwise called PIEs) and position-independent code (PIC for short) in shared objects. On x86-64 for example, the full 47 bits of user-space address availability, minus page offset bits, would be more consistently used for defensive randomization, rather than being curtailed by overly conservative placement algorithms or insufficient PRNG output variability.

MacOS's ASLR Apple's Mac OS X also received an implementation of ASLR with version 10.5 OS X Leopard in October 2007 [84]. This first introduction of ASLR was, at that juncture, quite limited in its protective scope [28]. The principal outcome of Leopard's ASLR was the varying placement of shared system library addresses, and other common exploited memory areas, such as a program's execution stack and its dynamically managed heap, were not yet subject to this defense mechanism. This library-centric randomization persisted through the Mac OS X 10.6 Snow Leopard release in August 2009 [83], an iteration that primarily focused on refining system performance and completing the crucial architectural shift to 64-bit computing.

A substantial broadening of ASLR's protective capabilities was added with OS X 10.7 Lion, which appeared in July 2011 [10, 124]. Lion introduced an entire userland ASLR, which means that randomization was applied comprehensively to the stack, the heap, and the executable images for both 32-bit and 64-bit applications [108]. The goal of system-wide randomization was further advanced by OS X 10.8 Mountain

Lion in July 2012. This version extended ASLR to the kernel space, implementing Kernel ASLR (KASLR) [29] to vary the locations of the kernel, kernel extensions (kexts), and other system frameworks at each boot of the system.

Android ASLR The Android operating system's reliance on the Zygote process creation model, which is a system that forks new applications from a pre-initialized parent process, was a core reason behind the requirement for the development of custom ASLR variations. A further reason would be that this model, while efficient, can lead to inherited memory layouts in child processes, a problem that standard Linux ASLR does not address [18]. The early iterations of Android did not incorporate ASLR, a design choice that simplified exploit development, placing it behind other platforms that had already integrated such mitigations. [112]. Although later versions of Android did introduce ASLR, with the first implementation being for Android 4.0 Ice Cream Sandwich (ICS) in 2011 [135], its practical efficacy has been influenced by the ecosystem and the architectural design of Android's Zygote application instantiation mechanism. Specifically, new applications are typically forked from a pre-initialized Zygote instance [4], and if Zygote's memory layout is not sufficiently re-randomized for each fork, the child applications may inherit a degree of predictability, which weakens the ASLR's protective qualities [46]. Furthermore, analyses have indicated potential weaknesses in Android's pseudo-random number generators (or PRNGs for short) and their seeding process. Given that the strength of the randomization can be affected by the quality of the used pseudo-random number generators and their seeding process, this led to concern with regard to the effective entropy [20, 93].

Despite that, Android demonstrated a sustained effort to improve the capabilities of ASLR through its later versions of Android. Particularly, the next available update, Android 4.1 Jelly Bean, improved heap randomizations and introduced support for PIEs, permitting the randomization of the main executable's base address, which was previously loaded at a fixed location [126]. Android 5.0 Lollipop then built upon the support for PIEs and made them a requirement for all dynamically linked native binaries. Lastly, in August 2016, marked the release of Android 7.0 Nougat, and with it, a new layer of unpredictability through library load order randomization, thereby adding variance to the order in which shared libraries are mapped into a process' address space [162].

2.2.2 The need for fine-grained randomization

While base address randomization techniques obscure the starting locations of memory segments, they offer no protection against attacks once a single pointer into a code segment is leaked, especially with the existence of tools such as ROPGadget [2] that automate the process of scanning a binary file to discover usable instructions. The internal layout of functions and instructions within these segments remains deterministic, allowing an attacker with a single leaked address to de-randomize an entire library or executable by applying known, fixed offset. To counter such

[2] https://github.com/JonathanSalwan/ROPgadget

threats, a variety of more advanced mitigations were introduced. These mitigations can be broadly categorized into enforcement and obfuscation schemes. Enforcement schemes include methods like Control-Flow Integrity (CFI), whereas fine-grained randomization techniques are a prime representative of obfuscation methods, working at a much smaller scale. They alter the arrangements of smaller code units, such as individual functions or basic blocks within a program's executable image. The objective behind this is to disrupt an attacker's ability to locate and reuse specific gadgets, even if the general location of a larger code region is known or has been disclosed.

In contrast to base address randomization, which randomizes only the load offset of large program segments, a fine-grained approach modifies the internal structure of the code itself. While ASLR makes the absolute start addresses of the `.text` segment unpredictable, fine-grained techniques permute the relative offsets of smaller code units, such as functions or basic blocks within that segment [156, 68]. By reordering or inserting padding between these smaller units, these methods create a much more varied internal memory map, making it significantly harder for an attacker to construct precise exploit payloads that depend on the exact byte offsets of particular instruction sequences [156].

Bhatkar II A first example of this is an early system proposed by Bhatkar et al. [19] in 2005, which we will refer to as Bhatkar II. This is a progression from its earlier variant which increased the granularity of randomization by not only varying placements but also by reordering the sequence of functions and associated data items within the memory image. A main idea of this approach involved creating a function pointer for each function present in the codebase. Direct calls to functions were then substituted with indirect calls routed through these newly created pointers. This substitution converted direct references based on absolute memory addresses into indirect ones, thereby introducing an additional layer of indirection to hinder attackers. Safeguarding these pointers from modification was done by having them reside in read-only segments of the binary. In addition, to avoid the risk of switch statements being compiled into jump tables with fixed offsets, this strategy also compiled them into if-then-else conditional statements, thereby removing another source of static address information.

ASLP Not too long after this, Kil et al [86]. proposed a new fine-grained code randomization strategy, named Address Space Layout Permutation (ASLP), which, unlike Bhatkar II, did not involve source code transformation. Instead, its operation began by randomizing the base virtual addresses of executable and linkable format (ELF for short) segments, such as the `.text` and `.data` sections by directly manipulating the program's segment headers. Following this initial segment level randomization, ASLP proceeded to partition the `.text` segment into its constituent functions; a process that relied on extracting the name, address, and size of each function from the program's symbol table. Once functions were identified, ASLP computed a random permutation of these functions and then physically reordered the functions within the memory image according to this computed permutation, and then relocated both

code and data segments to reflect these changes. That being said, these relocations were fully dependent on the entries contained within the ELF relocation sections, such as `.rel.text` and `.rel.data`, which are generated by the static linker. Without this detailed, instruction-level guidance, ASLP's function reordering and segment adjustment would not be feasible.

2.2.3 Leakage-resistant randomization

While base address and fine-grained randomization techniques primarily focus on obscuring the initial code locations, they can still be compromised if an attacker can find runtime addresses through information disclosure vulnerabilities [40]. Leakage-resistant randomization, on the other hand, represents a distinct category of defense specifically developed to prevent such scenarios, with the goal of maintaining the unpredictability of code and data locations even in the presence of vulnerabilities that might otherwise reveal portions of the process's memory layout. As such, the principle behind this kind of randomization is to either make disclosed addresses bring no value to the attacker, or to actively protect address information from being read. This can be done through various methods, such as encrypting code pointers, introducing additional layers of indirection that are randomized and protected themselves [131], or using execute-only memory to prevent code from being read as data [25]. The end goal is to make sure that if an attacker can read some memory, the critical address information needed for tailoring an exploit remains hidden or changes too frequently to be reliably used.

Readactor Readactor, introduced by Crane et al. in 2015 [41] is an example of such a leakage-resistant code randomization scheme, as it addresses the threat of memory disclosure by combining execute-only memory with complex code-pointer hiding mechanisms. The system uses virtualization features present in processes to enforce execute-only permissions on code pages, and therefore directly counters attacks relying on reading code. To combat indirect disclosure, where attackers glean code locations from pointers in data segments, Readactor transforms direct code pointers into references to a separate, randomized area. This area itself is execute-only and its internal layout is de-correlated from the main program code, making leaked addresses of limited use to the attacker. Moreover, it extends its protections to both statically compiled code and dynamically generated code, such as that produced by just-in-time (or JIT for short) compilers found in web browsers, which are common targets of exploits [41, 17].

Isomeron Another approach to leakage-resistant code randomization can be seen in Isomeron. Isomeron was developed by Davi et al. [43] in 2015 to provide resilience against just-in-time return-oriented-programming (or JIT-ROP) attacks. This system maintains two in-memory representations of the program code, an original one, and a fine-grained randomized "isomer". At each call to the function, Isomeron makes a probabilistic decision to execute either the original or the diversified version of the

called function. This random switching between code isomers means that an attacker, even with full memory disclosure, cannot reliably predict which version of a gadget will be active when their exploit attempts to use it, thereby disrupting the chain of ROP gadgets. The system is implemented using a custom dynamic binary instrumentation framework that is designed for this unique dual-path execution model.

2.3 Process

To fully understand the possibilities of code randomization, one must apprehend all the details and necessary steps in performing it. The process can be defined as a series of transformations applied to an executable, either before execution or at load time. This activity is dependent on the data structures within the binary file format and the operational logic of the system's program loader and dynamic linker. The subsections below will describe these components and their roles in making memory layout diversification possible, with a visualization of its result available in Figure 2.

ELF format The Executable and Linkable Format is designed to help linkers and loaders in fulfilling their tasks [94], structuring a program's compiled code and data through two parallel views: one for execution and one for linking. The program header table defines segments, which are contiguous memory regions like `.text` and `.data` that the system loader maps into a process' address space. The section header table, in contrast, provides a finer-grained view for the linkers and debuggers, detailing sections like the symbol table (`.symtab`) and the relocation entries for the code section (`.rel.text`) [149]. The symbol table lists every function and global variable by name, associating each with an address or offset. Relocation sections contain entries that specify how to patch instruction operands and data pointers. These relocations are resolved at two different points in a binary's lifecycle: static relocations are processed by the linker to resolve intra-library references when building the binary, whereas dynamic relocations are processed by the system's loader at runtime to correctly link against shared libraries whose load addresses are not known at compile time, a feature that many randomization schemes depend on [151].

Process loading Process loading is fundamentally an act of memory mapping arranged by the OS kernel. This sequence begins when a parent process issues an `execve` system call. The kernel's loader then inspects the ELF file's program header table, a data structure composed of entries that define the program's memory segments. For each descriptor marked `PT_LOAD`, the loader establishes a mapping in the new process' virtual address space. The `p_vaddr` and `p_memsz` fields direct the start address and size of the virtual memory region [49], while `p_offset` and `p_filesz` specify the corresponding data within the executable file [98]. The text segment is typically mapped read-only and execute, whereas the data segment is mapped read-write [9, 86]. This direct mapping from file to memory is a point of intervention for base address randomization schemes, which introduce an offset to the `p_vaddr` of each segment before the mapping occurs. After all such segments are mapped and the

stack is created, the program is set to start. The exact next step depends on the ELF type. For a statically-linked executable (`ET_EXEC`), control is transferred directly to its entry point. For a position-independent executable (`ET_DYN`), however, the kernel instead passes control to a user-space dynamic linker. This linker must then perform dynamic relocations, calculating absolute addresses from the binary's relative offsets before the program can begin.

Dynamic linking Many ELF programs use dynamic linking to resolve external function calls at runtime [1]. This task is handled by a dynamic linker in the user space [13], specified in the executable's `PT_INTERP` header. When such a program starts, the kernel loads both the executable and the dynamic linker into memory, then passes control to the linker. The dynamic linker inspects the executable's `.dynamic` section, which contains a list of needed shared libraries (`DT_NEEDED` entries) and other metadata [137]. It recursively loads these libraries and then performs relocations. Function calls are mediated by the Procedure Linkage Table (PLT) and the Global Offset Table (GOT). A call to an external function first targets a stub in the PLT. This stub's initial logic is to invoke the dynamic linker's resolution routine. This routine finds the function's real address, writes it into the corresponding entry in the GOT, and then transfers control to it. Subsequent calls through the same PLT sub will jump indirectly via the now-populated GOT entry, avoiding the overhead of re-resoluting everything [12].

Relocation The relocation process, which corrects references to symbols whose addresses are not known at compile time, is an operation with multiple stages. The first one is the static relocation, which is performed by the static linker (e.g. ld) [143]. When combining multiple object files into a single executable or shared library, the linker resolves references between them. For references to symbols that are defined in external shared libraries, such as `printf` in `libc.so`, the linker cannot determine the final address. Instead, it generates relocation entries in dedicated ELF sections [119] (e.g. `.rel.dyn`, `.rela.plt`). These entries act as placeholders, providing the linker with a complete recipe for each patch. Specifically, each entry details which part of the code to modify via the `r_offset` field, which symbol's address to use via an index packed into the `r_info` field, and the exact formula to apply via a relocation type also stored in `r_info`. An additional `r_addend` value is often included in this calculation.

The second stage is load-time relocation, handled by the dynamic linker. When a program is executed, the dynamic linker loads the main executable and all its dependent shared libraries into memory. For each loaded object, it consults the relocation sections. It uses the architecture-specific relocation types, which, for AArch64, these include:

`R_AARCH64_CALL26` which is applied to branch with link (`BL`) instructions. Its value is calculated as Symbol + Addend - Place, encoding the PC-relative offset to the target function. Since our reordering changes the distance between caller and callee, this relocation must be re-evaluated. The processing of these instructions can be seen below in Figure 3. Moreover, this is a static relocation that is normally resolved and discarded by the linker.

R_AARCH64_ADR_PREL_PG_HI21 that is applied to ADRP instructions. This computes the page-level difference between a symbol and the current instruction, forming the high 21 bits of a PC-relative address.

R_AARCH64_ADD_ABS_LO12_NC being typically paired with the former and applied to an ADD instruction. This provides the lowest 12 bits of the absolute address.

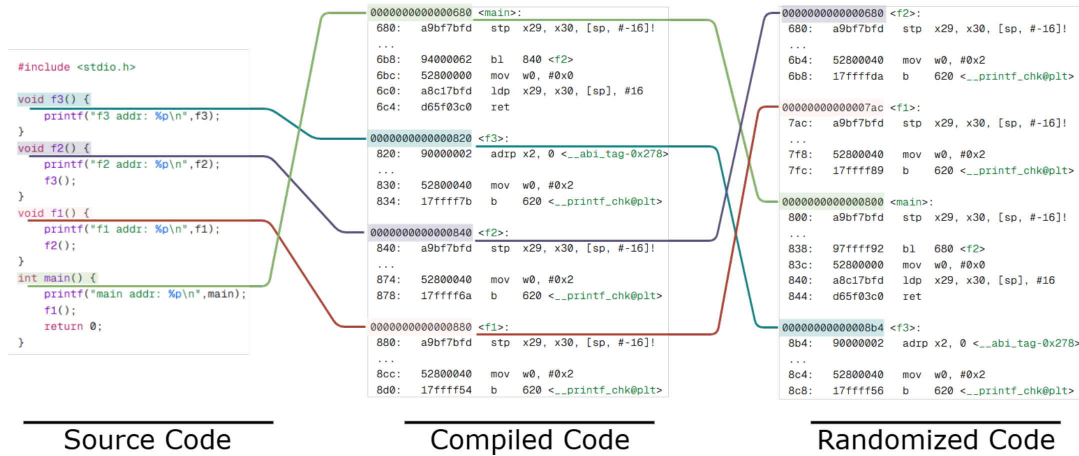


Figure 2: Code transformation before and after randomization

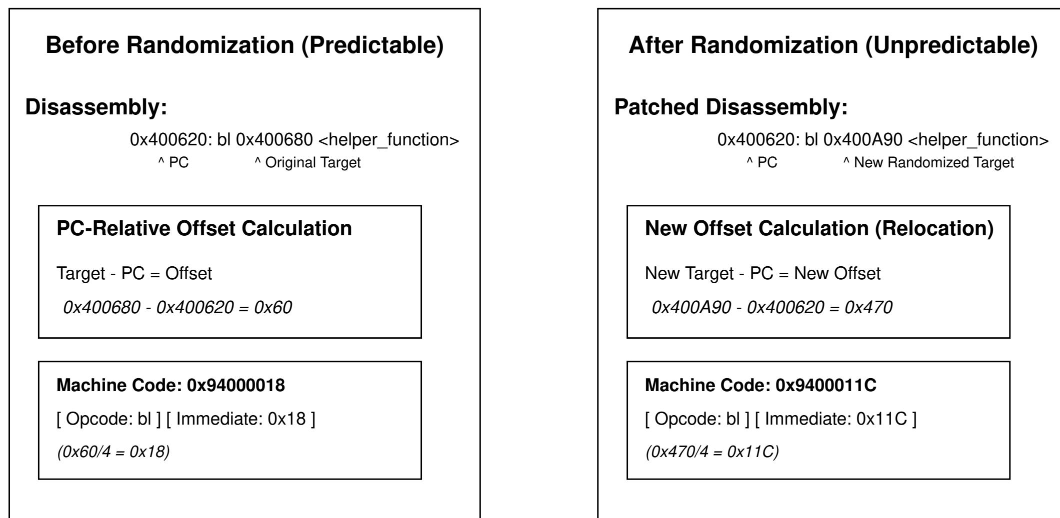


Figure 3: Relocation of a PC-Relative Branch (bl) Instruction After Randomization

2.4 Isolation

The effectiveness of any code randomization scheme is tied to the integrity of the randomization process itself. If an attacker can interfere with the agent performing the randomization or inspect the resulting memory layout, the defense is nullified.

When the operating system is not trusted to perform the randomization securely, it is possible to leverage architectural features that create protected execution contexts. The following sections will detail the privilege separation mechanism, particularly within the ARM architecture, that provides the necessary security guarantees for advanced, leakage-resistant diversification.

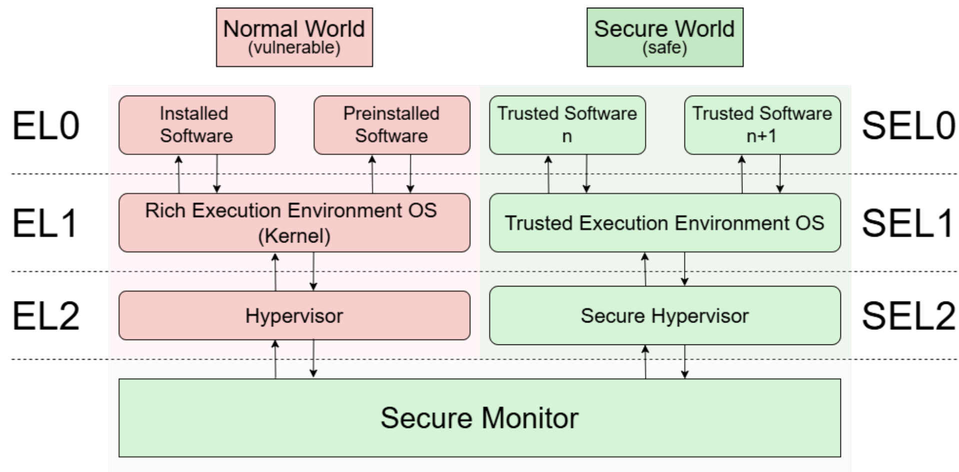


Figure 4: ARM privilege separation through execution levels

EL0 and EL1 Processor architectures enforce privilege separation through a system of distinct execution levels [15, 42], as seen in Figure 4 for ARM architectures. User applications, including both installed and preinstalled software execute at the lowest privilege level (i.e., EL0 on ARM). The main operating system kernel, which manages system resources within the Rich Execution Environment (REE), runs at a more privileged level (EL1 or Ring 0) and is specifically for supporting hardware-assisted virtualization [42]. The processor hardware strictly polices the boundary between these levels, preventing code at a lower level from directly accessing or modifying memory belonging to the higher one [154]. An application must request kernel services through a defined interface, such as system calls, which triggers a controlled transition to the more privileged state [155]. This hardware-enforced separation is the primary mechanism for protecting the OS kernel.

EL2 The hypervisor runs at EL2 and it is responsible for creating and managing one or more virtual machines. Moreover, as per Figure 4, the hypervisor sits between the physical hardware and the guest operating systems running in the REE, so while each guest OS runs at EL1 and believes it has full control over the hardware, it actually does not [56]. The hypervisor intercepts privileged operations from the guest kernels [87] (such as the modification of memory management registers) and emulates them in a way that maintains isolation between virtual machines. This allows multiple unaware operating systems to share the same physical processor securely. That being said, in an environment with one singular operating system, a hypervisor is not necessary.

Secure Monitor EL3 contains the Secure Monitor, which is a small piece of firmware that has the highest privilege level in the ARM architecture [154]. Its sole responsibility is to manage transitions between the Normal World and the Secure World, as illustrated by its central position in Figure 4. It does not act as an OS or hypervisor. Instead, it functions as a minimal, verifiable gatekeeper [136]. When code in the Normal World issues a Secure Monitor Call instruction, the hardware traps to the Secure Monitor. The monitor then saves the full context of the Normal World, switches the processor state to Secure, and delegates control to the relevant secure counterpart.

Secure Hypervisor As of the ARMv8.4-A specification, the ARM architecture extends virtualization capabilities into the Secure World through the introduction of Secure EL2 (S-EL2) [3]. This privilege level is positioned between the Secure Monitor at EL3 and the traditional TEE OS at S-EL1. A Secure Hypervisor, operating at S-EL2, can instantiate and manage multiple guest TEEs, each believing it has exclusive access to S-EL1. This facility permits the co-existence of distinct secure environments within the split world architecture, fully isolated from one another by hardware-enforced boundaries managed by the Secure Hypervisor.

TEEs The final layer of separation is the Trusted Execution Environment (TEE for short), which a hardware-isolated processing environment that runs parallel to the main operating system. On ARM processors, this isolation is traditionally implemented through TrustZone technology, which partitions the system into a standard Normal World and a hardware-isolated Secure World. More recently, ARM has also introduced its Confidential Compute Architecture [2] as an evolution of this principle. That said, a TEE provides security guarantees that exceed those of the standard EL0/EL1 separation. The TEE's threat model assumes that the entire Normal World software stack, from applications down to the hypervisor and kernel, may be suspicious [140]. Despite a full compromise of the REE, the TEE is designed to protect the full confidentiality and integrity of its own code and data [153]. This is achieved through hardware-backed memory isolation [91]. Given this security, our system uses a Trusted Application (otherwise known as a TA) inside the Secure World to perform our randomization, thereby keeping randomization schemes secure, as these operations are then protected from inspection or interference by an attacker who has gained control of the main OS. While a newer technology, particularly ARM's Confidential Compute Architecture, offers a similar environment known as a Realm, our work makes use of the more-mature split-world model and its standardized GlobalPlatform APIs.

