rave: A Framework for Code and Memory Randomization of Linux Containers

Christopher N. Blackburn

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Ruslan Nikolaev
Haining Wang

July 9, 2021
Blacksburg, Virginia

Keywords: Migration, Binary Rewriting, Memory Corruption, Code-Reuse, Randomization

rave: A Framework for Code and Memory Randomization of Linux Containers

Christopher N. Blackburn

(ABSTRACT)

Memory corruption continues to plague modern software systems, as it has for decades. With the emergence of code-reuse attacks which take advantage of these vulnerabilities like Return-Oriented Programming (ROP) or non-control data attacks like Data-Oriented programming (DOP), defenses against these are growing thin. These attacks, and more advanced variations of them, are becoming more difficult to detect and to mitigate. In this arms race, it is critical to not only develop mitigation techniques, but also ways we can effectively deploy those techniques. In this work, we present rave - a framework which takes common design features of defenses against memory corruption and code-reuse and puts them in a real-world setting. Rave consists of two components: librave, the library responsible for static binary analysis and instrumentation, and CRIU-rave, an extended version of the battle-tested process migration tool available for Linux. In our prototype of this framework, we have shown that these tools can be used to rewrite live applications, like NGINX, with enough randomization to disrupt memory corruption attacks.

This work is supported in part by ONR under grant N00014-18-1-2022 and NAVSEA/NEEC/NSWC Dahlgren under grant N00174-20-1-0009.

rave: A Framework for Code and Memory Randomization of Linux Containers

Christopher N. Blackburn

(GENERAL AUDIENCE ABSTRACT)

Memory corruption attacks continue to be a concrete threat against modern computer systems. Malicious actors can take advantage of related vulnerabilities to carry out more advance, hard-to-detect attacks which give them control of the target or leak critical information. Many works have been developed to defend against these sophisticated attacks and their triggers (memory corruption), but many struggle to be adopted into the real-world for reasons such as instability or difficulty in deployment. In this work, we introduce rave, a framework which seeks to address issues of stability and deployment by designing a way for defenders to coordinate and apply mitigation techniques in a real-world setting.

Dedication

This is dedicated to my parents.

Acknowledgments

I would like to thank my parents for always offering their unconditional support throughout all events of my life. They have always encouraged and enabled me to do the things I love to do. I could not have asked for better guides in life.

I would also like to thank my advisor, Dr. Ravindran. He has worked with and supported me throughout this venture. I could not have done any of this without his support and guidance. He provided me with so many opportunities to be successful and has worked closely with me as I progressed this work. I have learned a lot and developed myself as an engineer under his wing.

I also am grateful for Dr. Ruslan Nikolaev and Dr. Haining Wang as they offered their time and agreed to be on my examination committee, helping me achieve my goals.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

As we continue to dive head first into a digital age, growing portions of our lives are submitted to the many software services available to us. As the amount of information guarded by these services increases (and as more services emerge), the security of these resources becomes more and more prevalent. The attack surface available to malicious actors has become almost incomprehensible.

Of the many types of attacks used to hijack applications or meddle with information, memory corruption continues to plague computer systems as it has for decades. Burow *et al.* [13] claim that around 70% of bugs at Microsoft are memory corruptions.

One of the most fundamental techniques in memory corruption exploitation is the buffer overflow (or underflow). This technique takes advantage of logic errors (e.g. lack of bounds checking) to read/write data outside an intended scope. For example, in stack-based buffer overflows (dubbed stack smashing [40]), attackers can take advantage of stack-allocated variables to execute malicious code. With an in-properly bounded stack buffer, an attacker could overwrite memory on the stack, causing a vulnerable program to unknowingly load malicious code and jump to that code (by replacing a stack-saved return address with the address of that code). Because the data-plane and control-plane of applications are interleaved [13],

applications are often vulnerable to more advanced types of attacks which prey on memory corruption. Some attacks [14, 28], even today, are largely undetectable.

