MARDU: Efficient and Scalable Code Re-randomization

Christopher Jelesnianski Jinwoo Yom Changwoo Min Yeongjin Jang[†]
Virginia Tech [†]*Oregon State University*

Abstract

Defense techniques such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) were role models in preventing early return-oriented programming (ROP) attacks by keeping performance and scalability in the forefront, making them widely-adopted. As code reuse attacks evolved in complexity, defenses have lost touch with pragmatic defense design to ensure security, either being narrow in scope or providing unrealistic overheads.

We present MARDU, an *on-demand system-wide re-randomization* technique that maintains strong security guarantees while providing better overall performance and having scalability most defenses lack. We achieve *code sharing with diversification* by implementing reactive and scalable, rather than continuous or one-time diversification. Enabling code sharing further minimizes needed tracking, patching, and memory overheads. The evaluation of MARDU shows low performance overhead of 5.5% on SPEC and minimal degradation of 4.4% in NGINX, proving its applicability to both compute-intensive and scalable real-world applications.

CCS Concepts

• **Security and privacy** → **Software security engineering**; **Systems security**;

Keywords

Code Randomization, Return-Oriented Programming, Code Reuse, Code Sharing

ACM Reference Format:
Christopher Jelesnianski Jinwoo Yom Changwoo Min Yeongjin Jang[†] . 2020. MARDU: Efficient and Scalable Code Re-randomization. In *The 13th ACM International Systems and Storage Conference (SYSTOR '20), June 2–4, 2020, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3383669.3398280

1 Introduction

Computing remains susceptible to memory vulnerabilities as few state-of-the-art techniques are practical enough to be adopted mainstream. While code injection [2, 4, 27] has been solved with light-weight techniques like Data Execution Prevention (DEP) [26, 33], stronger adversaries like code reuse remain to be dealt with efficiently or securely.

Code reuse attacks, like *return-oriented programing (ROP)* and ret-into-libc [36], 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 bewtween attackers and defenders.

Both coarse- and fine-grained ASLR, while light-weight, are vulnerable to attacks. Whether an entire code regions or basic blocks layout is randomized in memory, respectively, a single memory disclosure can result in exposing the entire code layout, regardless. Execute-only memory (XoM) was introduced to prevent direct memory disclosures, by enabling memory regions to be marked with execute-only permissions. However, code inference attacks, code reuse attacks that work by indirectly observing and deducing information about the code layout circumvented these limitations. Various attack angles revealed that one-time randomization is simply not sufficient. This fostered the next generation of defenses, such as CodeArmor [10] and Shuffler [41], introducing continuous runtime re-randomization to strengthen security guarantees; they rely on background threads to run randomization and approximated use of thresholds to proactively secure vulnerable code. Despite being non-intrusive and easily deployable, these techniques require additional system resources and do not support code sharing.

Control Flow Integrity (CFI) innovations are in a similar state, like unrealistic performance tradeoffs or the inability to scale system-wide relying on background threads for each thread or process [14, 22]. Compared to randomization, CFI guards against ROP by strictly guiding control flow to only legimate destinations via type analysis or offline runtime analysis. While these may provide stricter enforcement, this class of techniques have their own set of issues like large equivalence classes for remaining edge cases.

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

MARDU builds on the insight that thresholds, like time intervals [10, 41] and the amount of leaked data [39], are a security loophole and a performance shackle in re-randomization; MARDU does not rely on a threshold in its design. This allows MARDU to avoid the paradox between security and performance. MARDU takes adventage of the event trigger design and Intel MPK. Using Intel MPK, MARDU provides XoM protection against *both* variations of remote and local JIT-ROP, with negligible overhead. MARDU also combines XoM with immutable trampolines by covering them from read access while decoupling the function entry from its function body to make it impossible for attackers to infer and obtain ROP gadgets. Note that immutable code pointer approaches (*e.g.*, using trampolines and indirection tables) *all* inherently share the vulnerability to only full-function reuse (except TASR [7]). We consider eliminating full-function code reuse and data-oriented programming [23] to be an orthongal problem. That being said, MARDU can complement CFI solutions adding minimal overhead and adding valuable randomization to make code reuse more difficult.

It is crucial to note that current re-randomization techniques do not take scalability into consideration. Support for *code sharing* has been forgotten, as applying re-randomization per process counters the principles of memory deduplication. Additionally, 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 keeps performance and scalability at its forefront and does not require expensive code pointer tracking and patching. Furthermore, MARDU does not incur significant overhead from continuous re-randomization triggered by overconservative time intervals or benign I/O system calls as in Shuffler [41] and ReRanz [39], 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 [7, 10, 17, 39, 41]). To summarize, we make the following contributions:

- **ROP attack & defense analysis.** Our background §2, describes current randomization defenses as well as remaining prevalent ROP attacks that challenge current works and should be addressed. This motivates the innovation that MARDU brings to improving security, performance, and scalability of sequent randomization techniques.
- **MARDU defense framework.** We present the design of MARDU §4, a comprehensive ROP defense technique capable of addressing all currently known ROP 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 & prototype.** We have built a prototype of MARDU and evaluated it §6 with both compute-intensive benchmarks and real-world applications.

