Dynamic and Secure Memory Transformation in Userspace

Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran

Virginia Tech, Blacksburg, VA, USA
{rlyerly,xiaoguang,binoy}@vt.edu

Abstract. Continuous code re-randomization has been proposed as a way to prevent advanced code reuse attacks. However, recent research shows the possibility of exploiting the runtime stack even when performing integrity checks or code re-randomization protections. Additionally, existing re-randomization frameworks do not achieve strong isolation, transparency and efficiency when securing the vulnerable application. In this paper we present Chameleon, a userspace framework for dynamic and secure application memory transformation. Chameleon is an out-of-band system, meaning it leverages standard userspace primitives to monitor and transform the target application memory from an entirely separate process. We present the design and implementation of Chameleon to dynamically re-randomize the application stack slot layout, defeating recent attacks on stack object exploitation. The evaluation shows Chameleon significantly raises the bar of stack object related attacks with only a 1.1% overhead when re-randomizing every 50 milliseconds.

1 Introduction

Memory corruption is still one of the biggest threats to software security [43]. Attackers use memory corruption as a starting point to directly hijack program control flow [18, 22, 9], modify control data [24, 25], or steal secrets in memory [20]. Recent works have shown that it is possible to exploit the stack even under new integrity protections designed to combat the latest attacks [23, 31, 24, 25]. For example, position-independent return-oriented programming (PIROP) [23] leverages a user controlled sequence of function calls and un-erased stack memory left on the stack after returning from functions (e.g., return addresses, initialized local data) to construct a ROP payload. Data-oriented programming (DOP) also heavily relies on user-controlled stack objects to change the execution path in an attacker-intended way [24–26]. Both of these attacks defeat existing code

This is the author's version of the work posted here per publisher's guidelines for your personal use. Not for redistribution. The final authenticated version is published in the Proceedings of the 25th European Symposium on Research in Computer Security (ESORICS 2020), Guildford, United Kingdom, September 14-18, 2020.

re-randomization mechanisms, which continuously permute the locations of functions [50] or hide function locations to prevent memory disclosure vulnerabilities from constructing gadget chains [14].

In this work, we present Chameleon, a continuous stack randomization framework. Chameleon, like other continuous re-randomization frameworks, periodically permutes the application's memory layout in order to prevent attackers from using memory disclosure vulnerabilities to exfiltrate data or construct malicious payloads. Chameleon, however, focuses on randomizing the stack – it randomizes the layout of every function's stack frame so that attackers cannot rely on the locations of stack data for attacks. In order to correctly reference local variables in the randomized stack layout, Chameleon also rewrites every function's code, further disrupting code reuse attacks that expect certain instruction sequences (either aligned or unaligned). Chameleon periodically interrupts the application to rewrite the stack and inject new code. In this way, Chameleon can defeat attacks that rely on stack data locations such as PIROP or DOP.

Chameleon is also novel in how it implements re-randomization. Existing works build complex runtimes into the application's address space that add non-trivial performance overhead from code instrumentation [17, 14, 50, 1]. Chameleon is instead an *out-of-band* framework that executes in userspace in an entirely separate process. Chameleon attaches to the application using standard OS interfaces for observation and re-randomization. This provides strong isolation between Chameleon and application – attackers cannot observe the re-randomization process (e.g., observe random number generator state, dump memory layout information) and Chameleon does not interact with any user-controlled input. Additionally, cleanly separating Chameleon from the application allows much of the re-randomization process to proceed in parallel. This design adds minimal overhead, as Chameleon only blocks the application when switching between randomized stack layouts. Chameleon can efficiently re-randomize an application's stack layout with randomization periods in the range of tens of milliseconds.

In this paper, we make the following contributions:

- We describe Chameleon, a system for continuously re-randomizing application stack layouts,
- We detail Chameleon's stack randomization process that relies on using compiler-generated function metadata and runtime binary reassembly,
- We describe how Chameleon uses the standard `ptrace` and `userfaultfd` OS interfaces to efficiently transform the application's stack and inject newly-rewritten code,
- We evaluate the security benefits of Chameleon and report its performance overhead when randomizing code on benchmarks from the SPEC CPU 2017 [42] and NPB [4] benchmark suites. Chameleon's out-of-band architecture allows it to randomize stack slot layout with only 1.1% overhead when changing the layout with a 50 millisecond period,
- We describe how Chameleon disrupts a real-world attack against the popular nginx webserver

The rest of this paper is organized as follows: in Section 2, we describe the background and the threat model. We then present the design and implementation of Chameleon in Section 3. We evaluate Chameleon security properties and performance in Section 4. We discuss related works in Section 5 and conclude the paper in Section 6.

2 Background and Threat Model

Before describing Chameleon, we first describe how stack object related attacks target vulnerable applications, including detailing a recent presented position-independent code reuse attack. We then define the threat model of these attacks.

2.1 Background

Traditional code reuse attacks rely on runtime application memory information to construct the malicious payload. Return-oriented programming (ROP) [36, 40] chains and executes several short instruction sequences ending with `ret` instructions, called *gadgets*, to conduct Turing-complete computation. After carefully constructing the ROP payload of gadget pointers and data operands, the attacker then tricks the victim process into using the ROP payload as stack data. Once the ROP payload is triggered, gadget pointers are loaded into the program counter (which directs control flow to the gadgets) and the operand data is populated into registers to perform the intended operations (e.g., prepare parameters to issue an attacker-intended system call). Modern attacks, such as JIT-ROP [41], utilize a memory disclosure vulnerability to defeat coarse-grained randomization techniques such as ASLR [29] by dynamically discovering gadgets and constructing gadget payloads.

