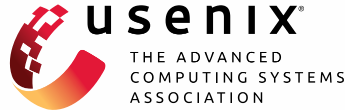


Shuffler: Fast and Deployable Continuous Code Re-Randomization

David Williams-King and Graham Gobieski, *Columbia University;* Kent Williams-King, *University of British Columbia;* James P. Blake and Xinhao Yuan, *Columbia University;* Patrick Colp, *University of British Columbia;* Michelle Zheng, *Columbia University;* Vasileios P. Kemerlis, *Brown University;* Junfeng Yang, *Columbia University;* William Aiello, *University of British Columbia*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Shuffler: Fast and Deployable Continuous Code Re-Randomization

David Williams-King[1] Graham Gobieski[1] Kent Williams-King[2]
James P. Blake[1] Xinhao Yuan[1] Patrick Colp[2] Michelle Zheng[1]
Vasileios P. Kemerlis[3] Junfeng Yang[1] William Aiello[2]

[1]Columbia University [2]University of British Columbia [3]Brown University

Abstract

While code injection attacks have been virtually eliminated on modern systems, programs today remain vulnerable to code reuse attacks. Particularly pernicious are Just-In-Time ROP (JIT-ROP) techniques, where an attacker uses a memory disclosure vulnerability to discover code gadgets at runtime. We designed a code-reuse defense, called *Shuffler*, which continuously re-randomizes code locations on the order of milliseconds, introducing a real-time deadline on the attacker. This deadline makes it extremely difficult to form a complete exploit, particularly against server programs that often sit tens of milliseconds away from attacker machines.

Shuffler focuses on being fast, self-hosting, and nonintrusive to the end user. Specifically, for speed, Shuffler randomizes code asynchronously in a separate thread and atomically switches from one code copy to the next. For security, Shuffler adopts an "egalitarian" principle and randomizes itself the same way it does the target. Lastly, to deploy Shuffler, no source, kernel, compiler, or hardware modifications are necessary.

Evaluation shows that Shuffler defends against all known forms of code reuse, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We observed 14.9% overhead on SPEC CPU when shuffling every 50 ms, and ran Shuffler on real-world applications such as Nginx. We showed that the shuffled Nginx scales up to 24 worker processes on 12 cores.

1 Introduction

At present, programs hardened with the latest mainline protection mechanisms remain vulnerable to code reuse attacks. In a typical scenario, the attacker seizes control of the instruction pointer and executes a sequence of existing code fragments to form an exploit [54]. This is fundamentally very difficult to defend against, as the program must be able to run its own code, and yet the attacker should be prevented from running out-of-order instruction sequences of that same code. One popular mitigation is to deny the attacker knowledge about the program's code through randomization. Unfortunately, memory disclosure vulnerabilities are common in the real world, with 500–2000 discovered per year over the last three years [20]. Such vulnerabilities can be used to read the program's code, at runtime, and unravel any static randomization in a so-called Just-In-Time ROP (JIT-ROP) attack [55].

We propose a system, called *Shuffler*, which provides a deployable defense against JIT-ROP and other code reuse attacks. Other such defenses have appeared in the literature, but all have had significant barriers to deployment: some utilize a custom hypervisor [4, 17, 33, 57]; others involve a modified compiler [7,10,13,40,42], runtime [10, 42], or operating system kernel [4, 7, 17]. Note that there is a security risk in any solution that requires additional privileges, as an attacker can potentially gain access to that elevated privilege level. Also, modified components present a large barrier to the adoption of the system and have less chance of incorporating upstream patches and updates, so users may continue to run vulnerable software versions. In comparison, Shuffler runs in userspace alongside the target program, and requires no system modifications beyond a minimal patch to the loader. Shuffler can be deployed amongst existing cloud infrastructure, adopted by software distributors, or used at small scale by individual security-conscious users.

Shuffler operates by performing continuous code re-randomization at runtime, within the same address space as the programs it defends. Most defenses operating at the same level of privilege as their target do not consider defending their own attack surface. In contrast, we bootstrap into a self-hosted and self-modifying *egalitarian* environment—Shuffler actually shuffles itself. We also defend all of a program's shared libraries, and handle multithreading and process `fork`s, shuffling each child independently. Our current prototype does not handle certain hand-coded assembly, but in principle, *all* executable code in a process's address space can be shuffled.

With Shuffler, we aim to rapidly obsolete leaked information by rearranging memory as fast as possible. Shuffler operates within a real-time deadline, which we call the *shuffle period*. This deadline constrains the total execution time available to any attack, since no information about the memory layout transfers from one shuffle period to the next. We achieve a shuffle period on the order of tens of milliseconds, so fast that it is nearly impossible to form a complete exploit. Shuffler creates new function permutations asynchronously in a separate thread, and then atomically migrates program execution from one copy of code to the next. This migration requires a vanishingly small global pause time, as program threads continue to execute unhindered 99.7% of the time (according to SPEC CPU experiments). Thus, if the host machine has a spare CPU core, shuffling at faster rates does not significantly impact the target's performance. Shuffler's default behaviour is to use a fixed shuffling rate, but it can work with different policies. For instance, if the system is under reduced load, a new vulnerability is announced, or an intrusion detection system raises an alarm, the shuffling rate can be increased dynamically.

Our system operates on program binaries, analyzing them and performing binary rewriting. This analysis must be complete and precise; missing even a single code pointer and failing to update it upon re-randomization can cause correctness issues. Because of the difficulty of binary analysis, we leverage existing compiler and linker flags to preserve symbols and relocations. Some (but not all [46]) vendors strip symbol information from binaries to impede reverse engineering, but reversing stripped binaries is still feasible using disassemblers like IDA Pro [27]. We anticipate that vendors would be willing to include (obfuscated) symbols and relocations in their binaries, given the additional defensive possibilities. For instance, relocations enable shuffling but are also required for executable base address randomization on Windows. In the open-source Linux world, high-level build systems are already designed to support the introduction of additional compiler flags [26], allowing distribution-wide security hardening [25, 29, 58].

Evaluation shows that our system successfully defends against all known forms of code reuse, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We ran Shuffler on a range of programs including web servers, databases, and Mozilla's SpiderMonkey Javascript interpreter. We successfully defend against a Blind ROP attack on Nginx, and against a JIT-ROP attack on a toy web server. Shuffler incurs 14.9% overhead on SPEC CPU when shuffling every 50 ms, and has good scalability on Nginx when shuffling up to 24 workers every 50 ms. We show that a 50 ms shuffle period is orders of magnitude faster than the time required by existing JIT-ROP attacks, which take 2.3 to 378 seconds to complete [52, 55].

Our main contributions are as follows:

1. **Deployability:** We design a re-randomization defense against JIT-ROP and code reuse, which runs without modification to the source, compiler, linker, or kernel, and with minimal changes to the loader.
2. **Speed:** We introduce a real-time deadline on the order of milliseconds for any disclosure-based attack, using a new asynchronous re-randomization architecture that has low latency and low overhead.
3. **Egalitarianism:** We describe how we bootstrap our defense into a self-hosting environment, thus avoiding any expansion of the trusted computing base.
4. **Augmented binary analysis:** We show that complete and precise analysis is possible on binaries by leveraging information available from today's compilers (namely, symbols and relocations).

2 Background and Threat Model

Attack taxonomy Many attacks seen in the wild against running programs are based on control-flow hijacking. An attacker uses a memory corruption vulnerability to overwrite control data, like return addresses or function pointers, and branches to a location of their choosing [2]. In the early days, that location could be a buffer where the attacker had directly written their desired exploit code, thus enacting a so-called *code injection* attack. Nowadays, the widespread deployment of Write-XOR-Execute (W^X) [15] ensures that pages cannot be both executable and writable, which has led to the effective demise of code injection.

In response, attackers began to create *code reuse* attacks, stitching together pieces of code already present in a program's code section. The first and simplest such attack was return-to-libc (`ret2libc`) [51, 56], where an attacker redirects control flow to reuse whole `libc` functions, such as `system`, after setting up arguments on the stack. A more sophisticated technique called Return-Oriented Programming (ROP) [54] was soon discovered, where an attacker stitches together very short instruction sequences ending with a return instruction (or other indirect branch instructions [9, 36])—sequences known as gadgets. The terminating return instruction allows the attacker to jump to the next gadget, and the attacker may set up the stack to contain the addresses of a desired "chain" of gadgets. ROP has been shown to be Turing-complete, and there are tools known as ROP compilers which can automatically generate ROP chains [52].

Defenses against code reuse The research community has proposed two main categories of defenses against code reuse. The first is Control Flow Integrity (CFI) [1], which tries to ensure that every indirect branch taken

by the program is in accordance with its control-flow graph. However, both coarse-grained CFI [61, 62] and fine-grained CFI [47] can be bypassed through careful selection of gadgets [11, 23, 28].