2 Code Layout (Re-)Randomization

Realizing that maintaining fixed layout across crash-probes still allowed code layout/contents to be indirectly infered by attackers, re-randomization techniques addressed JIT-ROP and BROP [7, 10, 17, 19, 31, 39, 41] by continuously shuffling code (and data) at runtime, making information leaks or code probing useless. Re-randomization techniques can be categorized into two core design elements: *1) Re-randomization triggering condition* and *2) Code pointer semantics*.

Triggering conditions include timing and sensitive system calls. Techniques have logical reasoning about their thresholds, such as randomizing faster than typical network latency [10, 41] or triggering on known sensitive system calls such as `fork()` and `write()` [7, 31, 39]. These techniques add unnecessary overhead due to excessive re-randomization and remain vulnerable to attacks that can be acheived within smaller timing windows.

Prevailing code pointer semantics include using raw code addresses, as in ASR3 [19] and TASR [7], where tracking of code (or all) pointers is required during runtime, which is compute intensive to update values after re-randomization. Some defenses have opted to shift this overhead to a background thread, but this then creates a scalability bottleneck. Another method is using trampolines; these techniques store an indirect index [41] or immutable trampoline address, as in ReRanz [39] as code pointers. Re-randomization only affects the sensitive code layout skipping expensive code pointer tracking/updating, leaving the code layout secured. An even more minimalist pointer semantic is using offsets to code addresses [10]; this avoids pointer tracking by having an immutable offset from the random base version address to refer to functions. Randomization here is efficient because only a single base address is updated. However, both trampolines and base address offsets share the drawback that leaking code pointers will reveal an immutable piece of information (function index/offset) valid across re-randomizations, allows the attacker to reuse and perform full function reuse.

We motivate MARDU presenting two pertinent attack classes current *randomization* techniques are suseptible to.

Low-profile JIT-ROP. Existing defenses utilize either timing thresholds [10, 17, 19, 41], transmitted data amount by output system calls [39], or crossing of an I/O system call boundary [7] as a trigger for layout re-randomization. However, low-profile JIT-ROP attacks can exploit pre-defined randomization time intervals or carry out an attack without involving any I/O system call invocations. By bypassing these triggering conditions, code layout remains unchanged within the given interval and vulnerable to JIT-ROP.

Code pointer offsetting. Even with re-randomization, techniques might be susceptible to code pointer manipulation if code pointers are not protected from having arithmetic operations applied by attackers [7, 10]. Particularly, techniques directly using code address [7] or code offset [10], could allow attackers to reach a ROP gadget if the gadget offset is known beforehand. Ward *et al.* [40] has recently demonstrated that this attack is possible against TASR. This attack essentially 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 ROP gadgets.

3 Threat Model and Assumptions

MARDU's threat model follows that of similar re-randomization works [7, 10, 41]. We assume attackers can perform arbitrary read/write by exploiting software vulnerabilities in the victim program. We also assume all attack attempts are 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 loadding/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 and XoM are enabled). This includes trusting Intel Memory Protection Keys (MPK) [24], a mechanism that provides XoM; attacks targeting hardware (side-channel attacks, *e.g.*, Spectre [28], Meltdown [30]) are out of scope.

4 MARDU Design

4.1 Goals

MARDU aims to improve current state-of-the-art defense techniques to enable practical code re-randomization. Our design goals focus on scalability, performance, and security.

Scalability. Most proposed exploit mitigation mechanisms overlook the impact of required additional system resources, such as CPU or memory 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 a pay-as-you-go pricing Cloud. Oxymoron [6] and PageRando [11] are the only defenses, to our knowledge, that allow code sharing of randomized code, thus minimizing their memory footprint. Additionally, most re-randomization defenses [10, 39, 41] require per-process background threads, which not only cause additional CPU usage but also contention with the application process, especially as the number of processes increases. Therefore, to apply MARDU system-wide, we design MARDU to minimize the system resources required.

Performance. Many prior approaches [10, 12, 13, 41] demonstrate decent runtime performance on average (<10%, *e.g.*, <3.2% in CodeArmor); however, they also show a few remarkably slow cases (*i.e.*, >55%). We design MARDU to run with an acceptable average overhead (≈5%) with minimal performance outliers across a variety of application types.

