Securely Sharing Randomized Code That Flies

CHRISTOPHER JELESNIANSKI, JINWOO YOM, and CHANGWOO MIN, Virginia Tech
YEONGJIN JANG, Oregon State University

Address space layout randomization was a great role model, being a light-weight defense technique that could prevent early return-oriented programming attacks. Simple yet effective, address space layout randomization was quickly widely adopted. Conversely, today only a trickle of defense techniques arebeing integrated or adopted mainstream. As code reuse attacks have evolved in complexity, defenses have strived to keep up. However, to do so, many have had to take unfavorable tradeoffs like using background threads or protecting only a subset of sensitive code. In reality, these tradeoffs were unavoidable steps necessary to improve the strength of the state of the art. In this article, we present MARDU, an on-demand system-wide runtime re-randomization technique capable of scalable protection of application as well as shared library code that most defenses have forgone. We achieve code sharing with diversification by implementing reactive and scalable rather than continuous or one-time diversification. Enabling code sharing further removes redundant computation like tracking and patching, along with memory overheads required by prior randomization techniques. In its baseline state, the code transformations needed for MARDU security hardening incur a reasonable performance overhead of 5.5% on SPEC and minimal degradation of 4.4% in NGINX, demonstrating its applicability to both compute-intensive and scalable real-world applications. Even when under attack, MARDU only adds from less than 1% to up to 15% depending on application size and complexity.

CCS Concepts: • **Security and privacy** → **Software security engineering**; **Systems security;**

Additional Key Words and Phrases: Code randomization, return-oriented programming, code reuse, code sharing

ACM Reference format:
Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang. 2022. Securely Sharing Randomized Code That Flies. *Digit. Threat.: Res. Pract.* 3, 3, Article 32 (September 2022), 25 pages.
https://doi.org/10.1145/3474558

1 INTRODUCTION

Code reuse attacks have grown in depth and complexity to circumvent early defense techniques such as **address space layout randomization (ASLR)** [63]. Examples like **return-oriented programming (ROP)** and ret-into-libc [58] utilize a victim's code against itself. ROP leverages innocent code snippets (i.e., *gadgets*) to construct malicious payloads. Reaching this essential gadget commodity versus defending it from being exploited has made an arms race between attackers and defenders. Both coarse- and fine-grained ASLR, although light-weight, are vulnerable to attack. Whether or not an entire code region or basic block layout is randomized in memory, a single memory disclosure can result in exposing the entire code layout [59]. **Execute-only memory (XoM)** was introduced to prevent direct memory disclosures by enabling memory regions to be marked with execute-only permissions [16, 18]. However, code inference attacks, a code reuse attack variant that works by indirectly

This work was supported in part by the U.S. Office of Naval Research under grants N00014-18-1-2022.
Authors' addresses: C. Jelesnianski, J. Yom, and C. Min, Virginia Tech; emails: {kjski, jinwoo7, changwoo}@vt.edu; Y. Jang, Oregon State University; email: yeongjin.jang@oregonstate.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2022 Copyright held by the owner/author(s).
2576-5337/2022/09-ART32 $15.00
https://doi.org/10.1145/3474558

observing and deducing information about the code layout, circumvented these limitations [9]. Various attack angles have revealed that one-time randomization is simply not sufficient, and that stronger adversaries remain to be dealt with efficiently and securely [8, 47].

This fostered the next generation of defenses, such as Shuffler [74] and CodeArmor [12], which introduced continuous runtime re-randomization to strengthen security guarantees. However, these techniques are not designed for a system's scalability in mind. Despite being performant, non-intrusive, and easily deployable, they rely on background threads to run re-randomization or approximated use of timing thresholds to proactively secure vulnerable code. Specifically, these mechanisms consume valuable system resources even when not under attack (e.g., consuming CPU time per each threshold for re-randomization), leaving less compute power for the task at hand. Additionally, no continuous re-randomization techniques currently support code sharing, and thereby they use much more physical memory by countering the operating system memory deduplication technique of using a page cache [6]. Under these defenses, the resource budget required for running an application is much higher than is traditionally expected, making active randomization techniques not scalable for general multi-programming computing.

Control flow integrity (CFI) is another protection technique that guards against an attacker attempting to subvert a program's control flow. CFI in general is already widely deployed on Windows, Android [65], and iOS, and has support in compilers like LLVM [62] and `gcc` [64]. CFI enforces the integrity of a program's control flow based off a constructed control flow graph as well as derived equivalence classes for each forward-edge. However, building a fully precise control flow graph for enforcing CFI is challenging and remains an open problem. Going in an orthogonal direction to CFI could avoid these inherent challenges.

In this article, we introduce MARDU to refocus the defense technique design, showing that it is possible to embrace core fundamentals of both performance and scalability while ensuring comprehensive security guarantees.

MARDU builds on the insight that thresholds, like time intervals [12, 74] or the amount of leaked data [70], are a security loophole and a performance shackle in re-randomization; MARDU does not rely on a threshold at all in its design. Instead, MARDU only activates re-randomization when necessary. MARDU takes advantage of an event trigger design and acts on permissions violations of Intel **Memory Protection Keys (MPK)** [37]. Using Intel MPK, MARDU provides efficient XoM protection against *both* variations of remote and local **just-in-time RoP (JIT-ROP)**. With XoM in place, MARDU leverages Readactor's [16, 17] immutable trampoline idea such that although trampolines are not re-randomized, they are protected from read access and effectively decouple function entry points from function bodies, making it impossible for an attacker to infer and obtain ROP gadgets.

It is crucial to note that few re-randomization techniques factor in the overall scalability of their approach. Support for *code sharing* is quite challenging, especially for randomization-based techniques. This is because applying re-randomization per process counters the principles of memory deduplication. In addition, the prevalence of multi-core has excused the reliance on *per-process background threads* dedicated to performing compute-extensive re-randomization processes; even if recent defenses have gained some ground in the arms race, most still lack effective comprehensiveness in security for the system resource demands they require in return (both CPU and memory). MARDU balances performance and scalability by not requiring expensive code pointer tracking and patching. Furthermore, MARDU does not incur a significant overhead from continuous re-randomization triggered by conservative time intervals or benign I/O system calls as in Shuffler [74] and ReRanz [70], respectively. Finally, MARDU is designed to both support code sharing and not require the use of any additional system resources (e.g., background threads as used in numerous works [8, 12, 23, 70, 74]). To summarize, we make the following contributions:

- *ROP attack and defense analysis*: Our background in Section 2 describes four prevalent ROP attacks that challenge current works, including JIT-ROP, code-inference, low-profile, and code pointer offsetting attacks. With this, we classify and exhibit current state-of-the-art defenses standings on three fronts: security,

Table 1. Classifications of ASLR-Based Code Reuse Defenses

Type	Defenses	Security					Performance			Scalability	
		Gran.	A1	A2	A3	A4	Perf.	Avg.	Worst	Code Sharing	No Addi. Process
Load-time ASLR	Fine-ASLR [14, 33, 34, 39, 43, 51, 72]	Fine	✕	✕	N/A	✕	✓	0.4%	6.4%	✗	✓
	Oxymoron [6]	Coarse	✕	✕	N/A	✕	✓	2.7%	11%	✓	✓
	Pagerando [15]	Coarse	✕	✕	N/A	✕	✓	1.09%	6.5%	✓	✓
	Isomeron [19]	Fine	●	●	N/A	▲	✗	19%	42%	✗	✗
Load-time+XoM	Readactor/Readactor++ [16, 18]	Fine	●	✕	N/A	▲	✓	8.4%	25%	✗	✓
	LR 2 [10]	Fine	●	✕	N/A	▲	✓	6.6%	18%	✗	✓
	kRˆ X [54]	Fine	●	✕	N/A	▲	✓	2.32%	12.1%	✗	✓
Re-randomization	RuntimeASLR [47]	Coarse	✕	●	N/A	✕	✗	N/T	N/T	✓	✓
	TASR [8]	Coarse	▲	●	✕	✕	✗	2.1%†	10.1%†	✗	✗
	ReRanz [70]	Fine	▲	●	✕	▲	✓	5.3%	14.4%	✗	✗
	Shuffler [74]	Fine	●	●	✕	▲	✗	14.9%	40%	✗	✗
	CodeArmor [12]	Coarse	●	●	✕	✕	✗	3.2%	55%	✗	✗
Our approach	MARDU	Fine	●	●	●	▲	✓	5.5%	18.3%	✓	✓

Note: Gray highlighting emphasizes the attack (*A1–A4*) that largely invalidated each type of defense. ● indicates that the attack is blocked by the defense (attack resistant). ✕ indicates the defense is vulnerable to that attack. ▲ indicates that the attack is not blocked but is still mitigated by the defense (attack resilient). ✓ indicates that the defense meets performance/ scalability requirements. ✗ indicates that the defense is unable to meet performance/scalability requirements. N/A in the *A3* column indicates that the attack is not applicable to the defense due to lack of re-randomization; N/T in the *Performance* column indicates that either SPEC CPU2006 or `perlbench` is not tested. Specifically in the *A1* column, ▲ indicates that the defense cannot prevent the JIT-ROP attack within the application boundary that does not use system calls; in the *A4* column, ✕ indicates that an attack may reuse both ROP gadgets and entire functions, whereas ▲ indicates that an attack can only reuse entire functions. †Note that in TASR, the baseline is a binary compiled with `-Og`, necessary to correctly track code pointers. Previous work [70, 74] reported performance overhead of TASR using regular optimization (`-O2`) binary is ≈30% to 50%. MARDU provides strong security guarantees with competitive performance overhead and good system-wide scalability compared to existing re-randomization approaches.

performance, and scalability. Our findings show that most defenses are not as secure or as practical as expected against current ROP attack variants.

- *MARDU defense framework*: We present the design of MARDU in Section 4, a comprehensive ROP defense technique capable of addressing the most popular and known ROP attacks, excluding only full-functioncode reuse attacks.
- *Scalability and shared code support*: To the best of our knowledge, MARDU is the first framework capable of re-randomizing shared code throughout runtime. MARDU creates its own calling convention to both leverage a shadow stack and minimize overhead of pointer tracking. This calling convention also enables shared code (e.g., libraries) to be re-randomized by any host process and maintain security integrity for the rest of the entire system.
- *Evaluation and prototype*: We built a prototype of MARDU and evaluate it in Section 6 with both compute-intensive benchmarks and real-world applications.