Position-independent ROP (PIROP) [23] proposes a novel way to reuse existing pointers on the stack (e.g., function addresses) as well as relative code offsets to construct the ROP payload. PIROP constructs the ROP payload agnostic to the code's absolute address. It leverages the fact that function call return addresses and local variables may remain on the stack even after the function returns, meaning the next function call may observe stack local variables and code pointers from the previous function call. By carefully controlling the application input, the attacker triggers specific call paths and constructs a stack with attacker-controlled code pointers and operand data. This stack construction procedure is called *stack massaging* (Figure 1 (a)). The next step modifies some bits of the code pointers to make them point to the intended gadgets (Figure 1 (b)). This is called *code pointer and data operand patching*. Since code pointers left from stack massaging point to code pages, it is possible to modify some bits of the pointer using relative memory writes to redirect it to a gadget on the same code page. Fundamentally, PIROP assumes function calls leave their stack slot contents on the stack even after the call returns. By using a temporal sequence of different function calls to write pointers to the stack, PIROP constructs a skeleton of the ROP payload. Very few existing defenses break this assumption.

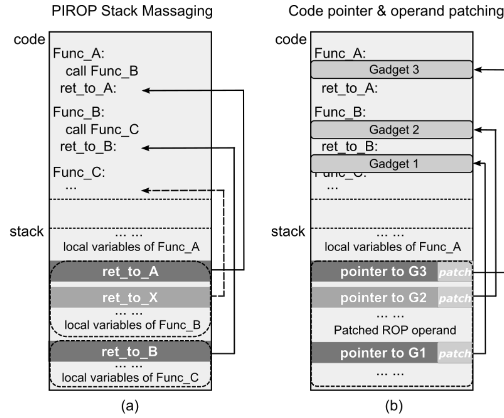

Fig. 1. Position Independent ROP. (a) *Stack massaging* uses code pointers that remain on the stack after the function returns. (b) *Code pointer and operand patching* write part of the massaged stack memory with relative memory writes to construct the payload.

2.2 Threat Model and Assumptions

The attacker communicates with the target application through typical I/O interfaces such as sockets, giving the attacker the ability to send arbitrary input data to the target. The attacker has the target application binary, thus they are aware of the relative addresses inside any 4K memory page windows. The attacker can exploit a memory disclosure vulnerability to read arbitrary memory locations and can use PIROP to construct the ROP payload. The application is running using standard memory protection mechanisms such that no page has both write and execute permissions; this means the attacker cannot directly inject code but must instead rely on constructing gadget chains. However, the gadget chains crafted by the attacker can invoke system APIs such as `mprotect` to create such regions if needed. The attacker knows that the target is running under Chameleon's control and therefore knows of its randomization capabilities. We assume the system software infrastructure (compiler, kernel) is trusted and therefore the capabilities provided by these systems are correct and sound.

3 Design

Chameleon continuously re-randomizes the code section and stack layout of an application in order to harden it against temporal stack object reuse (i.e. PIROP), stack control data smashing and stack object disclosures. As a result of running under Chameleon, gadget addresses or stack object locations that are leaked by memory disclosures and that help facilitate other attacks (temporal stack object reuse, payload construction) are only useful until the next randomization, after which the attacker must re-discover the new layout and locations of sensitive data. Chameleon continuously randomizes the application (hereafter

called the target or child) quickly enough so that it becomes probabilistically impossible for attackers to construct and execute attacks against the target.

Similarly to previous re-randomization frameworks [50, 17], Chameleon is transparent – the target has no indication that it is being re-randomized. However, Chameleon's architecture is different from existing frameworks in that it executes outside the target application's address space and attaches to the target using standard OS interfaces. This avoids the need for bootstrapping and running randomization machinery inside an application, which adds complexity and high overheads. Chameleon runs all randomization machinery in a separate process, which allows generating the next set of randomization information in parallel with normal target execution. This also *strongly isolates* Chameleon from the target in order to make it extremely difficult for attackers to observe the randomization process itself. These benefits make Chameleon easier to use and less intrusive versus existing re-randomization systems.

3.1 Requirements

Chameleon needs a description of each function's stack layout, including location, size and alignment of each stack slot, so that it can randomize each stack slot's location. Ideally, Chameleon would be able to determine every stack slot's location, size and alignment by analyzing a function's machine code. In reality, however, it is impossible to tell from the machine code whether adjacent stack memory locations are separate stack slots (which can be relocated independently) or multiple parts of a single stack slot that must be relocated together (e.g., a struct with several fields). Therefore, Chameleon requires metadata from the compiler describing how it has laid out the stack.

While DWARF debugging information [21] can provide some of the required information, it is best-effort and does not capture a complete view of execution state needed for transformation (e.g., unnamed values created during optimization). Instead, Chameleon builds upon existing work [5] that extends LLVM's stack maps [30] to dump a complete view of function activations. The compiler instruments LLVM bitcode to track live values (stack objects, local variables) by adding stack maps at individual points inside the code. In the backend, stack maps force generation of a per-function record listing stack slot sizes, alignments and offsets. Stack maps also record locations of all live values at the location where the stack map was inserted. Chameleon uses each stack map to reconstruct the frame at that location. The modified LLVM extends stack maps to add extra semantic information for live values, particularly whether a live value is a pointer. This allows Chameleon to detect at runtime if the pointer references the stack, and if so, update the pointer to the stack slot's randomized location. The metadata also describes each function's location and size, which Chameleon uses to patch each function to match the randomized layout. All of the metadata is generated at compile time and is lowered into the binary.