With the addition of defenses like Data Execution Prevention (DEP) [16], which prevents the execution of data regions, and Address Layout Randomization (ASLR) [22], which randomizes the base address of loaded memory regions, these types of code injection attacks have thinned out. However, even with these staples of security, memory corruptions attacks persist as a major threat: in the the CWE's 2020 report of the top 25 most dangerous software weaknesses, several entries involve memory corruption [23].

1.1.1 Code-Reuse Attacks

Since code injection techniques have become increasingly difficult to execute, exploiters have begun to leverage existing application code, which brings us to the *code-reuse* class of attacks (still rooted in and usually preceded by some type of memory corruption). In this class, an attacker can reuse fragments of code to alter the execution of a program or to corrupt/leak memory. One type of attack that has gained significant traction and advancement in recent years is Return-Oriented Programming (ROP) [43]. In ROP, attackers string together fragments of code called *gadgets*, which are snippets of instructions containing no control transfer instructions. These gadget chains can be linked together in arbitrary ways and has been shown to be Turing-complete. There are several other attacks that have branched from ROP's ideology, like PIROP which circumvents ASLR [25] or Blind ROP (BROP) [11], which allows the attacker to chain gadgets together without foreknowledge of the target binary.

There are many different ways to defend against these types of attacks. For example, with control-flow integrity (CFI) [6], we can detect anomalies in program execution. Upon detection, we can take some action, which could include the termination of the program as to

avoid leaking sensitive data. Another effective method of defense against both code-reuse and memory corruption has been some form of randomization [31]. The main idea of randomization is to provide toolchains or methods for diversifying memory or code. This makes it more difficult to chain together gadgets since useful gadgets may be broken, moved, or completely eliminated through randomization. Nevertheless, the more advanced extensions of ROP are still able to bypass these modes of security. Namely, non-control data attacks, such as Data-Oriented Programming (DOP) [28], are effective because they do not break control flow (i.e. they trigger no abnormalities in a programs normal execution). Additionally, DOP relies heavily on the predictability of the stack, which many works do not attempt to randomize.

1.1.2 Cloud Security

Works that thwart these more advanced variations of code-reuse and memory corruption attacks do exist [7, 35, 48]. However, some of these these defenses struggle to be as flexible as their attacker counterpart. That is, they exist largely in experimental settings or are difficult to deploy. Many vulnerable (and targeted applications) exist in a cloud environment and cannot afford to use these tools.

Many cloud providers use containers [38] to serve clients. Containers offer a more lightweight way to group processes and applications compared to more coarse-grained virtualization. At the same time, they provide much stronger isolation as opposed to process-process isolation. Because of these features, containers are much more flexible when it comes to application deployment. In many cases, it may be desirable to run several containers for a single application then load balance or deploy those containers dynamically through a cluster controller like Kubernetes [10]. However, because some prior arts can be difficult to deploy or inte-

grate, containers are left without defense. This thesis aims to parallel existing defenses in a more practical setting by providing code diversification to commonly used applications and containers.

1.2 Thesis Contributions

The contributions of this thesis are as follows:

- **Librave:** a library built to assist with the static analysis and rewriting of binaries. This library is used to generated new versions of the binary's text section. Specifically, it applies a randomization that permutes the stack slot locations of callee-preserved registers to defend against memory corruption attacks. In addition, it is able to rewrite a live stack space to match the randomized code layout.
- **CRIU-rave:** Linked with librave, this extended version of CRIU supports the randomization of processes upon process restore. This includes a randomized version of the binary and a re-written stack to match the new code layout.

Altogether, rave aims to be a framework through which code transformation techniques can be applied to vulnerable programs. Since too often previous works [5, 6, 18, 27, 48] cannot coordinate with each other, are unstable, or are not easily deployable, there is a need for this type of framework. For example, Chameleon [35] relies heavily on custom built tools which restrict what applications it can run. Other works incur too high an overhead to function in real-world settings while others simply don't defend against the most sophisticated types of code-reuse and memory corruption attacks [6, 18, 27]. With rave, we are able to overcome some of these issues by more closely studying the underlying issues in these attacks and by integrating with real-world applications like CRIU.