The second category of defense is code randomization, performed at load-time to make the addresses of gadgets unpredictable. Module-level Address Space Layout Randomization (ASLR) is currently deployed in all major operating systems [49, 60]. Fine-grained randomization schemes have been proposed at the function [6], basic block [59], and instruction [38] level. These defenses spurred a noteworthy new attack called Just-In-Time ROP (JIT-ROP) in 2013 [55]. In JIT-ROP, the attacker starts with one known code address, recursively reads code pages at runtime with a memory disclosure vulnerability, then compiles an attack using gadgets in the exfiltrated code. The authors conclude that no load-time randomization scheme can stand against this attack.

Defenses in the JIT-ROP era The first defenses against JIT-ROP concentrated on preventing recursive gadget harvesting. Oxymoron [5] and Code Pointer Integrity [40] proposed an inaccessible table to hide the true destination of `call` instructions. Other works proposed execute-only memory, either with a custom hypervisor [17, 57] or software emulation [4, 33]. Unfortunately, preventing the direct disclosure of memory pages is insufficient. Indirect JIT-ROP [14, 24] shows that harvesting code pointers from data pages allows the location of gadgets to be inferred, without ever being read. Leakage-resilient diversification [10, 17] combines execute-only memory with fine-grained ASLR and function trampolines. Thus, code pages cannot be read and their contents cannot be inferred through pointers. This defense is currently still effective, though implementing execute-only memory without extensive system modifications remains challenging.

Continuous re-randomization Following a handful of early re-randomization schemes [19, 34], researchers began to realize that continuous re-randomization can defend against JIT-ROP. If code is re-randomized between the time it is leaked and when a gadget chain is invoked, the attack will fail because the gadgets no longer exist.

For instance, Remix [13] continuously re-randomizes the basic block ordering within functions, so that gadgets no longer stay at constant offsets. The system utilizes an LLVM compiler pass to add padding NOPs so that there will be enough space to reorder blocks. However, this intra-function randomization is vulnerable to attacks that leverage function locations or reuse function pointers.

The closest system to Shuffler is TASR [7]. TASR is a source-level technique which performs re-randomization based on pairs of read/write system calls, between any program output (which may leak information) and any program input (which may contain an exploit). However, TASR requires kernel and compiler modifications, is currently only applicable to C programs, and has high performance overhead, as we discuss in Section 5.5.

Finally, another form of ROP called Blind ROP [8] targets servers that fork workers. Since the workers inherit the parent's address space layout, Blind ROP brute forces them without worrying about causing crashes. RuntimeASLR [42] uses heavyweight instrumentation to allow re-randomization of the child process on fork.

2.1 Threat Model

Shuffler is built upon continuous re-randomization. We aim to defend against all known forms of code reuse attacks, including ROP, direct JIT-ROP, indirect JIT-ROP, and Blind ROP. We assume that protection against code injection (W^X) is in place, and that an x86-64 architecture is in use. Our system does not require (and, in fact, is orthogonal to) other defensive techniques like intra-function ASLR, stack smashing protection, or any other compiler hardening technique.

On the attacker's side, we assume:

1. The attacker is performing a code reuse attack, and not code injection (handled by W^X [15]) or a data-only attack [12] (outside the scope of Shuffler).

2. The attacker has access to 1) a memory disclosure vulnerability that may be invoked repeatedly to read arbitrary memory locations, and 2) a memory corruption vulnerability for bootstrapping exploits.

3. Any memory read or write that violates memory permissions (or targets an unmapped page) will cause a detectable crash, and the attacker has no meta-information about page mappings.[1]

4. The attacker knows the re-randomization rate and can time their attack to start at the very beginning of a shuffling period, maximizing the time that code addresses remain the same.

Our technique is particularly effective when defending long-lived processes and network-facing applications, such as servers. Note that network-based attackers have additional latency induced by communication delays, each time they invoke a vulnerability; see Section 6.3 for details.

3 Design

This section presents the design goals of Shuffler, along with its architecture, and outlines significant technical challenges.

[1] Such as access to `/proc/<pid>/maps`.

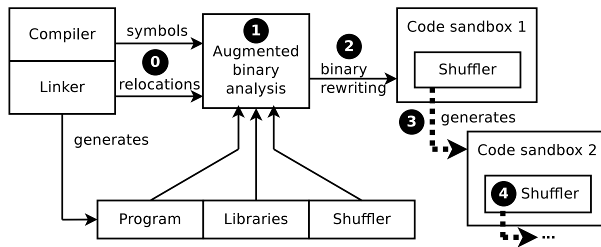


Figure 1: Shuffler architecture. We use symbols and relocations (0) for augmented binary analysis (1), rewrite code into shufflable form (2), and asynchronously create new code copies at runtime (3), while self-hosting (4).

3.1 Goals

The main goals of Shuffler are:

- **Deployability:** We aim to reduce the burden on end-users as much as possible. Thus, we require no direct access to source code, no static binary rewriting on disk, and no modifications to system components (except our small loader patch).
- **Security:** Our goal is to defeat all known code reuse attacks, without expanding the trusted computing base. We constrain the lifetime of leaked information by providing a configurable shuffling period, mitigating code reuse and JIT-ROP attacks.
- **Performance:** Because time is an integral part of our security model, speed is of the essence. We aim to provide low runtime overhead, and also low total shuffling latency to allow for high shuffling rates.

3.2 Architecture

Shuffler is designed to require minimal system modifications. To avoid kernel changes, it runs entirely in userspace; to avoid requiring source or a modified compiler, it operates on program binaries. Performing re-randomization soundly requires complete and precise pointer analysis. Rather than attempting arbitrary binary analysis, we leverage symbol and relocation information from the (unmodified) compiler and linker. Options to preserve this information exist in every major compiler. Thus, we are able to achieve completely accurate disassembly in what we call *augmented binary analysis*—as shown in Figure 1 part (1) and detailed in Section 3.3.

At load-time, Shuffler transforms the program's code using binary rewriting (Figure 1 part (2)). The goal of rewriting is to be able to track and update all code pointers at runtime. We avoid the taint tracking used by related work [7,42] because it is expensive and would introduce races during asynchronous pointer updates. Instead, we leverage our complete and accurate disassembly to transform all code pointers into unique identifiers—indices into a *code pointer table*. These indices cannot be altered after load time (the potential security implications of this choice are discussed in Section 6), but they trade off very favorably against performance and ease of implementation. We handle return addresses (dynamically generated code pointers) differently, encrypting them on the stack rather than using indices, thereby preventing disclosure while maintaining good performance.

Our system performs re-randomization at the level of functions within a specific *shuffle period*, a randomization deadline specified in milliseconds. Shuffler runs in a separate thread and prepares a new shuffled copy of code within this deadline, as shown in Figure 1 part (3). This step is accelerated using a Fenwick tree (see Section 4.4). The vast majority of the re-randomization process is performed asynchronously: creating new copies of code, fixing up instruction displacements, updating pointers in the code table, *etc.* The threads are globally paused only to atomically update return addresses. Since any existing return addresses reference the old copy of code, we must revisit saved stack frames and update them. Each thread walks its own stack in parallel, following base pointers backwards to iterate through stack frames (a process known as *stack unwinding*); see Section 3.3 for details.

Shuffler runs in an *egalitarian* manner, at the same level of privilege as target programs, and within the same address space. To prevent our own code from being used in a code reuse attack, Shuffler randomizes it the same way it does all other code (Figure 1 part (4)). In fact, our scheme uses binary rewriting to transform all code in a userspace application (the program, Shuffler, and all shared libraries) into a single code sandbox, essentially turning it into a statically linked application at runtime. Bootstrapping from original code into this self-hosting environment is challenging, particularly without substantially changing the system loader.

3.3 Challenges

Changing function pointer behaviour Normal binary code is generated under the assumption that the program's memory layout remains consistent and function pointers have indefinite lifetime. Re-randomization introduces an arbitrary lifetime for each block of code, and so re-randomization becomes an exercise in avoiding dangling code pointers. Failing to update even one such pointer may cause the program to crash, or worse, fall victim to a use-after-free attack.

Hence, we need to accurately track and update every code pointer during the re-randomization process. We opt to statically transform all code pointers into unique identifiers—namely, indices into a hidden *code pointer table*. Relying on accurate and complete disassembly (discussed next), we transform all initialization points to use indices. Then, wherever the code pointer is copied

throughout memory, it will continue to refer to the same entry in the table. This scheme does not affect the semantics of function pointer comparison. Iterating through and updating the pointer values stored in the table can be done quickly and asynchronously.