3 Problem Statement

3.1 Threat Model

Our threat model adopts a zero-trust security posture [85, 139], which presumes no implicit trust in the host operating system for performing the randomization and encryption. An adversary with control over the OS could otherwise compromise these procedures and predict pseudorandom outputs or extract cryptographic keys. This model assumes an adversary who aims to execute a code-reuse attack, such as ROP, against a protected application. However, we fully acknowledges that an attacker with kernel control can use tools like `ptrace` to observe a process. The goal of our system is not to make the running process invisible, but rather to make the information an attacker can gather from such observation ephemeral and insufficient for a practical attack. As such, the model is designed to reflect two primary scenarios: the scalable, offline development of an exploit, and the more complex, real-time attack against a running process. Therefore, we grant the adversary with the following capabilities:

AC1: Real-time Process Observation We assume that the adversary has persistent access to the Normal World OS, but has failed in escalating privileges, and thus does not have root permissions. However, we will allow the adversary to read the process's memory map via `/proc/[pid]/maps`, a file that details the virtual address ranges, permissions, and backing sources for each memory segment in that instance.

AC2: Unrestricted Access to the File System The attacker has persistent read/write access to the file system, allowing them to exfiltrate any application binary for offline analysis as long as it is within the permissions of a non sudo user. This capability is in line with the threat of universal exploit development, where the vulnerability of one application can be weaponized against all its users.

AC3: Advanced Static and Dynamic Analysis The adversary is equipped with a full suite of reverse-engineering tools. They can perform static analysis on any binary they can get access to understand its structure and identify potential gadgets. They can also perform dynamic analysis on unprotected applications to observer their runtime behavior.

AC4: Limited Information Disclosure We assume the adversary can exploit an information disclosure vulnerability to read the content of a single, arbitrary memory address. Such a leak, occurring after the application is loaded and decrypted, would reveal the runtime location of a chosen function or Randomization Unit (RU).

AC5: Tampering with Inputs The adversary can modify any data in the Normal World before it is passed to the Secure World, including the application binary on disk before it is loaded.

This model presumes the integrity of the hardware-based TEE. The adversary is considered incapable of breaching the CPU-enforced isolation that separates the Secure World from the Normal World, as depicted in Figure 4. Consequently, direct inspection of a Trusted Application's memory, extraction of its cryptographic keys, or manipulation of its execution flow are outside the scope of the attacker's abilities. The Trusted Computing Base for this system encompasses both the CPU hardware that enforces isolation and the software that runs within the Secure World [90]. This software component of the trusted computing base, which is comprised of the TEE OS and our randomization and cryptographic logic, is assumed to be correctly implemented and free of exploitable vulnerabilities. Physical and side-channel attacks against the processor hardware are not considered in this model.

3.2 Requirements

The design and implementation of Preshuf are guided by a set of requirements deemed as core for defining its security, performance, and functionality to, in turn, ensure its adaptability at larger scale. These requirements are:

R1: Functional Preservation. The randomization process must maintain the functional correctness of the application. All internal code and data references must be correctly updated to reflect the new memory layout, and thereby guaranteeing the maintained proper execution and behavior of the application.

R2: Secure Randomization Environment. The integrity and confidentiality of the randomization process must be guaranteed, even on a compromised host operating system. The mechanism must protect all critical operations, including permutation generation and application, from both inspection and tampering by the untrusted Normal World.

R3: At-Rest Confidentiality and Integrity. The randomized binary must be protected at rest to prevent offline analysis. The confidentiality and integrity on persistent storage must be cryptographically enforced, mitigating threats from an adversary with file system access.

R4: Low Performance Overhead. The performance overhead introduced by the security measures should be minimal. The end-to-end process, from invocation to process execution should be nearly indistinguishable from running a non-protected application, imposing no significant latency on the user.

R5: User and Developer Transparency. The system must operate transparently without requiring user interaction or modifications to an application's source code. The protection mechanism should integrate into the standard process execution flow, automatically handing decryption and loading.

3.3 Threat of Runtime Interception

A significant runtime threat is process interception, a technique where a threat actor uses debugging interfaces, such as x86dbg to attach to another running process [118]. Once any debugger gets attached, the malicious process can pause the target's execution, inspect its memory, analyze the registers, and single-step through each of its operations [161]. For any randomization scheme that executes in the Normal World, this presents a direct attack vector. Assuming an encrypted binary file and a decryptor, an adversary could use such an interception mechanism to pause the decryptor, read the newly randomized binary directly from its memory, and thereby completely bypass our at-rest protections.

4 Design

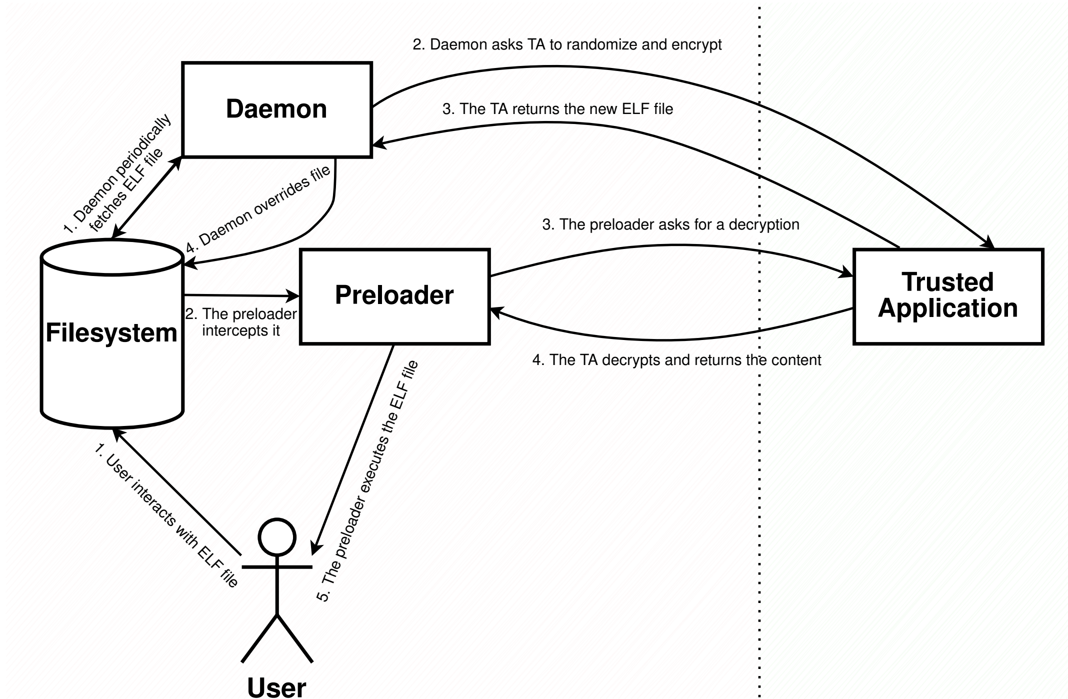


Figure 5: Component interaction

The architecture of our Preshuf tool is designed around a split-component model, utilizing the split-world framework in order to provide fine-grained code randomization with a minimal performance overhead. The design consists of three primary components, particularly: an asynchronous daemon, a TA within the TEE, and a runtime preloader. In here, the daemon operates in the background to periodically re-randomize and encrypt binaries, whereas the TA executes this logic in a hardware-isolated environment where the Normal World has no access to. Finally, the preloader will then intercept the user trying to launch an application to then manage the on-demand decryption with the TA, and finally load the program into memory. The following subsections will first detail the role of the OP-TEE environment in our system, then examine the specific algorithms for the randomization procedure and cryptography. Lastly, we will analyze the performance bottlenecks and implications of this architectural model.

4.1 OP-TEE

The security of our system relies on a minimal Trusted Computing Base that is rooted in the hardware security features, so as to perform randomization and encryption in an environment isolated from the main operating system, as per Figure 5. To achieve this, we have selected OP-TEE (short for Open Portable Trusted Execution Environment), which is an open-source TEE designed for ARM processors and it

implements the GlobalPlatform TEE specification. This specification standardizes the interface between an insecure OS, which is referred to as the Normal World, and the Secure World, which is called the TEE [63]. The architecture enforces a strict, hardware-level isolation between these two worlds, and the way it functions is that a Client Application (addressed to as a CA) running in Normal World can request services from a Trusted Application (TA for short) running in the Secure World. The CA however, is not able to inspect or interfere with the TA's execution or its protected memory.

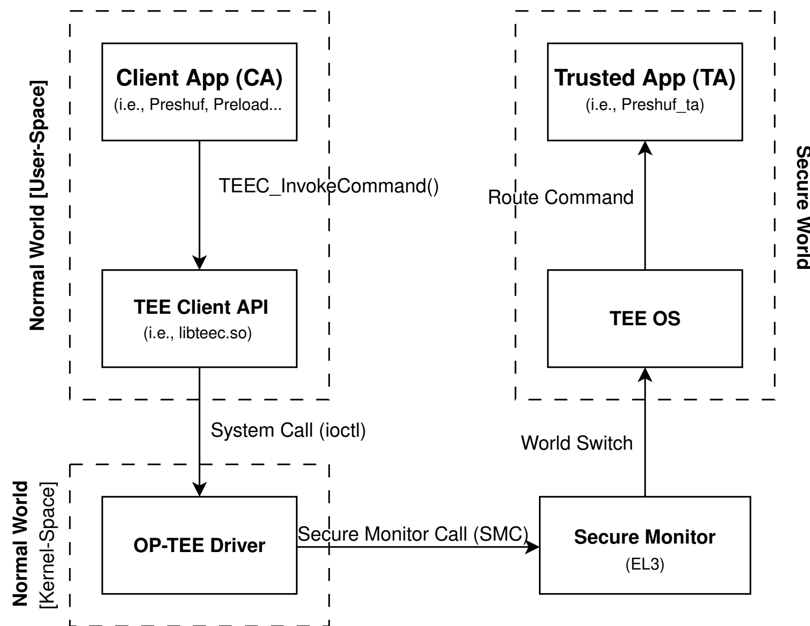


Figure 6: TEE Client API call flow from Normal World to Secure World

The selection of a TEE-based architecture is a direct response to the threat of runtime process interception. When the processor transitions into the Secure World to execute a Trusted Application, the Normal World, including its kernel and any attached debuggers, is paused [99]. The hardware itself enforces a strict isolation boundary, making the TEE's memory space completely inaccessible and invisible to any process in the Normal World. An attempt to inspect the memory of the Trusted Application performing the randomization or decryption would fail, as from the Normal World's perspective, there is nothing there to see [63]. This hardware-enforced opacity ensures that no user-level application, regardless of its privileges, can intercept and analyze our operations requiring security **(R2)**.

This protection extends beyond direct memory inspection to cover the communication channel between the Normal World (where software, preloaders, and drivers work) and Secure World (where our randomization and encryption TA runs) [141, p. 62-63]. The interaction is managed by a secure monitor, which vets all transitions [63]. Instead

of a single, monolithic “protocol”, security is achieved through a series of defined steps. When an application calls the TA, it uses a standardized TEE Client API, This user-space API translates the request into a system call to a trusted kernel driver. It is this driver, operating with kernel privileges, that then executes the necessary SMC instruction, passing control to the Secure Monitor, as seen in Figure 6. For bulk data, like the binaries to be randomized, the preloader can allocate a memory buffer in the Normal World and pass a reference to the TA [67, p. 21] which can map this untrusted buffer into the its own trusted address space. The TA can then treat this input as hostile, validating its structure before processing. This model prevents an attacker from passing a malicious pointer to overwrite secure memory or from modifying the binary after it has been checked but before it has been used (similarly to a TOCTOU attack) in order to make sure that the interface to the TA is resistant to tampering [141, p. 70-74].

An adversary cannot manipulate the parameters passed to the TA, nor can they spoof messages, as the TEE protocol is built upon a secure and session-based communication model. The TEE OS authenticates the caller by binding every operation to a specific session context that is established during an initial, trusted handshake, thereby preventing any unauthorized process from injecting commands, and protecting shared memory buffers [64, 80]. Therefore, even the interface between the preloader and the TEE is hardened against tampering, preventing attacks that seek to corrupt the randomization process by feeding it malicious inputs, attempt to extract the randomization seeding, or other similar attacks.

Our Preshuf tool, which also is the host application, is built around this client-server model; acting as the CA, and being responsible for handling the tasks in the Normal World. Preshuf_ta is the TA counterpart of preshuf and it contains the randomization logic. The operational flow of preshuf begins when it establishes a session with the TA, then passes the target ELF binary (which contains the necessary metadata in its `.PRR` section), computes a new random layout for the functions, shuffles the code within its own memory, and fixes the relocations and symbol table entries. Once the binary is randomized and relocated, the TA then encrypts the binary using an AEAD (short for Authenticated Encryption with Associated Data) cipher. The resulting ciphertext and its associated authentication tag are then passed back to the Normal World.

This design provides the security benefit of leakage resistance: the final, randomized layout of the binary is never exposed in plaintext outside of the TEE, nullifying attacks that rely on offline analysis of a shuffled but unencrypted file. The confidentiality of the randomization process is also guaranteed, as the random permutation and any cryptographic keys used are generated, used, and stored exclusively within the Secure World, beyond the reach of a Normal World adversary. The TEE-based approach is architecturally preferable to a Normal World implementation too. A computationally intensive task like randomization and encryption, when executed within the TEE, occurs in a single, uninterrupted context. In contrast, a Normal World process would not only be vulnerable to an attacker **(AC1)** but also require continuous context switches between user and kernel space for memory protection, thereby introducing significant overhead.

4.2 Randomization and Cryptography

4.2.1 Performing the randomization

For its initial scope, preshuf operates on compiled ELF binaries and it targets functions as the default unit of permutation (a method known as function-granular). The granularity is configurable, allowing for different Randomization Unit (otherwise called RU) sizes. The process is initiated by a parse of the executable's symbol table (`.symtab`), to extract the function symbols. This table's entries contain the metadata required to identify and delineate each function, including a string table index for its name, size, and its original virtual address (`st_value`). The function entries are then treated as discrete RUs, and from there, a random permutation is then computed to define a new, contiguous sequence for these RUs, which are then copied into the `.text` segments in the permuted order, henceforth ensuring that even with a leaked memory address, the attacker will not have enough information to craft an attack, protecting our system from **(AC4)**.

With the new layout defined, we compute a positional delta for each RU to differentiate between its original and new base address. This delta is the most important part when it comes to re-establishing the binary's referential integrity. This is because the permutation invalidates all hardcoded addresses, as a relocation pass is required to patch all pointers and make sure the program remains executable **(R1)**. For this, the first structure that needs to be repaired is the symbol table, which can be accomplished by recomputing the address of each function stored in its `st_value` field with its new location.

Next, we correct the binary's relocation entries. These entries are essentially instructions for the runtime loader, telling it where and how to patch memory addresses. Since the code containing these patch locations has moved, the r_offset field of each relocation entry itself must be updated. Furthermore, or specific relocation types like `R_AARCH64_RELATIVE`, where the `r_addend` field holds an address that now points to a shuffled function, that value must also be adjusted. We adjust each affected fields by its RU's delta, making sure that the runtime loader applies its patch to the correct instruction or data pointer in the newly arranged code.

The final stage is to then patch the machine code itself. We iterate through the relocation entries, recomputing instruction-specific values, for instance, such as the `R_AARCH64_CALL26` relocation type that requires recalculating a branch instruction's immediate operand based on its new PC-relative offset. This step is the final one before with the goal of making the binary's internal structure coherent and executable. The algorithm composed of Functions 1,2, and 3, depicted below, presents a pseudocode abstraction of the above-discussed process for our tool.

Function 1: `Fine-Grained Randomize and Relocate`

1: **Input:** ELF file E, Metadata M {RUs $\mathcal{R}$, Symbols $\mathcal{S}$, Relocations $\mathcal{L}$, .text_info, .data_info, GOT}
2: **Output:** Randomized ELF binary E_r
3: $\pi \leftarrow$ RandomPermutation($|\mathcal{R}|$) ▷ Stage 1: Compute Randomized Layout
4: $\mathrm{A_{new}} \leftarrow$ M.text_base
5: **for** $i = 0$ to $|\mathcal{R}| - 1$ **do**
6: ru $\leftarrow \mathcal{R}[\pi(i)]$
7: $\mathrm{ru.addr_{new}} \leftarrow \mathrm{A_{new}}$
8: $\mathrm{ru.\Delta} \leftarrow \mathrm{ru.addr_{orig}} - \mathrm{ru.addr_{new}}$
9: $\mathrm{A_{new}} \leftarrow \mathrm{A_{new}} + \mathrm{ru.size}$
10: **end for**
11: $E_{base} \leftarrow$ BaseAddress(E) ▷ Stage 2: Update Metadata (Fixups)
12: DoFixups($\mathcal{S}, \mathcal{R}$) ▷ Correct symbol values
13: DoFixups($\mathcal{L}, \mathcal{R}$) ▷ Correct relocation offsets
14: DoFixups(M.GOT, $\mathcal{R}$) ▷ Correct GOT entries pointing to code
15: $E.e_entry \leftarrow \mathcal{S}$[M.entry_idx].addr ▷ Correct ELF entry point
16: $\mathrm{B_{tmp}} \leftarrow$ AllocateBuffer(M.text_size) ▷ Stage 3: Physically Reorder Code
17: $\mathrm{B_{off}} \leftarrow 0$
18: **for** $i = 0$ to $|\mathcal{R}| - 1$ **do**
19: ru $\leftarrow \mathcal{R}[\pi(i)]$
20: Copy($E_{base} + \mathrm{ru.off_{orig}}, \mathrm{B_{tmp}} + \mathrm{B_{off}}$, ru.size)
21: $\mathrm{B_{off}} \leftarrow \mathrm{B_{off}} + \mathrm{ru.size}$
22: **end for**
23: Write($\mathrm{B_{tmp}}, E_{base}$ + M.text_offset, $\mathrm{B_{off}}$)
24: RelocateSegment(E_{base}, M.text_info, $\mathcal{L}, \mathcal{S}$) ▷ Stage 4: Apply Relocations (Patch Segments)
25: RelocateSegment(E_{base}, M.data_info, $\mathcal{L}, \mathcal{S}$)
26: **return** E

Function 2: `DoFixups`

1: **Input:** Collection C (Symbols, Relocations, or GOT entries), RUs $\mathcal{R}$
2: **for all** $c \in C$ **do**
3: $ru \leftarrow$ FindRUByOriginalAddress($c.addr_{orig}$)
4: **if** ru is found **then**
5: $c.addr_{new} \leftarrow c.addr_{orig} - ru.\Delta$
6: **end if**
7: **end for**

Note: $c.addr$ refers to a symbol's value, a relocation's offset, or a GOT entry's target address.

Function 3: `RelocateSegment`

1: **Input:** Base address E_{base}, Segment Info Seg, Relocations $\mathcal{L}$, Symbols $\mathcal{S}$
2: **for all** $l \in \mathcal{L}$ **do**
3: **if** $\mathrm{l.offset}_{new}$ is within Seg **then**
4: $\mathrm{A_{patch}} \leftarrow Seg.\mathrm{mem_base} + (\mathrm{l.offset}_{new} - Seg.vaddr)$
5: $\mathrm{s_{target}} \leftarrow \mathcal{S}[l.\mathrm{sym_idx}]$
6: $\mathrm{A_{target}} \leftarrow \mathrm{s_{target}.addr_{new}}$
7: PatchInstruction($\mathrm{A_{patch}}, \mathrm{A_{target}}$, l.addend, l.type)
8: **end if**
9: **end for**

4.2.2 Using the Trusted Execution Environment

Having the randomization done through an application within an untrusted environment such as EL0 would expose the process to adversarial inspection, even without root privileges. Our threat model grants an attacker the ability to observe a running process **(AC1)**, which would allow them to analyze the randomization mechanism and its outputs, nullifying its effectiveness. The TEE's hardware-enforced isolation, on the other hand, provides the necessary protection against such analysis. When control transitions to the TA, all Normal World processes lose visibility, protecting the permutation's generation and application in a confident context [67, p. 54] and thereby countering the threat of real-time process observation **(AC1)**. While this secure the process, it does not secure the output artifact. An attacker with persistent storage access **(AC2)** could analyze a plaintext binary offline. To counter this, our system ensures the randomized binary is never exposed as a persistent file in plaintext, making post-randomization encryption mandatory before returning it to the Normal World.

As opposed to this, an in-kernel implementation of preshuf presents two significant architectural problems. Firstly, it violates the principle of least privilege by forcing the logic to run at the kernel's high privilege level (EL1), and secondly, it creates a tight coupling between the security feature and the OS, making the solution difficult to port and maintain. Our TEE-based architecture solves both issues. The TA runs in a low-privilege, sandboxes mode (S-EL0) within the Secure World, fulfilling the principle of least privilege. Moreover, by communicating through the standardized GlobalPlatform API, it remains a modular and portable component, independent of the Normal World OS specifics.

4.2.3 File encryption

To secure the randomized binary while on disk, preshuf_ta makes use of AEAD. This cryptographic primitive is of high importance to our security model because it addresses the threat of offline analysis **(AC2)**. While randomization within the TEE protects the permutation process itself, the resulting shuffled binary, if stored in plaintext, would still be vulnerable. An attacker with file system access could simply copy the shuffled file and reverse-engineer it at their leisure, completely negating the benefits of the randomization. AEAD prevents this by providing two security properties; namely, confidentiality through encryption, and integrity and authenticity via a Message Authentication Code (MAC), which prevents an attacker from accessing the plaintext files and maliciously modifying or tampering with the encrypted file **(AC5)** without detection **(R3)**. The key used for these operations is generated and managed exclusively within the TEE, never exposed to the Normal World, thus guaranteeing that decryption can only occur in a controlled manner at load-time.