2 CODE LAYOUT (RE-)RANDOMIZATION

In this section, we present a background on the code reuse attack and defense arms race. In summary, Table 1 illustrates the characteristics of each defense technique by their randomization category, attack resilience, and performance and scalability factors, and we describe these in detail in the following.

2.1 Attacks Against Load-Time Randomization

Load-time randomization without XoM. Code layout randomization techniques such as coarse-grained ASLR [63] and fine-grained ASLR [6, 14, 19, 33, 34, 39, 43, 51, 72], which depend on the granularity of layout randomization, fall into this category of code layout randomization. These techniques randomize the code layout only once, usually when loaded into memory, and its layout never changes thereafter during the lifetime of the program.

A1: Just-in-time ROP. An attacker with an arbitrary memory read capability may launch JIT-ROP [59] by interactively performing memory reads to disclose one code pointer. This disclosure can be used to then leapfrog and further disclose other addresses to ultimately learn the entire code contents in memory. Any load-time code randomization technique that does not protect code from read access including fine-grained ASLR techniques is susceptible to this attack.

Load-time randomization with XoM. In response to A1 (JIT-ROP), several works protect code from read access via destructive read memory [61, 73] or XoM [5, 10, 12, 16, 18, 26, 54, 61, 73]. By destroying, purposely corrupting code read by attackers, or fundamentally removing read permissions from the code area, respectively, these techniques prevent attackers from gaining knowledge about the code contents, nullifying A1.

A2: Blind ROP and code inference attacks. Even with XoM, load-time randomizations still are susceptible to **blind ROP (BROP)** [9] or other inference attacks [53, 60]. BROP infers code contents via observing differences in execution behaviors such as timing or program output while other attacks [53, 60] defeat destructive code read defenses [61, 73] by weaponizing code contents from only a small fraction of a code read. Therefore, maintaining a fixed layout over crash-probing or read access to code allows inferring code contents indirectly, letting attackers still learn the code layout.

2.2 Defeating A1/A2 via Continuous Re-randomization

Continuous re-randomization techniques [8, 12, 23, 27, 47, 70, 74] aim to defeat A1 and A2 by continuously shuffling code (and data) layouts at runtime to make information leaks or code probing done before shuffling useless. To illustrate the internals of re-randomization techniques, we describe the core design elements of re-randomization by categorizing them into two types: the re-randomization triggering condition and Code pointer semantics:

(1) By re-randomization triggering condition:
- ***Timing*:** Techniques [12, 74] shuffle the layout periodically by setting a timing window. For example, Shuffler [74] triggers re-randomization every 50 msec (< network latency) to counter remote attackers, and CodeArmor [12] can set the re-randomization period as low as 55 μsec.
- ***System-call history*:** Techniques [8, 47, 70] shuffle the layout based on the history of the program's previous system call invocations, such as invoking `fork()` (vulnerable to BROP) [47] or when `write()` (leak) is followed by `read()` (exploit) [8, 70].

(2) By code pointer semantics:
- ***Code address as code pointer*:** Techniques (ASR3 [27] and TASR [8]) use the actual code address as code pointers. In this case, leaking a code pointer lets the attacker have knowledge about an actual code address. Therefore, this design requires tracking of all code pointers (or all pointers) at runtime, which is computationally expensive, to update values after randomizing the code layout.
- ***Function trampoline address as code pointer*:** These techniques store an indirect index, such as a function table index (Shuffler [74]) or the address of a function trampoline (ReRanz [70]), as code pointers to avoid expensive pointer tracking. After re-randomization, the techniques only need to update the function table while all code pointers remain immutable. With this design, leaking a code pointer will tell the attacker about the function index in the table or trampoline but not about the code layout; however, because the function index is immutable across re-randomization, attackers may reuse leaked function indices.
- ***An offset to the code address as code pointer*:** This design also avoids pointer tracking by having an immutable offset from the random version address for referring to a function, as in CodeArmor [12]. The re-randomization is efficient because it only requires randomizing the version base address and does not require any update of pointers. With this design, leaking a code pointer will only tell the attacker about the offset to select a specific function; however, the offset is immutable across re-randomization, so attackers may reuse leaked function offsets.

2.3 Attacks Against Continuous Re-randomization

Based on our analysis of continuous re-randomization techniques, we define two attacks (A3 and A4) against them.

A3: Low-profile JIT-ROP. This attack class does not trigger re-randomization, either by completing the attack within a defense's pre-defined randomization time interval or without involving any I/O system call invocations. Existing defenses utilize one of timing [12, 23, 27, 74], amount of transmitted data by output system calls [70], or I/O system call boundary [8] as a trigger for layout re-randomization. Therefore, attacks within the application boundary, such as code reuse attacks in Javascript engines of web browsers where both information-leak followed by control flow hijacking attack may complete faster than the re-randomization timing threshold (e.g., <50 msec) or not interact with any I/O system calls (e.g., leaking pointers via type-confusion vulnerabilities). This bypasses these triggering conditions, leaving the code layout unchanged within the given interval and vulnerable to JIT-ROP.

A4: Code pointer offsetting. Even with re-randomization, techniques might be susceptible to a code pointer offsetting attack if code pointers are not protected from having arithmetic operations applied by attackers [8, 12]. An attacker may trigger a vulnerability to apply arithmetic operations to an existing code pointer. In particular, in techniques that directly use a code address [8] or a code offset [12], the target could be even a ROP gadget if the attacker knows the gadgets offset beforehand. Ward et al. [71] demonstrated that this attack is possible against TASR. A4 shows that maintaining a fixed code layout across re-randomizations and not protecting code pointers lets attackers perform arithmetic operations over pointers, allowing access to other ROP gadgets.

3 THREAT MODEL AND ASSUMPTIONS

Mardu's threat model follows that of similar re-randomization works [8, 12, 74]. We assume that attackers can perform arbitrary read/write by exploiting software vulnerabilities in the victim program. We also assume all attack attempts run in a local machine such that attacks may be performed any number of times within a short time period (e.g., within a millisecond).

Our trusted computing base includes the OS kernel, the loading/linking process such that attackers cannot intervene to perform any attack before a program starts, and that system userspace does not have any memory region that is both writable and executable or both readable and executable (e.g., DEP (W⊕X) and XoM (R⊕X) are enabled). We assume that all hardware is trusted and attackers do not have physical access. This includes trusting Intel MPK [37], a mechanism that provides XoM.

Mardu does not support native MPK applications that directly use `wrpkru` instructions. We further analyze the security of leveraging protection keys for userspace in Section 7. Finally, hardware attacks (e.g., side-channel attacks, Spectre [40], Meltdown [45]) are outside the scope of this work.

4 MARDU DESIGN

We begin with the design overview of Mardu in Section 4.1 and then go into further detail of the Mardu compiler in Section 4.2 and kernel in Section 4.3.

4.1 Overview

This section presents the overview of Mardu, along with its design goals and challenges, and outlines its architecture.

4.1.1 Goals. Our goal in designing Mardu is to shore up the current state of the art to enable a practical code randomization. More specifically, our design goals are as follows.

Scalability. Most proposed exploit mitigation mechanisms overlook the impact of required additional system resources, such as memory or CPU usage, which we consider a scalability factor. This is crucial for applying a defense system-wide and is even more critical when deploying the defense in pay-as-you-go pricing on the Cloud.

Oxymoron [6] and PageRando [15] are the only defenses, to our knowledge, that allow code sharing of randomized code. No other *re*-randomization defenses support code sharing, thus they require significantly more memory. Additionally, most re-randomization defenses [12, 70, 74] require per-process background threads, which not only cause additional CPU usage but also contention with the application process. As a result, approaches requiring per-process background threads show significant performance overhead as the number of processes increases. Therefore, to apply MARDU system-wide, we design MARDU to not require significant additional system resources, such as additional processes/threads or significant additional memory.

Performance. Many prior approaches [12, 16, 18, 74] demonstrate decent runtime performance on average (<10%, e.g., <3.2% in CodeArmor); however, they also show corner cases that are remarkably slow (i.e., >55%, see the *Worst* column in Table 1). We design MARDU to be competitive with prior code randomization approaches in terms of performance overhead to show that the security benefits MARDU provides are worth the minor tradeoff. In particular, we aim to ensure that MARDU's performance is acceptable even in its worst-case outliers across a variety of application types.

Security. No prior solutions provide a comprehensive defense against existing attacks (see Section 2). Systems with only load-time ASLR are susceptible to code leaks (A1) and code inference (A2). Systems applying re-randomization are still susceptible to low-profile attacks (A3) and code pointer offsetting attacks (A4). MARDU aims to either defeat or significantly limit the capability of attackers to launch code reuse attacks spanning from A1 to A4 to provide a best-effort security against known existing attacks.

4.1.2 Challenges. Naively combining the best existing defense techniques is simply not possible due to conflicts in their requirements. These are the challenges MARDU addresses.

Tradeoffs in security, performance, and scalability. An example of the tradeoff between security and performance is having fine-grained ASLR with re-randomization. Although such an approach can defeat code pointer offsetting (A4), systems cannot apply such protection because re-randomization must finish quickly to meet performance goals to also defeat low-profile attacks (A3). An example of the tradeoff between scalability and performance is having a dedicated process/thread for running the defense and performing the re-randomization. Usage of a background thread results in a drawback in scalability by occupying a user's CPU core, which can no longer be used for useful user computation. This tradeoff is exaggerated even more by systems that require one-to-one matching of a background randomization thread per worker thread. Therefore, a good design must find a breakthrough to meet *all* of the aforementioned goals.

Conflict in code diversification vs. code sharing. Layout re-randomization requires diversification of code layout per-process, and this affects the availability of code sharing. The status quo is that code sharing cannot be applied to any existing re-randomization approaches, making defenses unable to scale to protect many-process applications. Although Oxymoron [6] enables both diversification and sharing of code, it does not consider re-randomization, nor use a sufficient randomization granularity (page-level).

4.1.3 Architecture. We design MARDU to gain insight on how to properly balance and integrate opposing goals of security, scalability, and performance together. Next we introduce our approach for satisfying each aspect.