Some code pointers are dynamically generated, in particular, return addresses on the stack. We could dynamically allocate table indices, but on the x86 architecture, `call`/`ret` pairs are highly optimized, and replacing them with the table mechanism would involve a large performance degradation [22, 43]. Instead, we allow ordinary calls to proceed as usual, and at re-randomization time we unwind the stack and update return addresses to new values. Rather than leave return addresses exposed on the stack, we encrypt each address with an XOR cipher. Every callee is responsible for disguising the return address on the top of the stack, encrypting it at function entry and decrypting before any function exit. Callers, meanwhile, are responsible for erasing the (now unencrypted) return address immediately after the called function returns. Even though the address is never used by the program, it is still a (leakable) dangling reference. The encryption key can be unique to each function and changed during each stack unwind; see Section 4.1.

Augmented binary analysis The commonly accepted wisdom is that program analysis can be performed at the source level (requiring access to source code) or at the binary level (plagued with completeness issues). In this work, we propose a middle ground, *augmented binary analysis*, which involves analyzing program binaries that have additional information included by the compiler. Compiler-generated binaries are much more amenable to analysis than hand-crafted binaries. We use existing compiler flags and have no visibility into the source code, and yet can achieve complete disassembly.

The common problems with binary analysis are distinguishing code from data, and distinguishing pointers from integers. To tackle these issues, we require that (a) the compiler preserve the symbol table, and (b) that the linker preserve relocations. The symbol table indicates all valid `call` targets and makes disassembly straightforward—we iterate through symbols and disassemble each one independently; there is no need for a linear sweep or recursive traversal algorithm [53]. Relocations are used to indicate portions of an object file (or executable) that need to be patched up once its base address is known. Since each base address is initially zero, every absolute code pointer must have a relocation—but as object files are linked together, most code pointers get resolved and their relocations are discarded. We simply ask the linker to preserve these relocations.

These two augmentations enable complete and accurate disassembly, for any optimization level—at least on the $\sim$30 programs that we tested, many of which have sizable codebases. We describe the details of our augmented binary analysis in Section 4.2.

Bootstrapping into shuffled code As stated above, Shuffler defends its own code the same way it defends all other code—leading to a difficult bootstrapping problem. Shuffled code cannot start running until the code pointer table is initialized, requiring some unshuffled startup code. Shuffled and original code are incompatible if they use code pointers; the process of transforming code pointers to indices overwrites data that the original code accesses, and then the original code will no longer execute correctly. For example, if Shuffler naïvely began fixing code pointers while making code copies with `memcpy`, it would at some point break the `memcpy` implementation, because the latter uses code pointers for a jump table.[2] Hence, we would have to call new functions as they became available, and carefully order the function-pointer rewrite process to avoid invalidating any functions currently on the call stack.

Instead, we opted for a simpler and more general solution. Shuffler is split into two stages, a minimal and a runtime stage. The minimal stage is completely self-contained, and it can safely transform all other code, including `libc` and the second-stage Shuffler. Then it jumps to the shuffled second stage, which erases the previous stage (and all other original code). The second stage inherits all the data structures created in the first so that it can easily create new shuffled code copies. From this point on, Shuffler is fully self-hosting.

4 Implementation

Shuffler runs in userspace on x86-64 Linux. It shuffles binaries, all the shared libraries that a binary depends on, as well as itself. The shuffling process runs asynchronously in a thread, without impeding the execution of the program's threads. Figure 2 shows a running snapshot of shuffled code. Code pointers are directed through the code pointer table and return addresses are stored on the stack, encrypted with an XOR cipher. In each shuffle period, Shuffler makes a new copy of code, updates the code pointer table and sends a signal to all threads (including itself); each thread unwinds and fixes up its stack. Shuffler waits on a barrier until all threads have finished unwinding, then erases the previous code copy.

Our Shuffler implementation supports many system-level features, including shared libraries, multiple threads, forking (each child gets it own Shuffler thread), `{set,long}jmp`, system call re-entry, and signals. Shuffler does not currently support `dlopen` or C++ exceptions. Yet, it does expose several debugging features, notably, exporting shuffled symbol tables to GDB and printing shuffled stack traces on demand.

[2] This crash took place in an earlier prototype of Shuffler.

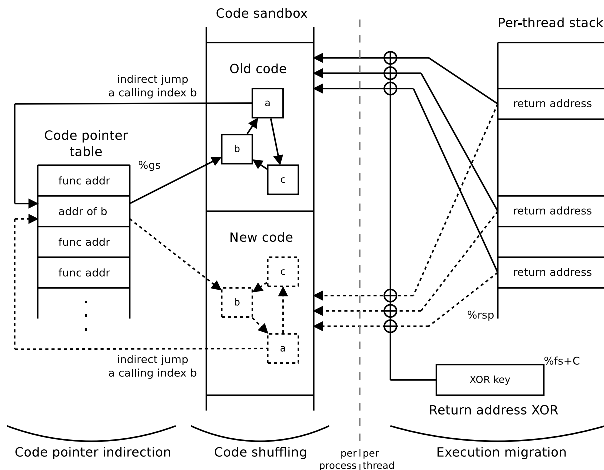


Figure 2: Overview of shuffled code at runtime, as Shuffler executes a shuffle pass. The old code is shown with solid lines and the new code with dotted lines.

Source instruction		Transformation
`lea funcptr, %rax`	→	`lea index, %rax`
`call *%rax`	→	`callq *%gs:(%rax)`
`callq *(%rax,%rbx,8)`	→	`mov (%rax,%rbx,8),%r11` `callq *%gs:(%r11)`
`jmp *%rax`	→	`jmpq *%gs:(%rax)`

`jmpq *(%rax,%rbx,8)`	→	`mov %r11, %fs:0x88`
		`mov (%rax,%rbx,8),%r11`
		`mov %gs:(%r11),%r11`
		`xchg %r11, %fs:0x88`
		`jmpq *%fs:0x88`

(a) Transforms to support the code pointer table.

Source instruction		Transformation

`# function begin`	→	`mov %fs:0x28,%r11`
		`xor %r11,(%rsp)`
		`# function begin`

`ret / jmp *%rax`	→	`mov %fs:0x28,%r11`
		`xor %r11,(%rsp)`
		`ret / jmp *%rax`

`call anything`	→	`call anything` `mov $0x0, -8(%rsp)`

(b) Transforms to support return address encryption.

Figure 3: Binary rewriting transformations performed by Shuffler. `%fs:0x28` is the stack canary, `%r11` is a scratch register, and `%fs:0x88` is a scratch variable.

4.1 Transformations to Support Shuffling

Code pointer abstraction We allocate the code pointer table at load-time and set the base address of the `GS` segment (selected by the `%gs` register) at it. Then, we transform every function pointer at its initialization point from an address value to an index into this table. We use relocations generated by the compiler and preserved by the linker flag `-q` to find all such code pointers. Pointer values are deduplicated as they are assigned indices in the table, for more efficient updating. Jump tables are handled similarly, with indices assigned to each offset within a function that is used as a target. Note that indices may also be assigned dynamically by Shuffler (*e.g.,* so that `setjmp` works across shuffle periods).

We must also transform the code so that indices are invoked properly. As shown in the Figure 3a, every instruction which originally used a function pointer value is rewritten to instead indirect through the `%gs` table. This adds an extra memory dereference. Since x86 instructions can contain at most one memory reference, if there is already a memory dereference, we use the caller-saved register `%r11` as scratch space. For (position-dependent) jump tables, there is no register we can safely overwrite, so we use a thread-local variable allocated by Shuffler as a scratch space (denoted as `%fs:0x88`).

Return-address encryption We encrypt return addresses on the stack with a per-thread XOR key. We reuse the stack canary storage location for our key; our scheme operates similarly to stack canaries, but does not affect the layout of the stack frame. As shown in Figure 3b, we add two instructions at the beginning of every function (to disguise the return address) and before every exit jump (to make it visible again); after each `call`, we insert a `mov` instruction to erase the now-visible return address on the stack. We again use `%r11` as a scratch register, since it is a caller-saved register according to the x86-64 ABI, and thus safe to overwrite.

Displacement reach A normal call instruction has a 32-bit displacement and must be within ± 2GB of its target to "reach" it. Shared libraries use Procedure Linkage Table trampolines to jump anywhere in the 64-bit address space. We wish to use only 32-bit calls and still enable function permutation; thus, we place all shuffled code at most 2GB apart, and transform calls through the PLT into direct function calls. Essentially, we convert dynamically linked programs into statically linked ones at runtime.

4.2 Completeness of Disassembly

We demonstrate the complete and precise disassembly of binaries that have been augmented with a symbol table and relocations. The techniques shown here are sufficient to analyze `libc`, `libm`, `libstdc++`, the SPEC CPU binaries, and the programs listed in our performance evaluation section. While shuffling these libraries and programs, we encountered myriad special cases. Figure 4 lists the main issues we faced, which would also need to be handled by other systems performing similar analyses. The issues boil down to: (a) dealing with inaccurate/missing metadata, especially in the symbol table; (b) handling special types of symbols and relocations; and (c) discovering jump table entries and invocations.