In selecting an AEAD algorithm, the security guarantees and performance characteristics of each variants were considered. AES-GCM is a well-established NIST standard, offering high performance where hardware acceleration is present, though its security is critically dependent on nonce uniqueness. Reusing a nonce with the same key in GCM is not ideal because it not only leaks the XOR of plaintexts but can

also allow an attacker to recover the authentication key, thereby allowing for universal forgery [44, 54]. ChaCha20-Poly1305, an IETG standard, provides an alternative with strong software performance and a different security profile that is less brittle with respect to nonce reuse, as, while reusing a nonce still compromises the confidentiality of the affected messages, it does not reveal the authentication key, thereby preserving the integrity of other communications [81].

The final candidate, ASCON, was recently named the official standard for lightweight cryptography by NIST [114]. Its design was tailored for high efficiency and a small code size, making it very well-suited for the resource-constrained environments and thereby ideal for our performance ideals. The algorithm's selection in the Lightweight Cryptography (LWC for short) competition [36] affirms its security and makes it a prime candidate for the modern applications that require authenticated encryption.

To encapsulate the necessary cryptographic outputs, `preshuf` defines a custom file format that encapsulates all required data for decryption. This format, as seen in Fig. 7, is constructed from several key components: a unique 3-byte magic number that serves as a file signature that allows the preloader to identify `preshuf`-protected binaries, a part of the nonce that we address as the nonce suffix, and the authentication tag. The magic bytes are at the top of the file, whereas the nonce suffix and the tag are at the bottom, and the reasoning for embedding all this data into the output file is so that the TEE can perform an authenticated decryption with all the data available in a singular, self-contained artifact.

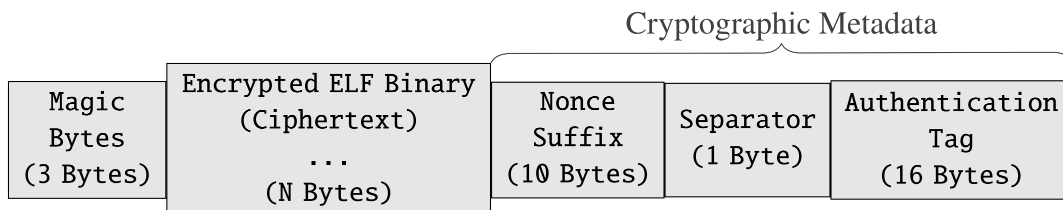


Figure 7: Structure of the Preshuf Encrypted File Format.

The at-rest encryption provides confidentiality but creates a functionality problem: the operating system can no longer interpret and run the protected binaries, so a mechanism is needed to make these files executable on demand, in a way that is transparent to the end-user **(R5)**. The system must appear to function as if no encryption were present. This challenge can be addressed by designing a preloader component; and so, for the system's security model to be complete, this runtime component must mediate the transition of executable process. The fundamental design of this component is loading sequence before the application's code begins to run. Upon interception, its responsibility is to inspect the `preshuf` protection, and then securely sort out the decryption process with the TEE. The final and most important step of this preloader would be to load the resulting plaintext directly into memory for execution, thereby preserving the confidentiality of the randomized code by bypassing the filesystem and preventing any offline analysis by an attacker with disk access **(AC2)**.

4.3 Bottlenecks and implications

In this subsection we will analyze the performance implications of our design choices, as efficiency **(R4)** is a core requirement to deliver the user transparency outlined in our prior requirements **(R5)**. The main point of consideration is the timing of the randomization process itself. Two primary strategies within our scope are a system-wide shuffle at boot-time or a targeted shuffle at load-time. The following paragraphs will outline the trade-offs of each and justify our architectural direction with `preshuf`.

Boot-time randomization The first architectural strategy to consider for implementing code randomization and that we will discuss, is to perform the operation during the system's startup sequence. This boot-time approach can then be paired with early-launch mechanisms so not lingering threats can monitor the shuffling process, thereby removing the need for a TEE for the initial shuffling process. The other benefit would be that it can provide broad, uniform protection by allowing a possible shuffling of shared libraries before any user code executes. The primary challenge with this model, however, is one of performance impact on boot speed. The process of reading, permuting, and fixing relocations for a multitude of system files is computationally intensive and is likely to introduce a startup delay directly proportional with the amount of files requiring shuffling on the system. On a minimal system, this added time may be insignificant, but on a fully-featured desktop or server environment, this latency can become a major source of user frustration. Because our system prioritizes immediate responsiveness and an ideal user experience **(R4)**, we reject the boot-time model to avoid this potentially significant and variable performance delay.

Preloader based randomization To mitigate the performance penalties associated with slow boot times, a second architectural idea would be to perform randomization on a per-application basis, at the moment of execution. This load-time model confines the performance impact to only the application being launched, leaving the system's overall responsiveness untouched during startup. The inherent trade-off is the introduction of a new latency on the path of application startup. This delay is the cumulative duration of several sequential operations. This delay is the cumulative duration of several sequential operations: reading the binary from storage, communicating with the TEE to initiate a secure session, executing the computationally expensive code permutation, and applying the subsequent relocation fixups. The central challenge of our work is to design a system where the total duration of these steps is sufficiently small that it does not create a noticeable slowdown of the device, so, while this options showcases great potential, it still is heavily dependent on the randomization process and its overhead.

Proposed solution Our proposed solution decouples the computationally expensive randomization from the time-sensitive application launch path **(R4)**. This is achieved through an asynchronous background process, managed by a persistent daemon, which continuously re-randomizes and encrypts the system's protected binaries. These

newly diversified versions are prepared in the background and, at designated intervals, atomically replace the previous set, as shown in Figure 8. The synchronous, load-time component is now dramatically simplified. When a user executes a program, the preloader's sole responsibility is to trigger the hardware-accelerated decryption of the current binary version. The expensive permutation and fixup tasks have already been completed offline

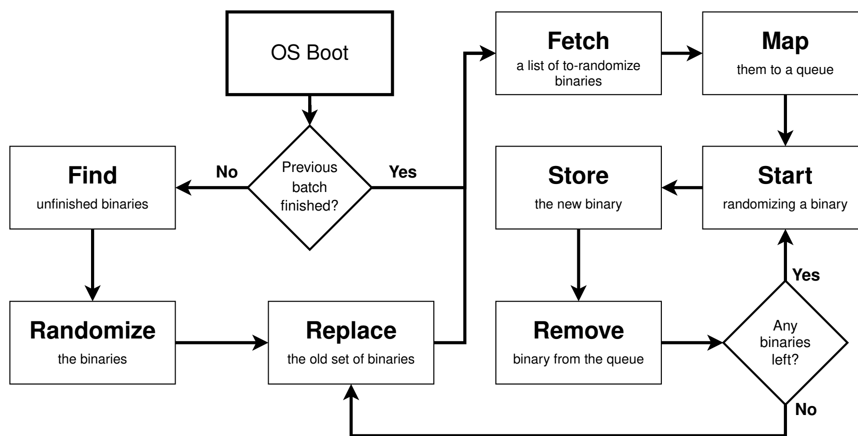


Figure 8: Design of Preshuf runtime

Benefits of solution This architecture provides a dual-advantage. From a performance workload entirely off the critical path, leaving only the highly optimized decryption operation to contribute to load-time latency. From a security standpoint, the system implements a form of refreshing defense. The continuous, background re-randomization means the system's memory layout can evolve over time without requiring a reboot. This offers a security posture that is at minimum equivalent to a full boot-time shuffle but with the added benefit of making leaked information obsolete after the next cycle complete, therefore drastically lowering the attack surface in case of an internal memory layout leakage.

5 Implementation

The development of Preshuf was done in collaboration with Huawei Technologies with the long term goal of its integration in the HarmonyOS ecosystem. As this environment is built exclusively for ARM64-based processors, this architecture was the necessary and logical platform for the research. All implementation and evaluation efforts were therefore tailored specifically to match this target architecture, and a high-level view of the process can be found in Figure 9 below.

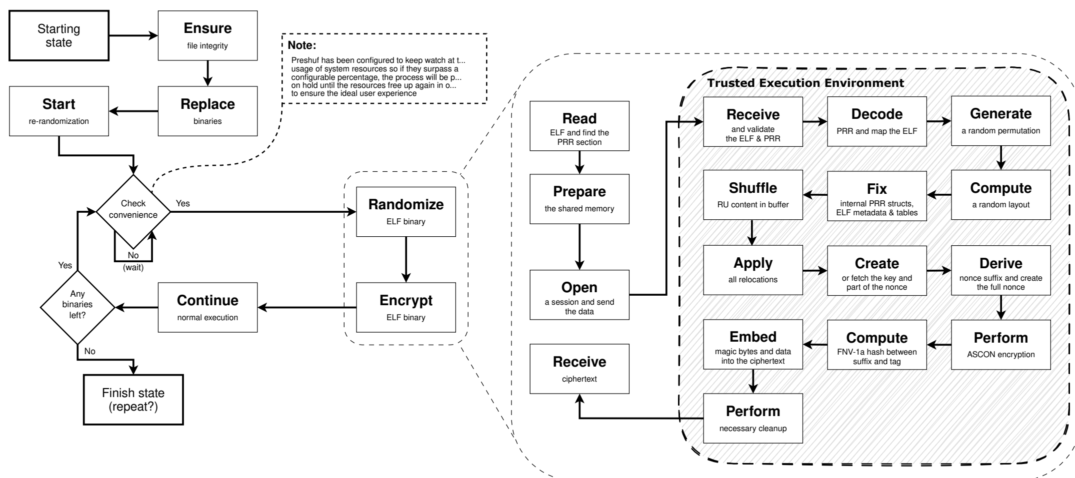


Figure 9: Full overview of Preshuf

That being said, a central component of Preshuf is the preparation of binaries through the introduction of a custom data structure for what we will call Preshuf Runtime Relocation (`.PRR`). This metadata is stored within a dedicated `.PRR` section added to the executable. Standard linkers typically resolve and discard the static relocation entries such as those in `.rel.text` that our fine-grained shuffling algorithm depends on to patch pointers to their permuted values. This "prr-injection" process begins by parsing the target binary's ELF and program headers to build an in-memory map of its structure. This map is then used to safely append the new `.PRR` data and update the necessary header fields to reflect the modified file layout. Then, the tool writes the content of existing sections, maintaining their alignment, before inserting the newly generated `.PRR` data. The content of this section follows a custom, serialized binary format. It begins with a unique 4-byte magic identified `"PRR\0"`, followed by a fixed-size C structure (`prr_info_st`) that contains the needed values as shown in Figure 10.

Data Type	Variable Name	Field Description
unsigned char[3]	**magic**	The "PRR" magic identifier.
size_t	**count_rus**	The number of Randomization Units (RUs).
size_t	**count_symbols**	The number of symbols in the symbol table.
size_t	**count_relas**	The number of relocation entries.
uint64_t *	**ru_start_addr**	Array of original RU start addresses.
uint64_t *	**ru_size**	Array of original RU sizes.
Elf64_Sym *	**symbols**	The symbol table entries.
Elf64_Rela *	**relas**	The relocation entries.

Figure 10: Key fields within the `prerand_info_st` structure.

Following this fixed header, the section contains a series of dynamically sized data blocks, each prefixed with its length in bytes. This allows the TA to decode the data with no prior knowledge of its size. These blocks include a complete copy of the application's string table, the array of RUs derived from the symbol table, the symbol table itself, and all relocation entries (`Elf64_Rela`) necessary for the fixup stage. Finally, the ELF's section header table is appended to the end of the file, with its own header entry updated to reflect the addition of the `.PRR` section, and the main ELF header's `e_shoff` field is adjusted to point to this new location.

5.1 Implementation of the Code Shuffling Tool in the TA

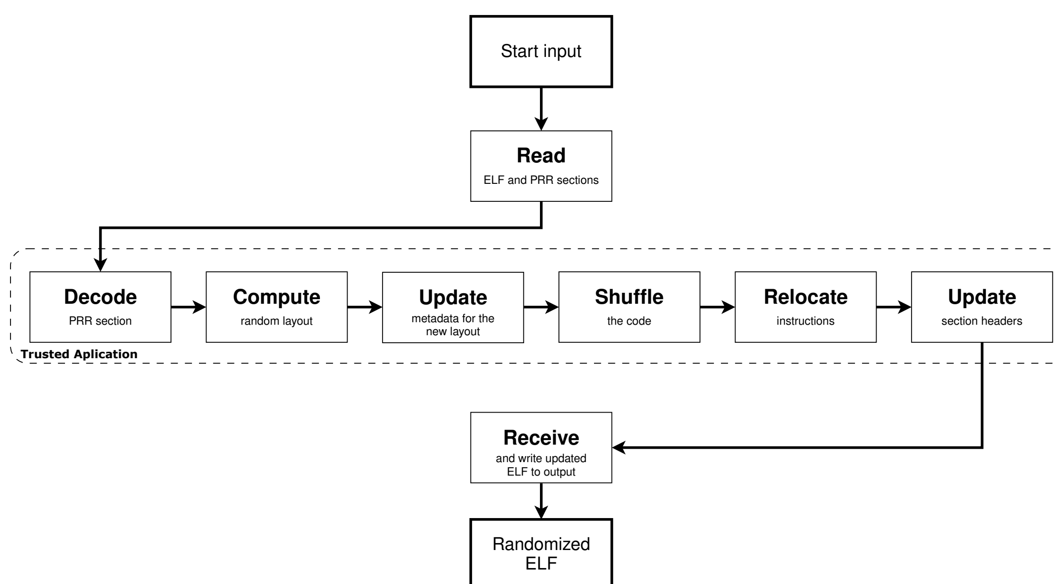


Figure 11: Elfshuf's workflow

Just as it can be seen in Figure 11, `preshuf` begins its randomization process with a parsing stage that dissects the input ELF binary to build an in-memory model of its structure. This initial phase involves parsing the main ELF header to identify the

locations of the program and section header tables. From there, the tool extracts the contents of key sections, most important the symbol table (`.symtab`), the string table (`.strtab`), and all static relocation tables (such as `.rela.text`). The first stage of Preshuf's randomization process partitions the executable's code into discrete RUs. To accomplish this, the tool performs a detailed parse of the input ELF binary, extracting the key sections, of which, the most notable being the symbol table, the string table, and all static relocation tables. By default, each function identified in the symbol table is treated as a single RU; for each, the tool recording its original virtual address, size, and file offset.

With the binary's executable code mapped and partitioned into a set of RUs, the system proceeds to the re-ordering phase. As detailed in Stage 1 of Algorithm 1, a random permutation is generated for the list of RUs, which dictates the new shuffled layout. The tool then calculates a new virtual address for each RU and the corresponding positional delta. Before any machine code is moved, the tool performs a metadata fixup pass using these deltas. This fixup process, formalized in Algorithm 2, updates all address-dependent entries to reflect the new layout: the `st_value` field of each symbol `Elf64_Sym` is corrected to its new virtual address, the **r_offset** field of every relocation entry (`Elf64_Rela`) is adjusted to its new location, and the program's main entry point address in the ELF header is updated.

The final stage applies these changes to the binary, detailed in Stages 3 and 4 of Algorithm 1. First, the code is reordered by copying the machine code of each RU from its original location into a temporary buffer, following the new randomized sequence. This buffer, now containing the fully shuffled `.text` segment, is then written back into the ELF file. With the code physically moved, the tool performs the final patching of the machine code itself, a process detailed in Algorithm 3. It iterates through the corrected relocation entries, applying each one to its corresponding instruction. For instance, for a branch instruction with an `R_AARCH64_CALL26` relocation, the tool recalculates the 26-bit immediate operand to reflect the new PC-relative distance to its target. Similarly, for a data access using an `ADRP/ADD` instruction pair, it re-computes both the high 21-bit and low 12-bit immediate values based on their respective relocation entries. After all relocations have been applied, the binary is left in a fully shuffled, internally consistent, and executable state.

5.2 Implementation of the Encryption Tool in the TA

The encryption process is the final stage of the process, and it is executed exclusively within the TA to guarantee that no plaintext randomized binary is ever exposed to the untrusted Normal World, as per Figure 12. The goal of this is to counter the considerable threat of offline file analysis, as defined by an attacker with unrestricted file system access **(AC2)** and advanced static analysis tools **(AC3)**. The system uses Authenticated Encryption with Associated Data (otherwise referred to as AEAD), a class of ciphers that provide both confidentiality and data integrity at the same time [129]. Particularly, confidentiality through encryption and integrity through an authentication tag to avoid potential tampering **(AC5)**. This design choice is important as it prevents an attacker from both reading the shuffled binary at rest and tampering

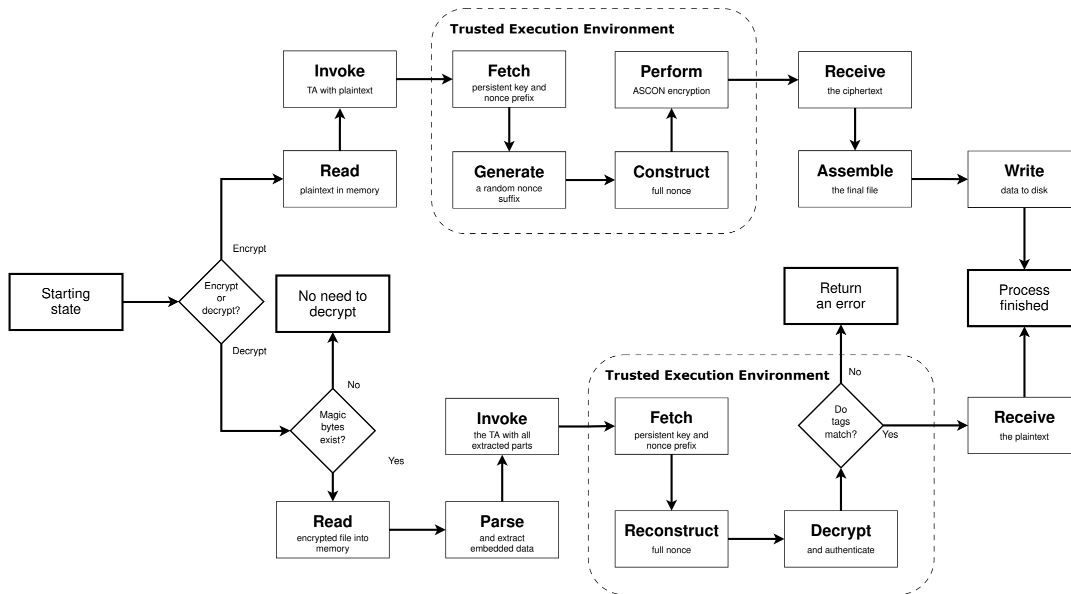


Figure 12: Elfcrypt's workflow

with the encrypted file without detection, thus properly defending against **AC2, AC3** and **AC5**. While the cryptographic backend is modular to allow the use of various AEAD implementations, the primary implementation uses ASCON [47, 48], was selected as the winner of the NIST Lightweight Cryptography competition [111], chosen for its high performance in software and suitability for resource-constrained environments.

The security of the entire process is based on the TA's key and nonce management scheme. The cryptographic key is not ephemeral; instead, it is a persistent asset managed by the OP-TEE secure storage subsystem. Upon first execution, a 16-byte key is generated using the hardware's true random number generator and saved to a protected area of persistent memory. On subsequent operations, the TA retrieves this same key, guaranteeing cryptographic continuity. This key is never exposed outside the Secure World's hardware-enforced memory protection. Nonce management, which is important to the security of the ASCON stream cipher, is handled through a split-nonce mechanism to take up less space in the TEE while maintaining uniqueness. A unique 6-byte nonce prefix is stored in secure memory alongside the key for each file, while a fresh, 10-byte random nonce suffix is generated by the TA and then concatenated with the persistent prefix to form the full 16-byte nonce, guaranteeing that a unique nonce is used for every file encryption, thus preventing key reuse attacks.

Upon successful encryption of the randomized binary, the TA constructs a self-contained file designed for efficient and secure processing by the runtime preloader. The final output is prefixed with a 3-byte magic number prefix that acts as a unique signature, allowing the preloader to instantly identify protected files without complex parsing. The main body of the file contains the ciphertext of the randomized and encrypted ELF binary. The cryptographic metadata is appended as a suffix at the end of the file; specifically, the 10-byte random nonce suffix is written, followed by

a unique 1-byte separator, and finally the 16-byte ASCON authentication tag. This structure guarantees that a single read of the file provides the preloader with all the components necessary for authenticated decryption: the ciphertext to decrypt, the nonce suffix to reconstruct the full nonce, and the tag to verify the file's integrity.

5.3 Preloader-based Startup Interception

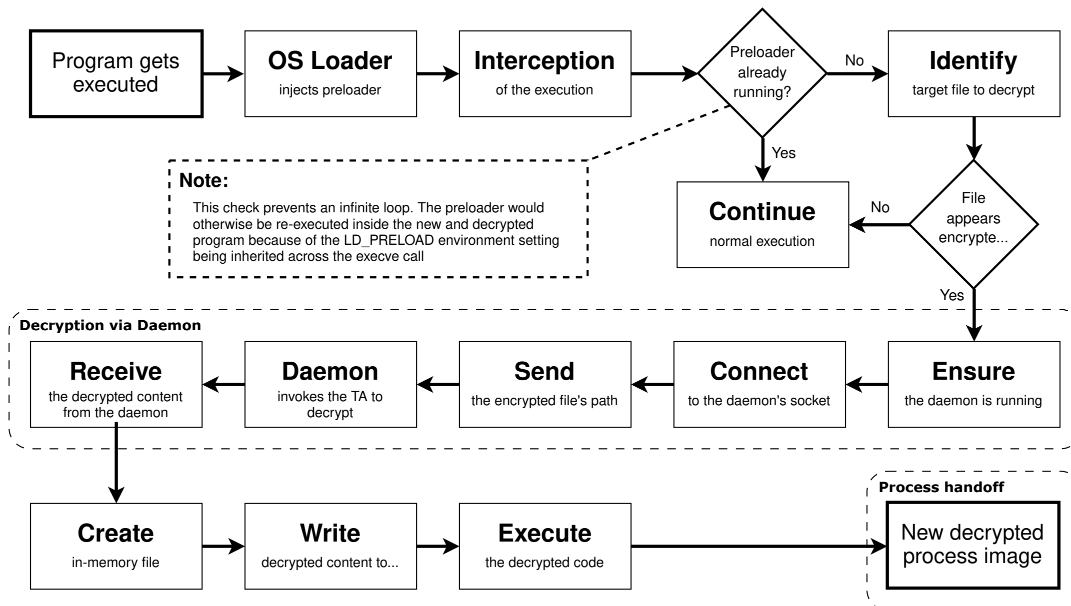


Figure 13: The Preloader's workflow

As seen in Figure 13, the preloader's operation is initiated by the dynamic linker, which loads it into a process's address space before the main application, a behavior triggered by the `LD_PRELOAD` environment variable [145]. Its primary function is to intercept the program's startup sequence by providing its own `__libc_start_main` function. This implementation first dynamically resolves a pointer to the original `__libc_start_main` from the standard C library using `dlsym(RTLD_NEXT, ...)`. It then determines the path of the executable to inspect, reading it from `/proc/self/exe` over the potentially misleading `argv[0]`[3], and then identifies the script file in cases involving shell interpreters. Once the target file is identified, the preloader performs a check for the additional data on its first and last few bytes. If the signature is not present, it calls the original function pointer and passes through all the original arguments to resume a normal execution. If this signature is found, it initiates the decryption stage and then executes the resulting plaintext via `fexecve`, a path that replaces the current process and does not return to the original.