This information could potentially be inferred heuristically, e.g., from a decompiler

Chameleon also needs to rewrite stack slot references in code to point to their new locations and must transform existing execution state, namely stack memory and registers, to adhere to the new randomized layout. To switch between different randomized stack layouts (named *randomization epochs*), Chameleon must be able to pause the target, observe current target execution state, rewrite the existing state to match the new layout and inject code matching the new layout. Chameleon uses two kernel interfaces, `ptrace` and `userfaultfd`, to monitor and transform the target. `ptrace` [48] is widely used by debuggers to inspect and control the execution of *tracees*. `ptrace` allows *tracers* (e.g., Chameleon) to read and modify tracee state (per-thread register sets, signal masks, virtual memory), intercept events such as signals and system calls, and forcibly interrupt tracee threads. `userfaultfd` [27] is a Linux kernel mechanism that allows delegating handling of page faults for a memory region to user-space. When accesses to a region of memory attached to a `userfaultfd` file descriptor cause a page fault, the kernel sends a request to a process reading from the descriptor. The process can then respond with the data for the page by writing a response to the file descriptor. These two interfaces together give Chameleon powerful and flexible process control tools that add minimal overhead to the target.

3.2 Re-Randomization Architecture

Chameleon uses the mechanisms described in Section 3.1 to transparently observe the target's execution state and periodically interrupt the target to switch it to the next randomization epoch. In between randomization epochs, Chameleon executes in parallel with the target to generate the next set of randomized stack layouts and code. Figure 2 shows Chameleon's system architecture. Users launch the target application by passing the command line arguments to Chameleon. After reading the code and state transformation metadata from the target's binary, Chameleon forks the target application and attaches to it via `ptrace` and `userfaultfd`. From this point on, Chameleon enters a *re-randomization loop*. At the start of a new randomization cycle, a *scrambler* thread iterates through every function in the target's code, randomizing the stack layout as described below. At some point, a re-randomization timer fires, triggering a switch to the next randomization epoch. When the re-randomization event fires, the *event handler* thread interrupts the target and switches the target to the next randomization epoch by dropping the existing code pages and transforming the target's execution state (stack, registers) to the new randomized layout produced by the scrambler. After transformation, the event handler writes the execution state back into the target and resumes the child; it then blocks until the next re-randomization event. As the child begins executing, it triggers code page faults by fetching instructions from dropped code pages. A *fault handler* thread handles these page faults by serving the newly randomized code. In this way the entire re-randomization procedure is transparent to the target and incurs low overheads. We describe each part of the architecture in the following sections.

Randomizing stack layouts. Chameleon randomizes function stack layouts by logically permuting stack slot locations and adding padding between the slots.

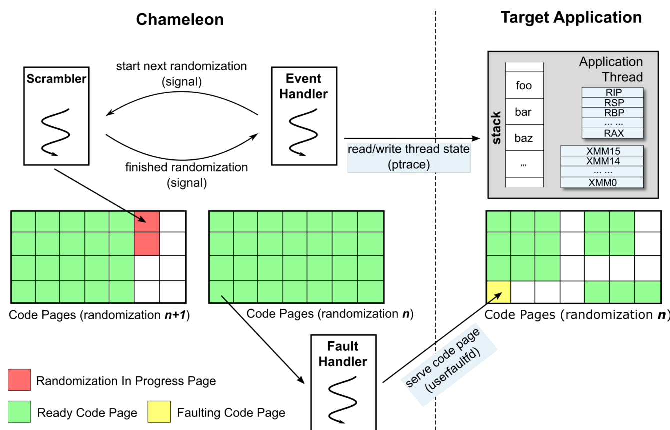

Fig. 2. Chameleon system architecture. An *event handler* thread waits for events in a target thread (e.g., signals), interrupts the thread, and reads/writes the thread's execution state (registers, stack) using `ptrace`. A *scrambler* thread concurrently prepares the next set of randomized code for the next re-randomization. A *fault handler* thread responds to page faults in the target by passing pages from the current code randomization to the kernel through `userfaultfd`.

Chameleon also transforms stack memory references in code to point to their randomized locations. When patching the code, Chameleon must work within the space of the code emitted by the compiler. If, for example, Chameleon wanted to change the size of code by inserting arbitrary instructions or changing the operand encoding of existing instructions, Chameleon would need to update all code references affected by change in size (e.g., jumps between basic blocks, function calls/returns, etc.). Because finding and updating all code references is known to not be statically solvable [47], previous re-randomization works either leverage dynamic binary instrumentation (DBI) frameworks [32, 7, 17] or an indirection table [50, 3] in order to allow arbitrary code instrumentation. There are problems with both approaches – the former often have large performance costs while the latter does not actually re-randomize the stack layout, instead opting to try and hide code pages from attackers. Chameleon instead applies stack layout randomization without changing the size of code to avoid these problems. In order to facilitate randomizing all elements of the stack, Chameleon modifies the compiler to *(1)* pad function prologues and epilogues with `nop` instructions that can be rewritten with other instructions and *(2)* force 4-byte immediate encodings for all memory operands.