Issue	Description	How to handle
Missing symbol sizes	Internal GCC functions have a symbol size of zero.	Hard-code sizes; `_start` is 42 bytes.
Fall-through symbols	Functions implicitly fall through to the following function.	Attach a copy of the following code.
Overlapping symbols	Some functions are a strict subset of an enclosing function.	Binary search for targets very carefully.
Symbol aliases	Symbol tables have many names for the same function.	Pick one representative name.
Ambiguous names	One LOCAL name, multiple versions (`bsloww` in libm).	Look up address resolved by the loader.
Pointers to static functions	For pointers to functions within the same module, the offset is known, and object files contain no relevant relocations.	Determine if `lea` instructions target a known symbol (not completely sound).
`noreturn` function calls	GCC always generates a NOP after calls to `noreturn` functions like `longjmp`, but omits unwind information.	Detect when at a NOP following a call and use unwind info from at the call.
COPY relocations	Object initialized in one library, then `memcpy`'d to another.	Track data symbols, not just code.
IFUNC symbols	Return pointer to actual function to call (cached in PLT).	Statically evaluate from `lea` refs.
Conditional tail recursion	Does not appear in normal GCC-generated code. Used in hand-coded assembly by glibc (`lowlevellock.h`).	Can do XOR'ing both before and after, works whether or not the jump is taken.
Indirect tail rec.	Difficult to tell apart from jump-table jumps.	Use a function epilogue heuristic.
Finding jump tables	Jump tables are not clearly delineated.	See the text for a discussion on this.

Figure 4: Special cases in augmented binary disassembly.

Jump tables One major challenge is identifying whether relocations are part of jump tables, and distinguishing between indirect tail-recursive jumps and jump-table jumps. If we fail to realize a relocation in a jump table, we will calculate its target incorrectly and the jump will branch to the wrong location; if we decide that a jump table's jump is actually tail recursive, we will insert return-address decryption instructions before it, corrupting `%r11` and scrambling the top of the stack.

GCC generates jump tables differently in position-dependent and position-independent code (PIC). Position-dependent jump tables use 8-byte direct pointers, and are nearly always invoked by an instruction of the form `jmpq *(%rax,%rbx,8)` at any optimization level. PIC jump tables use 4-byte relative offsets added to the address of the beginning of the table—and the `lea` that loads the table address may be quite distant from the final indirect jump. To find PIC jump tables, we use outgoing `%rip`-relative references from functions as bounds and check if they point at sequences of relocations in the data section.[3] Note that `R_X86_64_PC32` relocations must have 4 bytes added to their value (the displacement size) if present in an instruction, and they must not if present in a jump table.

It is difficult to tell whether a `jmpq *%rax` instruction is used for indirect tail recursion, or a PIC jump table. In our system, we must distinguish these to decide whether to decrypt the return address or not. We do this with a heuristic that pairs function epilogues with function prologues. We use a linear sweep to record `push` instructions in the function's first basic block, and keep a log of the `pop` instructions seen since the last jump (within a window size). If an indirect jump is preceded by `pop` instructions that are in the reverse order of the `push` instructions, we assume we have found a function epilogue and that the jump is indirect tail recursive.

4.3 Bootstrapping and Requirements

We carefully bootstrap into shuffled code using two libraries (stage 1 and stage 2) so that the system never overwrites code pointers for the module that is currently executing. These libraries are injected into the target using `LD_PRELOAD`.[4] Rather than reimplement loader functionality, we defer to the system loader to create a valid process image, and then take over before the program—or even its constructors—begin executing.

The constructor of stage 1 is called before any other via the linker mechanism `-z initfirst`.[5] Then, by setting breakpoints in the loader itself, stage 1 makes sure all other constructors run in shuffled code. The last constructor to be called (a side effect of `LD_PRELOAD`) is stage 2's own constructor; stage 2 creates a dedicated Shuffler thread, erases the original copy of all other code, and resumes execution at the shuffled ELF entry point.

4.3.1 Full Shuffling Requirements

Compiler flags We require the program binary and all dependent libraries to be compiled with `-Wl,-q`, a linker flag that preserves relocations. Since we require symbols and DWARF unwind information, the user must avoid `-s`, which strips symbols, and `-fno-asynchronous-unwind-tables`, which elides DWARF unwind information. For simplicity, we do not support some DWARF 3 and 4 opcodes, so the user may need to pass `-gdwarf-2` when compiling

[3]Fortunately, GCC only emits jump tables of size five or more, which makes this heuristic very accurate.

[4]`LD_PRELOAD=./libshuffle0.so:./libshuffle.so`

[5]We require a patch to fully use this mechanism; see Section 4.3.1.

C++. Finally, we found that some SPEC CPU programs required `-fno-omit-frame-pointer`, due to a limitation in our DWARF unwind implementation.

System modifications The `-z initfirst` loader feature currently only supports one shared library, and `libpthread` already uses it. To maintain compatibility with `libpthread`, we patched the loader to support constructor prioritization in multiple libraries. Our 24-line patch transforms a single variable into a linked list. (We have submitted our patch to `glibc` for review.)

Since shuffled functions must be within ± 2GB of each other, we simplify Shuffler's task and map all ELF `PT_LOAD` sections into the lower 32 bits of the address space (1-line change to the loader). Since `glibc` and `libdl` refer directly to variables in the loader with only 32-bit displacements, we also place the loader itself into that region, preresolving its relocations with `prelink` [3]. Finally, we disabled a manually-constructed jump table in the `vfprintf` of `glibc`, which used computed `goto` statements (1-line change). No other library changes were necessary.

4.4 Implementation Optimizations

Generating new code The Shuffler thread maintains a large code *sandbox* that stores shuffled (and currently executing) functions. In each shuffle period, every function within the sandbox is duplicated and the old copies are erased. The sandbox is split in half so that one half may be easily erased with a single `mprotect` system call.[6] Performance suffers if each function is written to an independent location in the sandbox. The bottleneck is in issuing many `mprotect` system calls (we do not want to expose the whole sandbox by making it writable). Instead, we maintain several *buckets* (64KB–1MB) and each function is placed in a random bucket; when a bucket fills up, it is committed with an `mprotect` call and a fresh bucket is allocated. The Memory Protection Keys (MPK) feature on upcoming Intel CPUs [16] may allow buckets to be created even more efficiently.

Generating function addresses with high entropy (*i.e.,* uniformly at random) is a challenging task. The simplest allocator would pick random addresses repeatedly until a free location is found, but this may require many attempts due to fragmentation. Instead, we use a Fenwick Tree (or Binary Indexed Tree) [30,32] for our allocations. Our tree keeps track of all valid addresses for new buckets, storing disjoint intervals; it also tracks the sum of interval lengths (*i.e.,* the amount of free space). We can select a random number less than this sum and be assured that it maps to some valid free location, and compute this mapping in logarithmic time. This guarantees that each allocation is selected uniformly at random.

[6]This also clears the old code from the instruction cache, since Linux's updates to the Translation Lookaside Buffer (TLB) flush the appropriate cache lines as per Section 4.10.4 of the Intel manual [39].

Stack unwinding Stack unwinding is performed by parsing the DWARF unwind information from the executable. This information is used by exception handling code, and by the debugger to get accurate stack traces. We found that the popular library `libunwind` [35] was quite unwieldy, used unwind heuristics, and made it difficult to add an address-translation mechanism. Hence, we wrote a custom unwind library with a straightforward DWARF state machine, using binary search to translate between shuffled and original addresses. We generate DWARF information for new code inserted through binary rewriting, and also record the points where return addresses are (or are not) encrypted.

Binary rewriting Shuffler's load-time transformations are all implemented through binary rewriting. We disassemble each function with diStorm [21] and produce intermediate data structures which we call *rewrite blocks*. Rewrite blocks are similar to basic blocks but may be split at arbitrary points to accommodate newly inserted instructions. Through careful block splitting, we can choose whether incoming jumps execute or skip over new instructions as appropriate. This data structure also allows fast linear updates of internal offsets for jump instructions. We promote 8-bit jumps to 32-bit jumps (iteratively) if the jump targets have become too far away. Once jumps and other data structures are consistent, the final code size is known and we create the first shuffled copy of a function. The runtime shuffling process copies the shuffled version of each function to a new location and patches it without invoking the rewriting procedure.

5 Performance Evaluation

Unless otherwise noted, performance results were measured on a dual-socket 2.8GHz Westmere Xeon X5660 machine, with 64GB of RAM and 24 cores (hyperthreading enabled), running Ubuntu 16.04 with GCC 4.8.4.

5.1 SPEC CPU2006 Overhead

We ran Shuffler on all C and C++ benchmarks in SPEC CPU2006, over a range of different shuffling periods. The SPEC baseline was compiled with its default settings (`-O2`). The shuffled versions were compiled the same way with the addition of `-Wl,-q` (see Section 4.3.1), and also `-fno-omit-frame-pointer` due to a limitation in our DWARF unwind implementation. Since Shuffler does not yet support C++ exceptions, we replaced exceptions with conventional control flow in `omnetpp` (20-line change) and `povray` (15 lines).