On the other hand, if the signature check confirms the file is a protected binary, the preloader begins the decryption process, which is offloaded to a persistent, background

[3]For interpreted languages, `/proc/self/exe` points to the interpreter executable, not the user's script. The preloader therefore examines the program's arguments to locate the correct target file for potential decryption.

daemon for performance and security. The preloader first attempts to connect to this daemon through a UNIX domain socket, but if the connection fails, it assumes the daemon is not running and takes responsibility for starting it. To prevent race conditions where multiple preloaders might try to start the daemon simultaneously, it uses a file-based lock. The way this is implemented is by opening the lock file and attempting to acquire a non-blocking and exclusive lock using `flock(lock_fd, LOCK_EX | LOCK_NB)`. If this fails with `EWOULDBLOCK`, it indicates that the lock is held and the preloader will retry with a blocking `flock(lock_fd, LOCK_EX)` call, which pauses the preloader until the original process has finished starting the daemon. Once the lock is acquired, it forks a new process that executes the main `prerand` with the `start-daemon` argument, and then waits for the daemon's socket to become available. With a connection established, the preloader sends the full path of the protected binary to the daemon for it to then resolve the authenticated decryption and then stream the resulting plaintext back to the preloader over the socket, preceded by its total size.

With the decrypted, randomized plaintext now held in a memory buffer, the preloader's final task is to execute it securely, without ever writing the plaintext to the disk. To accomplish this, it makes a `memfd_create` system call, which creates an anonymous file descriptor backend entirely by RAM [146]. The preloader writes the entire decrypted binary into this memory file, and, for an `ELF` executable, it then uses the `fexecve` system call to load and run the program directly from the file descriptor, completely bypassing the filesystem [144]. Before this final execution step however, it sets a specific internal variable which is checked for at startup so that it will not re-intercept a new process while resolving an original one, thus avoiding an infinite loop. In the case of a decrypted script, the process is slightly different: It first constructs a new argument list, invokes `/bin/sh` and passes the path to the memory file `/proc/self/fd/...` as the script to be executed.

5.4 Architectural Synthesis and Operational Flow

The Preshuf architecture shown at the beginning of this section in Figure 9 integrates the above-defined daemon, TA, and preloader into a cohesive operational flow. This system implements a refreshing defense by periodically re-randomizing application binaries, ensuring proper "freshness" of the files. The process is driven by a background daemon that operates during periods of low system activity. It takes existing application binaries, already prepared with `.PRR` metadata, and sends them to the TA for a complete security overhaul. Inside the TEE's protected environment, the binaries are first re-shuffled using a freshly generated random permutation and then immediately encrypted with a persistent, hardware-backed key. The resulting encrypted files are then used to atomically replace their older counterparts on the disk. This replacement is safe for a live system, as running applications in Linux operate on an in-memory representation of their binary, decoupled from the on-disk file that was used to launch them.

This design, in essence, creates a moving target defense that directly counters an attacker with user-level privileges. An attacker with access to the file system **(AC2)** only ever finds encrypted binaries whose internal layout is unknown. Even if

an attacker were to acquire a randomized and unencrypted binary to then decrypt it manually, the background daemon would eventually replace it, thus rendering any intelligence gained from that analysis invalid for future exploits. At runtime, the preloader acts as a secure gatekeeper, identifying protected files by their magic number and handing them off to the TA for decryption. Because the decryption and loading process happens just-in-time and directly into memory, the plaintext binary is never written to disk. This prevents a user-level attacker from inspecting or tampering with randomized code before it executes, and thus preserving its confidentiality until the end of the process.

5.5 Necessary OP-TEE modifications

The standard `OP-TEE QEMU` platform, while a convenient development environment, is architecturally misaligned with the persistence requirements of our security model. It defaults to a stateless, RAM-based filesystem that is destroyed on shutdown, thereby rendering our batch daemon's function of producing and storing hardened binaries across boots impossible. To guarantee the alignment between the two, our first necessary modification was to transform this ephemeral environment into a stateful one. By reconfiguring the integrated `Buildroot` build system, we instructed it to produce a persistent `ext4` block device image instead of the default `cpio` archive intended for use as an in-memory `initramfs`. This guarantees a system that is more conventional, with writable root filesystem, which is a necessity for the refreshing, system-bound persistent randomization strategy, where the newly protected binaries must be stored for future execution either at a later point in time or in subsequent boots of the system.

The second set of modifications was required to produce software compatible with our randomization tool by using `-emit-relocs` and `-no-relax`. This is because the tool's randomization and relocation-fixing logic requires a structural blueprint of the binary it is processing, in the form of static relocation entries. Despite this, standard build processes are optimized for production and strip this metadata due to their irrelevance to the customer. To ensure that this is something that does not happen in our situation, we established a new system-wide compilation policy by modifying the global linker flags within Buildroot. In principle, the usage of `-emit-relocs` was to preserve these relocation entries and `-no-relax` to prevent linker optimizations that may remove or corrupt them. These policies mean that the entire Normal World OS, including the many core utilities consolidated within the BusyBox executable, is compiled from the ground up to be compatible with Preshuf, creating a fully "instrumentable" environment.

Lastly, to make sure that our performance analysis was not limited to an idealized, emulated platform, we extended this custom environment to physical hardware. We selected the Raspberry Pi 3 as representative embedded system and created a parallel Buildroot environment for it. We applied the same fundamental modifications: the global linker settings were adjusted to produce relocation-rich binaries, and the SD card served as the persistent storage medium. This allowed us to deploy and benchmark our complete system on a resource-constrained device, providing a realistic assessment

of the performance overhead of our randomization and decryption processes, the results of which are presented in the analysis that follows in the upcoming section.

6 Analysis of results

6.1 Testing Criteria

Our performance criteria were designed to evaluate the system's impact on the user, as outlined in our requirement for low overhead (**R4**). To do this, we selected a range of test applications, from large, multi-function BusyBox applets (≈3.4 MB) to small, single-task C programs (≈ 81 KB, available in Appendix A), to ensure our results were representative. All binaries were CPU-bound and non-interactive, a choice that avoids interference from I/O wait times. Since the overhead introduced by Preshuf's randomization and decryption logic is almost entirely computational, through this selection of binaries we make sure that our measurements accurately reflect the latency attributable to our system, with as little I/O "noise" as possible.

For full transparency and to observe what constitutes the overhead of our system, we decided on two measurements: the TA processing time, which is meant to show the direct overhead of the secure logic in the Trusted Application, and the total execution time, which measures the final, user-perceived latency. In order to achieve better statistical power, we have averaged over 50 iterations per test case and made sure to conduct these tests under ideal conditions and with a persistent daemon connection so as we are able to avoid the overhead of establishing a connection with the TEE. We also recorded these values through separate batches to guarantee the validity of our findings.

6.2 Performance Evaluation

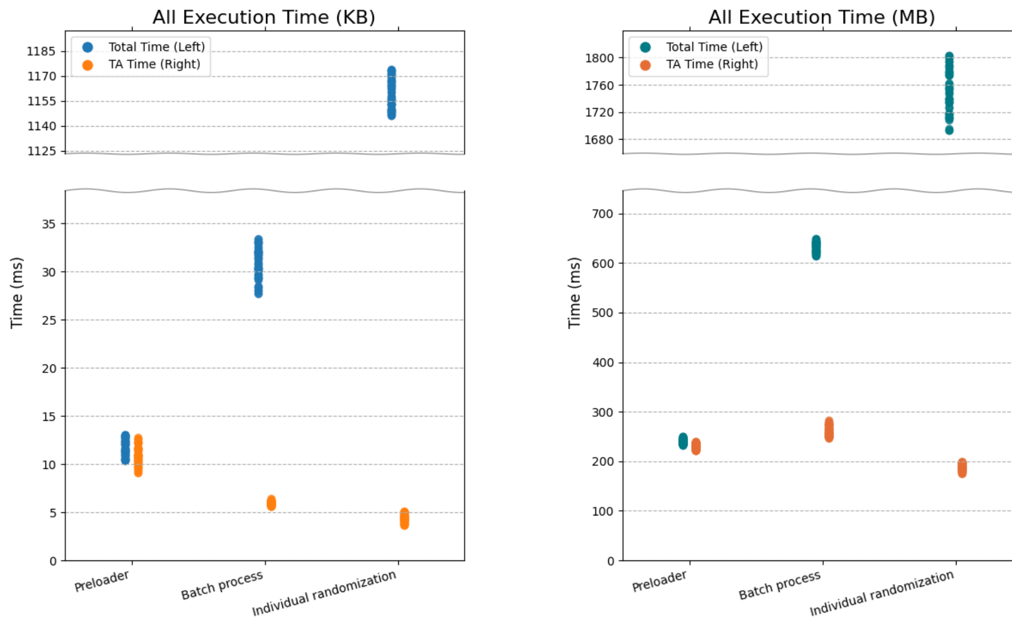
(a) Performance Metrics for Small Binaries in Milliseconds

Metric	Operation	Lowest	Mean	99th Percentile	Highest
Total Time (ms)	Preloader	10.36	11.87	13.02	13.03
	Batch Process	27.70	30.73	33.35	33.40
	Individual Randomization	1146.24	1159.95	1173.35	1173.75
TA Time (ms)	Preloader	9.09	10.85	12.69	12.78
	Batch Process	5.60	5.95	6.30	6.40
	Individual Randomization	3.64	4.41	5.11	5.11

(b) Performance Metrics for Large Binaries in Milliseconds

Metric	Operation	Lowest	Mean	99th Percentile	Highest
Total Time (ms)	Preloader	231.89	240.71	249.68	249.79
	Batch Process	614.10	632.14	647.29	648.80
	Individual Randomization	1692.65	1751.60	1803.22	1803.43
TA Time (ms)	Preloader	221.54	230.13	239.41	239.86
	Batch Process	247.10	261.64	280.89	282.80
	Individual Randomization	175.34	188.40	198.66	198.79

Figure 14: Performance overhead analysis using QEMU in milliseconds (ms).



(a) Performance on small binaries (KB). **(b)** Performance on large binaries (MB).

Figure 15: Comparative performance analysis of Preshuf on QEMU.

The analysis in Figures 14 and 15 encompass three main operations: the Trusted Application randomization process, the preloader's addition to overhead, from interception until the execution the program, and the background processing cycle of the batch daemon on a per-randomization basis. We presented the resulting statistics in both tabular and graphical formats, with the overhead in milliseconds. The measurements above are shown with an accepted margin of error of up to a tenth of a millisecond for inherent system timing variations.

As it can be seen, the preloader's load-time latency averages 11.87ms and 240.71ms for small, and respectively, large binaries. This overhead is fundamentally bounded by the cryptographic primitive, as the expensive ELF parsing and permutation logic is handled offline. This represents another direction, drifting away from individual randomization, where the entire sequence must be executed synchronously, given its impracticality for on-demand use due to higher average times. That being said, the batch process represents a further optimization to the randomization process, and it achieves a lower processing time by keeping a persistent TEE session. This then amortizes the cost of the secure channel setup across multiple operations and allows for a more efficient, continuous data flow in the longer run.

While QEMU provides a consistent baseline for performance evaluation, its results must be contextualized due to its nature as a cross-architecture emulator. The test environment runs on an `x86` host, which relies on dynamic binary translation through its Tiny Code Generator [14] to execute the ARMv8-A instructions of the guest. This process introduces a per-instruction translation overhead not present on native hardware, making direct performance extrapolation unreliable. Having that in mind,

in order to obtain realistic metrics, the evaluation was extended to a physical device: a Raspberry Pi 3 Model B, a platform representative of the target embedded systems, and a device commonly used in Huawei's HSSL. The following analysis will compare the performance data from both the emulated and native environments.

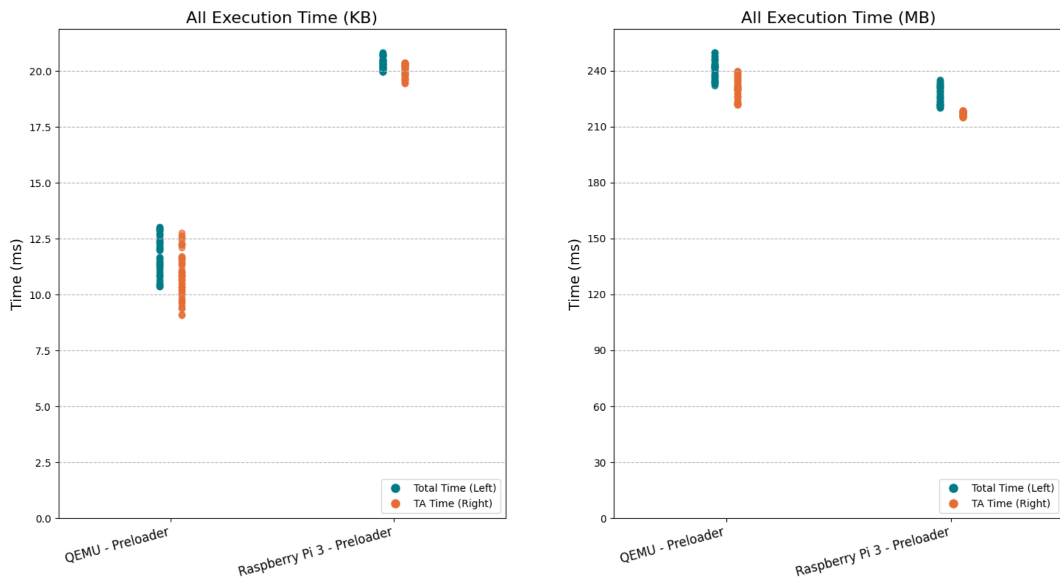
(a) Preloader Performance on Small Binaries (KB)

Platform	Metric	Lowest	Mean	99th Percentile	Highest
QEMU	Total Time (ms)	10.36	11.87	13.02	13.03
	TA Time (ms)	9.09	10.85	12.69	12.78
RPi3	Total Time (ms)	19.97	20.37	20.83	20.84
	TA Time (ms)	19.99	19.45	20.38	20.39

(b) Preloader Performance on Large Binaries (MB)

Platform	Metric	Lowest	Mean	99th Percentile	Highest
QEMU	Total Time (ms)	231.89	240.71	249.68	249.79
	TA Time (ms)	221.54	230.13	239.41	239.86
RPi3	Total Time (ms)	220.03	226.54	234.72	234.96
	TA Time (ms)	215.01	216.99	218.76	218.77

Figure 16: Comparative performance analysis of the preloader on QEMU vs. RPi3.



(a) Preloader comparison (KB)

(b) Preloader comparison (MB)

Figure 17: Comparative performance analysis of the preloader on QEMU and RPI3

The data from Figure 17 suggests a performance inversion, primarily being determined by their own bottleneck at each scale. Particularly, for smaller binaries,

any Input/Output operations reliant on the storage will be affected by its limitations, meaning that in our case, the Raspberry Pi 3 with a microSD card is bound to deliver results with a higher latency than our QEMU emulation, which resides on an SSD. In contrast, for large binaries, the task becomes compute-bound, thus being dominated by the AEAD decryption. More specifically, in here, the Raspberry Pi 3 executes the ARM cryptographic instructions natively, whereas QEMU must use its Tiny Code Generator to translate each ARM instruction into host-native x86 code dynamically, therefore adding per-instruction overhead [22, 74]. While the host CPU is faster, the cumulative cost of such translations for a large cryptographic workload outweighs the initial advantage concerning the Input/Output, thus, making the execution on the Raspberry Pi 3 more efficient.

6.3 Performance measurement

The implementation of the background daemon directly addresses requirement **R4**, of low performance overhead. In our minimal test environment, the process consumed on average 5–7% of the CPU resources while active, reaching up to 10% in exceptional cases. This resource consumption is largely attributable to the constrained hardware; for comparison, the host-side daemon algorithm on a modern development machine consumes less than 2% CPU for the same tasks. Moreover, to further satisfy the transparency requirement **(R5)**, the daemon incorporates a system load throttle. Particularly, this component actively monitors CPU usage and suspends all randomization operations whenever utilization exceeds a set limit. By restricting its activity to periods of low system demand, the daemon therefore avoids introducing any perceptible latency or performance degradation, making its operations unnoticeable to the user.

6.4 Security Evaluation

The security posture of the Preshuf architecture is evaluated by its capacity to neutralize the adversarial capabilities defined in the Threat Model (Section 3.1). The primary objective of this system is to prevent code-reuse attacks by invalidating the attacker's knowledge of the memory layout of a vulnerable binary file, and so, the following evaluation will structured into three parts, each one of them targeting a different defensive layer. Specifically, the integrity and unpredictability of the randomization process, the confidentiality of the binary at rest, and the security of the runtime decryption and loading sequence.

Security of randomization The security guarantees of Preshuf are split between the TEE and Normal World OS, where the TEE's hardware-enforced isolation makes certain that the randomization process is confidential and integral. Once the preloader executes the plaintext binary through `fexecve`, however, the resulting process runs within the Normal World's security context. The issue with this is that if an intruder in the system is capable of escalating privileges to root and attain the `CAP_SYS_PTRACE` capability [101], they can use the `ptrace` [102] system call and gain read-write access

to the process' memory and register, thereby ultimately being able to map the currently active memory layout and identifying gadget addresses for a JIT-ROP attack [43, 138].

While this is a known limitation, the Preshuf architecture mitigates it by rendering the attacker's intelligence ephemeral. The core mitigation for this is a refreshing defense, where the background daemon periodically replaces the on-disk binary with a freshly-randomized version. This process of atomic replacement guarantees that the memory layout of the next application instance will be different and unpredictable. An attacker, therefore, cannot develop a universal exploit; the JIT-ROP payload they might craft for one process would then be useless for the next. The system effectively trades absolute runtime opacity for a model where the cost and complexity of a successful attack are raised considerable, especially against memory disclosures. With that in mind, a discussion regarding further hardening of defenses against live memory analysis are explored in the Future Work, in Section 9.1.

Security of the encryption The security of the encrypted binary is a product of the cryptographic primitives and the secure architecture in which they are deployed. The choice of ASCON, an AEAD cipher, provides a unified defense against both offline analysis **(AC2)** and tampering **(AC5)**. This then yields us a mixture of confidentiality and integrity, which is achieved through a persistent, TEE-managed key, and respectively, a 16-byte authentication tag. This type of protection means that an attacker with file-system access is left with an opaque blob of data that they can neither read nor modify without detection.

The strength of this system comes from its holistic design. More precisely, the key never leaves the TEE, nonce reuse is prevented by generating a fresh random suffix for each encryption, and the file format itself being self-contained, providing the preloader and TA with everything needed for a secure decryption. Moreover, the use of the magic number header plays a role in both identification for decryption, and optimization of the identification phase, not requiring the decryption to begin its mathematically-intensive decryption in order to decide whether it was the correct file. An attacker is therefore comprehensively blocked from any offline integration with the binary. They cannot reverse-engineer it to find gadgets, and they cannot alter it to bypass the randomization, thereby limiting their capabilities to only being able to engage with them at runtime, under the constrained conditions that were described in the previous section.

Preloader to prevent destructive attacks The preloader is the runtime control mechanism in the the Preshuf architecture and it creates a security boundary between the encrypted on-disk artifact and the in-memory running process, thus making it a non-bypassable component for protected binaries. An important architectural distinction to keep in mind is that the preloader is a cryptographic client, not a provider; meaning that is requests services in form of decryption from the TA, which then executes the operation within the Secure World. The main security feature of the preloader is its use of a diskless loading process that makes use of the `memfd_create` and `fexecve` such that the plaintext binary will never be written to persistent storage, and the preloader's

own memory is the only place where the plaintext binary can be found. This leaves the preloader's own memory as the only theoretical attack surface for a privileged attacker, who would have the capability of capturing the transient plaintext buffer by using `ptrace`. This, however, doesn't expose the system's cryptographic keys, and above all, the intelligence gained would be ephemeral, valid only for a brief moment before the process image is replaced, making successful interception a matter of a difficult, real-time race condition.

6.5 Analysis of Randomization Effectiveness and Attack Complexity

The entropy of our randomization process is a measure of the combinatorial complexity an attacker must overcome in order to achieve their goal. As such, while the total number of possible memory layouts for a binary with N RUs is $N!$, providing a theoretical maximum entropy of $\log_2(N!)$, this figure does not represent the practical challenge for a targeted attack, which does not not require a thorough understanding of the entire layout. Rather, a better and more accurate way of taking measure of the entropy of the randomization process is to take into account that code reuse attacks involve finding a chain of k individual gadgets, each residing in different RUs. The challenge in finding all k of these RUs in their appropriate, permuted locations is thus a problem of permutations, which makes the effective entropy that an attacker must overcome in finding this particular sequence of RUs more accurately described by the $\log_2(N!/(N-k)!)$ formula, where N is the total number of RUs and k is the number of RUs required for the exploit chain.