Scalability: Sharing randomized code. MARDU manages the cache of randomized code in the kernel, making it capable of being mapped to multiple userspace processes, not readable from userspace, and not requiring any additional memory.

Scalability: System-wide re-randomization. Since code is shared between processes in MARDU, per-process randomization, which is CPU intensive, is not required; rather, a single process randomization is sufficient for the *entire* system. For example, if a worker process of the NGINX server crashes, it re-randomizes upon exit all associated mapped executables (e.g., `libc.so` of all processes, and all other NGINX workers).

Scalability: On-demand re-randomization. MARDU re-randomizes code only when suspicious activity is detected. This design is advantageous because MARDU does not rely on per-process background threads nor a

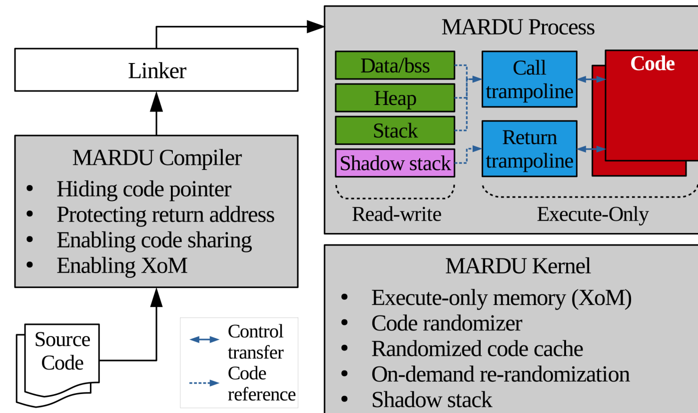


Fig. 1. Overview of Mardu.

re-randomization interval unlike prior re-randomization approaches. Particularly, Mardu re-randomization is performed in the context of a crashing process, thereby not affecting the performance of other running processes.

Performance: Immutable code pointers. The described design decisions for scalability also help reduce performance overhead. Mardu neither tracks nor encrypts code pointers, so code pointers are not mutated upon re-randomization. Although this design choice minimizes performance overhead, other security features (e.g., XoM, trampoline, and shadow stack) in Mardu ensure a comprehensive ROP defense.

Security: Detecting suspicious activities. Mardu considers any process crash or code probing attempt as a suspicious activity. Mardu's use of XoM makes any code probing attempt trigger a process crash and consequently system-wide re-randomization. Therefore, Mardu counters direct memory disclosure attacks as well as code inference attacks requiring initial code probing [53, 60]. We use Intel MPK [37] to implement XoM for Mardu; leveraging MPK makes hiding, protecting, and legitimately using code much more efficient and simple with page permissions compared to virtualization-based designs that require nested address translation during runtime.

Security: Preventing code and code pointer leakage. In addition to system-wide re-randomization, Mardu minimizes the leakage of code and code pointers. Besides XoM, we use three techniques. First, Mardu applications always go through a trampoline region to enter into or return from a function. Thus, only trampoline addresses are stored in memory (e.g., stack and heap), whereas non-trampoline code pointers remain hidden. Mardu does not randomize the trampoline region, so tracking and patching are not needed upon re-randomization. Second, Mardu performs fine-grained function-level randomization within an executable (e.g., `libc.so`) to completely disconnect any correlation between trampoline addresses and code addresses. This provides high entropy (i.e., roughly $n!$, where n is the number of functions). In addition, unlike re-randomization approaches that rely on shifting code base addresses [8, 12, 47], Mardu is not susceptible to code pointer offsetting attacks (A4). Finally, Mardu stores return addresses—precisely, trampoline addresses for return—in a shadow stack. This design makes stack pivoting practically infeasible.

Design overview. As shown in Figure 1, Mardu is composed of compiler and kernel components. The Mardu compiler enables trampolines and a shadow stack to be used. The Mardu compiler generates PC-relative code so that randomized code can be shared by multiple processes. In addition, the compiler generates and attaches additional metadata to binaries for efficient patching.

The Mardu kernel is responsible for choreographing the runtime when a Mardu-enabled executable is launched. The kernel extracts and loads Mardu metadata into a cache to be shared by multiple processes. This metadata is used for first load-time randomization as well as re-randomization. The randomized code is cached and shared by multiple processes; while allowing sharing, each process will get a different random virtual

address space for the shared code. The MARDU kernel prevents read operations of the code region, including the trampoline region, using XoM such that trampoline addresses do not leak information about non-trampoline code. Whenever a process crashes (e.g., XoM violation), the MARDU kernel re-randomizes all associated shared code such that all relevant processes are re-randomized to thwart an attacker's knowledge immediately.

4.2 MARDU Compiler

The MARDU compiler generates a binary able to (1) hide its code pointers, (2) share its randomized code among processes, and (3) run under XoM. MARDU uses its own calling convention using a trampoline region and shadow stack.

4.2.1 *Code Pointer Hiding.*

Trampoline. MARDU hides code pointers without paying for costly runtime code pointer tracking. The key idea for enabling this is to split a binary into two regions in process memory: *trampoline* and *code* regions (as shown in Figures 2 and 3). A trampoline is an intermediary call site that moves control flow securely to/from a function body, protecting the XoM hidden code region. There are two kinds of trampolines: call and return trampolines. A *call trampoline* is responsible for forwarding control flow from an instrumented call to the *code region* function entry, whereas a *return trampoline* is responsible for returning control flow semantically to the caller. Each function has one call trampoline to its function entry, and each call site has one return trampoline returning to the following instruction of the caller. Since trampolines are stationary, MARDU does not need to track code pointers upon re-randomization because only stationary call trampoline addresses are exposed to memory.

Shadow stack. Unlike the x86 calling convention using `call`/`ret` to store return addresses on the stack, MARDU instead stores all return addresses in a shadow stack and leaves data destined for the regular stack untouched. Effectively, this protects all backward-edges. A MARDU `call` pushes a return trampoline address onto the shadow stack and jumps to a call trampoline; an instrumented `ret` directly jumps to the return trampoline address at the current top of the shadow stack. MARDU assumes a 64-bit address space and ability to leverage a segment register (e.g., `%gs`); the base address of the MARDU shadow stack is randomized by ASLR and is hidden in `%gs`, which cannot be modified in userspace and will never be stored in memory.

Running example. Figure 2 is an example of executing a MARDU-compiled function `foo()`, which calls a function `bar()` and then returns. Every function call and return goes through trampoline code that stores the return address to a shadow stack. The body of `foo()` is entered via its call trampoline ❶. Before `foo()` calls `bar()`, the return trampoline address is stored onto the shadow stack. Control flow then jumps to `bar()`'s trampoline ❷, which will jump to the function body of `bar()` ❸. `bar()` returns to the address in the top of the shadow stack, which is the return trampoline address ❹. Finally, the return trampoline returns to the instruction following the call in `foo()` ❺.

4.2.2 *Enabling Code Sharing Among Processes.*

PC-relative addressing. The MARDU compiler generates PC-relative (i.e., position-independent) code so that it can be shared among processes loading the same code in different virtual addresses. The key challenge here is *how to incorporate PC-relative addressing with randomization.* MARDU randomly places code (at function granularity) while trampoline regions remain stationary. This means any code using PC-relative addressing must be correspondingly patched once its randomized location is decided. In Figure 2, all jump targets between the trampoline and code, denoted in yellow rectangles, are PC-relative and must be adjusted. All data addressing instructions (accessing global data, GOT, etc.) must also be adjusted.

Fixup information for patching. With this policy, it is necessary to keep track of these instructions to patch them properly during runtime. Similar to **compiler-assisted code randomization (CCR)** [42], MARDU makes its runtime patching process simple and efficient by leveraging the LLVM compiler to collect and generate metadata, like fixups and relocations, into the binary describing exact locations for patching and their file-relative

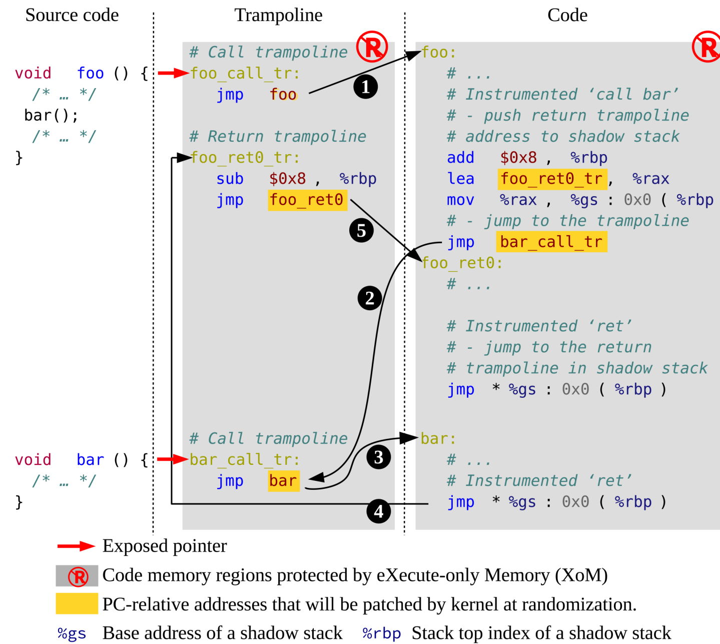


Fig. 2. Illustrative example executing a MARDU-compiled function `foo()`, which calls a function `bar()` and then returns.

offset. Reading this information from the newly generated section in the executable, this fixup information makes patching as simple as just adjusting PC-relative offsets for given locations (see Figure 3). However, CCR only uses that information once, relying on a binary rewriter for a single static user-side binary executable randomization at function and basic-block granularity. Contrary to CCR, MARDU leverages the metadata added to allow processes to behave as runtime code rewriters, and re-randomize on-demand. The overhead of runtime patching is negligible because MARDU avoids "stopping the world" when patching the code to maintain internal consistency compared to other approaches, putting the burden on the crashed process. We elaborate on the patching process in Section 4.3.2.

Supporting a shared library. A call to a shared library is treated the same as a normal function call to preserve MARDU's code pointer hiding property—that is, MARDU refers to the call trampoline for the shared library call via a procedure linkage table or **global offset table (GOT)** whose address is resolved by the dynamic linker as usual. Although MARDU does not specifically protect GOT, we assume that the GOT is already protected. For example, Fedora systems that support MPK have been hardened to enforce lazy binding will use a read-only GOT [22].