Chameleon both permutes the ordering and adds random amounts of padding between stack slots; the latter is configurable so users can control how much memory is used versus how much randomness is added between slots. Figure 3 shows how Chameleon randomizes the following stack elements: *(1) Callee-saved*

x86-64 backends typically emit small immediate operands using a 1-byte encoding

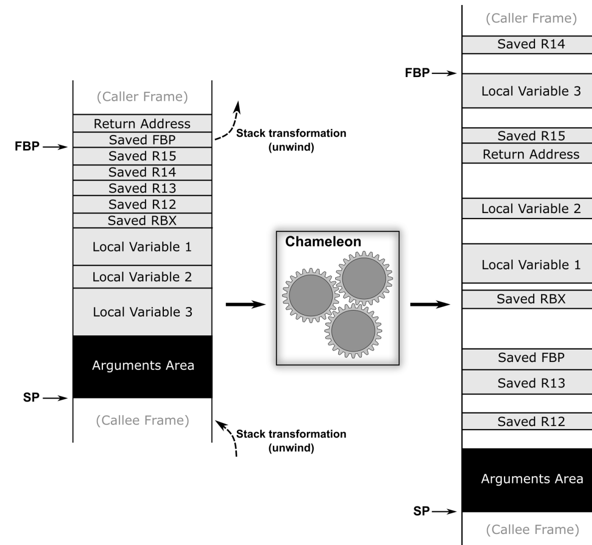

Fig. 3. Stack slot randomization. Chameleon permutes the ordering and adds random amounts of padding between slots.

registers: the compiler saves and restores callee-saved registers through `push` and `pop` instructions. Chameleon uses the `nop` padding emitted by the compiler to rewrite them as `mov` instructions, allowing the scrambler to place callee-saved registers at arbitrary locations on the stack. Chameleon also randomizes the locations of the return address and saved frame base pointer by inserting `mov` intructions in the function's prologue and epilogue. *(2) Local variables*: compilers emit references to stack-allocated variables as offsets from the frame base pointer (FBP) or stack pointer (SP). Chameleon randomizes the locations of local variables by rewriting a variable's offset to point to the randomized location. Chameleon does not currently randomize the locations of stack arguments for called functions as the locations are dictated by the ABI and would require rewriting both the caller and callee with a new parameter passing convention. We plan these transformations as future work.

Serving code pages. Chameleon needs a mechanism to transparently and efficiently serve randomized code pages to the child. While Chameleon could use `ptrace` to directly write the randomized code into the address space of the child application, this would cause large delays when swapping between randomization epochs for applications with large code sections – Chameleon would have to bulk write the entire code section on every re-randomization. Instead Chameleon uses `userfaultfd` and page faults, which allows quicker switches between epochs by demand paging code into the target application's address space.

At startup, Chameleon attaches a `userfaultfd` file descriptor to the target's code memory virtual memory area (VMA). `userfaultfd` descriptors can

only be opened by the process that owns the memory region for which faults should be handled. Chameleon cannot directly open a `userfaultfd` descriptor for the target but can induce the target application to create a descriptor and pass it to Chameleon over a socket before the target begins normal execution. Chameleon uses `compel` [15], a library that facilitates implanting *parasites* into applications controlled via `ptrace`. Parasites are small binary blobs which execute code in the context of the target application. For Chameleon, the parasite opens a `userfaultfd` descriptor and passes it to Chameleon through a socket. To execute the parasite, Chameleon takes a snapshot of the target process' main thread (registers, signal masks). Then, it finds an executable region of memory in the target and writes the parasite code into the target. Because `ptrace` allows writing a thread's registers, Chameleon redirects the target thread's program counter to the parasite and begins execution. The parasite opens a control socket, initializes a `userfaultfd` descriptor, passes the descriptor to Chameleon, and exits at well-known location. Chameleon intercepts the thread at the exit point, restores the thread's registers and signal mask to their original values and restores the code clobbered by the parasite.

After receiving the `userfaultfd` descriptor, Chameleon must prepare the target's code region for attaching (`userfaultfd` descriptors can only attach to anonymous VMAs [44]). Chameleon executes an `mmap` system call inside the target to remap the code section as anonymous and then registers the code section with the `userfaultfd` descriptor. The controller then starts the fault handler thread, which serves code pages through the `userfaultfd` descriptor from the scrambler thread's code buffer as the target accesses unmapped pages.

Switching between randomization epochs. The event handler begins switching the target to the new set of randomized code when interrupted by the re-randomization alarm. The event handler interrupts the target, converts existing execution state (registers, stack memory) to the next randomization epoch, and drops existing code pages so the target can fetch fresh code pages on-demand.

The event handler issues a `ptrace` interrupt to grab control of the target. At this point Chameleon needs to transform the target's current stack to match the new stack layout. The compiler-emitted stackmaps only describe the complete stack layout at given points inside of a function, called *transformation points*. To switch to the next randomization epoch, Chameleon must advance the target to a transformation point. While the thread is interrupted, Chameleon uses `ptrace` to write trap instructions into the code at transformation points found during initial code disassembly and analysis. Chameleon then resumes the target thread and waits for it to reach the trap. When it executes the trap, the kernel interrupts the thread and Chameleon regains control. Chameleon then restores the original instructions and begins state transformation.

Chameleon unwinds the stack using stackmaps similarly to other re-randomization systems [50]. During unwinding, however, Chameleon shuffles stack objects to their new randomized locations using information generated by the scrambler

Chameleon uses the `int3` instruction

(Figure 3). For each function, the scrambler creates a mapping between the original and randomized offsets of all stack slots. This mapping is used to move stack data from its current randomized location to the next randomized location. To access the target's stack, Chameleon reads the target's register values using ptrace and the target's stack using /proc/<target pid>/mem. After reaching a transformation point, Chameleon reads the thread's entire stack into a buffer, located using the target's stack pointer. Chameleon passes the stack pointer, register set and buffer containing stack data to a stack transformation library to transform it to the new randomized layout.

The final step in re-randomization is to map the new code into the target's address space using userfaultfd. Chameleon executes an madvise system call with the MADV_DONTNEED flag for the code section in the context of the target, which instructs the kernel to drop all existing code pages and cause fresh page faults upon subsequent execution. The fault handler begins serving page faults from the code buffer for the new randomization epoch and the target is released to continue normal execution. At this point the target is now executing under a new set of randomized code. The event handler thread signals the scrambler thread to begin generating the next randomization epoch. In this way, switching randomization epochs blocks the target only to transform the target thread's stack and drop the existing code pages. The most expensive work of generating newly randomized code happens in parallel with the target application's execution, highlighting one of the major benefits of cleanly separating re-randomization into a separate process from the target application.