That being said, this model, while useful for quantifying the combinatorial complexity, represents an idealized scenario. Its accuracy is contingent on several assumptions about both the target binary and the attacker's methods. An example of that would be that the model presumes that an adversary's required gadgets chain must span k distinct RUs. An attacker could circumvent this defense if a single, sufficiently complex RU contains all the necessary gadgets for a complete exploit. Moreover, the model also further assumes that gadgets are unique; should the same useful gadget exist in multiple RUs, an attacker's search is simplified, as any instance will suffice. An adversary, for instance, may also not require a specific, fixed set of k gadgets, but can instead construct functionally equivalent ROP chains from a larger pool of available alternatives, further reducing the search complexity. It is under the constraints of our defined "guessing" attacker, who lacks arbitrary memory disclosure, that this entropy calculation holds most strongly. An attacker with such a disclosure primitive could locate one gadgets and then scan the local memory region to find others, therefore bypassing the permutation defense.

In addition, the effectiveness of a randomization scheme is defined by the spatial distribution of its protected components. To analyze our fine-grained permutation defense in isolation, we disabled the operating system's main base ASLR, sampled 5000 runs, and generated a scatter plot for both the addresses of the main executable and other program functions, as seen below in Figure 18. The benchmarks were carried

out on code with 8 functions, of similar sizes (source code available in Appendix B.1) and the interaction of this, our permutation logic, and the system's memory allocator, resulted in a seemingly-distinct offset in our final plot. That is, because while the permutation itself is cryptographically random, the allocator's algorithms for finding a contiguous virtual memory space for the `.text` segment are not. The visible vertical gaps are an artifact of this allocation process, where functions of common sizes are permuted but still collectively occupy a memory region whose placement is not truly random within the application's memory space. Preshuf's permutation is therefore successful in its primary goal of breaking the internal structure of the binary, but the plots showcase that true spatial unpredictability requires further synergy with either the internal structure of the binary through artificial padding, or alternative methods.

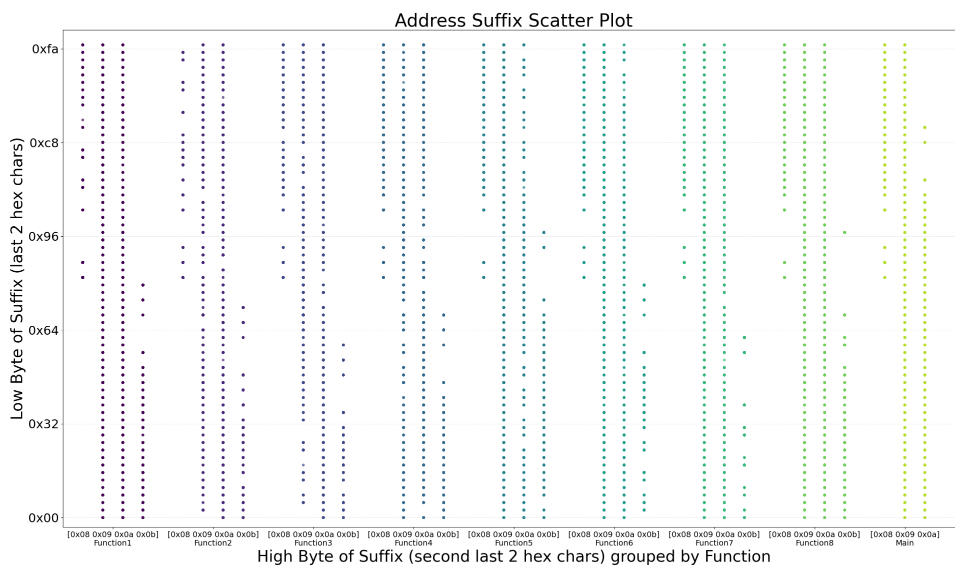


Figure 18: Distribution of Function Addresses under Preshuf's Fine-Grained Randomization without OS ASLR

As per Figure 18, for an attacker to guess the address one a specific function in our sample code, they would, on average, have a 1-in-170 chance, assuming a uniform distribution. This however, represents an idealized scenario. The practical distribution of randomized addresses is often influenced by the non-uniform behavior of the system's memory allocator, a subtlety that is not apparent at first but can be noticed in Figure 19

In addition to the above, one way to further strengthen the efficiency of randomization and therefore increase the entropy significantly is by integrating Preshuf's fine-grained randomization with the system's native base address randomization. This is achieved by letting the OS apply a single, large random delta to the `mmap` base address of the `.text` segment through its base ASLR, while Preshuf independently permutes the RUs within that mapped region. Combining the two would therefore solve the regional predictability of the memory allocator, while our internal shuffling would then destroy the fixed relative offsets that base ASLR alone would preserve. The results, shown in the now scatter plots below (Figures 20 and 21), represent a

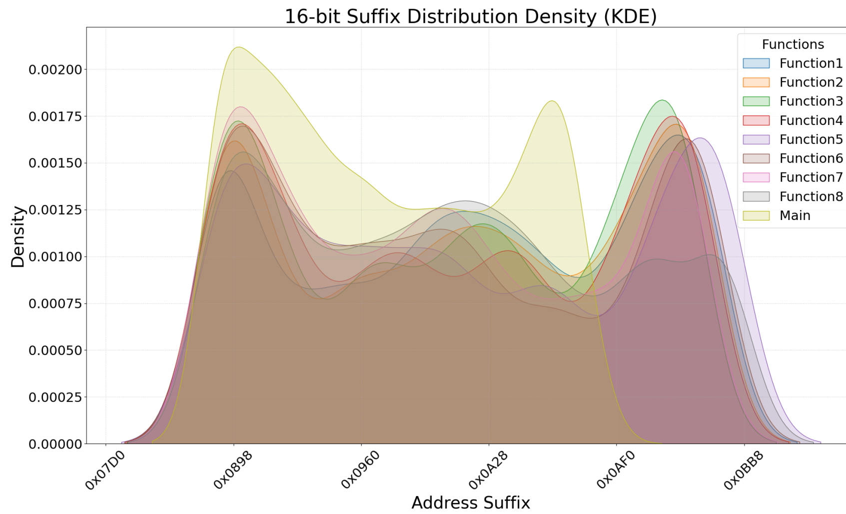


Figure 19: Preshuf Distribution KDE

fully comprehensive randomization where function locations are both unpredictable and closer to being uniformly distributed.

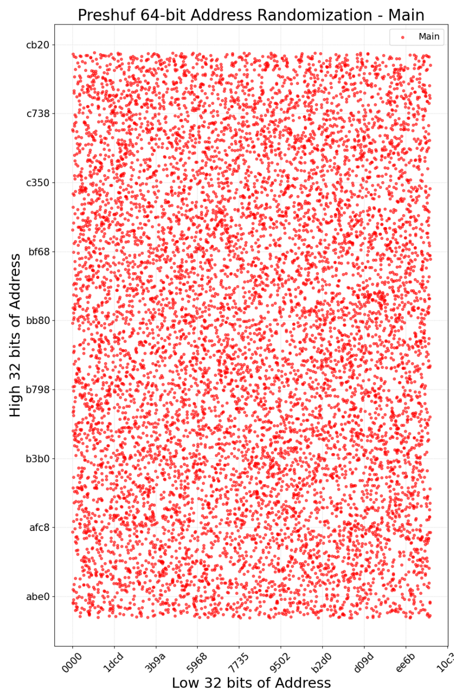


Figure 20: Distribution of the Main Executable's Entry Point Address

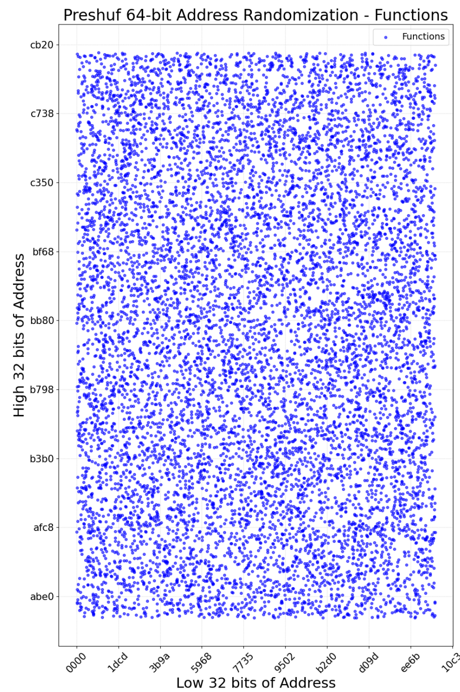


Figure 21: Distribution of Program Function Addresses

At first glance, the distributions in Figures 20 and 21 appear completely identical, however, this visual congruence is due to the large scale of the 48-bit address space depicted. The Y-axis, representing the high 24 bits of the address, spans a multi-terabyte range, while the X-axis covers the 16 megabytes of the lower 24 bits. On this

macroscopic scale, the internal re-ordering of functions, which can result in address changes of about a few thousand bytes, is visually negligible. As such, the plots do not show a lack of distinct randomization, but rather, they show that the OS's base ASLR, despite not altering the internal structure of the file, provides a dominant displacement across the virtual address space, thereby creating a high-entropy baseline for our internal permutation. To illustrate this fine-grained difference, the following figure (Figure 22) presents a highly magnified view, comparing the localized placement of a single randomized function with the main function from each dataset.

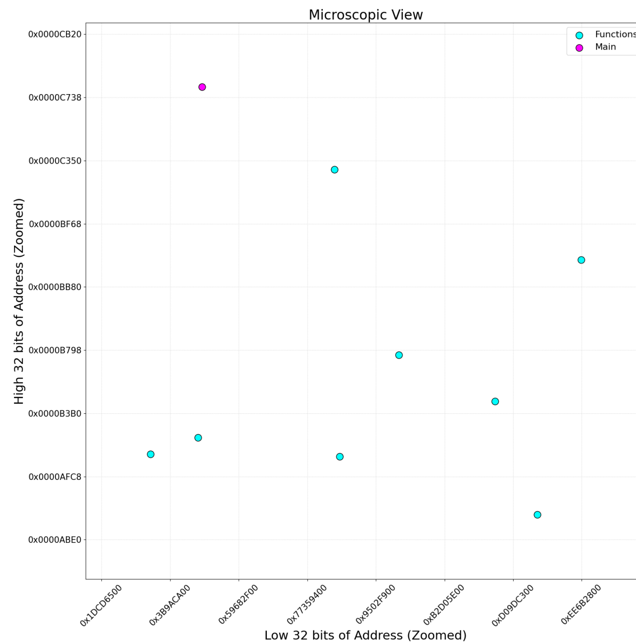


Figure 22: Microscopic view of Main and Functions

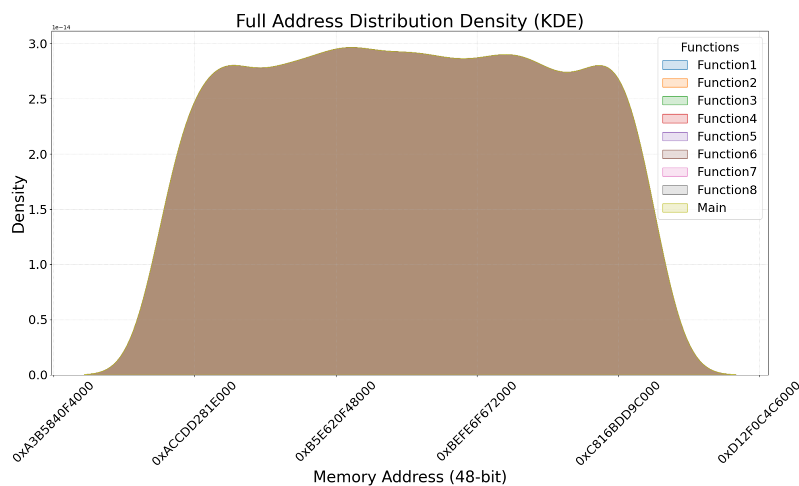


Figure 23: ASLR+Preshuf Distribution KDE

To quantify the effectiveness of this randomization beyond a visual inspection, we

ought to analyze its statistical distribution. The following Kernel Density Estimation plot (Figure 23) visualizes the probability density of the randomized addresses and allows for a quantitative assessment of its uniformity, and ultimately, resilience against probabilistic attacks.

Figure 23 also confirms the high quality of the randomization, visualizing a probability density function that is nearly uniform across a vast portion of the address space. The broad, “flat plateuau” directly indicates that the addresses are not concentrated in any specific spots, but rather, almost evenly distributed all-throughout the address space range, ultimately meaning that an attacker gains no probabilistic advantage when attempting to guess a location at large. Moreover, the overlapping distributions for all individual functions further demonstrate that the randomization is applied consistently, without any sign of biases for specific parts of the code.

6.6 Formal Adversarial Models and Notation

In order for us to be able to formally evaluate the security of our system against brute-force attacks, let us define both the variables and conceptual models that will then form the basis of our analysis.

Table 3: Adversarial Models and Probabilistic Concepts

Concept / Model	Description and Rationale
Adversarial Model 1 (MA1)	Models an attack against a single, static process where the memory layout is fixed. This scenario represents the baseline security of the permutation itself, without the benefit of the refreshing defense (i.e., a minimum security guarantee)
Adversarial Model 2 (MA2)	Models Preshuf’s refreshing defense. The memory layout is periodically reset, invalidating the attacker’s knowledge after a set time. This is the more realistic model for attacks that occur over any significant period
Sampling without Replacement	The probabilistic process governing MA1. Each failed attempt eliminates one possibility from the search space, providing the adversary with perfect, cumulative information.
Trial Block	Represents a sequence of attempts that an adversary can make within a single time window, during which the memory layout is static
Geometric Distribution	The probabilistic process behind MA2 at the level of trial blocks. Since each block is independent due to re-randomization, the number of blocks until success follows this “memoryless” distribution

Table 4: Mathematical Notation

Symbol	Meaning	Context and Role
R	The number of Randomization Units (RUs).	The input variable with a minimum granularity of a function. The combinatorial security of the system which scales factorially with R.
N	Total number of possible memory layouts.	The size of the total search space the adversary must overcome, defined as $N = R!$.
X	Random variable for the total attempts until success.	The primary value we seek to quantify. A higher expected value for X indicates a stronger defense.
k	An index representing a specific attempt number.	A variable used throughout the summations to denote one particular trial.
$E[\cdot]$	The Expected Value operator.	Calculates the average outcome of a random variable over many trials.
ΔT	The time interval for re-randomization.	The duration a specific memory layout remains static before Preshuf's daemon refreshes it.
M	Maximum attempts possible within ΔT.	Quantifies the adversary's speed relative to the system's refresh rate. M is a function of attacker capability.
K	Random variable for the number of trial blocks.	In the refreshing model (MA2), this represents the number of ΔT windows required for success.
p_M	Probability of success within a single trial block.	The chance an adversary succeeds within one ΔT window, given they can perform M attempts.

MA1: Attack on a Static Process Instance This static attack is equivalent to sampling without replacement, as each incorrect guess is removed from the set of possibilities, so the probability of success on any specific trial is uniform. The probability mass function is therefore $P(X = k) = \frac{1}{N}$, and the expected number of attempts is the mean of this discrete uniform distribution:

$$E[X] = \sum_{k=1}^{N} k * P(X = k) = \sum_{k=1}^{N} k * \frac{1}{N} = \frac{1}{N} \sum_{k=1}^{N} k$$

Using the formula for the sum of the first N integers, this resolves to:

$$E[X] = \frac{1}{N} * \frac{N(N+1)}{2} = \frac{N+1}{2} = \frac{R!+1}{2}$$

From an adversary's perspective, this means their task is to systematically guess

their way through an insurmountable search space, with their expected success point being halfway through. This means, that for any program that contains more than a few dozen functions, the necessary effort will go far beyond the reach of modern computation, ultimately making the static layout theoretically undiscoverable by blind guessing.

MA2: Attack Across Refreshed Process Instances In this model, the re-randomization after time ΔT creates windows of opportunity. We define M as the maximum number of attempts an adversary can execute within a single interval. With that, the attack essentially becomes a sequence of trial blocks, where the likelihood of success within a single block, p_M, is derived from sampling without replacement, and thus is:

$$p_M = 1 - P(\text{failure in M attempts}) = 1 - \frac{N - M}{N} = \frac{M}{N}$$

The overall attack can be modeled as a sequence of these blocks. Let's let the random variable K be the number of blocks required to achieve success. K follows a geometric distribution with success probability p_M, and so, the expected number of blocks is:

$$E[K] = \frac{1}{p_M} = \frac{N}{M}$$

The total number of attempts, X, is the sum of attempts in all failed blocks plus the number of attempts in the successful block. The $K - 1$ failed blocks contribute M attempts each. Within the final, successful block, the successful attempt is uniformly distributed from 1 to M, with an expected value of $\frac{(M+1)}{2}$. Using the law of total expectation we would have:

$$E[X] = E[(K - 1)M + \frac{M + 1}{2}]$$

By the linearity of expectation, this then becomes:

$$E[X] = (E[K] - 1)M + \frac{M + 1}{2} = \left(\frac{N}{M} - 1\right) M + \frac{M + 1}{2}$$

$$E[X] = N - M + \frac{M}{2} + \frac{1}{2} = N - \frac{M - 1}{2}$$

$$E[X] = N - \frac{M - 1}{2} = R! - \frac{M - 1}{2}$$

Otherwise said, in practical terms, this equation confirms that the attack cost remains factorially high under Preshuf's refreshing defense model. The term $\frac{M-1}{2}$ amounts to a negligible discount on this cost, representing the limited benefit an attacker gains from working on a static target for a brief period. Moreover, the principle behind our system still holds, namely that the attacker's progress is confined to a singular ΔT window of time, which, upon expiration, will erase all the progress.

Therefore, unless an attacker can perform a number of attempts M that is on the same order of magnitude of $R!$ itself within the interval ΔT, a condition that is practically unfeasible, they cannot succeed in crafting an attack.

7 Related Work

The development of defenses against memory exploitation is a result of the long arms race history between attackers and defenders. The first set of defenses sought to invalidate memory attacks and the eventual static knowledge of a program's structure, however, such defenses are short-lived, with newer tools adapting to bypass them. This section of the thesis aims to cover the spectrum of these techniques, alongside other defenses tangential to our work. The discussion will start with an analysis of foundational ASLR as implemented in Microsoft Windows which will be used as a baseline for our upcoming discussions, then transition to fine-grained randomization schemes that have an approach similar to ours, and then divert onto other diversification strategies that operate at the instruction-set and compiler levels.

Windows Vista's ASLR Windows, as one of the most used operating systems worldwide, introduced its own implementation of ASLR for Windows Vista in 2006 [60]. This implementation of ASLR was provided on a per-image basis, meaning that individual executable files (`.exe`) and dynamic link libraries (`.dll`) could opt into randomization by setting a specific bit (0x40) in their Portable Executable (PE) header's `DLLCHARACTERISTICS` field [75]. This decision to enable ASLR for a given binary is made during compilation, more specifically, through the use of the `/dynamicbase` linker flag. The linker responds to this flag by setting a bit within the PE header's `DLLCHARACTERISTICS` field. This bit's role is to inform the OS's loader that the binary contains the necessary base relocation information to be safely loaded at an address rather than its preferred one. When the loader detects this flag, it applies a boot-session-wide random offset to the image's base address, an offset selected from a pool of 256 possibilities and aligned to a 64KB boundary [97, 45], with it further extending to the initial placement of the process' stack and heap to increase the unpredictability of the newly defined memory structure.

While the ASLR was integrated by default for the core system files in Windows Vista, the opt-in feature applied primarily to third-party application vendors. Moreover, beyond this scope and irrespective of the set bit, the randomization also applied to the locations of the Process Environment Block and Thread Environment Block (PEB & TEB for short), which serve as the primary source of runtime information for a process, and respectively, its threads [160]. In addition, the most notable part about Vista's design was that while the PEB, stack, and heap location varied per program execution, the positions of program code, data segments, basic service set segments, and libraries only changed upon system reboot [160]. The effectiveness of this ASLR was heavily constrained by the 4GB of address space allocated, thereby providing approximately 8 bits of entropy, which made brute-force attacks very feasible [61]. This limitation was later addressed for 64-bit systems with `/HIGHENTROPYVA` support, allowing ASLR to use the entire 64-bit address space for significantly greater entropy.

That being said, as we now know, the primary weakness of base ASLR is its susceptibility to information leakage vulnerabilities, which is the primary motivator behind the various different fine-grained approaches. Should it not be for granular ASLR, a single leaked pointer from a randomized module would be sufficient to

calculate its base address, thereby neutralizing the protection for that entire image. Apart from brute forcing methods [108], this is something that attackers frequently achieve by targetting predictable, unrandomized data structures that contain pointers into protected memory regions. In the case of Windows, the PEB, for instance, can be accessed through the GS segment register on 64-bit Windows, providing a stable entry point for an attacker to walk internal data structures and find the base addresses of libraries such as `ntdll.dll` [55, 8, 152]. An arbitrary-read primitive, even a limited one, can thus be weaponized to either de-randomize a process' address space [72, 66, 130, 167] or craft malicious attacks [147, 21, 134, 127, 148], and with that in mind, the following paragraphs will go over some previously researched, unique frameworks that while successful in defeating base ASLR, would fail against more fine-grained approaches.

This susceptibility to information leakage, which allowed attackers to neutralize the defense with a single known address, did not in itself grant code execution, but rather, it just reset the conditions to a state of a non-randomized binary where the task of identifying and chaining gadgets was still required. To overcome this and achieve a more scalable exploitation, since 2008 [26], offensive security researchers focused on automating the payload construction process, leading directly to the development of Automatic Exploit Generators (generally abbreviated as AEGs). These systems formalize the exploit generation process and treat them like a computational search problem through methods such as symbolic execution to find out the precise inputs needed to hijack a program's control flow [5]. By discovering code gadgets and satisfying the constraints for chaining them, AEGs can autonomously synthesize complete ROP payloads from a proof of concept crash.