Security. No prior solutions provide a comprehensive defense against existing attacks (see §2). As systems applying re-randomization are still susceptible to low-profile attacks and code pointer offsetting attacks, MARDU aims to either defeat or significantly limit the capability of attackers to provide best-effort security against these existing attacks.

4.1.1 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-grain ASLR with re-randomization. Although such an approach can defeat code pointer offsetting, systems cannot apply such protection because re-randomization must finish quickly to meet performance goals to also defeat low-profile attacks. An example of the tradeoff between scalability and performance is having a dedicated process/thread for performing re-randomization. However, this results in a drawback in scalability by requiring more CPU time in the entire system. Therefore, a good design must find a breakthrough to meet *all* of 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).

4.1.2 Architecture

We design MARDU to innovate beyond tradeoffs in security, performance, and scalability aspects. We introduce our approach for satisfying each aspect below:

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 NGINX server crashes, it re-randomizes upon exit all associated mapped executables (*e.g.*, `libc.so` of all processes, and all other NGINX workers).

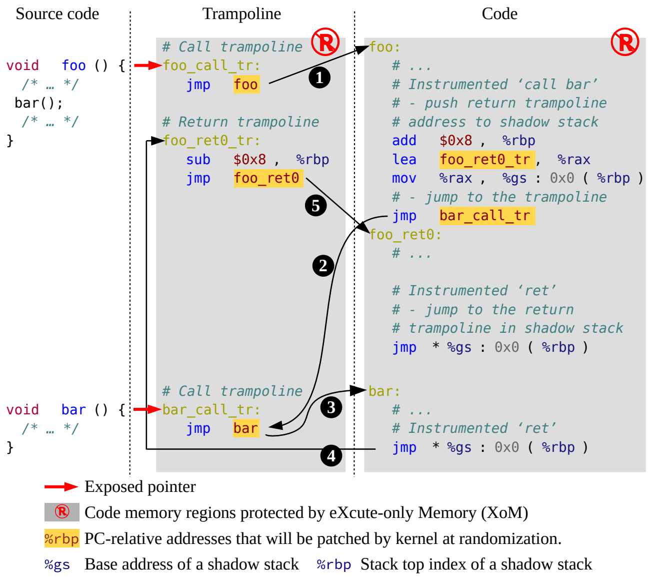
Scalability: On-demand re-randomization. MARDU re-randomizes code only when suspicious activity is detected. By doing so, MARDU does not rely on per-process background threads nor re-randomization interval unlike prior re-randomization approaches. Particularly, MARDU re-randomization is performed in the context of a crashing process, thereby not affecting the performance of other running processes.

Performance: Immutable code pointers. These scalability design decisions also help improve performance. MARDU neither tracks nor encrypts nor mutates code pointers upon re-randomization. This minimizes performance overhead, while 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 process crash and system-wide re-randomization. Therefore, MARDU counters direct memory disclosure attacks as well as code inference attacks requiring initial code probing [35, 37]. We use Intel MPK [24] to implement XoM so MARDU does not impose any runtime overhead unlike virtualization-based designs.

Security: Preventing code & code pointer leakage. In addition to system-wide re-randomization, MARDU minimizes 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) while non-trampoline code pointers remain hidden. MARDU does not randomize the trampoline region so that 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) and unlike re-randomization approaches that rely on shifting code base addresses [7, 10, 31], MARDU is not susceptible to code pointer offsetting attacks. Finally, MARDU stores return addresses–precisely, trampoline addresses for return–in a shadow stack. This makes stack pivoting practically infeasible.

Design overview. MARDU is composed of compiler and kernel components. The MARDU compiler enables trampolines and a shadow stack to be used. The compiler also generates


Source code
Trampoline
Code
❶
❷
❸
❹
❺
Exposed pointer
Code memory regions protected by eXcute-only Memory (XoM)
%rbp PC-relative addresses that will be patched by kernel at randomization.
%gs Base address of a shadow stack
%rbp Stack top index of a shadow stack

```
void  foo () {
  /* … */
 bar();
  /* … */
}

void  bar () {
  /* … */
}
```

```
# Call trampoline
foo_call_tr:
   jmp   foo

# Return trampoline
foo_ret0_tr:
   sub   $0x8 ,  %rbp
   jmp   foo_ret0

# Call trampoline
bar_call_tr:
   jmp   bar
```

```
foo:
   # ...
   # Instrumented 'call bar'
   # - push return trampoline
   # address to shadow stack
   add   $0x8 ,  %rbp
   lea   foo_ret0_tr,  %rax
   mov   %rax ,  %gs : 0x0 ( %rbp )
   # - jump to the trampoline
   jmp   bar_call_tr
foo_ret0:
   # ...

   # Instrumented 'ret'
   # - jump to the return
   # trampoline in shadow stack
   jmp  * %gs : 0x0 ( %rbp )

bar:
   # ...
   # Instrumented 'ret'
   jmp  * %gs : 0x0 ( %rbp )
```

Figure 1: Illustrative example executing a MARDU-compiled function `foo()`, which calls a function `bar()` and then returns.

and attaches metadata to binaries for efficient patching. Meanwhile, the MARDU kernel is responsible for choreographing the runtime of a MARDU enabled executable.