Effect of shuffling rate Figure 5 shows the overhead observed by the single-threaded SPEC benchmarks at different shuffling rates, excluding the overhead of the

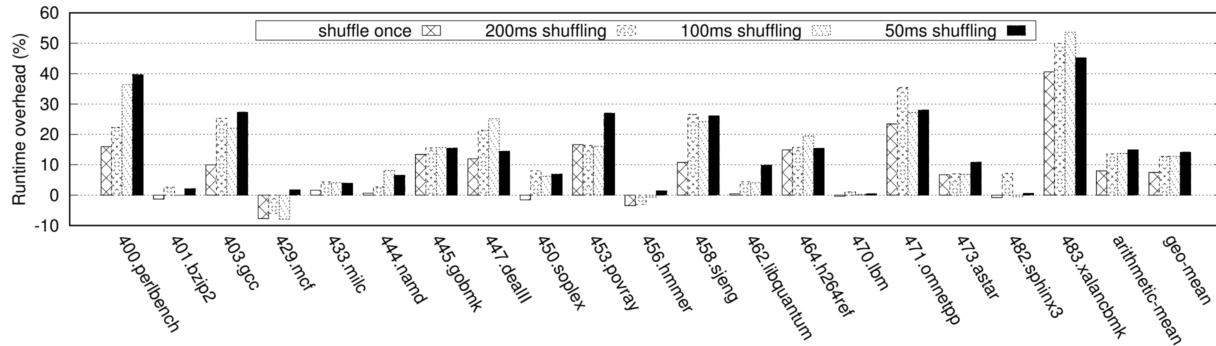


Figure 5: Shuffler performance (shown as overhead percentage) on SPEC CPU2006 at different shuffling rates.

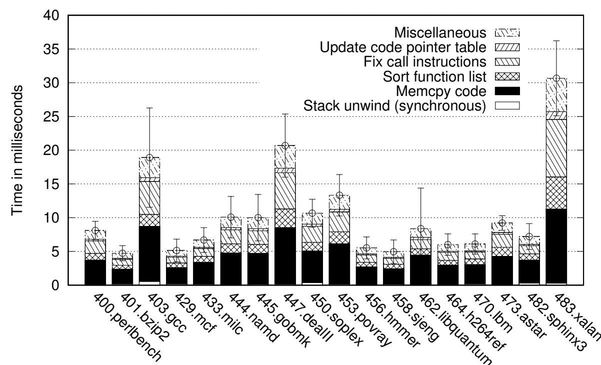


Figure 6: SPEC CPU continuous shuffling breakdown. Synchronous (stack unwind) overhead is barely visible at the bottom. Data for omnetpp was not gathered.

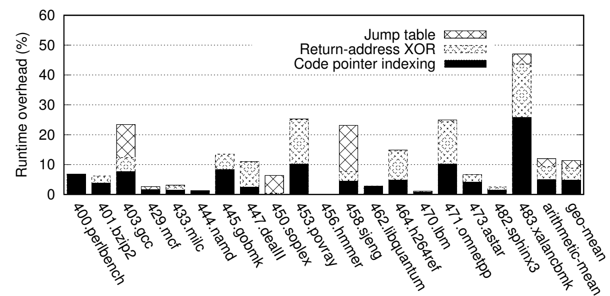


Figure 7: Static transformation overheads in SPEC CPU.

Shuffler thread. The average overheads are 7.99% (shuffling once), 13.5% (200ms shuffling), 13.7% (100ms shuffling), and 14.9% (50ms shuffling). Considering that thousands of shuffles were performed in each case (the runtime per program is from 3.5–10 minutes), the observed overhead is acceptable. Note that faster shuffling rates do not cause significant slowdown, because the static code rewriting cost is paid only once (up-front).

Asynchronous overhead By design, Shuffler offloads the majority of the shuffling computations onto another CPU core (see Figure 6). We assume that the protected system is not at full capacity and has sufficient cycles to execute the Shuffler thread concurrently.

We can, however, approximate the shuffling overhead: the asynchronous shuffling time divided by the shuffling period yields the CPU load. Assuming gcc asynchronously shuffles in 25 milliseconds, it would use 50% of the offload core in a shuffle period of 50 milliseconds, and 25% in a shuffle period of 100 milliseconds. We confirmed this approximation by measuring the reported CPU usage once per second, as each SPEC CPU program ran. The true overheads were within a few percentage points of the approximation. For instance, xalancbmk was predicted to use 61.31% of the CPU in the Shuffler thread and in fact used 58.64%. This overhead is examined in more detail in Section 5.2.

Synchronous overhead The only synchronous work in Figure 6 is the short time when the program thread is interrupted via a signal to perform stack unwinding. Shuffler's stack unwind performance is linear in the call stack depth, processing 3247 stack frames per millisecond (including the thread barrier synchronization time between Shuffler and the program threads). Most SPEC programs have modest call stack depths, except xalancbmk, where certain stages have call stacks at least 20,000 deep (up to 45,000), and take up to 6 ms to unwind. The highest average unwind time is 0.53 ms for gcc; the Shuffler thread unwinds itself in ∼0.025 ms.

5.1.1 Static overhead on SPEC CPU

In Figure 7, we break down the overhead observed due to static code transformations (when only shuffling once). This overhead is purely from the inserted instructions. The average overhead is 2.68% due to jump table rewriting, 4.36% due to return address encryption, and 4.78% due to code pointer abstraction. Jump table numbers are relative to a baseline with jump tables; everything else, to one without (the baselines only differ by 0.45%).

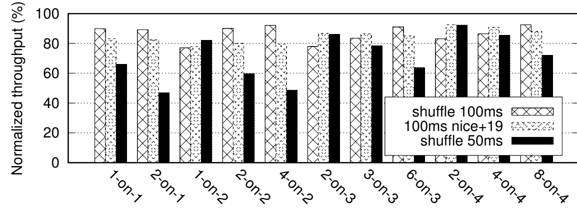
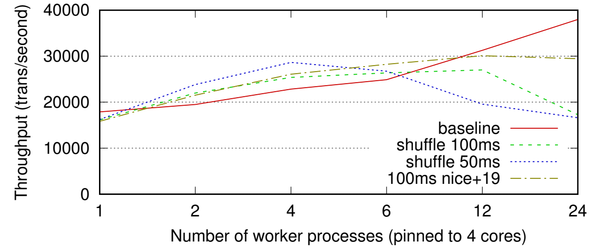
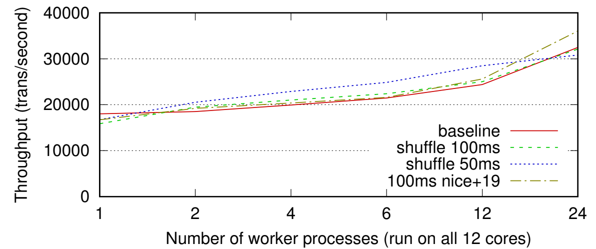


Figure 8: Shuffler thread impact on Nginx throughput. t-on-n means t worker processes pinned to n cores.



(a) Nginx workers and Shuffler threads pinned to 4 cores.



(b) Shuffled Nginx running on all 12 available cores.

Figure 9: Shuffled Nginx performance at a larger scale.

Jump tables Jump table overhead can be high, because our transformation to support code pointer indices is inefficient for position-dependent jump tables (see Section 4.2). With greater compiler integration or more thorough binary rewriting, this overhead can be reduced.

Return-address encryption The return-address encryption overhead increases as the program makes more function calls. The 4.36% overhead is higher than for a straightforward stack canary scheme. However, it also provides disclosure resilience for return addresses, which is essential for our method. Other strong shadow stack schemes are available [22], with comparable performance. We could use dynamically allocated table indices for return addresses, but disrupting `call`/`ret` pairs has high performance overhead [22, 43].

Code pointer abstraction The code pointer abstraction overhead is high when the program makes a large number of indirect calls. For instance, `xalancbmk` makes 3.35 million indirect calls on the test input size, 3.60 billion calls on train, and likely an order of magnitude more on ref. This overhead is mostly unavoidable; the layer of indirection introduced by these transformations is what allows Shuffler to invalidate old code addresses without using (code) pointer tracking. We confirmed with the Linux `perf` tool that the percentage overhead from code pointer abstraction corresponds to the percentage of the newly inserted instructions.

5.2 Nginx Overhead

We ran performance experiments on the Nginx 1.4.6 web server. Our setup used two dual hex-core machines on a dedicated gigabit network, each with Turbo mode and hyperthreading disabled (hence 12 cores each). The client machine was the same one used for SPEC CPU, and the server had two 2.50GHz Xeon E5-2640 CPUs.