Mayhem A first and historical example of this is Mayhem, a tool developed in 2012 [30] that was engineered to be ran on x86 binaries without requiring source code or pre-existing vulnerability information, starting only with a proof-of-concept input that causes a crash. The way it operates is by taking a program crash and, based on the input and metadata, generating an exploitable formula that includes constraints necessary for redirecting the instruction pointer and executing a payload that is then passed to a Satisfiability Modulo Theories solver and if a satisfying assignment is found, it represents the precise input required for a successful exploit [6]. To handle more complex cases where memory addresses depend on user input, a common scenario in binary analysis, Mayhem also introduces an index-based memory model, which is a technique that the authors note was of utmost importance for generating 40% of their discovered exploits.

Revery Another significant AEG, Revery, approaches the problem from a different perspective: instead of finding new bugs, it focuses on verifying the exploitability of a known crash [158]. Revery introduced a technique *"under-constrained symbolic execution"* and it means that the execution would start from the crash location as opposed to the start of the program. This method drastically pruned the search space by only symbolizing data directly relevant to the fault, leaving the rest of the program

state concrete. The system then defines specific exploit primitives, such as control over the instruction pointer, as symbolic goals and uses a solver to find an input that satisfies these conditions, effectively turning a crash proof-of-concept into a working exploit.

EoLeak The previously discussed AEGs excel at payload construction but their effectiveness in more modern environments is completely dependent on the knowledge of the target's memory layout, which ASLR is intended to protect. To counteract this specific preliminary step, researchers at Tsinghua University built EoLeak [166], an AEG that does not have control-flow hijacking as its main goal, but rather, automating the discovery of information leakage vulnerabilities. Its system operates using dynamic binary analysis and taint tracking to monitor the flow of attacker-controlled input [166]. For this, an effective execution path would be one where the tainted input influences the value of a pointer that is then subsequently written to an output channel, effectively leaking a runtime memory address and providing an attacker with the information needed to defeat ASLR.

LAEG & AAHEG Building upon the principle of information disclosure, Leak-based Automatic Exploit Generation (LAEG) integrates this capability into a complete, end-to-end exploit chain for stack-based buffer overflows. The way this operates is by first using a provided information leakage vulnerability to defeat ASLR, and it then performs dynamic binary analysis to precisely calculate the fixed offsets between the leaked address and the required ROP gadgets. With the memory layout de-randomized, LAEG then proceeds to synthesize a full payload [113]. While effective for stack vulnerabilities, this model does not address the significantly more complex domain of heap exploitation, which required manipulating the stateful and non-linear structure of a program's heap. This challenge, however, was tackled a year later through Automatic Advanced Heap Exploit Generation (otherwise called AAHEG), being a specialized AEG that operates on source code rather than binaries. AAHEG parses the code into an abstract syntax tree to build a formal model of the program's heap operations. This allows it to automate the sequences of allocations, frees, and writes necessary to corrupt heap metadata, create overlapping chunks, or trigger a use-after-free condition, which would then lead to control over the instruction pointer [159].

FormatAEG Rounding out the set of automated exploits for the major memory corruption types, FormatAEG reveals another way of exfiltration data, particularly through string formatting vulnerabilities. Moreover, just like the previous tools, FormatAEG also integrates ASLR bypasses directly into its exploitation process. However, while the previously discussed AEGs address stack and heap overflows, format string bugs bring their own special challenges and opportunities. FormatAEG operates by encoding a call to a write function such as `printf`, and gradually building payloads with format specifiers like `%x` and `%p` to read raw data directly from the stack, thereby leaking saved return addresses and stack canaries to bypass ASLR. Once aware of the memory layout, FAEG would build a final payload through the `%n`

specifier that would then produce an arbitrary write, most commonly overwriting a stored return address or function pointer in order to achieve control-flow hijacking

Hacking Blind The aforementioned AEGs all demonstrate a different approach to the automation of exploitation once a memory address is known. As opposed to this, another methodology is the concept of "hacking blind" [21], which challenges the assumption that an information leak is even necessary to obtain such an address. While this approach initially worked by using the termination state of a forked process, whether crash or not, as a side channel. Modern variants of it make use of transient execution attacks, where a speculative memory access to a guessed address can leak information through a microarchitectural side channel (e.g., cache timing) without faulting, thereby avoiding the need to crash the process. Despite the increased sophistication of the side channel, the fundamental goal still remains finding the base address. The success of the attack though, still hinges on the weakness of base ASLR, which is that once this address is discovered, the internal layout of all functions and gadgets is static and predictable.

This approach applies to network servers that handle requests in forked processes and works by brute-forcing the randomized base address of a shared library. The attack uses the process's termination state (crash or no crash) as a side channel to validate its guesses. The success of this technique does however hinge on the key weakness of ASLR, which is that once the library's base address is discovered, the internal layout of all functions and gadgets is static and predictable.

The success of all the above-described AEGs is entirely dependent on the principle of relative address constancy; namely that whether an attacker obtains a pointer via a leak or brute-forces a base address, they can reliably calculate the location of any required gadget using a known, fixed offset. One of the benefits of fine-grained randomization is that it targets this assumption by introducing a high degree of entropy within the code segment itself. As we have previously discussed in this thesis, permuting the order of functions implies that the offset between any two randomly selected functions is guaranteed to be a random variable between different executions, and not a constant. Fundamentally, this will break the attack chain and force the attacker to discover the location of every single necessary ROP gadget independently, a task with a factorial complexity directly proportional to the number of gadgets required. This then implies a significantly greater cost of exploitation, up from finding one piece of information to finding n times as many, all with a difficulty almost as high as the previous, thereby neutralizing the automated and scalable nature of these tools

Moreover, on top of the previously mentioned fine grained code randomization tools in 2.2.2, three other such systems pique our interest, primarily due to their relevance to our work. Starting with Marlin, which provides fine grained code randomization by permuting functions at program load time. The system ensures that the code will stitch the reordered components back together, thus breaking the contiguity of the original program layout [68], thus defeating code-reuse attacks through an approach which is fairly similar to ours. Moreover, just like our original implementation, its main

limitation is that the randomized layout is static throughout execution, offering a single secret that can be broken; a factor which acted as a motivator for our preloader-centric framework. As a last remark, the taken approach for benchmarking Marlin was one similar to ours, presenting an overhead of 0.87s on average [69], more than twice as high as ours.

FASLR is another protected ASLR scheme related to our work, this time the similarity being how its operations are being done within the TEE. More particularly, the way FASLR functions is by first loading the application into an enclave and then randomizing its memory layout so that its randomization will then be "shielded" against a compromised host OS, a threat model also similar to ours. FASLR, however, applies this randomization at functional granularity, randomizing the entry addresses of individual code elements as they are called, rather than rewriting a full binary statically [107].

Lastly, Mardu features another interesting design choice through its implementation of a continuous re-randomization defense by refreshing the code layout at runtime. Mardu operates on-demand and thus triggers a new randomization pass upon detecting a security-sensitive event, such as a program crash or an eXecute-Only Memory (XoM) violation. This mechanism is designed to invalidate address leaks over time, offering a security advantage against attackers with a long-term presence in the system [78, 79]. In addition, Mardu's on-demand approach to randomization sparks the question of whether we can further optimize our system by introducing either on-demand or on-page-fault decryption as opposed to load-time to minimize the waiting time of the end-user.

Instruction Set Randomization Moving away from defenses that randomize the location of existing code, another branch of research focuses on diversifying the content of the code itself in order to defeat a different attack vector; namely, direct code injection. This category of defense is exemplified by Instruction Set Randomization (ISR for an abbreviation), and it is a technique that was designed to prevent an attacker from executing previously-injected shellcode [82, 24]. The main idea behind ISR is to assign a unique, secret key to each running process, which is then used to transform, typically via a fast XOR operation, the opcodes of all instructions before they are loaded into memory. A supposed threat actor's injected payload, which is encoded with the standard, public instruction set, would therefore be meaningless to the protected process [11].

Making this possible is done through the processor's instruction-fetching mechanism, which must be modified to apply the inverse transformation, therefore using the secret process key to decode the randomized opcodes back into their original form just before execution. When the CPU attempts to fetch and decode the attacker's injected code, it will apply the secret key to the standard opcodes, resulting in a stream of invalid or nonsensical instructions that in almost all cases will lead to a program crash rather than successful exploitation [58]. This method aims to provide a stronger defense against attacks that rely on writing and executing new code in memory, though its practical adoption has been limited by its requirement for specialized hardware

support or a significant performance overhead from software-based emulation [34].

The defense model of ISR is particularly relevant to our work as it addresses a new threat vector–direct code injection–that code layout randomization alone does not. Preshuf is designed to defeat code-reuse attacks by making existing code hard to find, while ISR is designed to defeat code-injection attacks by making new code impossible to run. In a comprehensive security architecture, these two techniques are therefore complementary, not competing, thus leaving room for questioning on whether our preloader-centric defense could be altered in a way that it would not only prevent code reuse, but also code injection. Nevertheless, this principle of diversifying the code's content, rather than just its location, also forms the basis of another major category of defense known as Compiler-Based Diversification, which we will discuss more about in the coming paragraphs.

Compiler-Based Diversification Compiler-Based Diversification can be understood as a practical implementation of a "moving target defense", but one that operates across a population of deployed software rather than within a single running process. Just like with code reuse, this approach addresses a risk of software monoculture, where an attacker can develop an exploit on their own machine with the certainty that it will work on a plethora of other different devices [57]. By integrating randomization into the compiler, distributors can generate thousands of unique binary variants for the same application. Each user, organization, or distributor, would therefore receive a different variant, ultimately, making the software population heterogeneous [77].

This diversity is created through various randomized compiler passes, where, for instance, the compiler could permute the order of functions in the resulting binary [88] in a manner like Preshuf, randomize the allocation of registers [92], or even substitute equivalent instruction sequences [157]. The security benefit for this one is probabilistic, in a sense that, an attacker who develops an ROP exploit for one variant has a negligible, but not zero, chance of that exploit succeeding on any other variant deployed in the wild [35]. While this does not protect a single, compromised instance from runtime analysis, it breaks the model of scalable, widespread attacks and forces adversaries to re-target each individual machine, significantly increasing the cost and effort of exploitation just like the aim of Preshuf.

7.1 Benchmarks of related approaches

These defensive strategies present us different kinds of performance overheads, differing from implementation to implementation. Fine-grained randomization schemes like marlin or Mardu impose a significant cost at application launch or runtime, as they must perform various computationally intensive and expensive binary analyzes, permutations, and relocation fixings. Preshuf's asynchronous model attempts to mitigate this by moving these heavy tasks to a background process, leaving only the decryption at load time, whose overhead heavily depends on the size of the binary file itself. Instruction Set Randomization, on the other hand, presents a starkly different profile however, as, with specialized hardware, the overhead is a minimal, with

constant cost added to every instruction decode cycle. Without such hardware though, the performance penalty of software-based emulation would become prohibitively high for practical use. In contrast, Compiler-Based Diversification moves the entire computational burden to the software developer at compile time. The end-user would experience no direct load-time randomization overhead, and any performance impact would be a static artifact of the diversified code generation, which may result in very minor execution inefficiencies due to less optimal instruction sequences, or altered cache locality. That being said, a more thorough insight into these overheads can be seen below, in Table 5.

Table 5: Comparative Analysis of Code Diversification Defenses by Performance Overhead

	Scheme	**Granularity**	**Dependencies**	**Overhead**
Base ASLR	PaX ASLR [120]	Memory Regions	OS Kernel Support	<0.1% On-Load
	ASLR-NG [109]	Memory Regions	OS Kernel Support	<0.1% On-Load
	Apple	Memory Regions	OS Kernel Support	<0.1% On-Load
	Android [162]	Memory Regions	Zygote Model	<0.1% On-Load
Fine-Grained	Bhatkar II [19]	Functions	Source Code	~5-15% Per-Call
	ASLP [86]	Functions, Sections	Relocation Info	~1% On-Load
	Marlin [68]	Functions	Relocation Info	~870ms On-Load
	FASLR [107]	Functions	TEE / Enclave	<10% On-Load
Leakage Resistant	Readactor [41]	Code Pages	Virtualization (EPT)	3-7% (Runtime)
	Mardu [79]	Functions	Binary Rewriting	2-5% (Runtime)
	Isomeron [43]	Functions (Isomers)	DBI	5-20% (Runtime)
Content	ISR [82]	Opcodes	Modified CPU	<1% (if hardware) >900% (otherwise)
	CBD [77]	Functions, Registers	Altering Compiler	0% (User-facing)
Our Work	Preshuf	Functions	Relocation Info, TEE	<250ms On-Load as per Figure 16

While the preceding table provides a structured overview, it is important to

mention that creating a direct and quantitative comparison of the reported overheads is complicated by a significant heterogeneity across the cited works. This is primarily attributable to two confounding factors: Firstly, the selection of benchmark applications differs substantially; performance overhead is highly sensitive to the workload, with larger, more complex applications often showing a different relative impact than smaller, CPU-bound programs. Secondly, the evaluation environments, including processor architectures and system configurations, are not uniform. The influence of such environmental factors is something we confirm in our own analysis in Figures 16 and 17, which show a performance inversion between emulated and physical hardware based on the task's bottleneck. The table is therefore best interpreted as a qualitative guide to the nature and timing of the overhead for each scheme, rather than a direct numerical ranking.

8 Discussion

Having presented the design, implementation, and evaluation of the Preshuf system, this chapter moves to a broader discussion of the work. The objective here is to analyze the architectural trade-offs, contextualize the system's security guarantees within the larger landscape of memory safety, and explore potential avenues for future enhancement. The following subsections will begin by addressing the inherent limitations of the current architecture, particularly regarding its dependency on memory confidentiality and overhead. Building upon this analysis, we will then propose and discuss several significant architectural extensions designed to harden the system's integrity, improve its perceived performance, and address practical deployment considerations. This discussion serves to frame Preshuf not as a final product, but as a foundational architectural for a new class of TEE-hardened moving target defense.

8.1 Limitations of the Preshuf architecture

The security guarantees provided by Preshuf are evaluated against the adversarial capabilities defined in the Threat Model in Section 3.1 and a principal limitation of our current architecture is its dependency on the confidentiality of the randomized memory layout post-execution. Preshuf is designed to be effective primarily against an adversary who is limited to guessing attacks, but it does not, by itself, provide comprehensive leakage resistance. With several high-value targets for a disclosure vulnerability, an attacker with access to the function pointer array from the stack could reveal all randomized addresses of all functions and thereby the entire permutation. Moreover, an adversary could target the process' loaded ELF metadata, where, structures such as the GOT, symbol table, or the dynamic symbol table contain explicit mappings between symbol names and their randomized virtual addresses. A vulnerability that permits reading from these sections would effectively nullify the randomization. Preshuf's security model is therefore complementary to, and not a replacement for, defenses that provide memory safety to prevent such disclosures from occurring.

8.2 Enhancing system integrity through attestation

The Preshuf model is structured around the presumption of an intact Normal World loader chain with the main idea that if the attacker breaks the preloader, the system would no longer be functional and thereby them failing in achieving their goal. However, the alternative option could be substituting the preloader with a malicious variant that accomplishes the same task but has additional malicious steps. With both of these outcomes presenting more downsides for the system user as opposed to the attacker, one way to counter this class of threat would be to have the system's integrity be cryptographically verified through a remote attestation protocol. This mechanism aims to extend the trust boundary of the TEE by using it as a hardware-based Root of Trust for Measurement to produce an appraisal of the untrusted host.

The way this would operate is that upon the launch of the operating system, an attestation sequence would be initiated that would invoke the TA. The TA would then

measure a series of specified integrity targets within the Normal World that would then include a cryptographic hash of the preloader binary itself, the configuration state of the kernel's ASLR, and potentially the hashes of critical libraries the system and preloader depend upon. The TEE would then extend these measurements into a secure register and, acting as a Root of Trust for Reporting, it would sign this state with a private key fused into the processor hardware. This signed report, often referred to as a "quote", would be computationally infeasible to forge. This quote would then be transmitted along with a nonce from the verifier to prevent replay attacks to an attestation service. This verifier would then have the goal of comparing the measurements within the quote against a golden reference integrity manifest. Only upon successful cryptographic validation of the quote and matching of the measurements does the verifier release the secrets needed for the system to execute in a manner similar to its previous boot, whereas, if the validation fails, the system would enter a "patching" state, fixing the core structure of our system.

8.3 Architectural tradeoffs in decryption timings

As for the decryption timing, in our current state, we employ a load-time decryption model, where the entire application binary is decrypted and mapped into memory before the actual execution begins. This approach has the advantage of architectural simplicity and predictable performance, in a way that after the initial decryption phase, the application runs at native speed with no further security-related overhead. The primary drawback, however, is the introduction of a user-perceptible startup latency, which is directly proportional to the size of the binary. This motivates a further exploration of alternative models that could distill this computational cost into later stages of the execution and thereby keep the application launch delay to a minimum.

A first alternative is an on-demand, function-granular decryption model where the preloader seeks to map the encrypted binary into memory and then replace the entry point of each RU with a placeholder. This placeholder acts as a stub that would decrypt the target RU in-place, and then patch the placeholder to redirect subsequent calls directly to the now-decrypted code, followed by transferring control back to the function. This architecture would virtually eliminate most startup latency, essentially moving it to appear over the span of the application's entire runtime, and only the functions that are actually executed would ever be decrypted, as opposed to edge-case functions that are unlikely to be executed under normal executions. This potential efficiency gain for large applications with many unused code paths could come at the cost of introducing a small, but potentially noticeable latency on the first invocation of every function, which could manifest as "jitter" in interactive applications.

A second alternative then is on-page-fault, page-granular decryption. This approach would use the processor's Memory Management Unit to mediate decryption. The preloader's goal here would be to map the entire encrypted .text section into the process's virtual address space but mark the corresponding page table entries as not-present. The first attempt to execute code on any given page would trigger a page fault, which would be caught by a custom handler. This handler would identify the faulting page, decrypt its content in-place, update the PTE to mark the page as present

and executable, and then resume normal execution. This model also offers near-zero startup latency and amortizes the decryption cost, with a potentially more effective efficiency optimization than the on-demand counterpart because the interception is handler by hardware. The primary trade-off here is the complexity of implementing the fault handler, which typically requires privileged, kernel-level integration. It still introduces runtime latency, although less frequent, making it occur on the first access to a page of code rather than the first call to every function. The choice between these models would thus be essentially a trade between predictability with front-loaded latency, and distributed, event-driven latency.

8.4 Mitigating metadata storage overhead

A significant practical consideration for the Preshuf architecture is the storage overhead that is introduced by the custom `.PRR` section, which is injected into each binary and contains a complete copy of the symbol table, the string table, and all static relocation entries (e.g., `Elf64_Rela` structures) that are normally discarded by the linker. For a large binary with thousands of functions and cross-references, the size of this metadata can be substantial, often measured in megabytes, however, a more thorough breakdown can be found below, in Table 6 and Figure 24 or in Appendix C. This overhead directly impacts on-disk storage, and for embedded or mobile systems where storage is a constrained resource, it presents a notable drawback. This section discusses several architectural extensions to mitigate this storage cost.

Table 6: Summary of Metadata Storage Overhead after PRR-Injection

Metric	Value
Number of Binaries Analyzed	131
Total Size Before Injection	23.76 MB
Total Size After Injection	38.97 MB
Average Size Increase per Binary	87.24 KB
Average Percentage Increase	61.4%
Median Percentage Increase	61.2%
Minimum Percentage Increase	32.1% (apt-extracttemplates)
Maximum Percentage Increase	85.5% (store)

With Table 6 and Figure 24 showcasing the possible increase in size of our new binaries, we can notice the importance of optimizing this. One way, and possibly the most direct approach of doing so would be through applying lossless compression to the serialized data within the `.PRR` section. The "prr-injection" tool could be modified to compress the metadata blocks (symbols, relocations, and strings) using a more lightweight, high-speed algorithm like LZ4 or Zstandard before they are written to the binary. The choice of the algorithm is just as important as the one for the encryption though, as it must offer a high decompression speed together with good compression,

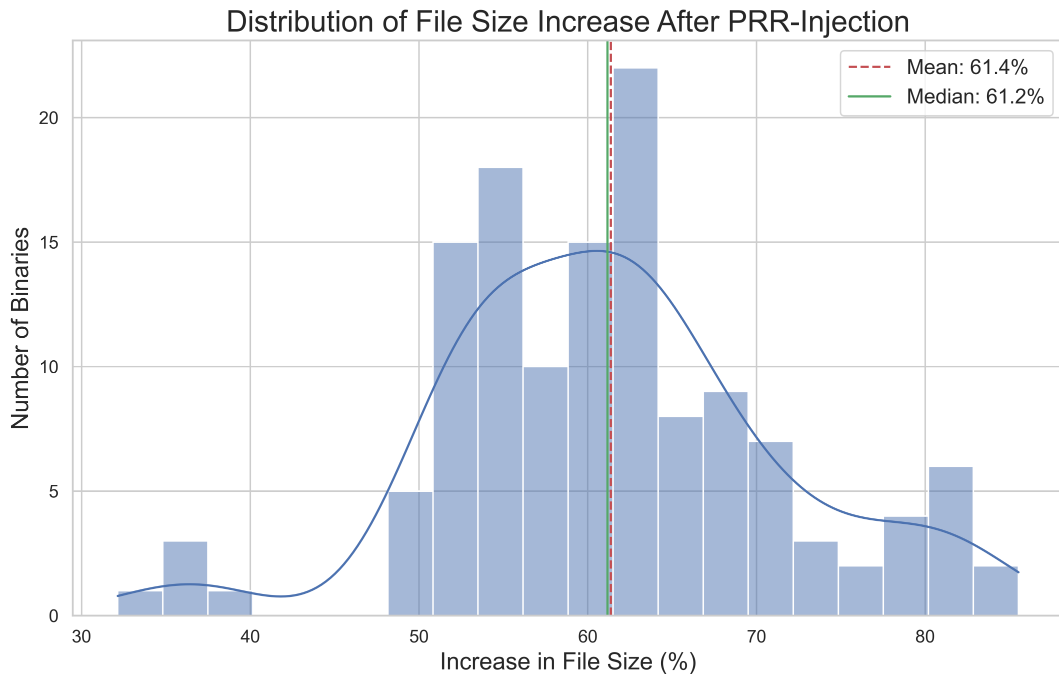


Figure 24: Metadata Storage Overhead Visualized

as this operation would be on the critical path of the randomization process within the TA. Upon receiving the binary, the Preshuf TA would first decompress the `.PRR` section in memory before proceeding with the parsing and randomization logic. This approach would then maintain the simplicity of a self-contained binary but then would add a one-time decompression penalty inside the TEE. Given that the data within this section can often be repetitive (because of the common symbol names, patterned relocation entries, etc.), it is expected to have a high compression ratio, significantly reducing the on-disk footprint.