Multi-process applications. Chameleon supports multi-process applications such as web servers that fork children for handling requests. When the target forks a new child, the kernel informs Chameleon of a fork event. The new process inherits ptrace status from the parent, meaning the event handler also has tracing privileges for the new child. At this point, the controller instantiates a new scrambler, fault handler and event handler for the new child. Chameleon hands off tracer privileges from the parent to the child event handler thread so the new handler can control the new child. In order to do this, Chameleon first redirects the new child to a blocking read on a socket through code installed via parasite. The original event handler thread then detaches from the new child, allowing the new event handler thread to become the tracer for the new child while it is blocked. After attaching, the new event handler restores the new child to the fork location and removes the parasite. In this way, Chameleon always maintains complete control of applications even when they fork new processes.

3.3 A Prototype of Chameleon

Chameleon is implemented in 6092 lines of C++ code, which includes the event handler, scrambler and fault handler. Chameleon extends code from an open source stack transformation framework [5] to generate transformation metadata

This file allows tracers to seek to arbitrary addresses in the target's address space to read/write ranges of memory

and transform the stack at runtime. Chameleon uses DynamoRIO [7] disassemble and re-assemble the target's machine code. Currently Chameleon supports x86-64. Chameleon's use of `ptrace` prevents attaching other `ptrace`-based applications such as GDB. However, it is unlikely users will want to use both, as GDB is most useful during development and testing. However, Chameleon could be extended to dump randomization information when the target crashes to allow debugging core dumps in debuggers.

4 Evaluation

In this section we evaluate Chameleon's capabilities both in terms of security benefits and overheads:

- What kinds of security benefits does Chameleon provide? In particular, how much randomization does it inject into stack frame layouts? This includes describing a real-world case study of how Chameleon defeats a web server attack. (Section 4.1)
- How much overhead does Chameleon impose for these security guarantees, including how expensive are the individual components of Chameleon and how much overhead does it add to the total execution time? (Section 4.2)

Experimental Setup. Chameleon was evaluated on an x86-64 server containing an Intel Xeon 2620v4 with a clock speed of 2.1GHz (max boost clock of 3.0GHz). The Xeon 2620v4 contains 8 physical cores and has 2 hardware threads per core for a total of 16 hardware threads. The server contains 32GB of DDR4 RAM. Chameleon is run on Debian 8.11 "Jessie" using Linux kernel 4.9. Chameleon was configured to add a maximum padding of 1024 bytes between stack slots. Chameleon was evaluated using benchmarks from the SNU C version of the NPB benchmarks [4, 38] and SPEC CPU 2017 [42]. Benchmarks were compiled with all optimizations (`-O3`) using the previously described compiler, built on clang/LLVM v3.7.1. The single-threaded version of NPB was used.

4.1 Security Analysis

We first analyze both the quality of Chameleon runtime re-randomization in the target and describe the security of the Chameleon framework itself. Because Chameleon, like other approaches [17, 50, 45], relies on layout randomization to disrupt attackers, it cannot make any guarantees that attacks will not succeed. There is always the possibility that the attacker is lucky and guesses the exact randomization (both stack layout and randomized code) and is able to construct a payload to exploit the application and force it into a malicious execution. However, with sufficient randomization, the probability that such an attack will succeed is so low as to be practically impossible.

Target Randomization: Chameleon randomizes the target in two dimensions: *randomizing the layout of stack elements* and *rewriting the code* to match the

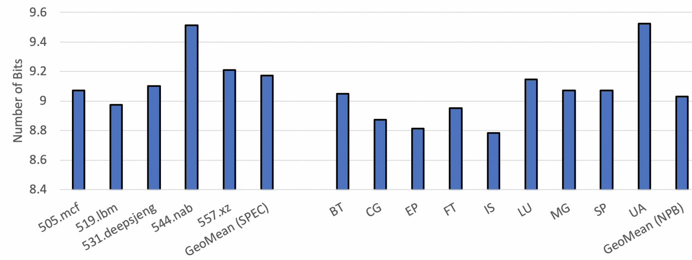

Fig. 4. Average number of bits of entropy for a stack element across all functions within each binary. Bits of entropy quantify in how many possible locations a stack element may be post-randomization – for example, 2 bits of entropy mean the stack element could be in $2^2 = 4$ possible locations with a $\frac{1}{4} = 25\%$ chance of guessing the location.

randomized layout. We first evaluated how well *randomizing the stack disrupts attacks that utilize known locations of stack elements*. When quantifying the randomization quality of a given system, many works use *entropy* or the number of randomizable states as a measure of randomness. For Chameleon, entropy refers to the number of potential locations a stack element could be placed, i.e., the number of randomizable locations.

Figure 4 shows the average entropy created by Chameleon for each benchmark. For each application, the y-axis indicates the geometric mean of the number of bits of entropy across all stack slots in all functions. Chameleon provides a geometric mean 9.17 bits of entropy for SPEC and 9.03 bits for NPB. Functions with more stack elements have higher entropy as there are a larger number of permutations. SPEC's benchmarks tend to have higher entropy because they have more stack slots. While an attacker may be able to guess the location of a single stack element with 9 bits of entropy (probability of $\frac{1}{2^9} = 0.00195$), the attacker must chain together knowledge of multiple stack locations to make a successful attack. For an attack that must corrupt three stack slots, the attacker has a $0.00195^3 = 7.45 * 10^{-9}$ probability of correctly guessing the stack locations, therefore making successful attacks probabilistically impossible. It is also important to note that the amount of entropy can be increased arbitrarily by increasing the amount of padding between stack slots, which necessarily creates much larger stacks. We conclude that Chameleon makes it infeasible for attackers to guess stack locations needed in exploits.