4.3 MARDU Kernel

The MARDU kernel randomizes code at load-time and runtime. It maps already-randomized code, if it exists, to the address space of a newly `fork`-ed process. When an application crashes, MARDU re-randomizes all mapped binaries associated with the crashing process and reclaims the previous randomized code from the cache after all processes are moved to a newly re-randomized code. MARDU prevents direct reading of randomized code from userspace using XoM. MARDU is also responsible for initializing a shadow stack for each task.[1]

[1]In this article, the term *task* denotes both process and thread as the convention in Linux kernel.

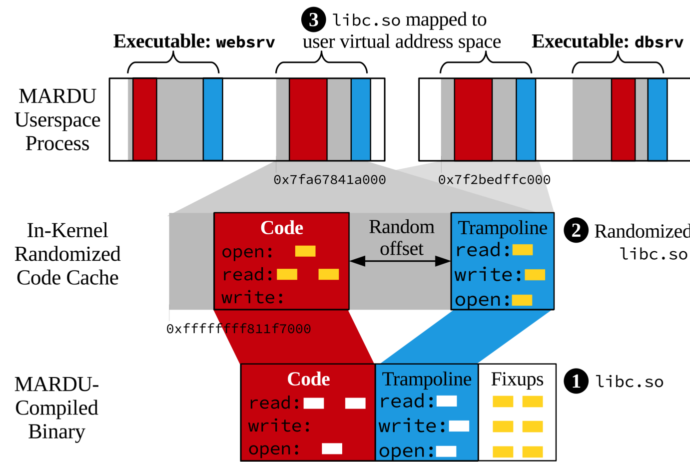


Fig. 3. The memory layout of two Mardu processes: `websrv` (top left) and `dbsrv` (top right). The randomized code in kernel (`0xffffffff811f7000`) is shared by multiple processes, which is mapped to its own virtual base address (`0x7fa67841a000` for `websrv` and `0x7f2bedffc000` for `dbsrv`).

4.3.1 Process Memory Layout. Figure 3 illustrates the memory layout of two Mardu processes. The Mardu compiler generates a PC-relative binary with trampoline code and fixup information ❶. When a binary is loaded to be mapped to a process with executable permissions, the Mardu kernel first performs a one time extraction of all Mardu metadata in the binary and associates it on a per-file basis. Extracting metadata gives Mardu the information it needs to perform (re-)randomization ❷. Note that load-time randomization and runtime re-randomization follow the exact same procedure. Mardu first generates a random offset to set apart the code and trampoline regions and then places functions in a random order within the code region. Once functions are placed, Mardu then uses the cached Mardu metadata to perform patching of offsets within both the trampoline and code regions to preserve program semantics. With the randomized code now semantically correct, it can be cached and mapped to multiple applications ❸.

Whenever a new task is created (`clone`), the Mardu kernel allocates a new shadow stack and copies the parent's shadow stack to its child; it is placed in the virtual code region created by the Mardu kernel. The base address of the Mardu shadow stack is randomized by ASLR and is hidden in segment register `%gs`. Any crash, such as brute-force guessing of base addresses, will trigger re-randomization, which invalidates all prior information gained, if any. To minimize the overhead incurred from using a shadow stack, Mardu implements its own compact shadow stack without comparisons [11]. For our shadow stack implementation, we reserve one register, `%rbp`, to use exclusively as a stack top index of the shadow stack to avoid costly memory access.

4.3.2 Fine-Grained Code Randomization.

Allocating a virtual code region. For each randomized binary, the Mardu kernel allocates a 2-GB *virtual* address region[2] (Figure 3 ❷), which will be mapped to userspace virtual address space with coarse-grained ASLR (Figure 3 ❸).[3] The Mardu kernel positions the trampoline code at the end of the virtual address region and returns the start address of the trampoline via `mmap`. The trampoline address remains static throughout program execution even after re-randomization.

[2]We note that for the unused region, we map all of those virtual addresses to a single abort page that generates a crash when accessed to not waste real physical memory and also detect potential attack attempts.

[3]We choose 2-GB because in x86–64 architecture, PC-relative addressing can refer to a maximum of ±2-GB range from `%rip`.

Randomizing the code within the virtual region. To achieve a high entropy, the MARDU kernel uses fine-grained randomization within the allocated virtual address region. Once the trampoline is positioned, the MARDU kernel randomly places non-trampoline code within the virtual address region; MARDU decides a *random offset* between the code and trampoline regions. Once the code region is decided, MARDU permutes the function order within the code region to further increase entropy. As a result, trampoline addresses do not leak information on non-trampoline code and an adversary cannot infer any actual codes' location from the system information (e.g., `/proc/<pid>/maps`), as they will get the same mapping information for the entire 2-GB region.

Patching the randomized code. After permuting functions, the MARDU kernel patches PC-relative instructions accessing code or data according to the randomization pattern. This patching process is trivial at runtime; the MARDU compiler generates fixup location information, and the MARDU kernel re-calculates and patches PC-relative offsets of instructions according to the randomized function location. Note that patching includes control flow transfer between trampoline and non-trampoline code, global data access (i.e., `.data`, `.bss`), and function calls to other shared libraries (i.e., procedure linkage table/GOT).

4.3.3 Randomized Code Cache. The MARDU kernel manages a cache of randomized code. When a userspace process tries to map a file with executable permissions, the MARDU kernel first looks up if there already exists a randomized code of the file. If cache hits, the MARDU kernel maps the randomized code region to the virtual address of the requested process. Upon cache miss, it performs load-time randomization as described earlier. The MARDU kernel tracks how many times the randomized code region is mapped to userspace. If the reference counter is zero or system memory pressure is high, the MARDU kernel evicts the randomized code. Thus, in normal cases without re-randomization, MARDU randomizes a binary file only once (load-time). In MARDU, the randomized code cache is associated with the `inode` cache. Consequently, when the `inode` is evicted from the cache under severe memory pressure, its associated randomized code is also evicted.

4.3.4 Execute-Only Memory. We designed XoM based on Intel MPK [37].[4] With MPK, each page is assigned to one of 16 domains under a *protection key*, which is encoded in a page table entry. Read and write permissions of each domain can be independently controlled through an MPK register. When randomized code is mapped to userspace, the MARDU kernel configures the XoM domain to be non-accessible (i.e., neither readable nor writable in userspace), and assigns code memory pages to the created XoM domain, enforcing execute-only permissions. If an adversary tries to read XoM-protected code memory, re-randomization is triggered via the raised XoM violation. Unlike EPT-based XoM designs [61] that require system resources to enable virtualization as well as have inherent overhead from nested address translation, our MPK-based design does not impose such runtime overhead.

4.3.5 On-Demand Re-randomization.

Triggering re-randomization. When a process crashes, MARDU triggers re-randomization of *all* binaries mapped to the crashing process. Since MARDU re-randomization thwarts the attacker's knowledge (i.e., each attempt is an independent trial), an adversary must succeed in her first try without crashing, which is practically infeasible.

Re-randomizing code. Upon re-randomization, the MARDU kernel first populates another copy of the code (e.g., `libc.so`) in the code cache and freshly randomizes it (Figure 4 ❶). MARDU leaves trampoline code at the same location to avoid mutating code pointers, but it does randomly place non-trampoline code (via new random offset) such that the new version does not overlap with the old one. Then, it permutes functions in the code. Thus, re-randomized code is completely different from the previous one without changing trampoline addresses.

Live thread migration without stopping the world. Re-randomized code prepared in the previous step is not visible to userspace processes because it is not yet mapped to userspace. To make it visible, MARDU first maps

[4]As of this writing, Intel Xeon Scalable Processors [38] and Amazon EC2 C5 instance [3] support MPK. Other than x86, ARM AArch64 architecture also supports XoM [4].

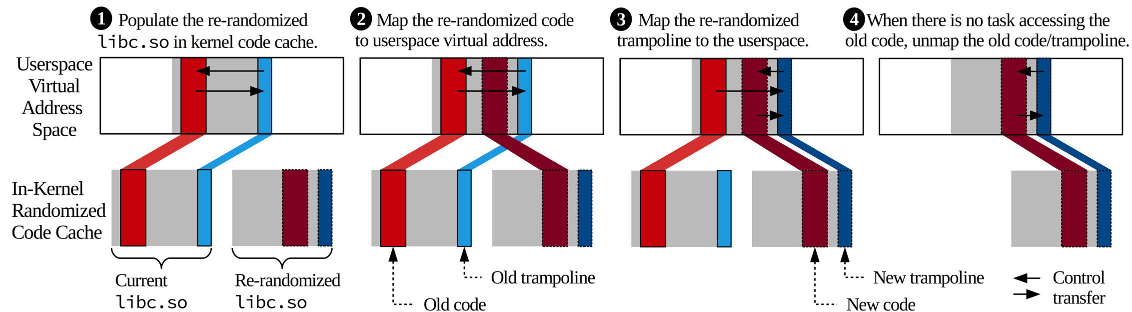


Fig. 4. Re-randomization procedure in Mardu. Once a new re-randomized code is populated ❶, the Mardu kernel maps new code and trampoline in order ❷, ❸. This makes threads crossing the new trampoline migrate to the newly re-randomized code. After it is guaranteed that all threads are migrated to the new code, Mardu reclaims the old code ❹. Unlike previous continuous per-process re-randomization approaches, our re-randomization is time-bound, efficient, and system-wide.

the new non-trampoline code to the application's virtual address space, as shown in Figure 4 ❷. The old trampolines are left mapped, making new code not reachable. Once Mardu remaps the virtual address range of the trampolines to the new trampoline code by updating corresponding page table entries ❸, the new trampoline code will transfer control flow to the new non-trampoline code. Hereafter, any thread crossing the trampoline migrates to the new non-trampoline code without stopping the world.