A different approach to this would be one that is inspired by Zygote, which addresses both storage and performance by externalizing metadata to a central database. The Preshuf daemon would act as a manager for this database, pre-loading the metadata for protected applications into a Normal World memory cache. Going about it this way converts the I/O-bounded task of reading a `.PRR` section into a much faster memory-to-memory copy when a randomization is requested. This would be completed with a second caching tier within the TEE. This way, the TA can create a persistent, secure cache of the metadata it receives for frequently used binaries. Such an approach would have a high likelihood of not only providing lower storage overhead, but also achieving a highly optimized state for repeated operations.

8.5 Final Remarks and the Broader Impact of Preshuf

The Preshuf architecture, as implemented, aims to drift from the original implementations of code randomization and decouple the high computational cost from the performance-sensitive application launch sequence. The broader impact of this is

twofold; namely, from a performance standpoint, it makes fine-grained randomization practical for resource-constrained systems or environments where the latency of traditional on-demand randomization would be prohibitive. From a security standpoint on the other hand, it materializes a refreshing defense that then makes adversarial knowledge long-term ephemeral. An exploit developed against one instance of a binary loses its validity after the next randomization cycle and thus greatly increasing the cost and complexity for the attacker attempting to achieve a persistent compromise.

This model does come with its own set of trade-offs though. It exchanges a one-time startup delay for a persistent, low-level CPU and I/O footprint from the daemon's operation. Furthermore, the entire security guarantee is predicated on the integrity of the Trusted Computing Base, and the system introduces a tangible storage overhead due to the inclusion of the `.PRR` metadata. Preshuf therefore servers as a comprehensive case study in architectural trade-offs, and offers a blueprint for making high-granularity moving target defenses both more performant and secure in real-world systems.

9 Conclusions

Fine-grained code randomization is a probabilistic defense against varied memory corruption exploits that seek to invalidate an attacker's foreknowledge of a program's internal layout. The practical adoption of this has seen hindrances due to fundamental performance problems, particularly surrounding the user-perceptible latency introduced by randomizing a binary at application launch time. Existing approaches either accept this startup delay or delay the cost to other parts, such as the system startup time, or during the runtime of the application. All of these approaches provide overheads that are unsuitable for many modern computing environments.

This thesis presents Preshuf, an architecture designed to resolve this performance-security conflict. The work details an asynchronous pre-shuffling model that moves the computationally expensive randomization process into a background task which is hardware-isolated within the TEE, and operates during convenient times for the system. Ultimately, this shifts the overhead from the randomization process to decryption; a field that sees more optimization potential given the high demand for lightweight cryptography. The system was implemented and evaluated on both emulated and physical systems through QEMU and a Raspberry Pi 3 to provide a realistic assessment of its overhead, and the primary results, in accordance to the research questions, are:

- **RQ1 - Performance Overhead Mitigation:** The asynchronous model mitigates launch-time overhead, with performance analysis showcasing that the latency introduced by the preloader is under 250ms for larger binaries, a noticeable decrease compared to the randomization which is around 1700ms.
- **RQ2 - Transparent and Automated Operation:** The system manages to achieve a transparent and automated defense through a decoupled architecture that separates the randomization logic from the runtime execution path, using an asynchronous daemon that operates with a low CPU footprint and incorporates a system load throttle in order to remain imperceptible to the user
- **RQ3 - TEE Capability and Guarantees:** The architecture confirms that the TEE is a capable environment for executing the randomization logic, and its hardware-enforced isolation protects the integrity and confidentiality of the process against non-root attackers
- **RQ4 - Combined Impact of AEAD Encryption:** The integration of AEAD encryption provides both confidentiality and integrity for the binaries on the system, ultimately defeating the prospect of offline analysis and tampering. The performance impact of this security gain is the decryption latency, a cost that can be measured by the preloader's TA execution time.

These results show that an asynchronous, TEE-based pre-shuffling architecture can deliver the security benefits of fine-grained randomization without the prohibitive startup costs of traditional on-demand models. This approach makes high-entropy moving target defenses more practical for resource-constrained systems, and, based

on this work, we could possibly observe further research directions for optimizing, hardening, or improving such a system.

9.1 Future Work

The Preshuf architecture, in its current form, provides strong at-rest and pre-execution security. That being said, it does have a limitation, particularly, runtime memory inspection. Once a protected process is launched, its memory image, though randomized, exists in plaintext and can be attached to and analyzed. To address this threat, a reactive hardening mechanism can be implemented to convert Preshuf into a hybrid proactive-reactive defense. Depending on our need, two architectural approaches can be taken to implement this detection mechanism, differing in their privilege level and security guarantees.

9.1.1 OS-Level Monitoring through the Linux Audit Framework

This extension would integrate our Preshuf with the Linux audit framework (otherwise written off as auditd [100]). The objective is to detect analysis attempts on protected binaries and trigger an immediate and high-priority re-randomization of the targeted application. Such an implementation would consist of the following components:

- **An audit rule** that will be established to specifically monitor known tools that would allow memory-level inspection, such as GDB [104] and LLDB [106]. This rule, configured via `auditctl`, will be defined to watch for the any debuggers attaching to processes in order to create audit log entries whenever a process attempts to attach to another.
- **A user-space monitoring agent** that will be developed to actively parse the audit logs and filter for events tagged with `preshuf_trace`. Then, upon detecting a relevant event, it will extract the PID of the target process and resolve its corresponding executable path via the `/proc/[pid]/exe` symbolic link.
- **A high-priority randomization trigger** that once the executable path is identified, will inspect for the first few bytes of the file to check for Preshuf's magic header. If it is present, it will send an inter-process communication message to the Preshuf daemon. This message will contain the path to the compromised binary and a high-priority flag, which would allow our compromised binary to stand out from the rest and be immediately re-randomized.
- **An enhancement to the daemon's logic** in order to include a high-priority queue. Upon receiving a high-priority re-randomization request, the daemon will immediately suspend its routine and reschedule the shuffling tasks. It will then fetch the specified binary, invoke the TEE to generate a new permutation and key, encrypt the result, and atomically replace the on-disk binary.

Through such a mechanism, we guarantee that any intelligence gathered by an attacker through means of debugging has a severely limited lifespan, ultimately limiting

the compromised layout's existence to a minimum to significantly lower the attack surface and force the attacker to repeat the process of analysis for each new instance and minimize the time-window in which a JIT-developed attack is effective. That being said, we acknowledge that this reactive mechanism is contingent on the integrity of the Linux audit subsystem, and that an adversary who has attained sufficient privileges could subvert this defensive layer by disabling `auditd` or altering its rules, making this approach useful only for user-level introspection.

9.1.2 Hardware-Enforced Monitoring through Hypervisor-Based Introspection

For threat models that assume a fully compromised guest kernel, not only will the attacker get access to more sophisticated tools such as the `ptrace(2)` system call [105], but the attacker will also have agency over other system features such as the audit logs. As such, the detection mechanism must be elevated to a privilege level that is architecturally superior and more isolated from the OS in order to provide an unavoidable monitoring guarantee. On the ARMv8-A architecture, this can be achieved through Hypervisor-Based Introspection, in a manner similar to [16], where a lightweight hypervisor at EL2 securely monitors the guest OS running at EL1. An attacker with root privileges inside the guest OS therefore would not be able to disable or modify the hypervisor's monitoring, as it is enforced by the CPU hardware. A possible implementation of such a feature would be as follows:

- **Configuring a trap** by making use of the hypervisor's rights, which controls the mapping of guest-physical to host-physical memory. Specifically, it would identify the guest-physical memory page containing the kernel's `sys_ptrace` handler, among other relevant ones, and revoke its execute permissions in the Stage 2 entry. Any attempt by the guest kernel at EL1 to execute code on this page, regardless of its own Stage 1 permissions, will trigger a virtualization fault and trap directly to the Hypervisor at EL2.

- **Intercepting the traps** and analyzing the system's state upon trapping a relevant instruction. Particularly, upon trapping, the hypervisor would introspect the guest CPU's state, examine the state and confirm the cause of the trap and the PID of the target process.

- **Out-of-band reaction** to the TEE granted that the guest OS is considered compromised and it cannot delegate a task back to it. As such, it would execute an SMC to trap the Secure Monitor running at the highest exception level, EL3. The SMC instruction would carry the necessary information, such as an identifier for the compromised binary, and then have the Secure Monitor act as a trusted dispatcher and forward the high-priority re-randomization request to the Preshuf TA running in the Secure World.

While highly theoretical and possibly impractical, such a Hardware-Based Introspection approach would provide a detection mechanism that is both invisible and

immune to subversion by an attacker who has achieved root privileges within the guest OS. The monitoring is enforced by the CPU's hardware virtualization extensions and managed from a higher exception level, representing the most hardened implementation of the reactive defense, ultimately making sure that even a fully compromised kernel cannot silently inspect protected processes.

References

[1] Varun Agrawal et al. "Architectural support for dynamic linking". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 691–702.

[2] Arm. *Arm Confidential Compute Architecture*. 2021. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture (visited on 09/14/2025).

[3] Arm Limited. *Secure virtualization*. 2024. URL: https://developer.arm.com/documentation/102142/0100/Secure-virtualization (visited on 07/11/2025).

[4] Alessandro Armando et al. "Breaking and fixing the Android launching flow". In: *Computers & Security* 39 (2013), pp. 104–115.

[5] Thanassis Avgerinos et al. "AEG: Automatic Exploit Generation." In: vol. 57. Jan. 2011. DOI: 10.1145/2560217.2560219.

[6] Thanassis Avgerinos et al. "Automatic Exploit Generation". In: *Communications of the ACM* 57.2 (Feb. 2014), pp. 74–84. DOI: 10.1145/2556942. URL: https://cacm.acm.org/research/automatic-exploit-generation/.

[7] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C Pierce. "The meaning of memory safety". In: *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. Springer. 2018, pp. 79–105.

[8] b00n. *Obtain DLL addresses from the PEB of a 64 bit process*. Rohitab.com. 2019. URL: http://www.rohitab.com/discuss/topic/45310-obtain-dll-addresses-from-the-peb-of-a-64-bit-process/ (visited on 08/22/2025).

[9] Michael Backes et al. "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*. 2014, pp. 1342–1353.

[10] Sean Barker et al. "An empirical study of memory sharing in virtual machines". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 273–284.

[11] Elena Gabriela Barrantes et al. "Randomized instruction set emulation to disrupt binary code injection attacks". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. 2003, pp. 281–289.

[12] Sean Bartell, Will Dietz, and Vikram S Adve. "Guided linking: dynamic linking without the costs". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–29.

[13] Jonathan Bartlett. "Dynamic Linking". In: *Learn to Program with Assembly: Foundational Learning for New Programmers*. Springer, 2021, pp. 187–201.

[14] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[15] Raz Ben Yehuda. "Manipulating the ARM Hypervisor and TrustZone". In: *JYU Dissertations* (2021).

[16] Raz Ben Yehuda et al. "Hypervisor memory acquisition for ARM". In: *Forensic Science International: Digital Investigation* 37 (2021), p. 301077.

[17] Lukas Bernhard et al. "JIT-picking: Differential fuzzing of JavaScript engines". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 351–364.

[18] Parnika Bhat and Kamlesh Dutta. "A survey on various threats and current state of security in Android platform". In: *ACM Computing Surveys (CSUR)* 52.1 (2019), pp. 1–35.

[19] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. "Efficient techniques for comprehensive protection from memory error Exploits." In: *USENIX Security Symposium*. Vol. 10. 1. 2005.

[20] Lorenzo Binosi et al. "The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2024, pp. 1360–1374.

[21] Andrea Bittau et al. "Hacking Blind". In: SP '14. USA: IEEE Computer Society, 2014, pp. 227–242. ISBN: 9781479946860. DOI: 10.1109/SP.2014.22. URL: https://doi.org/10.1109/SP.2014.22.

[22] Pierrick Bouvier. "QEMU: A tale of performance analysis". In: *Linaro Blog* (2025). Published on January 14, 2025. URL: https://www.linaro.org/blog/qemu-a-tale-of-performance-analysis/.

[23] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. Third. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN: 978-0-596-00565-8.

[24] Stephen W Boyd et al. "On the general applicability of instruction-set randomization". In: *IEEE Transactions on Dependable and Secure Computing* 7.3 (2008), pp. 255–270.

[25] Kjell Braden et al. "Leakage-Resilient Layout Randomization for Mobile Devices." In: *NDSS*. Vol. 16. 2016, pp. 21–24.

[26] David Brumley et al. "Automatic patch-based exploit generation is possible: Techniques and implications". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 143–157.

[27] Bulba and Kil3r. “Bypassing StackGuard and StackShield”. In: *Phrack Magazine* 56 (5 May 2000).

[28] Muhammad Arif Butt et al. “An in-depth survey of bypassing buffer overflow mitigation techniques”. In: *Applied Sciences* 12.13 (2022), p. 6702.

[29] Claudio Canella et al. “KASLR: Break it, fix it, repeat”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 481–493.

[30] Sang Kil Cha et al. “Unleashing mayhem on binary code”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

[31] Weiteng Chen et al. “{KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1093–1110.

[32] J-H. Cho et al. “Toward proactive, adaptive defense: A survey on moving target defense”. In: *IEEE Communication Surveys & Tutorials* 22 (1 Jan. 2020), pp. 709–745.

[33] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. “Mining specifications of malicious behavior”. In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2007, pp. 5–14.

[34] George Christou et al. “On architectural support for instruction set randomization”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 17.4 (2020), pp. 1–26.

[35] Joel Coffman et al. “ROP gadget prevalence and survival under compiler-based binary diversification schemes”. In: *Proceedings of the 2016 ACM Workshop on Software PROtection*. 2016, pp. 15–26.

[36] Computer Security Resource Center. “Lightweight Cryptography: NIST Selects Ascon”. In: *National Institute of Standards and Technology* (2023). Accessed: 2025-08-14. URL: https://csrc.nist.gov/news/2023/lightweight-cryptography-nist-selects-ascon.

[37] C. Cowan. *Re: Buffer overflow and OS/390*. https://tinyurl.com/dxuuj3jd. Feb. 1999.

[38] C. Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, Jan. 1998.

[39] Crispin Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: *Proceedings of the 7th USENIX Security Symposium*. San Antonio, TX: USENIX Association, 1998, pp. 63–77.

[40] Stephen Crane, Andrei Homescu, and Per Larsen. “Code randomization: Haven’t we solved this problem yet?” In: *2016 IEEE Cybersecurity Development (SecDev)*. IEEE. 2016, pp. 124–129.

[41] Stephen Crane et al. "Readactor: Practical code randomization resilient to memory disclosure". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 763–780.

[42] Christoffer Dall et al. "ARM virtualization: performance and architectural implications". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 304–316.

[43] Lucas Davi et al. "Isomeron: code randomization resilient to (just-in-time) return-oriented programming." In: *NDSS*. 2015, pp. 1–15.

[44] Joe Devlin and Eric Nonce. *Nonce disrespecting adversaries: Practical forgery attacks on GCM in TLS*. Black Hat USA 2016. Accessed: 2025-08-14. 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Devlin-Nonce-Disrespecting-Adversaries-Practical-Forgery-Attacks-On-GCM-In-TLS.pdf.

[45] Raquel Vázquez Díaz et al. "Address Space Layout Randomization Comparative Analysis on Windows 10 and Ubuntu 18.04 LTS". In: *Engineering Proceedings* 7.1 (2021). ISSN: 2673-4591. DOI: 10.3390/engproc2021007026. URL: https://www.mdpi.com/2673-4591/7/1/26.

[46] Yu Ding et al. "Android low entropy demystified". In: *2014 IEEE international conference on communications (ICC)*. IEEE. 2014, pp. 659–664.

[47] Christoph Dobraunig et al. "Ascon". In: *Submission to the CAESAR competition: http://ascon. iaik. tugraz. at* (2014).

[48] Christoph Dobraunig et al. "Ascon v1. 2: Lightweight authenticated encryption and hashing". In: *Journal of Cryptology* 34.3 (2021), p. 33.

[49] Ulrich Drepper. "How to write shared libraries". In: *Retrieved Jul* 16 (2006), p. 2009.

[50] T. Durden. "Bypassing PaX ASLR protection". In: *Phrack Magazine* 59 (9 June 2002).

[51] Zakir Durumeric et al. "The matter of heartbleed". In: *Proceedings of the 2014 conference on internet measurement conference*. 2014, pp. 475–488.

[52] J. Edge. "Linux ASLR vulnerabilities". In: *LWN.net* 2009 (330866 Apr. 2009).

[53] M. W. Eichin and J. A. Rochlis. "With microscope and tweezers: An analysis of the Internet virus of November 1988". In: *Proceedings of the 1989 IEEE Symposium on Security and Privacy*. IEEE, 1989, pp. 326–343.

[54] Elttam. *Key Recovery Attacks on GCM*. https://www.elttam.com/blog/key-recovery-attacks-on-gcm/. Accessed: 2025-08-14.

[55] Mohamed Fakroud. *Digging into Windows PEB*. Red Teaming's Dojo - GitBook. URL: https://mohamed-fakroud.gitbook.io/red-teamings-dojo/windows-internals/peb (visited on 08/22/2025).

[56] Giulia Ferri et al. “Towards the hypervision of hardware-based control flow integrity for ARM platforms”. In: *ITASEC*. 2019.

[57] Stephanie Forrest, Anil Somayaji, and David H Ackley. “Building diverse computer systems”. In: *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE. 1997, pp. 67–72.

[58] Jianming Fu, Xu Zhang, and Yan Lin. “An instruction-set randomization using length-preserving permutation”. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. IEEE. 2015, pp. 376–383.

[59] M. Gallagher et al. “Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn”. In: *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS ’19)* (Providence, RI, USA). New York, NY, USA: ACM, Apr. 2019, pp. 469–484.

[60] Jonathan Ganz and Sean Peisert. *ASLR: How robust is the randomness?* IEEE, 2017.

[61] Charalampos Gartaganis. “Comparative analysis of the Windows security XP, Vista, 7, 8 and 10”. MA thesis. Πανεπιστήμιο Πειραιώς, 2017.

[62] Héctor Marco Gisbert. “Universitat Politecnica de Valencia”. In: (2015).

[63] GlobalPlatform. *TEE System Architecture Version 1.2 Public Release*. Tech. rep. GPD_SPE_009. GlobalPlatform, Nov. 2018. URL: https://globalplatform.org.

[64] *GlobalPlatform API*. https://optee.readthedocs.io/en/latest/architecture/globalplatform_api.html. Accessed: 2025-08-14.

[65] RongJie Gu et al. “An automatic and generic early-bird system for internet backbone based on traffic anomaly detection”. In: *Networking-ICN 2005: 4th International Conference on Networking, Reunion Island, France, April 17-21, 2005, Proceedings, Part I 4*. Springer. 2005, pp. 740–748.

[66] Yufei Gu and Zhiqiang Lin. “Derandomizing Kernel Address Space Layout for Memory Introspection and Forensics”. In: Mar. 2016, pp. 62–72. DOI: 10.1145/2857705.2857707.

[67] Lachlan J. Gunn et al. “Hardware Platform Security for Mobile Devices”. In: *Foundations and Trends® in Privacy and Security* 3.3-4 (2022), pp. 214–394. ISSN: 2474-1558. DOI: 10.1561/3300000024. URL: http://dx.doi.org/10.1561/3300000024.

[68] Aditi Gupta et al. “Marlin: A fine grained randomization approach to defend against ROP attacks”. In: *International Conference on Network and System Security*. Springer. 2013, pp. 293–306.

[69] Aditi Gupta et al. “Marlin: Mitigating code reuse attacks using code randomization”. In: *IEEE Transactions on Dependable and Secure Computing* 12.3 (2014), pp. 326–337.

[70] Matt Hand. *Evading EDR: The Definitive Guide to Defeating Endpoint Detection Systems.* No Starch Press, 2023.

[71] Red Hat. *Position-Independent Executables (PIE)*. 2012. URL: https://www.redhat.com/en/blog/position-independent-executables-pie (visited on 06/18/2025).

[72] Christophe Hauser et al. "Sleak: Automating address space layout derandomization". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. 2019, pp. 190–202.

[73] Jason Hiser et al. "ILR: Where'd My Gadgets Go?" In: *2012 IEEE Symposium on Security and Privacy*. 2012, pp. 571–585. DOI: 10.1109/SP.2012.39.

[74] Ding-Yong Hong et al. "Efficient and retargetable dynamic binary translation on multicores". In: (2015).

[75] M. Howard. *Address Space Layout Randomization in Windows Vista*. May 2006.

[76] *iRMX 86 Application Loader Reference Manual*. Intel Corporation. 1984.

[77] Todd Jackson et al. "Compiler-generated software diversity". In: *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, 2011, pp. 77–98.

[78] Christopher Jelesnianski et al. "Making Code Re-randomization Practical with MARDU". In: *arXiv preprint arXiv:1909.09294* (2019).

[79] Christopher Jelesnianski et al. "Mardu: Efficient and scalable code re-randomization". In: *Proceedings of the 13th ACM International Systems and Storage Conference*. 2020, pp. 49–60.

[80] Hang Jiang et al. "An Effective Authentication for Client Application Using ARM TrustZone". In: Dec. 2017, pp. 802–813. ISBN: 978-3-319-72358-7. DOI: 10.1007/978-3-319-72359-4_50.

[81] Simon Josefsson and Cas Cremers. "ChaCha20-Poly1305 Cipher Suites for TLS". In: *RFC 8439*. Aug. 2018. URL: https://datatracker.ietf.org/doc/html/rfc8439.

[82] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. 2003, pp. 272–280.