To generate client load, we used the multithreaded Siege [31] benchmarking tool. We used a request size of 100 bytes with 32 concurrent connections. This configuration ensures that the server is CPU-bound; larger sizes may exceed network bandwidth, while more connections cause CPU scheduling delays on the client machine. Measurements are reported as the average of five 30-second runs. Siege reported a latency of less than 10 milliseconds, and a concurrency level between 30.86 and 31.76, for all baseline and shuffled test cases.

Shuffler thread overhead First, we investigated the performance of Shuffler threads in Nginx. In the beginning, Nginx has one master process and one Shuffler thread, and then it forks into a user-specified number of worker processes (each with their own Shuffler thread). In our evaluation, we pinned all Nginx workers and their associated Shuffler threads to a case-dependent number of cores, and excluded the master and its Shuffler thread by pinning them to a different core on the same socket.

The results are shown in Figure 8. In the 1-on-1 case, there is one Nginx worker process and its Shuffler thread on a single core. These two threads will compete for scheduling time slices on the same core, and whenever the Shuffler thread is scheduled, throughput is stalled (since Nginx can only run on the same core). Shuffler takes about 15 milliseconds to shuffle Nginx, so we would expect 15% slowdown at 100 millisecond shuffling and 30% slowdown at 50 millisecond shuffling. The measurements track this expectation quite closely.

Some cases have greater overcommitting, *e.g.,* 4-on-2 has four Nginx workers plus four Shuffler threads on two cores. Overhead is still reasonable, and the throughput is around 85%-90% of the baseline. Setting the Shuffler threads to lower priority (nice +19) at 100 ms does not increase throughput here, although it does help when a greater portion of the system is in use (see below).

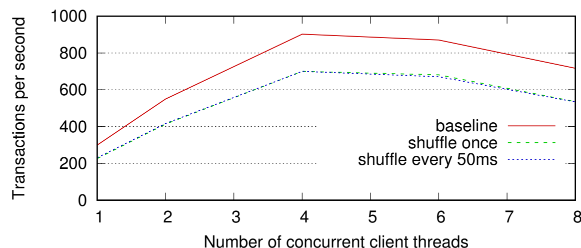


Figure 10: MySQL transaction throughput as measured by SysBench. Shuffle once and shuffle every 50ms incurs the same overhead.

Full-scale Nginx overhead In our second set of Nginx experiments, we pinned all threads (including the master process) to a certain number of cores. Figure 9a shows the results when pinned to four cores on the same socket, and Figure 9b shows the results with no pinning (*i.e.,* all 12 cores available for scheduling). In the four-core case, the overhead starts to get very high with 12 and 24 workers. This is because the Linux scheduler must try to place all worker threads, Shuffler threads, and the master (for a total of 26 or 50 threads) onto a mere four cores. To assist the scheduler, we made each Shuffler thread set its nice value to +19 (low priority) at 100 ms, which results in longer shuffling latencies but greater throughput since Nginx worker threads get more CPU time.

In the case of no CPU pinning (Figure 9b), Shuffler performance tracks the baseline very well. There is less overcommitting here: even in the 24 worker case, each core has two workers and two Shuffler threads to schedule. In the nice+19 case, shuffling latencies (for 24-on-12) are high with average 18.1 ms and std. dev. 266, instead of the original average 17.4 ms, std. dev. 39. Overall, we measured small speedups over the baseline, which is likely experimental noise; Shuffler threads do not significantly impact the overall system performance. This full-system experiment incorporates the master process overhead, as well as kernel I/O threads, which normally ignore userspace CPU pinning (and use idle cores).

5.3 Other Macro Benchmarks

MySQL We shuffled MySQL continuously every 50 ms (asynchronous shuffling takes 30 ms), querying its 10 million row database using SysBench on localhost. The machine had 24 cores and MySQL used the default of 16 threads. Figure 10 shows that the performance overhead (30.9%) is almost completely due to static rewriting, and shuffling every 50ms has the same performance as shuffling once. This is partially because unlike Nginx, where workers are separate processes and thus require separate Shuffler threads, MySQL worker threads are all randomized by a single Shuffler thread.

Program	Code + Syms/Relocs	Data Structs + Overhead
Shuffler	0.16MB + 0.15MB	(included below)
SQLite	2.20MB + 1.63MB	32.2MB + 23.7MB
Nginx	3.14MB + 2.68MB	45.7MB + 37.7MB
Xalan	4.36MB + 5.09MB	76.7MB + 44.3MB

Figure 11: Program size and Shuffler overhead.

So using multithreaded workers instead of multiprocess workers can amortise Shuffler's performance overhead, with an appropriate tradeoff in security (see Section 6.2).

SQLite SQLite has a reasonably small codebase which only takes the Shuffler thread 5 milliseconds to shuffle. We shuffled it at 20 ms for a week without incident.

Mozilla's SpiderMonkey We shuffled the JavaScript engine SpiderMonkey and it passed its test suite of 3600 test cases. We had to disable JIT code generation (IonMonkey); Shuffler could in future handle JIT code if it was informed of when new code chunks were generated.

5.4 Memory Overhead

Figure 11 reports the code/relocation/symbol section sizes for programs and their libraries. Shuffler's total memory overhead consists of: an in-flight copy of all code sections; the code pointer table (1MB); one signal stack (64KB) per thread; metadata structures like relocation and symbol hash tables; and the current permuted list of functions (32 bytes per function). For allocation efficiency, code copies are stored in a preallocated 160MB sandbox. We use a custom `malloc` implementation [41], and report its bookkeeping/fragmentation overhead separately. The permuted function list is destroyed and recreated for each shuffle period.

5.5 TASR Performance Comparison

The closest re-randomization system to Shuffler is TASR [7], which has a reported overhead of 0–10% (2.1% average) on SPEC CPU. However, those numbers are against a baseline compiled with `-Og`, which only performs optimizations that preserve debugging information. Such optimizations are fairly limited: we found that SPEC CPU with `-Og` is 30% slower than with the normal optimization level `-O2`. In other words, TASR's performance overhead is 30-40% relative to the true baseline (while Shuffler's is under 15%). Unfortunately, using `-Og` is intrinsic to any scheme like TASR that requires accurate tracking of source-level variables.

Additionally, TASR's scheme of randomizing on I/O system call pairs provides strong guarantees, but seems unlikely to scale to real-world server applications. In the case of Nginx, we measured that processing a 100KB request takes 0.22 milliseconds. Let us assume that TASR can randomize Nginx in 15 milliseconds (note that this

is Shuffler's rate—TASR is likely to take even longer since it injects and runs a pointer updater process). Since TASR re-randomizes after each request, it would incur 15 milliseconds of latency per 0.22 milliseconds of useful work, resulting in 1.5% of the original throughput. The scheme could be extended to allow multiple requests to run in parallel, but this would still require 68 threads on 68 cores to maintain the original throughput.

6 Security Analysis

In this section we show how Shuffler defends against existing attacks assuming all its mechanisms are in place, including code pointer indirection, return address encryption, and continuous shuffling every r milliseconds. Then we discuss other possible attacks against the Shuffler infrastructure, and follow up with some case studies.

6.1 Analysis of Traditional Attacks

Normal ROP It is fairly obvious that a traditional ROP attack will fail when the target is being shuffled, because the addresses of gadgets are hard-coded into the exploit. Shuffler's code sandbox currently has 27 bits of entropy (a 31-bit sandbox should be possible as per Section 4.1) and gadgets could be anywhere in the sandbox. Thus, if the ROP attack uses N distinct gadgets, the chance of it succeeding is approximately 2^{-27N}. Any attack which desires better odds needs to incorporate a memory disclosure component to discover what Shuffler is doing.

Indirect JIT-ROP Indirect JIT-ROP relies on leaked code pointers and computes gadgets accordingly. Because code pointers are replaced with table indices, the attacker cannot gather code pointers from data structures; nor can the attacker infer code pointers from data pointers, since the relative offset between code and data sections changes continuously. While the attacker can disclose indices, these are not nearly as useful as addresses: they can only be used to jump to the beginning of a function, and they cannot reveal the locality of nearby functions. We assume indices are randomly ordered at load time, with gaps (traps) in the index space to prevent an attacker from easily brute-forcing it [18]. The table itself is a potential source of information, but the table's location is randomized and it is continuously moved (see Section 6.2 below). Return addresses are encrypted with an XOR cipher, so disclosing them does not reveal true code addresses. In fact there are no sources of code pointers accessible to an attacker by way of memory disclosure, and so indirect JIT-ROP is impossible by construction.

Direct JIT-ROP In direct JIT-ROP [55], the attacker is assumed to know one valid code address, and employs a memory disclosure recursively, harvesting code pages and finding enough gadgets for a ROP attack. A control flow hijack is used to kick off the exploit execution.