Safely reclaiming the old code Mardu can safely reclaim the code only after all threads migrate to the new code ❹. Mardu uses *reference counting* for each randomized code to check if there is a thread accessing the old code. After the new trampoline code is mapped ❸, Mardu sets a reference counter of the old code to the number of all *runnable* tasks[5] that map the old code. It is not necessary to wait for the migration of a non-runnable, sleeping task, because it will correctly migrate to the newest randomized code region when it passes through the return trampoline, which refers to the new layout when it wakes up. The reference counter is decremented when a runnable task enters into the Mardu kernel due to system call or preemption. When calling a system call, the Mardu kernel will decrement reference counters of all code that needs to be reclaimed. When the task returns to userspace, it will return to the return trampoline and the return trampoline will transfer to the new code. When a task is preempted out, it may be in the middle of executing the old non-trampoline code. Thus, the Mardu kernel not only decrements reference counters but also translates `%rip` of the task to the corresponding address in the new code. Since Mardu permutes at function granularity, `%rip` translation is merely adding an offset between the old and new function locations.

Summary. Our re-randomization scheme has three nice properties: (1) time boundness of re-randomization, (2) almost zero overhead of running process, and (3) system-wide re-randomization. Because Mardu migrates *runnable* tasks at system call and scheduling boundaries, it ensures that Mardu re-randomization will always guarantee the process to use the newly secure version of Mardu-enabled code once awoken or has crossed the system call boundary. Just as important, processes will *never* have access to the attacker exposed code ever again once crossing those boundaries. If another process crashes in the middle of re-randomization, Mardu will not trigger another re-randomization until the current randomization finishes. However, as soon as the new randomized code is populated ❶, a new process will map the new code immediately. Therefore, the old code cannot be observed more than once. The Mardu kernel populates a new randomized code in the context of a crashing process. All other runnable tasks only additionally perform reference counting or translation of `%rip` to the new code. Thus, its runtime overhead for runnable tasks is negligible. To the best of our knowledge, Mardu is the first system to perform system-wide re-randomization allowing code sharing.

[5] A task in a `TASK_RUNNING` status in Linux kernel.

5 IMPLEMENTATION

We implemented Mardu on the Linux x86-64 platform. The Mardu compiler is implemented using LLVM 6.0.0, and the Mardu kernel is implemented based on Linux kernel 4.17.0, modifying 3549 and 4009 lines of code, respectively. We used `musl libc` 1.1.20 [1], a fast, light-weight C standard library for Linux. We chose `musl libc` because `glibc` is not able to be compiled with LLVM/Clang. We manually wrapped all inline assembly functions present in `musl` to allow them to be properly identified and instrumented by the Mardu compiler. We modified 164 lines of code in `musl libc` for the wrappers.

5.1 Mardu Compiler

Trampoline. The Mardu compiler is implemented as backend target-ISA (x86) specific `MachineFunctionPass`. This pass instruments each function body as described in Section 4.2.

Re-randomizable code. The following compiler flags are used by the Mardu compiler: `-fPIC` enables instructions to use PC-relative addressing; `-fomit-frame-pointer` forces the compiler to relinquish use of register `%rbp`, as register `%rbp` is repurposed as the stack top index of a shadow stack in Mardu; `-mrelax-all` forces the compiler to always emit full 4-byte displacement in the executable, such that the Mardu kernel can use the full span of memory within our declared 2-GB virtual address region and maximize entropy when performing patching; and last, the Mardu compiler ensures that code and data are segregated in different pages via using `-fno-jump-tables` to prevent false-positive XoM violations.

5.2 Mardu Kernel

Random number generation. Mardu uses a cryptographically secure random number generator in Linux based on hardware instructions (i.e., `rdrand`) in modern Intel architectures. Alternatively, Mardu can use other secure random sources such as `/dev/random` or `get_random_bytes()`.

5.3 Limitation of Our Prototype Implementation

Assembly code. Mardu does not support inline assembly as in `musl`; however, this could be resolved with further engineering. Our prototype uses wrapper functions to make assembly comply with the Mardu calling convention.

Setjmp and exception handling. Mardu uses a shadow stack to store return addresses. Thus, functions such as `setjmp`, `longjmp`, and `libunwind` that directly manipulate return addresses on stack are not supported by our prototype. Adding support for these functions could be resolved by porting these functions to understand our shadow stacks semantics, as our shadow stack is a variant of compact, register-based shadow stack [11].

C++ support. Our prototype does not support C++ applications, as we do not have a stable standard C++ library that is `musl`-compatible.

6 EVALUATION

We evaluate Mardu by answering these questions:

- How secure is Mardu, when presented against current known attacks on randomization? (Section 6.1)
- How much performance overhead does the needed instrumentation of Mardu impose, particularly for compute-intensive benchmarks in a typical runtime without any attacks? (Section 6.2)
- How scalable is Mardu in terms of load-time, re-randomization time with and without ongoing attacks, and memory savings, particularly for concurrent processes such as in a real-world network facing server? (Section 6.3)

Applications. We evaluate the performance overhead of Mardu using SPEC CPU2006. This benchmark suite has realistic compute-intensive applications, ideal to see worst-case performance overhead of Mardu. We tested

all 12 C language benchmarks using input size *ref*; we excluded C++ benchmarks as our current prototype does not support C++. We choose SPEC CPU2006 over SPEC CPU2017 to easily compare Mardu to prior relevant re-randomization techniques. We test performance and scalability of Mardu on a complex, real-world multi-process web server with NGINX.

Experimental setup. All programs are compiled with optimization `-O2` and run on a 24-core (48-hardware threads) machine equipped with two Intel Xeon Silver 4116 CPUs (2.10 GHz) and 128 GB of DRAM.

6.1 Security Evaluation

We analyze the resiliency of Mardu against existing attacker models with load-time randomization (A1–A2, Section 6.1.1) and continuous re-randomization. (A3–A4, Section 6.1.2). Then, to illustrate the effectiveness of Mardu for a wider class of code reuse attacks beyond ROP, we discuss the threat model of NEWTON [68] with Mardu (Section 6.1.3).

6.1.1 Attacks Against Load-Time Randomization.

Against JIT-ROP attacks. Mardu asserts permissions for all code areas and trampoline regions as execute-only (via XoM); thereby, JIT-ROP cannot read code contents directly.

Against code inference attacks. Mardu blocks code inference attacks, including BROP [9], clone-probing [47], and destructive code read attacks [53, 60] via layout re-randomization triggered by an application crash or XoM violation. Every re-randomization renders all previously gathered (if any) information regarding code layout invalid and therefore prevents attackers from accumulating indirect information. Note that attacks such as address-oblivious code reuse [57] do not fall into the category of A2. This attack vector's process of control hijacking more closely resembles full-function code reuse rather than indirect exposure of code; address-oblivious code reuse leverages manipulation of data and function pointer corruption and does not require usage of `ret` gadgets.

Hiding shadow stack. Attackers with arbitrary read/write capability (A1/A2) may attempt to leak/alter shadow stack contents if its address is known. Although the location of the shadow stack is hidden behind the `%gs` register, attackers may employ attacks that undermine this sparse-memory based information hiding [21, 29, 50]. To prevent such attacks, Mardu reserves a 2-GB virtual memory space for the shadow stack (the same way Mardu allocates code/library space) and chooses a random offset to map the shadow stack; all other pages in the 2-GB space are mapped as an abort page. Regarding randomization entropy of shadow stack hiding, we take an example of a process that uses 16 pages for the stack. In such a case, the possible shadow stack positions are as follows:

$$\begin{aligned} \#\ of\ positions &= (MEMSIZE - STACKSIZE)/PAGESIZE \\ &= (2^{31} - 16 * 2^{12})/2^{12} = 524,272, \end{aligned} \tag{1}$$

thereby the probability of successfully guessing a valid shadow stack address is one in 524,272, practically infeasible. Even assuming if attackers are able to identify the 2-GB region for the shadow stack, they must also overcome the randomization entropy of the offset to get a valid address within this region (winning chance: roughly one in 2^{31}, as Mardu's shadow stack can start at an arbitrary address within a page and not align to the 4K-page boundary); any incorrect probe will generate a crash, trigger re-randomization, thwarting the attack.

Entropy. Mardu applies both function-level permutation and random start offset to provide a high entropy to the randomized code layout. In particular, Mardu permutates all functions in each executable at each time of randomization. In this way, randomization entropy (E_{func}) depends on the number of functions in the executable (n), and the entropy gain can be formulated as follows:

$$E_{func} = log_2(n!). \tag{2}$$

Additionally, Mardu applies a random start offset to the code area in 2-GB space in each randomization. Because the random offset could be anywhere in 2-GB range excluding the size of trampoline region and twice

the size of the program (to avoid overlapping), the entropy gain by the random offset (E_{off}) can be formulated as

$$E_{off} = log_2\left(2^{31} - sizeof(trampoline) - 2\times sizeof(program)\right), \tag{3}$$

and the total entropy that Mardu provides is

$$E_{\text{MARDU}} = E_{func} + E_{off}. \tag{4}$$

We take an example of `470.lbm` in SPEC CPU2006, a case which provides the minimum entropy in our evaluation. The program contains 16 functions, and the entire size of the program including trampoline instrumentation is less than 64 KB. In such a case, the total entropy is as follows:

$$\begin{aligned} E_{func} &= log_2(16!) > 44.25, \\ E_{off} &= log_2\left(2^{31} - 2\times 64K\right) > 30.99, \end{aligned} \tag{5}$$

$$E_{\text{MARDU}} = E_{func} + E_{off} > 74.24. \tag{6}$$

Therefore, even for a small program, Mardu randomizes the code with significantly high entropy (74 bits) to render an attacker's success rate for guessing the layout negligible.

6.1.2 Attacks Against Continuous Re-randomization.

Against low-profile attacks (A3). Mardu does not rely on timing nor system call history for triggering re-randomization. As a result, neither low-latency attacks nor attacks without involving system calls are effective against Mardu. Instead, re-randomization is triggered and performed by any Mardu instrumented application process that encounters a crash (e.g., XoM violation). Nonetheless, a potential A3 vector could be one that does not cause any crash during exploitation (e.g., attackers may employ crash-resistant probing [21, 24, 29, 41, 50]). In this regard, Mardu places all code in XoM within a 2-GB mapped region. Such a stealth attack could only identify multiples of 2-GB code regions and will fail to leak any layout information.