Next, we evaluated how Chameleon's code patching disturbs gadget chains. Attackers construct malicious executions by chaining together gadgets that perform a very basic and low-level operation. Gadgets, and therefore gadget chains, are very frail – slight disruptions to a gadget's behavior can disrupt the entire intended functionality of the chain. As part of the re-randomization process, Chameleon rewrites the application's code to match the randomized stack layout. A side effect of this is that gadgets may be disrupted – Chameleon may overwrite part or all of a gadget, changing its functionality and disrupting the gadget chain. To analyze how many gadgets are disrupted, we searched for gadgets in

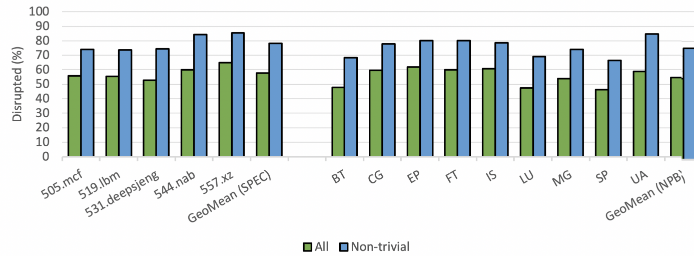

Fig. 5. Percent of gadgets disrupted by Chameleon's code randomization

the benchmark binaries and cross-referenced gadget addresses with instructions rewritten by Chameleon. We used Ropper, a gadget finder tool, to find all ROP gadgets (those that end in a return) and JOP gadgets (those that end in a call or jump) in the application binaries. We searched for gadgets of 6 instructions or less, as longer gadgets become increasingly hard to use due to unintended gadget side effects (e.g., clobbering registers).

Figure 5 shows the percent of gadgets disrupted as part of Chameleon's stack randomization process. When searching the binary, Ropper may return single-instruction gadgets that only perform control flow. We term these "trivial" gadgets and provide results with and without trivial gadgets. Chameleon disrupted a geometric mean of 55.81% gadgets or 76.32% of non-trivial gadgets. While Chameleon did not disrupt all gadgets, it disrupted enough that attackers will have a hard time chaining together functionality without having to use one of the gadgets altered by Chameleon. To better understand the attacker's dilemma, previous work by Cheng et al. [12] mentions that the shortest useful ROP attack produced by the Q ROP compiler [37] consisted of 13 gadgets. Assuming gadgets are chosen with a uniform random possibility from the set of all available gadgets, attackers would have a probability of $(1 - 0.5582)^{13} = 2.44 \times 10^{-5}$ of being able to construct an unaltered gadget chain, or a $(1 - 0.7632)^{13} = 7.36 \times 10^{-9}$ probability if using non-trivial gadgets. Therefore, probabilistically speaking it is very unlikely that the attacker will be able to construct gadget chains that have not been altered by Chameleon.

Defeating Real Attacks. To better understand how re-randomization can help protect target applications from attackers, we used Chameleon to disrupt a flaw found in a real application. Nginx [35] is a lightweight and fast open-source webserver used by a large number of popular websites. CVE-2013-2028 [16] is a vulnerability affecting nginx v1.3.9/1.4.0 in which a carefully crafted set of requests can lead to a stack buffer overflow. When parsing an HTTP request, the Nginx worker process allocates a page-sized buffer on the stack and calls `recv` to read the request body. By using a "chunked" transfer encoding and triggering a certain sequence of HTTP parse operations through specifically-sized messages, the attacker can underflow the size variable used in the `recv` operation

https://github.com/sashs/Ropper

on the stack buffer and allow the attacker to send an arbitrarily large payload. VNSecurity published a proof-of-concept attack [46] that uses this buffer overflow to build a ROP gadget chain that remaps a piece of memory with read, write and execute permissions. After creating a buffer for injecting code, the ROP chain copies instruction bytes from the payload to the buffer and "returns" to the payload by placing the address of the buffer as the final return address on the stack. The instructions in the buffer set up arguments and call the `system` syscall to spawn a shell on the server. The attacker can then remotely connect to the shell and gain privileged access to the machine.

Chameleon randomizes both the stack buffer and the return address targeted by this attack. There are four stack slots in the associated function, meaning the vulnerable stack buffer can be in one of four locations in the final ordering. Using a maximum slot padding size of 1024, Chameleon will insert anywhere between 0 and 1024 bytes of padding between slots. The slot has an alignment restriction of 16, meaning there are $\frac{1024}{16} = 64$ possible amounts of padding that can be added between the vulnerable buffer and the preceding stack slot. Therefore, Chameleon can place the buffer at $4 * 64 = 256$ possible locations within the frame for 8 bits of entropy. Thus, an attacker has a probability of $\frac{1}{2^8} = 0.0039$ of guessing the correct buffer location. Additionally, the attacker must guess the location of the return address, which could be at $4 * \frac{1024}{8} = 512$ possible locations to initiate the attack, meaning the attacker will have probability of $7.62 * 10^{-6}$ of correctly placing data to start the attack.