4.2 MARDU Compiler

MARDU compiler generates a binary able to 1) hide its code pointers, 2) share its randomized code among processes, and 3) run under XoM. To this end, 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 needing costly runtime tracking. To accomplish this we split a binary into two regions in process memory: *trampoline* and *code* regions (as shown in Figure 1 and Figure 2). 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 trampolines* are responsible for forwarding control flow from an instrumented call to the *code region* function entry, while *return trampolines* are 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 caller. Since function call trampoline addresses are stationary, MARDU does not need to track code pointers upon re-randomization.

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 to 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.

Running example. Figure 1 is an example of executing a MARDU-compiled function `foo()`, which calls a function `bar()`. Every function call and return goes through trampolines that store 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 to 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 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 key challenge here is *how to incorporate PC-relative addressing with randomization* so that code can be shared amongst processes. MARDU randomly places code (at function granularity) while trampoline regions are stationary. This means any code using PC-relative addressing must be correspondingly patched once its randomized location is decided. In Figure 1, all jump targets between the trampoline and code, denoted in yellow rectangles, are PC-relative and must be patched. All data addressing instructions (*e.g.*, accessing global data, *etc.*) must also be patched.

Fixup information for patching. PC-relative addressing makes it necessary to track these instructions to patch them throughout runtime. To make patching simple and efficient, MARDU compiler generates metadata into the binary describing exact locations for patching via their file-relative offset; this metadata is then used to adjust PC-relative offsets for those locations as needed (see Figure 2). The overhead of runtime patching is negligible because MARDU avoids "stopping the world" to maintain internal consistency compared to other approaches, putting the burden on the crashed process instead. We elaborate on the patching process in §4.3.2.

Supporting shared libraries. Calls to a shared library are treated the same as internal function calls to preserve MARDU's code pointer hiding property; that is, MARDU refers to the call trampoline for the shared library function via the procedure linkage table (PLT) or global offset table (GOT) whose address is resolved by the dynamic linker as usual. While MARDU does not specifically protect GOT, we assume that GOT is already protected using MPK [16, 32].

4.3 MARDU Kernel

MARDU kernel is responsible for maintaining a secure and efficient runtime. This is done by: creating a MARDU XoM enabled virtual code region, initializing a shadow stack for each task[1], randomizing/patching code, and reclaiming stale randomized code. These actions are described in detail next.

[1] A *task* denotes both process and thread as the convention in Linux kernel.

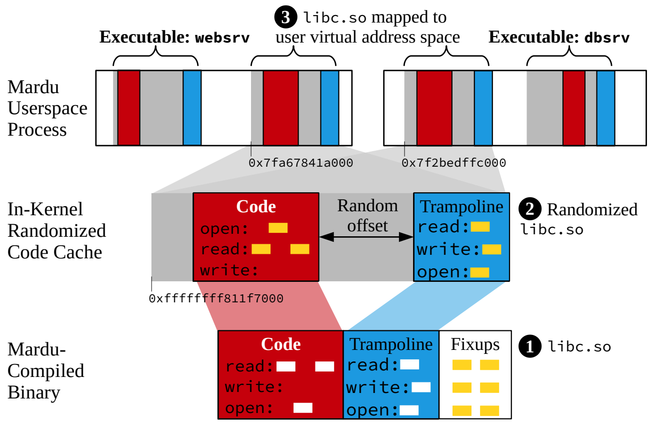


Figure 2: Memory layout of two MARDU processes: `websrv` (top left) and `dbsrv` (top right). The randomized code in kernel (`0xffffffff811f7000`) is shared by multiple processes, which is mapped to its own virtual base address (`0x7fa67841a000` for `websrv` and `0x7f2bedffc000` for `dbsrv`).

4.3.1 Process Memory Layout Figure 2 illustrates the memory layout of two MARDU processes. MARDU compiler generates a PC-relative binary with trampoline code and fixup metadata ❶. When a binary is loaded, MARDU kernel first extracts all MARDU metadata in the binary and associates it on a per-file basis. This metadata gives MARDU the information needed to perform (re-)randomization ❷.