Against code pointer offsetting attacks (A4). Attackers may attempt to launch this attack by adding/subtracting offsets to a pointer. To defend against this, Mardu decouples any correlation between trampoline function *entry* addresses and function *body* addresses (i.e., no fixed offset), so attackers cannot refer to the middle of a function for a ROP gadget without actually obtaining a valid function body address. Additionally, the trampoline region is also protected with XoM, thus attackers cannot probe it to obtain function body addresses to launch A4. Mardu limits available code reuse targets to only exported functions in the trampoline.

6.1.3 Beyond ROP Attacks.

Attack analysis with NEWTON. To measure the boundary of viable attacks against Mardu, we present a security analysis of Mardu based on the threat model set by NEWTON [68]. In this regard, we analyze possible writable pointers that can change the control flow of a program (write constraints) as well as possible available gadgets in Mardu (target constraints), which will reveal what attackers can do under this threat model. In short, Mardu allows only the reuse of exported functions via call trampolines.

For write constraints, attackers cannot overwrite real code addresses such as return addresses or code addresses in the trampoline. Mardu only allows attackers to overwrite other types of pointer memory, such as object pointers and pointers to the call trampoline. For target constraints, attackers can reuse only the exported functions via call trampoline. Note that a function pointer is a reusable target in any re-randomization techniques using immutable code pointers [12, 70, 74]. Although Mardu allows attackers to reuse function pointers in accessible memory (e.g., a function pointer in a structure), such live addresses will never include real code addresses or return addresses, and will be limited to addresses only referencing call trampolines. Under these write and target constraints, inferring the location of ROP gadgets from code pointers (e.g., leaking code addresses or adding an offset) is not possible.

6.2 Performance Evaluation

Runtime performance overhead with SPEC CPU2006. Figure 5 shows the performance overhead of SPEC with MARDU trampoline-only instrumentation (which does not use a shadow stack) as well as with a full MARDU implementation. Both of these numbers are normalized to the unprotected and uninstrumented baseline, compiled with vanilla Clang. Note that this performance overhead is the base incurred overhead of security hardening an application with MARDU. In the rare case that the application were to come under attack, on-demand re-randomization would be triggered inducing additional brief performance overheads. We discuss the performance overhead of MARDU under active attack in Section 6.3.

Figure 5 does not include a direct performance comparison to other randomization techniques, as MARDU is substantially different in how it implements re-randomization and the source code of closely related systems, such as Shuffler [74] and CodeArmor [12], is not publicly available. It is not based on timing nor system call history compared to previous works. This peculiar approach allows MARDU's average overhead to be comparable to the fastest re-randomization systems and its worst-case overhead to be significantly better than similar systems. The average overhead of MARDU is 5.5%, and the worst-case overhead is 18.3% (`perlbench`), in comparison to Shuffler [74] and CodeArmor [12], whose reported average overheads are 14.9% and 3.2%, and their worst-case overhead are 45% and 55%, respectively (see Table 1). TASR [8] shows a very practical average overhead of 2.1%; however, it has been reported by Shuffler [74] and ReRanz [70] that TASR's overhead against a more realistic baseline (not using compiler flag `-Og`) is closer to 30% to 50% overhead. This confirms that MARDU is capable of matching if not slightly improving the performance (especially worst-case) overhead while casting a wider net in terms of known attack coverage.

MARDU's two sources of runtime overhead are trampolines and the shadow stack. MARDU uses a compact shadow stack without a comparison epilogue whose sole purpose is to secure return addresses. Specifically, only four additional assembly instructions are needed to support our shadow stack. Therefore, we show the trampoline-only configuration to clearly differentiate the overhead contribution of each component. Figure 5 shows that MARDU's shadow stack overhead is negligible with an average of less than 0.3%, and in the noticeable gaps adding less than 2% in `perlbench`, `gobmk`, and `sjeng`. The overhead in these three benchmarks comes from the higher frequency of short function calls, making shadow stack updates not amortize as well as in other benchmarks. In the cases where Full MARDU is actually faster than the trampoline-only version (e.g., `bzip2`, `gcc`, and `h264ref`), we investigated and found that our handcrafted assembly for integrating the trampolines with the regular stack in the trampoline-only version can inadvertantly cause elevated amounts of branch misses, leading to the expected performance slowdown.

6.3 Scalability Evaluation

Runtime performance overhead with NGINX. NGINX is configured to handle a max of 1,024 connections per processor, and its performance is observed according to the number of worker processes. `wrk` [28] is used to generate HTTP requests for benchmarking. `wrk` spawns the same number of threads as NGINX workers, and each `wrk` thread sends a request for a 6,745-byte static html. To see worst-case performance, `wrk` is run on the same machine as NGINX to factor out network latency, unlike Shuffler. Figure 6 presents the performance of NGINX with and without MARDU for a varying number of worker processes. The performance observed shows that MARDU exhibits quite similar throughput to vanilla. MARDU incurs 4.4%, 4.8%, and 1.2% throughput degradation on average, at peak (12 threads), and at saturation (24 threads), respectively. Note that Shuffler [74] suffers from overhead from its *per-process* shuffling thread; just enabling Shuffler essentially doubles CPU usage. Even in their NGINX experiments with network latency (i.e., running a benchmarking client on a different machine), Shuffler shows 15% to 55% slowdown. This verifies MARDU's design that having a crashing process perform system-wide re-randomization, rather than a per-process background thread as in Shuffler, scales better.

Load-time randomization overhead. We categorize load-time to cold or warm load-time whether the in-kernel code cache (❷ in Figure 3) hits or not. Upon a code cache miss (i.e., the executable is first loaded in a system),

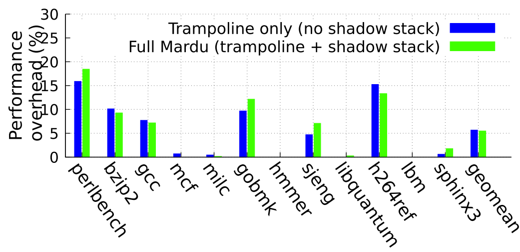


Fig. 5. MARDU performance overhead breakdown for SPEC.

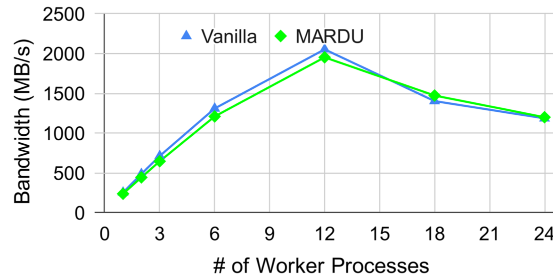


Fig. 6. Performance comparison of NGINX web server.

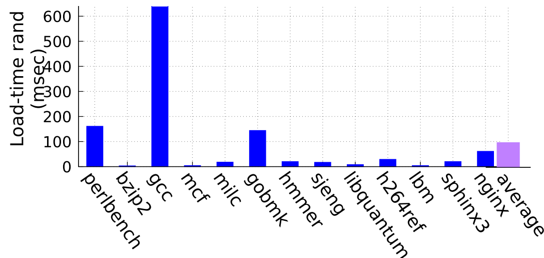


Fig. 7. Cold load-time randomization overhead.

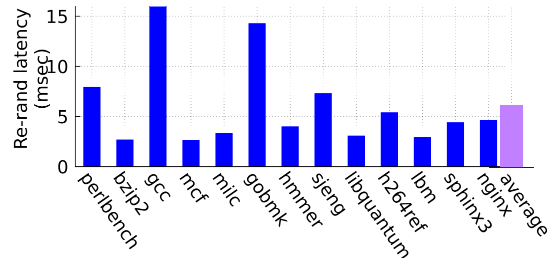
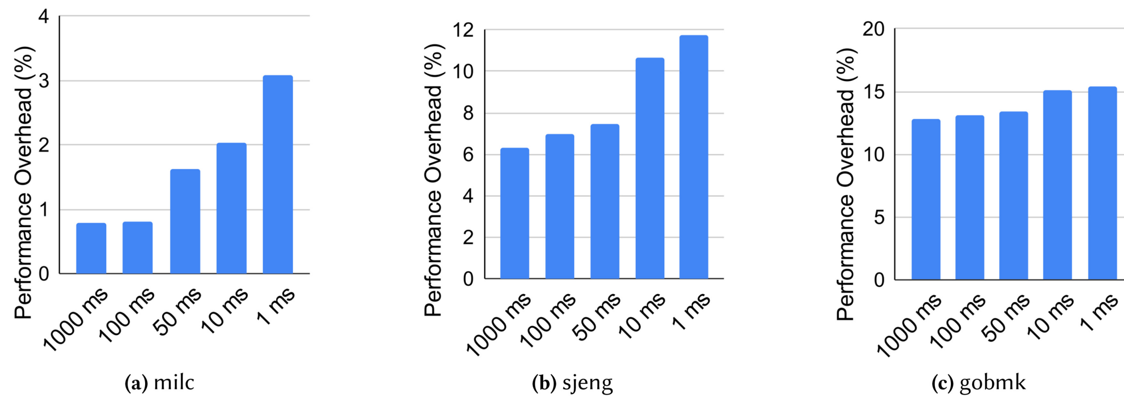


Fig. 8. Runtime re-randomization latency.

MARDU performs initial randomization including function-level permutation, start offset randomization of the code layout, and loading and patching of fixup metadata. As Figure 7 shows, all C SPEC benchmarks showed negligible overhead averaging 95.9 msec. `gcc`, being the worst case, takes 771 msec; it requires the most (291,699 total) fixups relative to other SPEC benchmarks, with ≈9,372 fixups on average. `perlbench` and `gobmk` are the only other outliers, having 103,200 and 66,900 fixups, respectively; all other programs have <<35K fixups (refer to Table 2). For NGINX, we observe that load-time is constant (61 msec) for any number of specified worker processes. Cold load-time is roughly linear to the number of trampolines. Upon a code cache hit, MARDU simply maps the already-randomized code to a user-process's virtual address space. Therefore, we found that warm load-time is negligible. Note that a cold load-time of `musl` takes about 52 msec on average. Even so, this is a one-time cost; all subsequent warm load-time accesses of fetching `musl` takes below 1 μsec, for any program needing it. Thus, load-time can be largely ignored.