Attacking Chameleon. We also analyzed how secure Chameleon is itself from attackers. Chameleon is most vulnerable when setting up the target as Chameleon communicates with the parasite over Unix domain sockets. However, these sockets are short lived, only available to local processes (not over the network) and only pass control flags and the `userfaultfd` file descriptor – Chameleon can easily validate the correctness of these messages. After the initial application setup, Chameleon only interacts with the outside world through `ptrace` and `ioctl` (for `userfaultfd`). The only avenue that attackers could potentially use to hijack Chameleon would be through corrupting state in the target binary/application which is then subsequently read during one of the re-randomization periods. Although it is conceivable that attackers could corrupt memory in such a way as to trigger a flaw in Chameleon, it is unlikely that they would be able to gain enough control to perform useful functionality; the most likely outcomes of such an attack are null pointer exceptions caused by Chameleon following erroneous pointers when transforming the target's stack. Additionally, because Chameleon is a small codebase, it could potentially be instrumented with safeguards and even formally verified. This is a large benefit of Chameleon's strong isolation – it is much simpler to verify its correctness. Thus, we argue that Chameleon's system architecture is safe for enhancing the security of target applications.

4.2 Performance

We next evaluated the performance of target applications executing under Chameleon's control. As mentioned in Sections 3.2, Chameleon must perform a number

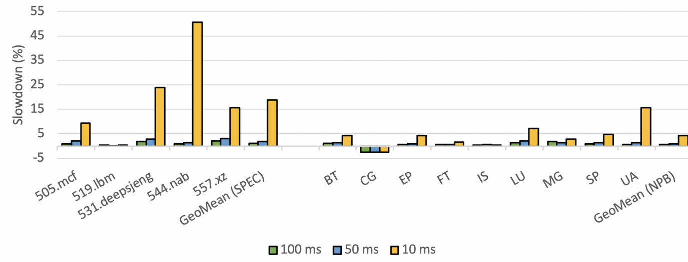

Fig. 6. Overhead when running applications under Chameleon versus execution without Chameleon. Overheads rise with smaller re-randomization periods, but are negligible in most cases.

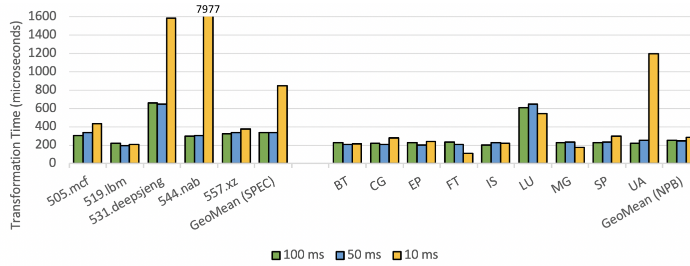

Fig. 7. Time to switch the target between randomization epochs, including advancing to a transformation point, transforming the stack and dropping existing code pages.

of duties to continuously re-randomize applications. In particular, Chameleon runs a scrambler thread to generate a new set of randomized code, runs a fault handler to respond to code page faults with the current set of randomized code, and periodically switches the target application between randomization epochs.

Figure 6 shows the slowdown of each benchmark when re-randomizing the application every 100ms, 50ms and 10ms versus execution without Chameleon. More frequent randomizations makes it harder for attackers to discover and exploit the current target application's layout at the cost of increased overhead. For SPEC, Chameleon re-randomizes target applications with a geometric mean 1.19% and 1.88% overhead with a 100ms and 50ms period, respectively. For NPB, the geometric means are 0.53% and 0.77%, respectively. Re-randomizing with a 10ms period raises the overhead to 18.8% for SPEC and 4.18% for NPB. This is due to the time it takes the scrambler thread to randomize all stack layouts and rewrite the code to match – with a 10ms period, the event handler thread must wait for the scrambler to finish generating the next randomization epoch before switching the target. With 100ms and 50ms periods, the scrambler's code randomization latency is completely hidden.

We also analyzed how long it took Chameleon to switch between randomization epochs as described in Section 3.2. Figure 7 shows the average switching cost for each benchmark. Switching between randomization epochs is an inex-

pensive process. For both 100ms and 50ms periods, it takes a geometric mean of 335μs for SPEC and 250μs for NPB to perform the entire procedure transformation. For these two re-randomization periods, only `deepsjeng` and `LU` take longer than 600μs. This is due to large on-stack variables (e.g., `LU` allocates a 400KB stack buffer) that must be copied between randomized locations. Nevertheless, as a percentage of the re-randomization period, transformations are inexpensive: 0.2% of the 100ms re-randomization period and 0.5% of the 50ms period. We also measured page fault overhead of 5.06μs per fault. While Chameleon causes page faults throughout the lifetime of the target to bring in new randomized pages, we measured that this usually added less than 0.1% overhead to applications.

There are several performance outliers for the 10ms transformation period. `deepsjeng`, `nab` and `UA`'s overheads increase drastically due to code randomization overhead. When the event handler thread receives a signal to start a re-randomization, it advances the target to a transformation point and blocks until the scrambler thread signals it has finished re-randomizing the code. Because these applications have higher code randomization costs, the event handler thread is blocked waiting for a significant amount of time.

We conclude that Chameleon is able to inject significant amounts of entropy in target applications while adding minimal overheads.

5 Related Works

Stack object-based attacks were proposed a long time ago but are regaining popularity due to the recent data oriented attacks and position-independent code reuse attacks [24–26, 23]. Traditional "stack smashing" attacks overflow the stack local buffer and modify the return address on the stack so that upon returning from the vulnerable function, the application jumps to the malicious payload [2]. There are a number of techniques proposed to prevent the return address from being corrupted, such as stack canaries and shadow stacks [13, 8, 49]. Stack canaries place a random value in between the function return address and the stack local buffer and re-checks the value before function returns. The program executes the warning code and terminates if the canary value is changed [13]. Shadow stacks further enforce backward control flow integrity by storing the function return values in a separate space [49, 8]. Both approaches focus on protecting direct control data on stack without protecting other stack objects.