Our argument against JIT-ROP is threefold. First, the attacker must be able to obtain the first valid code address, and as described for indirect JIT-ROP, there is no accessible source of code pointers in the program. Thus the attacker must resort to brute force or side channels (as for Blind ROP below). Second, once an attack has been completely constructed, there is no easy way to jump to an address of the attacker's choosing: indirect calls and jumps treat their operands as table indices, not addresses, while return statements mangle the return address before branching to a target. The attacker must therefore use a partial return address overwrite (described below in Section 6.2), which itself has a significant chance of failure.

Thirdly, and most importantly, the entire attack must be completed within the shuffle period of r milliseconds. No useful information carries over from one shuffle period to the next, and all previously discovered code pages and gadgets are immediately erased. If the attacker can do everything in r milliseconds, they win; thus, the defender should select a small enough r to disrupt any anticipated attacks. We discuss the attack time required in Section 6.3. The fastest published attack times are on the order of several seconds, not tens of milliseconds.

Blind ROP Blind ROP [8] tries to infer the layout of a server process by probing its workers, which are `fork`ed from the parent and have the same layout. The attack uses a timing channel to infer information about the parent based on whether the child crashed or not. Shuffler easily thwarts this attack because it randomizes child and parent processes independently.

6.2 Shuffler-specific Attacks

Breaking XOR encryption Our XOR encryption is less vulnerable to brute force than typical XOR ciphers. Leaking multiple return addresses does not allow the attack to easily construct linear relations, because there are two unknowns: random values (addresses) encrypted under a random key. The addresses are re-randomized during each shuffle period, and the XOR key could be too. If every function uses it own key, the attacker's task becomes even harder [10]. The keys are stored at unknown addresses in thread-local storage. While there is a small window of two instructions after calls during which the unencrypted return address is visible on the stack, this would be difficult to exploit because the attacker cannot insert any intervening instructions—though a determined attacker might try to do so from another thread.

It is possible to bypass XOR in other ways. For example, an attacker might partially overwrite an encrypted return address, attempting to increment the return address by a small amount without knowing the plaintext value. This could be used to initiate execution of a misaligned gadget, or to trampoline through a return instruction and jump straight to an attacker-controlled address. Such an

attack would be difficult; the attacker would need to find a function on the call stack with appropriate known code layout, and then brute-force several bits of the canary.

Ciphertext-only attacks The attacker could attempt to swap valid code pointer indices. This allows an attacker to jump to the beginning of functions whose address is taken, similar to the restrictions under coarse-grained Control Flow Integrity (CFI) [61, 62]—and such defenses have been bypassed [23, 36]. The mapping between indices and functions would have to first be discovered (subject to permutation and traps). We consider this a data-only attack [12]. As per Section 2.1, we do not attempt to add to the literature for data-only attacks.[7]

The attacker might swap valid encrypted return addresses on the stack. This is equivalent to jumping to call-preceded gadgets (as in coarse-grained CFI), but using only those functions which occur on the call stack. While such an attack may be theoretically possible, it has not been demonstrated in the literature—especially within the constraints of a single shuffle period, where return addresses change every *r* milliseconds.

Parallel attacks When Shuffler is defending a multi-threaded program, every thread uses the same shuffled code layout. Thus, an attacker might run a parallel disclosure attack, multiplying the information that may be gathered from a single-threaded program. However, parallel disclosure is limited by dependencies—often one page's address is computed from another's content, so the disclosures are not parallelizable. In the worst case, defending a parallel attack requires a linearly faster shuffling rate. Currently, the user can run a multiprocess program instead (like Nginx) to avoid this issue. We also used the `%gs` register to store our code pointer table intentionally so that code could be shared between threads. It would be fairly straightforward to use the thread-local `%fs` register instead to maintain separate code copies and pointer tables for each thread, at a corresponding increase in memory and CPU use.

Exploiting the Shuffler infrastructure Since Shuffler runs in an egalitarian manner in the same address space as the target, it may be vulnerable to attack. Shuffler's code is shuffled and defended in the same way as the target, and any specific functionality (*e.g.,* dynamic index allocation) is not accessible through static references. However, Shuffler's data structures might be disclosed at runtime—*e.g.,* to reveal the location of every chunk of code. We are careful to place sensitive information in exactly one data structure, the list of chunks, which is itself destroyed and moved in each shuffle period. There is a single global pointer to this list, which is stored in the `%gs` table along with code pointers.

[7] Thwarting this means updating indices at runtime; see Section 3.2.

Shuffler's code pointer table might itself be used to execute functions, or read or write function locations. As described earlier in Section 6.1, we assume that the table contains traps or invalid entries. This impedes execution of gadgets and requires the index-to-code mapping to be unravelled first. However, the table can be read and written directly with `%gs`-relative gadgets—which are not used by shuffled code but may occur at misaligned offsets. Writes can be disallowed using page permissions. Reads yield information that is only useful for one shuffle period; it is also a "chicken-and-egg" problem to rely on such a gadget to find one's gadgets.

Although the table contains many addresses that the attacker would like to disclose, we assume that the table location is randomized and is continuously moving during the shuffling process. The table's location is only stored in kernel data structures and the inaccessible model-specific register `%gs`. While x86 has a new instruction to read `%gs`, called `RDGSBASE`, it must be enabled through processor control flags (Linux v4.6 does not support that feature). Thus, the attacker must find the table's location through cache timing attacks or allocation spraying [37, 48], which has not been shown to be effective against a continuously moving target.

Finally, even if all of Shuffler's data is disclosed, the addresses for the next shuffle period can be made unpredictable by reseeding Shuffler's random number generator with the kernel-space PRNG `/dev/urandom`.

Shuffler thread compromise If the Shuffler thread crashes for whatever reason, the target program could continue executing its current copy of code unhindered (and undefended). To guard against this, we install signal handlers for common fatal signals. Our default policy is to terminate the process if a crash occurs in Shuffler code. We could also attempt to restart the Shuffler thread (as is done on fork). Instead of causing an outright crash, the attacker could attempt to hang the Shuffler thread, *e.g.,* by pretending that another thread has been created through data structure corruption. This particular technique would cause all threads to hang in the post-unwind synchronization barrier, inside Shuffler code, which is not very useful for an attacker. Still, if a user is concerned that the Shuffler thread may be compromised, an external watchdog can periodically ensure (*e.g.,* by examining `/proc/<pid>/maps`) that shuffling is still occurring.

6.3 Case Studies

Disclosing memory pages When conducting a JIT-ROP attack, the attacker has a tradeoff: either quickly scan memory pages for desired gadgets, which may require many source pages; or, spend more time looking for gadgets in a small number of pages, which can be computationally prohibitive. The original JIT-ROP [55] attack searches through 50 pages to find the gadgets for

an attack, and takes 2.3–22 seconds to carry out a full exploit. The ROP compiler Q [52] can attack executables as small as 20KB, but due to their use of heavyweight symbolic execution and constraint solving, their published real-world attack computation times are 40–378 seconds.

Fetching pages takes time because real memory disclosures do not execute instantaneously. The original JIT-ROP [55] attacks can harvest 3.2, 22.4, and 84 pages/second (*e.g.,* requiring between 12 and 312 milliseconds per page). We reproduced Heartbleed on OpenSSL 1.0.1f using Metasploit [45] and found that the attack takes 60ms to complete (17.2ms per additional disclosure), when the attacker is on the local machine.

Network communication latency For server programs, the network communication latency must be added to every memory disclosure's execution time. According to data from WonderProxy [50], long-distance packet speeds are about 22% the speed of light. We tested this by communicating between servers on the east and west coast of the United States, observing 65.94 and 67.57 ms ping times where 59.27 was predicted. Thus, every millisecond of round-trip ping implies a physical separation of 41 miles (66 km). For example, to perform a single disclosure and then a control-flow hijack against a server shuffled every 20 milliseconds, the attacker would need to be within 820 miles (1320 km).

Continuous re-randomization ensures that addresses are only valid for a short time period. One could eliminate this time window entirely by introducing artificial latency for requests. Each request response would be held in an outgoing queue until a re-randomization has occurred—increasing the server's latency, but guaranteeing that all leaked information is already out-of-date.

Small-scale JIT-ROP attack We created a small vulnerable server to simulate a JIT-ROP scenario. The program prints its stack canary and a known code address, using inline assembly to read the code pointer table. We have an 8-byte memory disclosure (a request which overruns a buffer and corrupts a pointer). We use this vulnerability repeatedly to leak a full 4KB page (which takes 8 milliseconds over loopback). Finally, we overwrite a return address to point at a leaked function. With 8 millisecond shuffling or faster, the attack crashes the target; at slower shuffling rates, the attack succeeds.