Re-randomization latency. Figure 8 presents time to re-randomize all associated binaries of a crashing process. The time includes creating and re-randomizing a new code layout, and reclaiming old code (❶–❹ in Figure 4). We emulate an XoM violation by killing the process via a `SIGBUS` signal and measured re-randomization time inside the kernel. The average latency of SPEC is 6.2 msec. The performance gained between load-time and re-randomization latency is from MARDU taking advantage of metadata being cached from load-time, meaning that no redundant file I/O penalty is incurred. To evaluate the efficiency of re-randomization on multi-process applications, we measured the re-randomization latency with varying number of NGINX worker processes up to 24. We confirm that latency is consistent regardless of the number of workers (5.8 msec on average, 0.5 msec standard deviation).

Re-randomization overhead under active attacks. In addition, a good re-randomization system should exhibit good performance not only in its idle state but also under stress from active attacks. To evaluate this, we stress test MARDU under frequent re-randomization to see how well it can perform, assuming a scenario that MARDU



(a) milc (b) sjeng (c) gobmk

Fig. 9. Overhead varying re-randomization frequency.

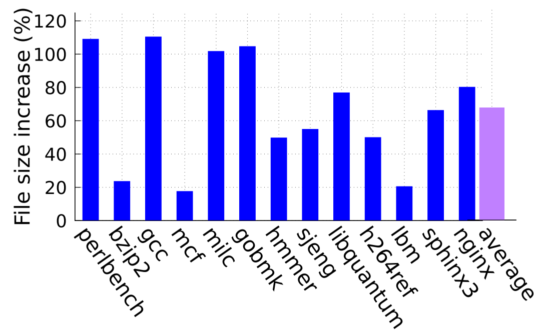


Fig. 10. File size increase with Mardu compilation.

is under attack. In particular, we measure the performance of SPEC benchmarks while triggering frequent re-randomization. We emulate the attack by running a background application, which continuously crashes at the given periods: 1 second, 100 msec, 50 msec, 10 msec, and 1 msec. SPEC benchmarks and the crashing application are linked with the Mardu version of `musl`, forcing Mardu to constantly re-randomize `musl` and potentially incur performance degradation on other processes using the same shared library. In this experiment, we choose three representative benchmarks, `milc`, `sjeng`, and `gobmk`, that Mardu exhibits a small, medium, and large overhead in an idle state, respectively. Figure 9 shows that the overhead is consistent and in fact is quite close to the performance overhead in the idle state observed in Figure 5. More specifically, all three benchmarks differ by less than 0.4% at a 1-second re-randomization interval. When we decrease the re-randomization period to 10 msec and 1 msec, the overhead is quickly saturated. Even at 1 msec re-randomization frequency, the additional overhead is under 6%. These results show that Mardu provides performant system-wide re-randomization even under active attack.

File size overhead. Figure 10 and Table 2 show how much binary files increase with Mardu compilation. In our implementation, file size increase comes from transforming the traditional x86-64 calling convention with the one designed for Mardu (besides calls to outside libraries). On average, Mardu compilation with trampolines increases the file size by 66%. One would assume that applications with more call sites incur a higher overhead, as we are adding five instructions for every `call` and four instructions for every `retq` (e.g., `perlbench`, `gcc`, `milc`, and `gobmk` are the only benchmarks with more than 100% increase, being 108%, 110%, 101%, and 104% respectively).

Table 2. Breakdown of MARDU Instrumentation

Benchmark	No. of Fixups				Binary Increase (bytes)		
	Call Tr.	Ret Tr.	PC-rel. addr	Total	Trampolines	Metadata	Total
perlbench	1,596	39,174	62,430	103,200	1,115,136	2,607,559	3,722,695
bzip2	66	926	896	1,888	17,568	78,727	96,295
gcc	4,015	118,617	169,067	291,699	3,074,672	6,276,870	9,351,542
mcf	23	94	208	325	1,824	19,056	20,880
milc	234	3,531	7,256	11,021	110,688	313,620	424,308
gobmk	2,388	22,880	41,632	66,900	726,176	3,085,208	3,811,384
hmmer	452	5,145	9,925	15,522	139,216	574,446	713,662
sjeng	129	1,368	5,418	6,915	58,912	250,234	309,146
libquantum	97	1,659	1,424	3,180	25,952	93,222	119,174
h264ref	508	5,874	14,824	21,206	278,240	714,629	992,869
lbm	16	75	260	351	1,920	16,549	18,469
sphinx3	308	4,958	8,010	13,276	103,920	409,814	513,734
NGINX	1,497	15,004	18,984	35,485	416,736	1,309,708	1,726,444
musl libc	4,400	10,009	7,594	22,003	192,153	1,238,071	1,430,224

Runtime memory savings. Although there is an upfront one-time cost for instrumenting with MARDU, the savings greatly outweigh this. To illustrate, we show a typical use case of MARDU in regard to shared code. `musl` is ≈800 KB in size, and instrumented is 2 MB. Specifically, `musl` has 14K trampolines and 7.6K fixups for PC-relative addressing, the total trampoline size is 190 KB, and the amount of loaded metadata is 1.2 MB (Table 2). Since MARDU supports code sharing, only one copy of `libc` is needed for the entire system. Backes and Nürnberger [6] and Ward et al. [71] also highlighted the code sharing problem in randomization techniques and reported a similar amount of memory savings by sharing randomized code. Finally, note that the use of our shadow stack does not increase the runtime memory footprint beyond the necessary additional memory page allocated to support the shadow stack and the increase in code size from our shadow stack instrumentation. MARDU solely relocates return addresses from the normal stack to the shadow stack.

System-wide performance estimation. Deploying MARDU system-wide for all applications and all shared libraries requires additional engineering effort of recompiling the entire Linux distribution. Instead, we get an estimate of how MARDU would perform on a regular Linux server during boot-time. We obtain this estimate based on the fact that MARDU's load-time overhead increases linearly with the total number of functions and call sites present in an application or library. We calculate the estimated boot overhead if the entire system were protected by MARDU. Referencing Figure 2, MARDU requires one trampoline per function containing one fixup, and one return trampoline per call site containing three fixups. In addition, all PC-relative instructions must be patched. Therefore, the total number of fixups to be patched is as follows:

$$Total\ \#\ Fixups = \#\ Functions + (\#\ Callsites * 3) + \#\ PC\ relative\ Instructions. \quad (7)$$

Extrapolating from MARDU's load-time randomization overhead in Figure 7, where `gcc` has the most fixups at 291,699 and takes 771 ms, this makes each fixup take approximately 2.6 μsec. We recorded all executables launched as well as all respective loaded libraries in our Linux server to calculate the additional overhead imposed by MARDU during boot-time. We included all programs run within the first 5 minutes of boot-time and looked at the current system load. In 5 minutes after the system booting, we recorded a total of 117 no longer active processes and recorded 265 currently active processes, using a total of 784 unique libraries. The applications contained a total of 8,862 functions, a total of 472,530 call sites, and 415,951 total PC-relative instructions. Using Equation (7), this gave a total of 1,842,403 fixups if all launched applications were MARDU-enabled. The libraries contained a total of 223,415 functions, a total of 4,450,488 call sites, and 2,514,676 total PC-relative instructions; this gave a total of 16,089,555 fixups for shared libraries. Using our estimation from `gcc`, we can approximate that patching all fixups including both application fixups and shared library fixups (a total of 17,931,958 fixups) for a MARDU-enabled Linux server will take roughly ≈46.6 additional seconds compared to a vanilla boot. To give a little more insight, application fixups contribute only ≈4.8 seconds of delay; the majority of overhead comes

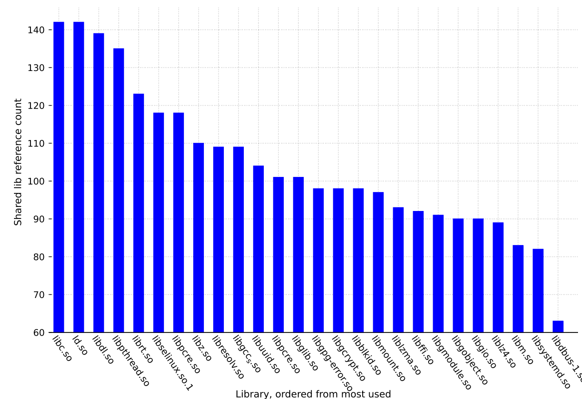


Fig. 11. Top 25 shared libraries with their reference count on our idle Linux server ordered from most linked to least linked libraries.

from randomization of the shared libraries. However, note that this delay is greatly amortized, as many libraries are shared by a large number of applications compared to the scenario where each library is not shared (e.g., statically linked) and needs a separate randomized copy for each application requiring it.

System-wide memory savings estimation. Similarly, we give a system-wide snapshot of memory savings observed when Mardu's randomized code sharing is leveraged. For this, we again use the same Linux server for system-wide estimation. The vanilla total file size of 784 unique libraries is approximately 787 MB. From our scalability evaluation (Figure 10) showing that Mardu roughly increases file size by 66% on average, this total file size would grow to 1,306 MB if all were instrumented with Mardu. Although this does appear to be a large increase, it is a one-time cost as code sharing is enabled under Mardu. From our Linux server having 265 processes, 127 of them had mapped libraries. If code sharing is not supported, each process needs its own copy of a library in memory. We counted each library use and multiplied by its size to get the total non-sharing memory usage. For our Linux server, this non-sharing would incur approximately an 8.8-GB overhead. This means that Mardu provides approximately 7.5-GB memory savings through its inherent code sharing design. This memory savings is compared to if these libraries were individually and separately statically linked to each of the running processes.

To get how many times each library is shared by multiple processes, we analyzed each process's memory mapping on our Linux server by investigating `/proc/{PID}/maps`. Figure 11 presents the active reference count for the 25 most linked shared libraries on our idle Linux server. The 25 most linked shared libraries are referenced more than ≈106 times on average, showing that dynamically linked libraries really do save a lot of memory compared to a non-shared approach. For the same 25 most linked shared libraries, we also demonstrate in Figure 12 the estimated memory savings obtained for each of those libraries if Mardu were used instead of an approach that does not support sharing of code. Notice that some of our biggest memory savings come from `libc.so` and `libm.so`, very commonly used libraries for which Mardu saves almost 0.80 GB and 0.25 GB of memory, respectively.