Allocating a virtual code region. For each randomized binary, MARDU kernel allocates a 2 GB *virtual* address region[2] ❷, which will be mapped to userspace[3] with coarse-grained ASLR ❸. MARDU kernel positions the trampoline code at the end of the virtual address region and that the trampoline address will remain static throughout program execution even after re-randomization. Note that MARDU kernel maps already-randomized code, if it exists, to the address space of a newly `fork`-ed process for even more efficiency.

Whenever a new task is created (`clone`), MARDU kernel allocates a new shadow stack and copies parent's shadow stack to its child; it is placed in the virtual code region created by 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. To maximize performance, MARDU implements a compact shadow stack [9]. In addition, we reserve one register, `%rbp`, to use exclusively as a stack top index of the shadow stack to avoid costly memory access.

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

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

4.3.2 Fine-Grain Code Randomization

Randomizing the code within the virtual region. Load-time randomization and run-time re-randomization follow the exact same procedure. To achieve a high entropy, MARDU kernel uses fine-grained randomization within the allocated virtual address region. After the trampoline is placed, MARDU kernel randomly places the code region 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 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, MARDU kernel patches PC-relative instructions accessing code or data according to the randomization pattern. This patching process is trivial at runtime; MARDU compiler generates fixup location information and MARDU kernel re-calculates and patches PC-relative offsets of instructions according to the randomized function location. With the randomized code semantically correct, it can be cached and mapped to multiple applications, Figure 2 ❸. Note that patching includes control flow transfer between trampoline and code regions, global data access, and function calls to other shared libraries.

4.3.3 Randomized Code Cache MARDU kernel manages a cache of randomized code. When a userspace process tries to map a file with executable permissions, MARDU kernel first looks up if there already exists a randomized code of the file. If cache hits, 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. 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, MARDU kernel evicts the randomized code. Thus, in normal cases without re-randomization, MARDU randomizes a binary file only once. 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 (XoM) We designed XoM based on Intel MPK [24][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, 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 [12, 38], our MPK-based design does not impose 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 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, MARDU kernel populates another copy of the code (*e.g.*, `libc.so`) in the code cache and randomizes it (Figure 3 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 the new non-trampoline code to the application's virtual address space, Figure 3 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 migration of non-runnable, sleeping task because it will correctly migrate to the newest randomized code region when it passes through the (virtually) static return trampoline, which refers to the new layout when it wakes up. The reference counter is

[4]As of this writing, Intel Xeon Scalable Processors [25] and Amazon EC2 C5 instance [3] support MPK. Other than x86, ARM AArch64 architecture also supports execute-only memory [5].

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

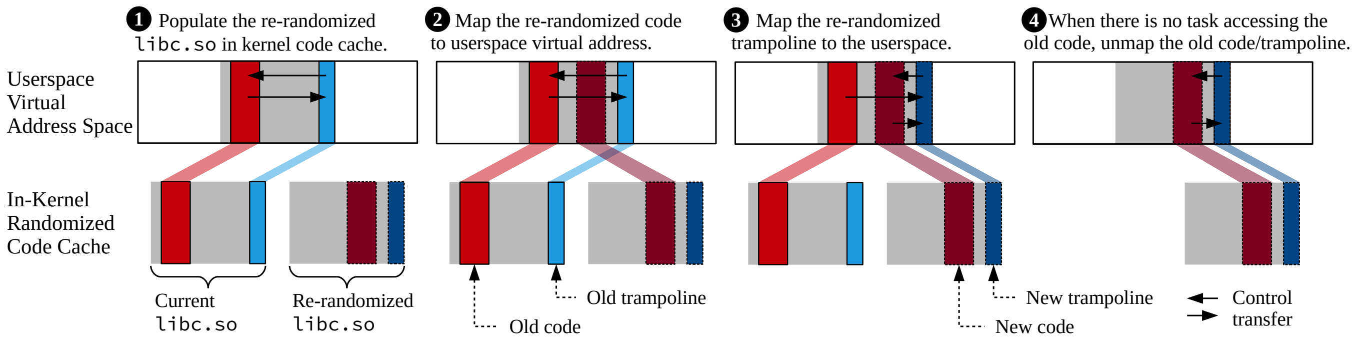

❶ Populate the re-randomized libc.so in kernel code cache.
❷ Map the re-randomized code to userspace virtual address.
❸ Map the re-randomized trampoline to the userspace.
❹ When there is no task accessing the old code, unmap the old code/trampoline.
Userspace Virtual Address Space
In-Kernel Randomized Code Cache
Current libc.so
Re-randomized libc.so
Old trampoline
Old code
New trampoline
New code
Control transfer