[83] Gregg Keizer. "Apple Missed Security Boat with Snow Leopard, Says Researcher". In: *Computerworld* (2009). URL: https://www.computerworld.com/article/1470056/apple-missed-security-boat-with-snow-leopard-says-researcher.html (visited on 06/02/2025).

[84] Angelos D Keromytis. "Randomized instruction sets and runtime environments past research and future directions". In: *IEEE Security & Privacy* 7.1 (2009), pp. 18–25.

[85] Muhammad Jamshid Khan. “Zero trust architecture: Redefining network security paradigms in the digital age”. In: *World Journal of Advanced Research and Reviews* 19.3 (2023), pp. 105–116.

[86] Chongkyung Kil et al. “Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software”. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE. 2006, pp. 339–348.

[87] Michael Kiperberg et al. “Hypervisor-based protection of code”. In: *IEEE Transactions on Information Forensics and Security* 14.8 (2019), pp. 2203–2216.

[88] Hyungjoon Koo et al. “Compiler-assisted code randomization”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 461–477.

[89] ZhongZheng Koo, Zakiah Ayop, and ZaheeraZainal Abidin. “Analysis of ROP attack on grsecurity/PaX Linux kernel security variables”. In: *International Journal of Applied Engineering Research* 12.23 (2017), pp. 13179–13185.

[90] Butler Lampson et al. “Authentication in distributed systems: theory and practice”. In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 265–310. ISSN: 0734-2071. URL: https://doi.org/10.1145/138873.138874.

[91] John R Lange, Nicholas Gordon, and Brian Gaines. “Low overhead security isolation using lightweight kernels and TEEs”. In: *2021 SC Workshops Supplementary Proceedings (SCWS)*. IEEE. 2021, pp. 42–49.

[92] Per Larsen et al. “SoK: Automated Software Diversity”. In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 276–291. DOI: 10.1109/SP.2014.25.

[93] Byoungyoung Lee et al. “From Zygote to Morula: Fortifying weakened ASLR on Android”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 424–439.

[94] J. R. Levine. *Linkers and Loaders*. San Diego, CA, USA: Academic Press, 2000.

[95] E. Levy. “Smashing the Stack for Fun and Profit”. In: *Phrack Magazine* 49 (14 1996).

[96] Kyung-Suk Lhee and Steve J Chapin. “Buffer overflow and format string overflow vulnerabilities”. In: *Software: Practice and Experience* 33.5 (2003), pp. 423–460.

[97] Lixin Li, James E Just, and R Sekar. “Address-space randomization for Windows systems”. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE. 2006, pp. 329–338.

[98] Yinbing Li and Jing Yan. “ELF-based computer virus prevention technologies”. In: *International Conference on Information Computing and Applications*. Springer. 2011, pp. 621–628.

[99] Zhiang Li et al. "On practicality of using ARM TrustZone trusted execution environment for securing programmable logic controllers". In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 2024, pp. 947–961.

[100] Linux Die.net. *`auditd(8)` – The Linux Audit daemon*. Manual page (man8). Accessed: September 2025. 2025. URL: https://linux.die.net/man/8/auditd.

[101] Linux Kernel Organization. *capabilities(7) — Linux manual page*. Accessed on 20 September 2025. 2025. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html (visited on 09/20/2025).

[102] Linux Kernel Organization. *ptrace(2) — Linux manual page*. Accessed on 20 September 2025. 2025. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 09/20/2025).

[103] *Linux kernel patch from the Openwall Project*. Tech. rep. https://tinyurl.com/2end2e9x. Solar Designer, 1997.

[104] Linux Man-pages Project. *`gdb(1)` – The GNU Debugger*. Manual page (man1). Accessed: September 2025. 2025. URL: https://man7.org/linux/man-pages/man1/gdb.1.html.

[105] Linux Man-pages Project. *`ptrace(2)` – process tracing*. Manual page (man2). Accessed: September 2025. 2025. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html.

[106] LLVM Project. *`lldb(1)` – The LLDB Debugger*. Manual page (man1). Accessed: September 2025. 2025. URL: https://lldb.llvm.org/man/lldb.html.

[107] Lan Luo et al. "FASLR: Function-based ASLR via TrustZone-M and MPU for resource-constrained IoT systems". In: *IEEE Internet of Things Journal* 9.18 (2022), pp. 17120–17135.

[108] Hector Marco-Gisbert and Ismael Ripoll. "On the Effectiveness of Full-ASLR on 64-bit Linux". In: *Proceedings of the In-Depth Security Conference*. 2014.

[109] Hector Marco-Gisbert and Ismael Ripoll Ripoll. "Address space layout randomization next generation". In: *Applied Sciences* 9.14 (2019), p. 2928.

[110] Hector Marco-Gisbert and Ismael Ripoll-Ripoll. "Exploiting Linux and PaX ASLR's weaknesses on 32-and 64-bit systems". In: *BlackHat Asia* (2016).

[111] Kerry McKay et al. "NIST Selects 'Lightweight Cryptography' Algorithms to Protect Small Devices". In: *NIST News* (Feb. 2023). Published on February 7, 2023. URL: https://www.nist.gov/news-events/news/2023/02/nist-selects-lightweight-cryptography-algorithms-protect-small-devices.

[112] Charlie Miller. "Mobile attacks and defense". In: *IEEE Security & Privacy* 9.4 (2011), pp. 68–70.

[113] Wei-Loon Mow, Shih-Kun Huang, and Hsu-Chun Hsiao. “LAEG: Leak-based AEG using dynamic binary analysis to defeat ASLR”. In: *2022 IEEE Conference on Dependable and Secure Computing (DSC)*. 2022, pp. 1–8. DOI: 10.1109/DSC54232.2022.9888796.

[114] National Institute of Standards and Technology. *NIST Finalizes Lightweight Cryptography Standard to Protect Small Devices*. https://www.nist.gov/news-events/news/2025/08/nist-finalizes-lightweight-cryptography-standard-protect-small-devices. Accessed: 2025-08-14. 2025.

[115] Arto Niemi. “The last line of defense in memory safety: A survey of code randomization”. In: *Unpublished manuscript* (2025).

[116] Vitaly Nikolenko. *CVE-2016-6187: Exploiting Linux kernel heap off-by-one*. 2016.

[117] Paul C. van Oorschot. “Memory errors and memory safety: C as a case study”. In: *IEEE Security & Privacy* 21.2 (2023), pp. 70–76. DOI: 10.1109/MSEC.2023.3236542.

[118] Pradeep Padala. “Playing with ptrace, Part I”. In: *Linux Journal* 103.5 (2002).

[119] Michael Pavento. “A Practical Guide to Open Source Software”. In: *Kilpatrick Townsend* (2012). This article is adapted from Michael Pavento, Open Source Software: A Practical Guide for In-House Counsel, in Association of Corporate Counsel (September 2012). URL: http://opensource.org/licenses/alphabetical.

[120] *PaX project: ASLR*. https://tinyurl.com/hutfnx3w. 2003.

[121] *PaX project: overall description*. https://tinyurl.com/3udj9ddf. 2000.

[122] *PaX project: RANDEXEC*. https://tinyurl.com/3stk7rf7. 2003.

[123] E. Perla. *A Guide to Kernel Exploitation: Attacking the Core*. Burlington, MA, USA: Elsevier Inc, 2011.

[124] Conor Pirry, Hector Marco-Gisbert, and Carolyn Begg. “A review of memory errors exploitation in x86-64”. In: *Computers* 9.2 (2020), p. 48.

[125] M. Pomonis et al. “kRˆX: Comprehensive kernel protection against just-in-time code reuse”. In: *EuroSys ’17: Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia). Association for Computer Machinery, Apr. 2017, pp. 420–436.

[126] Andrea Possemato et al. “Trust, but verify: A longitudinal analysis of Android OEM compliance and customization”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 87–102.

[127] Marco Prandini and Marco Ramilli. “Return-oriented programming”. In: *IEEE Security & Privacy* 10.6 (2012), pp. 84–87.

[128] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. "Practical fine-grained binary code randomization". In: *Proceedings of the 36th Annual Computer Security Applications Conference*. 2020, pp. 401–414.

[129] Phillip Rogaway. "Authenticated-encryption with associated-data". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 2002, pp. 98–107.

[130] G. F. Roglia et al. "Surgically returning into randomized lib(c)". In: *2009 Annual Computer Security Applications Conference*. IEEE, Dec. 2009, pp. 60–69.

[131] Robert Rudd et al. "Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity." In: *NDSS*. 2017.

[132] Takamichi Saito et al. "A Survey of Prevention/Mitigation against Memory Corruption Attacks". In: *2016 19th International Conference on Network-Based Information Systems (NBiS)*. 2016, pp. 500–505. DOI: 10.1109/NBiS.2016.11.

[133] H. Schacham. "The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)". In: *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria Virginia, USA). New York, NY, USA: Association for Computer Machinery, Oct. 2007, pp. 552–561.

[134] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of ACM CCS 2007*. 2007. URL: https://hovav.net/ucsd/dist/geometry.pdf.

[135] Himanshu Shewale et al. "Analysis of Android vulnerabilities and modern exploitation techniques". In: *ICTACT journal on communication technology* 5.1 (2014), pp. 863–867.

[136] Dongwook Shim and Dong Hoon Lee. "SOTPM: software one-time programmable memory to protect shared memory on ARM TrustZone". In: *IEEE Access* 9 (2020), pp. 4490–4504.

[137] Denis V Silakov. "Using virtualization to protect application address space inside untrusted environment". In: *Programming and Computer Software* 38.1 (2012), pp. 24–33.

[138] Kevin Z. Snow et al. "Just-In-Time code reuse: On the effectiveness of fine-grained address space layout randomization". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 525–539. DOI: 10.1109/SP.2013.43.

[139] V Stafford. "Zero trust architecture". In: *NIST special publication* 800.207 (2020), pp. 800–207.

[140] Haiyong Sun and Hang Lei. "A design and verification methodology for a TrustZone trusted execution environment". In: *IEEE Access* 8 (2020), pp. 33870–33883. DOI: 10.1109/ACCESS.2020.2974487. URL: https://ieeexplore.ieee.org/document/9000841.

[141] Jakub Szefer. "Principles of secure processor architecture design". In: Jan. 2019, pp. 113–124. ISBN: 978-3-031-00632-6. DOI: 10.1007/978-3-031-01760-5_10.

[142] László Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[143] *The compiler, assembler, linker, loader and process address space tutorial*. https://www.tenouk.com/ModuleW.html. Accessed: 2025-07-29.

[144] The Linux man-pages project. *fexecve – execute program specified via file descriptor (3)*. Linux Library Functions Manual. Accessed on October 6, 2025. URL: https://man7.org/linux/man-pages/man3/fexecve.3.html.

[145] The Linux man-pages project. *ld.so, ld-linux.so – dynamic linker/loader (8)*. Linux System Manager's Manual. Accessed on October 6, 2025. URL: https://man7.org/linux/man-pages/man8/ld.so.8.html.

[146] The Linux man-pages project. *memfd_create – create an anonymous file (2)*. Linux Programmer's Manual. Accessed on October 6, 2025. URL: https://man7.org/linux/man-pages/man2/memfd_create.2.html.

[147] Zhang Tianning et al. "SeBROP: blind ROP attacks without returns". In: *Frontiers of Computer Science* 16 (Aug. 2022). DOI: 10.1007/s11704-021-0342-8.

[148] Minh Tran et al. "On the expressiveness of return-into-libc attacks". In: *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*. Springer. 2011, pp. 121–141.

[149] Corporate UNIX Press. *System V Application Binary Interface (3rd ed.)* USA: Prentice-Hall, Inc., 1993. ISBN: 0131004395.

[150] Victor Van der Veen et al. "Memory errors: The past, the present, and the future". In: *Research in Attacks, Intrusions, and Defenses: 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15*. Springer. 2012, pp. 86–106.

[151] Freek Verbeek, Nico Naus, and Binoy Ravindran. "Verifiably correct lifting of position-independent x86-64 binaries to symbolized assembly". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2024, pp. 2786–2798.

[152] void-stack. *Exploring the PEB*. Void-Stack's Blog. 2022. URL: https://void-stack.github.io/blog/post-Exploring-PEB/ (visited on 08/22/2025).

[153] Shengye Wan et al. "RusTEE: developing memory-safe ARM TrustZone applications". In: *Proceedings of the 36th Annual Computer Security Applications Conference*. 2020, pp. 442–453.

[154] KC Wang. "ARM TrustZone and Secure Operating Systems". In: *Embedded and Real-Time Operating Systems*. Springer, 2023, pp. 793–845.

[155] KC Wang. "ARMv8 Architecture and Programming". In: *Embedded and Real-Time Operating Systems*. Springer, 2023, pp. 505–792.

[156] Pei Wang et al. "Quantitative assessment on the limitations of code randomization for legacy binaries". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2020, pp. 1–16.

[157] Shuai Wang, Pei Wang, and Dinghao Wu. "Composite software diversification". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 284–294.

[158] Yan Wang et al. "Revery: From Proof-of-Concept to Exploitable". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1914–1927. ISBN: 9781450356930. URL: https://doi.org/10.1145/3243734.3243847.

[159] Yu Wang, Yipeng Zhang, and Zhoujun Li. "AAHEG: Automatic Advanced Heap Exploit Generation Based on Abstract Syntax Tree". In: *Symmetry* 15.12 (2023). ISSN: 2073-8994. DOI: 10.3390/sym15122197. URL: https://www.mdpi.com/2073-8994/15/12/2197.

[160] Ollie Whitehouse. "An analysis of address space layout randomization on Windows Vista". In: (Jan. 2007).

[161] Reginald Wong. *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*. Packt Publishing Ltd, 2018.

[162] Xiaowen Xin. "Keeping Android safe: Security enhancements in Nougat". In: *Google Cloud Blog* (Sept. 2016). URL: https://cloud.google.com/blog/products/chrome-enterprise/keeping-android-safe-security-enhancements-nougat (visited on 07/02/2025).

[163] H. Xu and S. J. Chapin. "Improving address space randomization with a dynamic offset randomization technique". In: *SAC '06: Proceedings of the 2006 ACM Symposium on Agile Computing*. Association for Computer Machinery, Apr. 2006, pp. 384–391.

[164] J. Xu, Z. Kalbarczyk, and R. K. Iyer. "Transparent runtime randomization for security". In: *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*. IEEE, Oct. 2003, pp. 260–269.

[165] Wei Xu, Daniel C DuVarney, and R Sekar. "An efficient and backwards-compatible transformation to ensure memory safety of C programs". In: *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*. 2004, pp. 117–126.

[166] Song-Tao Yang et al. "Exploit-oriented automated information leakage". In: *Journal of Software* 33.6 (2022), pp. 2082–2096. ISSN: 1000-9825. DOI: 10.13328/j.cnki.jos.006570.

[167] Tianning Zhang et al. "De-randomizing the code segment with timing function attack". In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE. 2020, pp. 259–267.

A Benchmarked code for time data

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

typedef unsigned long u_l;

void f_one(){asm("nop;");}
// ...
void f_twenty(){asm("nop;");}

char ret_nul() {return '\0';}

void mul_31(u_l* v) {
  *v *= 31;
}

u_l do_xor(u_l a, u_l b) {
  return a ^ b;
}

int r_len() {
  return (rand() % 15) + 5;
}

char* r_str(int len) {
  char* s = malloc(len+1);
  if (!s) return NULL;
  int i = 0;
  for (; i < len; i++) {
    int r = rand() % 26;
    s[i] = 'a' + r;
  }
  s[len] = ret_nul();
  return s;
}

u_l hash(char* s) {
  u_l h = 0;
  int i = 0;
  for (; s[i]; i++) {
    h += s[i];
    mul_31(&h);
  }
  return h;
}
void* g_addr() {
  int x;
  return (void*)&x;
}

u_l xor_it(u_l val, void* p){
  return do_xor(val, (u_l)p);
}

void p_res(u_l res) {
  printf("r: %lu\n", res);
}

void done() {
  puts("done.");
}

int main(int argc,
         char** argv) {

  srand(time(NULL));
  char* str;
  int gen = 0;

  if (argc < 2) {
    str = r_str(r_len());
    gen = 1;
  } else {
    str = argv[1];
  }

  if (!str) return 1;

  u_l h = hash(str);
  void* adr = g_addr();
  u_l x = xor_it(h, adr);

  p_res(x);
  done();

  if (gen) {
    free(str);
  }

  return 0;
}
```

B Benchmarked code for entropy graphs

B.1 Code used for entropy benchmarking

```c
#include <stdio.h>
#include <stdlib.h>

void pointless_function1() {
    int x = 10;
    x = x * 2;
}

void pointless_function2() {
    char* str = "example";
    printf("String length: %zu\n", sizeof(str));
}

void pointless_function3() {
    int i = 0;
    for (; i < 5; i += 1) {
        printf("Hi!");
    }
}

void pointless_function4() {
    printf("%d\n", 5/2);
    printf("%f\n", 5.0/2.0);
}

int pointless_function5() {
    return 100 / 10;
}

void pointless_function6() {
    char c = 'A';
    c++;
}
void pointless_function7() {
    int* ptr = malloc(sizeof(int));
    if (ptr != NULL) {
        *ptr = 5;
        free(ptr);
    }
}

void pointless_function8() {
}

int main() {

    void (*fp[8])() = {
        pointless_function1,
        pointless_function2,
        pointless_function3,
        pointless_function4,
        pointless_function5,
        pointless_function6,
        pointless_function7,
        pointless_function8
    };

    for (int i = 0;
             i < 8;
             i += 1) {
        printf("%p\n",fp[i]);
    }

    printf("%p\n", &main);
    return 0;
}
```

B.2 Code used to extract the function addresses

```bash
#!/bin/bash

rm -f *.tmp times_by_function.txt

for i in {1..8}; do
    touch "func${i}.tmp"
done
touch "main.tmp"

for i in {1..5000}
do
    ./elfshuf runme

    output=$(./runme.shuffled)

    echo "$output" | sed -n '1p' >> func1.tmp
    echo "$output" | sed -n '2p' >> func2.tmp
    echo "$output" | sed -n '3p' >> func3.tmp
    echo "$output" | sed -n '4p' >> func4.tmp
    echo "$output" | sed -n '5p' >> func5.tmp
    echo "$output" | sed -n '6p' >> func6.tmp
    echo "$output" | sed -n '7p' >> func7.tmp
    echo "$output" | sed -n '8p' >> func8.tmp
    echo "$output" | sed -n '9p' >> main.tmp

    rm runme.shuffled runme.hrr
done

echo "Sampling complete. Consolidating data..."

{
    for i in {1..8}; do
        echo "Function${i}:"
        echo $(cat "func${i}.tmp")
    done
    echo "Main:"
    echo $(cat "main.tmp")
} > times_by_function.txt

rm -f *.tmp

echo "Done, output in times_by_function.txt"
```

C System Binaries and PRR Injection Overhead

Name	Before	After	Name	Before	After
apt	89K	122K	nice	96K	146K
apt-cache	206K	324K	nl	187K	298K
apt-cdrom	88K	121K	nohup	99K	155K
apt-config	89K	122K	nproc	96K	148K
apt-extracttemplates	84K	111K	numfmt	189K	288K
apt-ftparchive	463K	808K	od	187K	282K
apt-get	107K	170K	paste	98K	153K
apt-mark	110K	178K	pathchk	95K	145K
apt-sortpkgs	86K	117K	pinky	103K	165K
b2sum	109K	183K	pr	189K	290K
base32	99K	155K	printenv	94K	142K
base64	99K	155K	printf	107K	175K
basename	95K	146K	ptx	206K	349K
basenc	107K	180K	pwd	97K	151K
cat	99K	155K	readlink	103K	169K
cdrom	223K	365K	realpath	105K	175K
chcon	123K	225K	rm	121K	220K
chgrp	116K	206K	rmdir	100K	156K
chmod	114K	200K	rred	237K	399K
chown	118K	211K	rsh	223K	366K
cksum	97K	149K	runcon	98K	153K
comm	102K	164K	scp	535K	894K
copy	145K	264K	sed	307K	527K
cp	235K	420K	seq	107K	175K
csplit	194K	316K	sftp	528K	878K
cut	103K	166K	sha1sum	104K	168K
date	193K	305K	sha224sum	105K	171K
dd	195K	304K	sha256sum	105K	171K
df	205K	331K	sha384sum	105K	171K
dir	302K	497K	sha512sum	105K	171K
dircolors	102K	167K	shred	115K	200K
dirname	95K	144K	shuf	111K	191K
du	294K	484K	sleep	97K	150K
echo	96K	146K	sort	214K	363K
env	108K	176K	split	116K	199K
expand	100K	158K	ssh	1.9M	3.2M
expr	190K	309K	ssh-add	780K	1.3M
factor	181K	272K	ssh-agent	769K	1.2M
false	93K	139K	ssh-keygen	1.1M	1.8M
file	145K	264K	ssh-keyscan	978K	1.6M

Table 7 – continued from previous page

Name	Before	After	Name	Before	After
fmt	101K	163K	stat	200K	318K
fold	98K	152K	stdbuf	104K	169K
ftp	352K	598K	store	152K	282K
gpgv	251K	438K	stty	188K	287K
grep	302K	513K	sum	104K	168K
groups	97K	149K	sync	96K	146K
head	103K	166K	tac	186K	294K
hostid	95K	143K	tail	190K	293K
http	403K	735K	tee	100K	157K
id	105K	169K	test	102K	164K
join	109K	182K	timeout	104K	167K
kill	98K	152K	touch	189K	295K
link	95K	144K	tr	106K	175K
ln	191K	297K	true	93K	139K
logname	95K	144K	truncate	99K	154K
ls	302K	497K	tsort	107K	180K
md5sum	104K	168K	tty	94K	142K
mirror	261K	461K	uname	96K	148K
mkdir	192K	303K	unexpand	100K	159K
mkfifo	118K	213K	uniq	105K	171K
mknod	121K	219K	unlink	95K	144K
mktemp	104K	169K	uptime	103K	165K
mv	236K	422K	users	96K	148K
vdir	302K	497K	wc	106K	174K
who	109K	181K	whoami	95K	144K
yes	95K	145K			

Table 7: Original and final sizes of system binaries after PRR injection.