We also show the entire system snapshot (including all 784 unique libraries) in the form of a CDF for both the unique library link count in Figure 13 and cumulative memory savings in Figure 14 if Mardu were to be applied and used system-wide for all dynamically linked libraries. From Figure 13, it can be seen that approximately 150 libraries are in the 75th percentile of link count.

7 DISCUSSION AND LIMITATIONS

Applying Mardu to binary programs. Although our current Mardu prototype requires access to source code, applying Mardu directly to binary programs is possible. Mardu requires detecting all function call transfers

Fig. 12. Estimated runtime memory savings with a shared memory Mardu approach for the top 25 most linked libraries on our idle Linux server.

Fig. 13. CDF of shared library occurrence on an idle Linux server.

Fig. 14. CDF plot of estimated runtime memory savings with Mardu's shared memory approach.

(`call`/`ret`) and instrumenting them to use trampolines to keep control transfers semantically correct with re-randomization. A potential way to enable this is to apply techniques that can retrieve precise disassembly from a given binary, such as BYTEWEIGHT [7], to identify possible call targets. A more recent innovation, Egalito [75], a binary recompiler, showed that it is possible to raise (stripped) modern Linux binaries into a low-level intermediate representation. Their stand-alone, layout-agnostic intermediate representation is precise and allows arbitrary modifications. This approach would allow binary code or legacy binaries to be directly instrumented such that transfers utilize trampolines via binary re-writing. Leveraging reassemblable assembly, RetroWrite [20] also offers a binary transformation framework for 64-bit position-independent binaries. This work makes it plausible that Mardu could leverage their underlying binary-rewriting framework to instrument a popular and important class of binaries, as well as notably third-party shared libraries. Either of these recent approaches would enable Mardu to perform security hardening to software distributed as binaries to end users.

Security of protection keys for userspace. As briefly mentioned in Section 3, native MPK applications containing `wrpkru` instructions are not supported on a Mardu system. If there exists a running native MPK application that contains `wrpkru` instructions on the host system, there *does exist* an attack scenario that could victimize this native MPK application and break the guarantees and assumptions set by Mardu. To elaborate, if any applications have `wrpkru` instructions, it is possible for a cross-process MPK attack to occur such that all Mardu-guarded code (protected under an XoM domain) could be made accessible to the attacker (i.e., this could done via an attack vector not covered by Mardu, e.g., data/argument corruption), leveraging the secondary process containing `wrpkru` instructions, and effectively disabling Mardu.

To remedy this current limitation, Mardu could be extended to include and use HODOR [32] or ERIM [66] style approaches. Instrumenting hardware watchpoints to vet `wrpkru` instruction execution at runtime, or perform binary rewriting to generate functionally equivalent assembly where unintended `wrpkru` instructions occur in the code region, respectively, would then allow Mardu to support and properly protect applications containing native `wrpkru` instructions.

Full-function reuse attacks. Throughout our analysis, we show that existing re-randomization techniques that use a function trampoline or indirection table [12, 74] (i.e., use immutable (indirect) code pointer across re-randomization) cannot prevent full-function reuse attacks. This also affects MARDU; although limited to functions exposed in the trampoline, MARDU cannot defend against an attacker reusing such exposed immutable code pointers as gadgets by leaking code pointers, and we believe that this is a limitation of using immutable code pointers.

That being said, a possible workaround could be to utilize a monitoring mechanism limited to tracking function pointer assignment. Although this approach would be cumbersome, it will have a much smaller overhead because of its smaller scope leading to being more secure while producing less overhead than Shuffler [74].

Another possible solution to prevent these attacks could be pairing MARDU together with CFI [2, 13, 25, 30, 31, 46, 48, 49, 52, 55, 64, 67, 69, 76, 77], code pointer integrity/separation (CPI/CPS) [44], or other hardware-assisted solutions such as Intel's Control-flow Enforcement Technology (CET) [36] and ARM's pointer authentication code [56]. MARDU's defense is orthogonal to forward-edge protection like CFI. We say this because it is our hope that a technique (whether it is CFI-based or not) is made such that precise and efficient forward-edge security can be guaranteed; thus, applying both defenses can complement each other to provide better security. MARDU already provides precise backward-edge CFI via shadow stack, so forward-edge CFI can also be leveraged to further reduce available code reuse targets.

Note that completely eliminating full-function code reuse and data-oriented programming [35] with low performance overhead and system-wide scalability currently remains an open problem.

8 CONCLUSION

Current defense techniques are capable of defending against current ROP attacks; however, most designs inherently tradeoff well-rounded performance and scalability for their security guarantees. Hence, we introduce MARDU, a novel on-demand system-wide re-randomization technique to combat a majority of code reuse attacks. Our evaluation verifies MARDU's security guarantees against known ROP attacks and adequately quantifies its high entropy. MARDU's performance overhead on SPEC CPU2006 and multi-process NGINX averages 5.5% and 4.4%, respectively, showing that scalability can be achieved with reasonable performance. By being able to re-randomize on-demand, MARDU eliminates both costly runtime overhead and integral threshold components associated with current continuous re-randomization techniques. MARDU is the first code reuse defense capable of code-sharing *with* re-randomization to enable practical security that scales system-wide.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] Musl Libc. 2019. Home Page. Retrieved March 21, 2022 from https://wiki.musl-libc.org/.
[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*.
[3] Amazon. 2019. Amazon EC2 C5 Instances. Retrieved March 21, 2022 from https://aws.amazon.com/ec2/instance-types/c5/.
[4] ARM. 2019. ARM Compiler Software Development Guide: 2.21 Execute-Only Memory. Retrieved March 21, 2022 from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471j/chr1368698326509.html.
[5] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*.
[6] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium (Security'14)*.
[7] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (Security'14)*.

[8] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[10] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-resilient layout randomization for mobile devices. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS'16)*.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

[12] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (Euro S&P'17)*.

[13] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS'14)*.

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

[15] Stephen Crane, Andrei Homescu, and Per Larsen. 2016. Code randomization: Haven't we solved this problem yet? In *Proceedings of the 2016 IEEE Conference on Cybersecurity Development (SecDev'16)*. IEEE, Los Alamitos, CA, 124–129.

[16] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*.

[17] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Return to where? You can't exploit what you can't find. In *Proceedings of Black Hat USA*.

[18] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*.

[19] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS'15)*.

[20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP'20)*. IEEE, Los Alamitos, CA, 1497–1511.

[21] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*.

[22] Fedora. 2018. Hardening Flags Updates for Fedora 28. Retrieved March 21, 2022 from https://fedoraproject.org/wiki/Changes/HardeningFlags28.

[23] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, et al. 2019. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. 469–484.

[24] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS'16)*.

[25] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding control flows using Intel Processor Trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[26] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY'15)*.

[27] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium (Security'12)*.

[28] Will Glozer. 2019. A HTTP Benchmarking Tool. Retrieved March 21, 2022 from https://github.com/wg/wrk.

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

[30] Jens Grossklags and Claudia Eckert. 2018. τCFI: Type-assisted control flow integrity for x86–64 binaries. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'18)*.

[31] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent backward-edge control flow violation detection using Intel Processor Trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY'17)*.

[32] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*.

[33] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

[34] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: Transparent code randomization for just-in-time compilers. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*. 993–1004.

[35] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP'16)*. IEEE, Los Alamitos, CA, 969–986.

[36] Intel Corporation. 2019. Control-Flow Enforcement Technology Specification. Retrieved March 21, 2022 from. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[37] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Software Developer Manual. Retrieved March 21, 2022 from https://software.intel.com/en-us/articles/intel-sdm.

[38] Intel Corporation. 2019. Intel® Xeon® Scalable Processors. Retrieved March 21, 2022 from https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html.

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

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, et al. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

[41] Benjamin Kollenda, Enes Göktaş, Tim Blazytko, Philipp Koppe, Robert Gawlik, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, and Thorsten Holz. 2017. Towards automated discovery of crash-resistant primitives in binary executables. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN'17)*.

[42] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*.

[43] Hyungjoon Koo and Michalis Polychronakis. 2016. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM Symposium on Information, Computer, and Communications Security (ASIACCS'16)*.

[44] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, et al. 2018. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (Security'18)*.

[46] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and efficient CFI enforcement with Intel Processor Trace. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA'17)*.

[47] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to make ASLR win the clone wars: Runtime re-randomization. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS'16)*.

[48] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*.

[49] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. 914–926.

[50] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium (Security'16)*.

[51] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, 601–615.

[52] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium (Security'13)*.

[53] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. 2017. Breaking and fixing destructive code read defenses. In *Proceedings of the 12th ACM Symposium on Information, Computer, and Communications Security (ASIACCS'17)*. 55–67.

[54] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kRˆ X: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*.

[55] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS'15)*.

[56] Qualcomm Inc. 2017. Pointer Authentication on ARMv8.3. Retrieved March 21, 2022 from https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[57] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, et al. 2017. Address-oblivious code reuse: On the effectiveness of leakage resilient diversity. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS'17)*.

[58] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*.

[59] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

[60] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. 2016. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

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

[62] The Clang Team. 2021. Clang 12 Documentation: Control Flow Integrity. Retrieved March 21, 2022 from https://releases.llvm.org/12.0.0/tools/clang/docs/ControlFlowIntegrity.html.

[63] The PAX Team. 2003. Address Space Layout Randomization. Retrieved March 21, 2022 from https://pax.grsecurity.net/docs/aslr.txt.

[64] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (Security'14)*.

[65] Sami Tolvanen. 2018. Control Flow Integrity in the Android Kernel. Retrieved March 21, 2022 from https://security.googleblog.com/2018/10/posted-by-sami-tolvanen-staff-software.html.

[66] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (Security'19)*.

[67] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 927–940.

[68] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*.

[69] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

[70] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. 2017. ReRanz: A light-weight virtual machine to mitigate memory disclosure attacks. In *Proceedings of the 13th International Conference on Virtual Execution Environments (VEE'17)*.

[71] Bryan C. Ward, Richard Skowyra, Chad Spensky, Jason Martin, and Hamed Okhravi. 2019. The leakage-resilience dilemma. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS'19)*.

[72] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*.

[73] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM Symposium on Information, Computer. and Communications Security (ASIACCS'16)*.

[74] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[75] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147.

[76] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA.

[77] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*.

Received November 2020; revised June 2021; accepted July 2021