Figure 3: Re-randomization procedure in MARDU. Once a new re-randomized code is populated ❶, 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, almost zero overhead, and system-wide.

decremented when a runnable task enters into MARDU kernel due to system call or preemption. When calling a system call, 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, 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: time boundness of re-randomization, almost zero overhead of running process, and system-wide re-randomization. The re-randomization is guaranteed to finish at most within one scheduling quantum (*e.g.*, 1 msec) once the newly randomized code is exposed ❸. That is because MARDU migrates *runnable* tasks at system call and scheduling boundary. 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. 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 runtime re-randomization while allowing code sharing.*

5 Implementation

We implemented MARDU on the Linux x86-64 platform. MARDU compiler is implemented using LLVM 6.0.0 and MARDU kernel is implemented based on Linux kernel 4.17.0 modifying 3549 and 4009 lines of code (LOC), respectively. We used `musl libc` 1.1.20 [1], a fast, lightweight C standard library for Linux. We manually wrapped all inline assembly functions present in `musl` to allow them to be properly identified and instrumented by MARDU compiler.

5.1 MARDU Compiler

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

Re-randomizable code. The following compiler flags are used by 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 MARDU kernel can use the full span of memory within our declared 2GB virtual address region and maximize entropy when performing patching; lastly, MARDU compiler ensures 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 Intel architectures.

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 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. Modifying these functions is straightforward though as our shadow stack is a variant of compact, register-based shadow stack [9].

C++ support. Our prototype does not support C++ applications since we do not have a stable standard C++ library that is `musl`-compatible. Therefore handling C++ exceptions and protecting vtables is out of scope.

6 Evaluation

We evaluate MARDU by answering these questions:

- How secure is MARDU, when presented against current known attacks on randomization? (§6.1)
- How much performance overhead does MARDU impose, particularly for compute-intensive benchmarks? (§6.2)
- How scalable is MARDU in terms of load time, re-randomization time, and memory savings, paricularly for concurrent processes such as in a real-world network facing server? (§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 DRAM.

6.1 Security Evaluation

We analyze the resiliency of MARDU against existing attacker models pertinent to current re-randomization defenses.

MARDU Security Summary:
- *vs.* **JIT-ROP:** Execute-only memory blocks the attack.
- *vs.* **BROP/Code Inference:** Re-randomization blocks any code inference via crash.
- *vs.* **Low-profile JIT-ROP:** Execute-only memory and a large search space (2 GB dummy mappings) block JIT-ROP and crash-resistant probing.
- *vs.* **Code Pointer Offsetting:** Trampolines decouple function entry from function bodies blocking any type of code pointer offsetting; full function code reuse of exported functions remains possible.

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 [8], clone-probing [31], and destructive code read attacks [35, 37] 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.

Hiding shadow stack. Attackers with arbitrary read/write capability 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 [15, 21, 34]. 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. 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}); any incorrect probe will generate a crash, trigger re-randomization, thwarting the attack.

Entropy. MARDU permutes all functions in each executable and applies a random start offset to the code area in 2 GB space for each randomization providing high entropy to each new code layout. Thus, randomization entropy depends on the number of functions in the executable and the size of a code region (*i.e.*, $log_2(n! \cdot 2^{31})$ where n is the number of functions). To illustrate, `470.lbm` in SPEC provides the minimum entropy in our evaluation; it contains 26 functions and is less than 64 KB in size, but has 119.38 bits entropy using MARDU. Therefore, even for small programs, MARDU randomizes code with significantly high entropy to render attacker's success rate for guessing the layout negligible.

6.1.2 Attacks against Continuous Re-randomization

Against low-profile attacks. 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 vector could be one that does not cause any crash during exploitation (*e.g.*, attackers may employ crash-resistant probing [15, 18, 21, 29, 34]). In this regard, MARDU places all code in execute-only memory within 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. 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

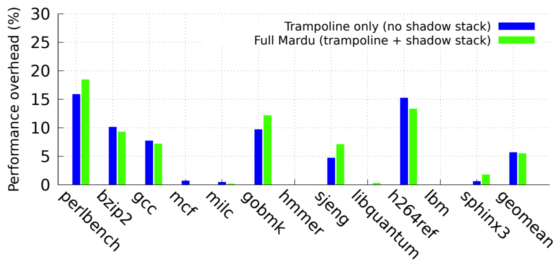


Figure 4: MARDU performance overhead breakdown for SPEC

code pointer offsetting. MARDU limits available code-reuse targets to only exported functions in the trampoline.