Real-world Blind-ROP attack We reproduced the Blind-ROP [8] attack against Nginx 1.4.0 (using CVE-2013-2028 [44]). We measured that the attack takes seven minutes to complete. When Nginx was shuffled, the attack was unable to find the Procedure Linkage Table or stack canary; it received false feedback since parent and child processes are randomized independently.

7 Discussion and Future Work

The commonly accepted wisdom is that performing analysis on binaries is challenging. In fact, while handcrafted binaries can be pathological, compiler-generated code is relatively straightforward to disassemble. Thus, building binary-level defenses is quite possible, especially for symbol- and relocation-augmented binaries.

We are able to perform continuous re-randomization quite efficiently. This is partially because program code size is small, and because the cost of code rewriting is paid only once up-front (not during each shuffle). However, while shuffling in a separate thread is excellent for efficiency, it can lead to unpredictable shuffling latencies, especially under load. Ideally, the target code would need to check in periodically with Shuffler and not run indefinitely. Also, while we currently use a single Shuffler thread, the shuffling process is parallelizable to multiple worker threads if higher shuffling rates are desired.

Most defensive techniques exist outside the infrastructure they defend, or declare themselves part of the trusted computing base. We hope that Shuffler's design will inspire more egalitarian techniques, and in general more techniques that pay attention to their own attack surface.

8 Conclusion

We present Shuffler, a system which defends against all forms of code reuse through continuous code re-randomization. Shuffler randomizes the target, all of the target's libraries, and even the Shuffler code itself—all within a real-time shuffling deadline. Our focus on egalitarian defense allows Shuffler to operate at the same level of privilege as the target, from within the same address space, enabling deployment in environments such as the cloud. We require no modifications to the compiler or kernel, nor access to source code, leveraging only existing compiler flags to preserve symbols and relocations. For the best possible performance, we perform shuffling asynchronously, making use of spare CPU cycles on idle cores. Programs spend 99.7% of their time running unhindered, and only 0.3% of their time running stack unwinding to migrate between copies of code. Shuffler can randomize SPEC CPU every 50 milliseconds with 14.9% overhead. We shuffled real-world applications including MySQL, SQLite, Mozilla's SpiderMonkey, and Nginx. Finally, Shuffler scales well on Nginx, up to a full system load of 24 worker processes on 12 cores.

9 Acknowledgements

We thank the anonymous reviewers, our shepherd Andrew Baumann, and Mihir Nanavati for their valuable comments. This paper was supported in part by ONR N00014-12-1-0166 and N00014-16-1-2263; NSF CCF-1162021, CNS-1054906, and CNS-1564055; an NSF CAREER award; and an NSERC PGS-D award.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proc. of ACM CCS* (2005).

[2] ALEPHONE. Smashing the stack for fun and profit. https://users.ece.cmu.edu/~adrian/630-f04/readings/AlephOne97.txt, 1997.

[3] ARCH WIKI. Prelink. https://wiki.archlinux.org/index.php/Prelink, 2015.

[4] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proc. of ACM CCS* (2014).

[5] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proc. of USENIX Security* (2014), pp. 433–447.

[6] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of USENIX Security* (2005), pp. 271–286.

[7] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., AND OKHRAVI, H. Timely rerandomization for mitigating memory disclosures. In *Proc. of ACM CCS* (2015), pp. 268–279.

[8] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIERES, D., AND BONEH, D. Hacking blind. In *Proc. of IEEE S&P* (2014), pp. 227–242.

[9] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proc. of ACM CCS* (2011), pp. 30–40.

[10] BRADEN, K., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., LIEBCHEN, C., AND SADEGHI, A.-R. Leakage-resilient layout randomization for mobile devices. In *Proc. of NDSS* (2016).

[11] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. of USENIX Security* (2015), pp. 161–176.

[12] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proc. of USENIX Security* (2005).

[13] CHEN, Y., WANG, Z., WHALLEY, D., AND LU, L. Remix: On-demand live randomization. In *Proc. of ACM CODASPY* (2016), pp. 50–61.

[14] CONTI, M., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., NEGRO, M., LIEBCHEN, C., QUNAIBIT, M., AND SADEGHI, A.-R. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proc. of ACM CCS* (2015), pp. 952–963.

[15] CORBET, J. x86 NX support. http://lwn.net/Articles/87814/, 2004.

[16] CORBET, J. Memory protection keys [lwn.net]. https://lwn.net/Articles/643797/, 2015.

[17] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *Proc. of IEEE S&P* (2015), pp. 763–780.

[18] CRANE, S. J., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., DE SUTTER, B., AND FRANZ, M. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proc. of ACM CCS* (2015), pp. 243–255.

[19] CURTSINGER, C., AND BERGER, E. D. Stabilizer: Statistically sound performance evaluation. In *Proc. of ACM SIGARCH* (Mar. 2013), pp. 219–228.

[20] CVEDETAILS. Vulnerability distribution of CVE security vulnerabilities by types. https://www.cvedetails.com/vulnerabilities-by-types.php, 2016.

[21] DABAH, G. distorm3. http://ragestorm.net/distorm/, 2003–2012.

[22] DANG, T. H., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. In *Proc. of ACM CCS* (2015), pp. 555–566.

[23] DAVI, L., LEHMANN, D., SADEGHI, A.-R., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proc. of USENIX Security* (Aug. 2014).

[24] DAVI, L., LIEBCHEN, C., SADEGHI, A.-R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proc. of NDSS* (2015).

[25] DEBIAN. Hardening - Debian Wiki. https://wiki.debian.org/Hardening, 2015.

[26] DEBIAN. sbuild - Debian Wiki. https://wiki.debian.org/sbuild, 2016.

[27] EAGLE, C. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[28] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proc. of ACM CCS* (2015), pp. 901–913.

[29] FEDORA. Harden All Packages - Fedora Project. https://fedoraproject.org/wiki/Changes/Harden_All_Packages, 2016.

[30] FENWICK, P. M. A new data structure for cumulative frequency tables. *Software: Practice and Experience 24*, 3 (1994), 327–336.

[31] FULMER, J. Siege home. https://www.joedog.org/siege-home/, 2012.

[32] GEEKSFORGEEKS. Binary indexed tree or Fenwick tree. http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/, 2015.

[33] GIONTA, J., ENCK, W., AND NING, P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proc. of ACM CODASPY* (2015), pp. 325–336.

[34] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. of USENIX Security* (2012), pp. 475–490.

[35] GNU. The libunwind project. http://savannah.nongnu.org/projects/libunwind/, 2014.

[36] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proc. of IEEE SOSP* (2014).

[37] GÖKTAS, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining information hiding (and what to do about it). In *Proc. of USENIX Security* (2016).

[38] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'd My Gadgets Go? In *Proc. of IEEE SOSP* (2012), pp. 571–585.

[39] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, Mar 2010.

[40] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *Proc. of USENIX OSDI* (2014), pp. 147–163.

[41] LEE, D. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000.

[42] LU, K., NÜRNBERGER, S., BACKES, M., AND LEE, W. How to make ASLR win the clone wars: Runtime re-randomization. In *Proc. of NDSS* (2016).

[43] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC Architecture. In *Proc. of USENIX Security* (2006).

[44] MITRE CORPORATION. CVE-2013-2028. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028, 2013.

[45] MOORE, H., ET AL. The Metasploit Project. http://www.metasploit.com/, 2009.

[46] MSDN. Symbols and symbol files - Windows 10 hardware dev. https://msdn.microsoft.com/en-us/library/ff558825.aspx, 2016.

[47] NIU, B., AND TAN, G. Modular control-flow integrity. In *Proc. of ACM PLDI* (2014).

[48] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *Proc. of USENIX Security* (2016).

[49] PAX TEAM. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[50] REINHEIMER, P. Miles per millisecond: A look at the WonderProxy network. https://wonderproxy.com/blog/miles-per-milisecond/, 2011.

[51] ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCHI, D. Surgically returning to randomized lib(c). In *Proc. of USENIX ACSAC* (2009), pp. 60–69.

[52] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *Proc. of USENIX Security* (2011), pp. 25–25.

[53] SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. Disassembly of executable code revisited. In *Proc. of IEEE WCRE* (2002), pp. 45–54.

[54] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of ACM CCS* (2007), pp. 552–61.

[55] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. of IEEE SOSP* (2013).

[56] SOLAR DESIGNER. lpr libc return exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html, 1997.

[57] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proc. of ACM SIGSAC* (2015), pp. 256–267.

[58] UBUNTU. Security/features - Ubuntu Wiki. https://wiki.ubuntu.com/Security/Features#Userspace_Hardening, 2016.

[59] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of ACM CCS* (2012), pp. 157–168.

[60] XU, J., KALBARCZYK, Z., AND IYER, R. Transparent runtime randomization for security. In *Proc. of IEEE SRDS* (2003), pp. 260–269.

[61] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Proc. of IEEE SOSP* (2013).

[62] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proc. of USENIX Security* (2013).