Recent works have shown that stack objects other than function return addresses could also be used to generate exploits. Göktas et al. proposed using function return addresses and the initialized data that function calls left on the stack to construct position-independent ROP payloads. This way of legally using function calls to construct the malicious payload on the stack is named "stack massaging" [23]. Similarly, attackers can also manipulate other non-control data on the stack to fully control the target. Hu et al. proposed a general approach to automatically synthesize such data-oriented attacks, named data-oriented programming (DOP) [24, 25]. They used the fact that non-control data corruption could potentially be used to modify the program's control flow and implement

memory loads and stores. By using a gadget dispatcher (normally a loop and a selector), the attacker could keep the program executing data-oriented gadgets. Note that both of these attacks leverage non-control data on the stack, bypassing existing control flow integrity checks.

Strict boundary checking could be a solution to preventing memory exploits. Such boundary checking could be either software-based [33, 28, 39] or hardware-based [34, 19]. For example, Intel MPX introduces new bounds registers and an instruction set for boundary checking [34]. Besides the relatively large performance overhead introduced by strict boundary checks, the integrity-based approaches cannot defeat the stack object manipulation caused by temporal function calls [23]. StackArmor statically instruments the binary and randomly allocates discontinuous stack pages [10]. Although StackArmor can break the linear stack address space into discrete pages, the function call locality allows position-independent code reuse to succeed within a stack page size [23]. Timely code randomization breaks the constant locations used in the the program layout, making it hard for attackers to reuse existing code to chain gadgets [50, 11, 6]. However, these approaches transform the code layout but not the stack slot layout, giving attackers the ability to exploit stack objects. Chameleon is designed to disrupt these kinds of attacks by continuously randomizing both the stack layout and code. By changing the stack layout, Chameleon makes it more difficult for attackers to corrupt specific stack elements.

6 Conclusion

We have presented the design, implementation and evaluation of Chameleon, a practical system for continuous stack re-randomization. Chameleon continually generates randomized stack layouts for all functions in the application, rewriting each function's code to match. Chameleon periodically interrupts the target to rewrite its existing execution state to a new randomized stack layout and injects matching code. Chameleon controls target applications from a separate address space using the widely available `ptrace` and `userfaultfd` kernel primitives, maintaining strong isolation between Chameleon and the target. The evaluation showed that Chameleon's lightweight user-level page fault handling and code transformation significantly raises the bar for stack exploitation with minimal overhead to target application.

The source code of Chameleon is publicly available as part of the Popcorn Linux project at http://popcornlinux.org.

7 Acknowledgments

This work is supported in part by the US Office of Naval Research (ONR) under grants N00014-18-1-2022 and N00014-16-1-2711, and by NAVSEA/NEEC under grant N00174-16-C-0018.

References

1. Misiker Tadesse Aga and Todd Austin. Smokestack: thwarting dop attacks with runtime stack layout randomization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 26–36. IEEE, 2019.
2. One Aleph. Smashing the stack for fun and profit. *http://www.shmoo.com/phrack/Phrack49/p49-14*, 1996.
3. Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. *Proc. 23rd Usenix Security Sym*, pages 433–447, 2014.
4. David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165. IEEE, 1991.
5. Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*, volume 52, pages 645–659. ACM, 2017.
6. David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.
7. Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Sept 2004.
8. Nathan Burow, Xinping Zhang, and Mathias Payer. Shining light on shadow stacks. *arXiv preprint arXiv:1811.03165*, 2018.
9. Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.
10. Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*. Citeseer, 2015.
11. Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61. ACM, 2016.
12. Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach for Defending against ROP Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.
13. Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, , and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, August 1998.
14. Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.
15. CRIU. CRIU Compel. https://criu.org/Compel, Accessed: 2019-04-14.
16. CVE-2013-2028. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028, Accessed: 2019-04-14.

17. Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (just-in-time) Return-oriented Programming. *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)*, 2015.
18. Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Conference on Security*, SEC'14, 2014.
19. Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
20. Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014.
21. DWARF Standards Committee. *The DWARF Debugging Standard*, February 2017.
22. Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.
23. Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242. IEEE, 2018.
24. Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.
25. Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.
26. Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882. ACM, 2018.
27. kernel.org. Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt, Accessed: 2019-04-14.
28. Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, page 22. ACM, 2018.
29. Linux Kernel Address Space Layout Randomization. http://lwn.net/Articles/569635/, Accessed: 2019-04-14.
30. LLVM Compiler Infrastructure. Stack maps and patch points in LLVM. https://llvm.org/docs/StackMaps.html, Accessed: 2019-04-14.
31. Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *NDSS*, 2017.
32. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building

customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.
33. Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.
34. Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28, 2018.
35. Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.
36. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.
37. Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, pages 25–41, 2011.
38. Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE international symposium on workload characterization (IISWC)*, pages 137–148. IEEE, 2011.
39. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.
40. Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.
41. Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.
42. Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017, Accessed: 2019-04-14.
43. László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal War in Memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.
44. The Linux man-pages project. mmap(2) - Linux manual page. http://man7.org/linux/man-pages/man2/mmap.2.html, April 2020.
45. Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 727–741. ACM, 2016.
46. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028). https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html, Accessed: 2019-04-14.
47. Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Machiry Aravind, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.
48. Wikipedia. Ptrace. http://en.wikipedia.org/wiki/Ptrace, Accessed: 2019-04-14.
49. Wikipedia. Shadow stack. https://en.wikipedia.org/wiki/Shadow_stack, Accessed: 2019-04-14.

50. 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. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*, pages 367–382, 2016.