6.2 Performance Evaluation

Runtime performance overhead with SPEC CPU2006. Figure 4 shows the performance overhead of SPEC with MARDU trampoline only instrumentation (which does not use a shadow stack) as well as with full MARDU implementation. Both of these numbers are normalized to the unprotected baseline, compiled with vanilla Clang. Figure 4 does not include a direct performance comparison to other randomization techniques as MARDU is substantially different in how it implements re-randomization. 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 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 [41] and CodeArmor [10], whose reported average overheads are 14.9% and 3.2%, and their worst-case overhead are 45% and 55%, respectively. TASR [7] shows a very practical average overhead of 2.1%; however, it has been reported by Shuffler [41] and ReRanz [39] that TASR's overhead against a more realistic baseline (not using compiler flag `-Og`) is closer to 30-50% overhead. This confirms 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 shadow stack. MARDU uses a compact shadow stack without a comparison epilogue whose sole purpose is to secure return addresses. Specifically, only 4 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 4 shows 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.

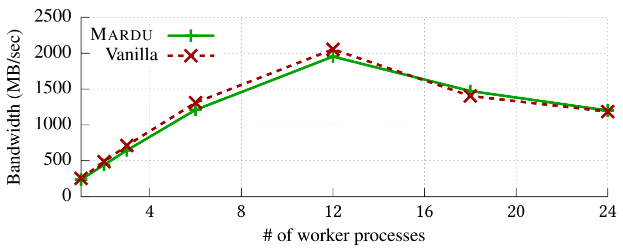


Figure 5: Performance comparison of NGINX web server

6.3 Scalability Evaluation

Runtime performance overhead with NGINX. NGINX is configured to handle a max of 1024 connections per processor, and its performance is observed according to the number of worker processes. `wrk` [20] 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 6745-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 5 presents the performance of NGINX with and without MARDU for a varying number of worker processes. The performance observed shows that MARDU exhibits very 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 [41] suffers from overhead from *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 15-55% 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.

Runtime memory savings. While 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 regards to shared code. `musl` is ≈800 KB in size, 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. Since MARDU supports code sharing, only one copy of `libc` is needed for the entire system. Backes *et al.* [6] and Ward *et al.* [40] 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 shadow stack does not increase runtime memory footprint because MARDU solely relocates return address from the normal stack to the shadow stack.

Load-time randomization overhead. We categorize load-time to cold or warm load-time whether the in-kernel code cache (❷ in Figure 2) hits or not. Upon a code cache miss (*i.e.*, the executable is first loaded in a system), MARDU performs initial randomization including function-level permutation,

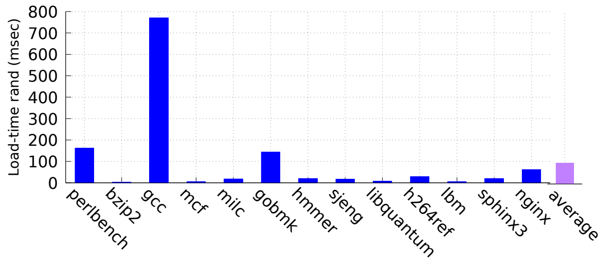


Figure 6: Cold load-time randomization overhead

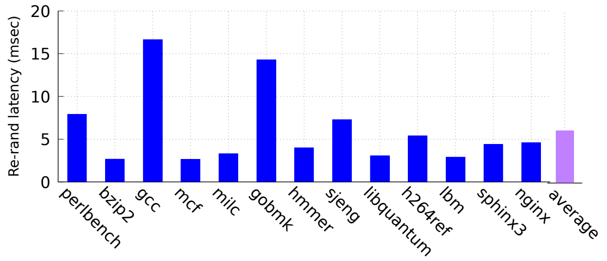


Figure 7: Re-randomization latency

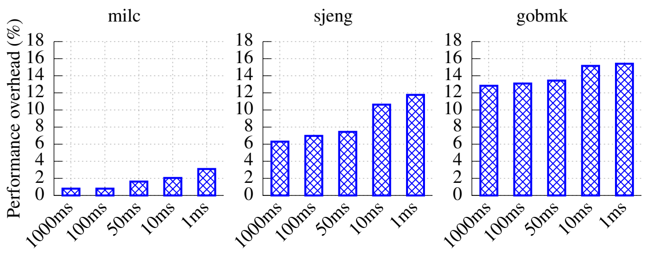


Figure 8: Overhead varying re-randomization frequency

start offset randomization of the code layout, and loading & patching of fixup metadata. As Figure 6 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. 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, for 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 1μsec, for any program needing it. Thus, load time can be largely ignored.

Re-randomization latency. Figure 7 presents time to re-randomize all associated binaries of a crashing process. The time includes creating & re-randomizing a new code layout, and reclaiming old code (❶-❹ in Figure 3). 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 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 latency is consistent regardless of number of workers (5.8 msec on average, 0.5 msec std. deviation).

Re-randomization overhead under active attacks. 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 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 sec, 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 8 shows that the overhead is consistent, and in fact, is very close to the performance overhead in the idle state observed in Figure 4. More specifically, all three benchmarks differ by less than 0.4% at a 1 sec 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 confirm that MARDU provides performant system-wide re-randomization even under active attack.

7 Discussion and Limitations

Residual attack surface. In terms of code-reuse attack surface, the following are traditional resources compromised and leveraged: 1) return addresses via direct and indirect function calls, 2) ROP gadgets within the code region, and 3) function entries for full function reuse. MARDU completely protects 1) and 2), using shadow stack and XoM, respectively, leaving only function entry 3). Evaluating SPEC, `musl`, and NGINX, shows the percentage of secured sensitive fragments is 95.6% on average. This means function entries via MARDU trampolines make up less than 4.4% of total leverage-able sensitive fragments. Applying orthogonal solutions such as CFI can complement MARDU to fill this gap as applying both techniques will make full function reuse more difficult.

Applying MARDU to binary programs. Although this prototype requires access to source code, applying MARDU directly to binary programs is possible. If one can precisely detect all indirect control transfers (`call/ret`), then the binary code can be directly instrumented such that transfers utilize trampolines instead. Therefore, MARDU should be practical enough to adopt with little additional engineering effort.

8 Conclusion

While current defense techniques are capable of defending against current ROP attacks, 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 code-reuse attacks. MARDU is the first code-reuse defense capable of code-sharing *with* re-randomization to enable practical security that scales system-wide. 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. Our evaluation verifies MARDU's security guarantees against known ROP attacks and adequately quantifies its high entropy. Furthermore, MARDU's performance overhead on SPEC CPU2006 and multi-process NGINX averages 5.5% and 4.4%, respectively, confirming practicality and scalability.

9 Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This work is supported in part by US Office of Naval Research under grants N00014-18-1-2022.

References

[1] 2019. musl libc. https://wiki.musl-libc.org/.

[2] One Aleph. 1996. Smashing the stack for fun and profit. *http://www.shmoo. com/phrack/Phrack49/p49-14* (1996).

[3] Amazon. 2019. Amazon EC2 C5 Instances. https://aws.amazon.com/ec2/instance-types/c5/.

[4] Autore Anonimo. 2001. Once upon a free ().. *Phrack Magazine* 11, 57 (2001).

[5] ARM. 2019. ARM Compiler Software Development Guide: 2.21 Execute-only memory. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471j/chr1368698326509.html.

[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)*. San Diego, CA.

[7] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado.

[8] 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 (Oakland)*. San Jose, CA.

[9] 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 (Oakland)*. San Francisco, CA.

[10] 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)*. Paris, France.

[11] Stephen Crane, Andrei Homescu, and Per Larsen. 2016. Code Randomization: Haven't We Solved This Problem Yet?. In *Cybersecurity Development (SecDev), IEEE*. IEEE, 124–129.

[12] 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 (Oakland)*. San Jose, CA.

[13] 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 (Oakland)*. San Jose, CA.

[14] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, BC, Canada.

[15] 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 (Oakland)*. San Jose, CA.

[16] Fedora. 2018. Hardening Flags Updates for Fedora 28. https://fedoraproject.org/wiki/Changes/HardeningFlags28.

[17] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. 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)*. Providence, RI, USA, 469–484.

[18] 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)*. San Diego, CA.

[19] 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)*. Bellevue, WA.

[20] Will Glozer. 2019. a HTTP benchmarking tool. https://github.com/wg/wrk.

[21] 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)*. Austin, TX.

[22] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, ON, Canada.

[23] 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 *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 969–986.

[24] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm.

[25] Intel Corporation. 2019. INTEL ® XEON ® SCALABLE PROCESSORS. https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html.

[26] Jonathan Corbet. 2004. x86 NX support. https://lwn.net/Articles/87814/.

[27] Michel Kaempf. [n. d.]. Vudo malloc tricks. Phrack Magazine, 57 (8), August 2001.

[28] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[29] 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)*. Denver, CO.

[30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[31] 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)*. San Diego, CA.

[32] Michael Larabel. 2017. Glibc Rolls Out Support For Memory Protection Keys. https://www.phoronix.com/scan.php?page=news_item&px=Glibc-Memory-Protection-Keys.

[33] Microsoft Support. 2017. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.

[34] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding.. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX.

[35] 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)*. Abu Dhabi, UAE, 55–67.

[36] 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*. Alexandria, VA.

[37] 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 (Oakland)*. San Jose, CA.

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

[39] 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)*. Xi'an, China.

[40] 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)*. Luxembourg.

[41] 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)*. Savannah, GA.