Through this study, we observe that most works have common design features which rave provides through several layers of abstraction. Rave takes advantage of this by providing tooling necessary to defend against attacks using similar techniques shown in previous works. It also leverages the hardened and rich CRIU environment to apply transformations to live processes and to improve overall stability. The current prototype of the rave framework is able to transform real-world applications like NGINX, while also introducing a high enough diversification to disrupt memory corruption attacks. We are able to quantify the quality of randomization showing that for applications like NGINX, on average, attackers will only be able to guess the locations of stack slots 19.1% of the time (assuming they only need to find one slot). The overhead incurred in CRIU-restore is hardly noticeable - even for large applications with over 40 thousand functions to analyze (MySQL), CRIU-rave analyzes and transforms them in under two seconds on the experimental machine.

1.2.1 Improving Container Security

Another benefit of integration with CRIU is that rave is able to introduce randomization (i.e. security benefits) to Linux container migration [42, 44]. Containers provide high levels of isolation for user programs, but do not inherently add any security benefits to said applications. The ability to migrate containers across machines is valuable in a cloud setting, yet there is a lack of mitigations for memory corruption and code-reuse attacks there. This further motivated the extension of CRIU.

1.3 Thesis Organization

Following this introduction, this thesis provides background information for the topic as well an overview of other related works in chapters 2 and 3. Next, the design and implementation of the primary contributions are described in chapter 4, which is followed by their evaluation in chapter 5. Finally, this thesis closes with concluding statements and future work in chapters 6 and 7.

Chapter 2

Background

This chapter provides an overview of relevant background information including insights on security and process migration. It also covers some related works which seek to address similar vulnerabilities rave was built to defend against.

2.1 Memory Corruption

By modifying the internal state of a program in its execution, we can hijack a program's control flow, disclose secret information in memory, or modify non-control data. This is called *memory corruption*. It is one of the leading exploits attackers use to gain control of a system [13, 23].

In order to understand the threat model, the subsequent subsections provide a foundation in understanding some application vulnerabilities and how we can defend against them and memory corruption attacks. As such, this will not be an exhaustive analysis of memory corruption techniques, and it will largely focus on stack-based vulnerabilities.

2.1.1 Buffer Overflows

Buffer overflows [40] are one of the most prevalent types of memory corruption techniques used in gaining control of a program. An overflow occurs when data is written outside of

a fixed-size buffer thus corrupting memory adjacent to the buffer in a program's memory space. These writes can happen directly by the attacker or indirectly through a number of function calls (like read, recv, etc.). In modern systems, the former is unlikely to occur because of process isolation, unless the attacker already has full control of the system. The same cannot be said about the latter. When receiving data from an outside source via some function, typically the size of the data is included with the incoming payload (if not, the size can be measured or restricted). If a program does not have proper bounds-checking logic and naively writes incoming data to some place in memory (i.e. a fixed-size buffer), it leaves itself in a rather vulnerable position. Without restricting memory reads and writes, attackers can send malicious payloads that overwrite critical data (which, in turn, allow them to hijack the control flow of a program).

While these memory vulnerabilities are less likely to manifest through using safe programming languages, like Rust [36], it's not always possible to (re)write applications in these newer languages. Much of the world runs on legacy code, and C continues to hold a lot of influence. Thus, we cannot just ignore buffer overflow vulnerabilities and discard them as the programmer's duty to guard (unfortunately, computers do exactly what we tell them to do).

While a buffer overflow can happen anywhere in memory, many attacks target a thread's execution stack. On the stack, functions store local variables, preserve register data, and return addresses which serve as a history of the program's control flow [37]. The layout of this memory will differ between functions and architectures, but there are several common features. For example, whenever a program is about to finish execution of some function, it will pop a return address from the stack then, using a special instruction (e.g. ret), set the program counter equal to that address, thus resuming program execution. Early exploits involve submitting a payload which includes malicious code in addition to rewriting the

return addresses stored on the stack. Once a function is ready to return to its caller, it will pop the overwritten address, setting the instruction pointer to another portion of the payload containing the malicious code.

Listing 2.1: Example of code vulnerable to buffer overflow

```
1   #include <stdio.h>
2   #include <string.h>
3
4   extern char *secret;
5   void foo(void) {
6       char buf[10], *c;
7       int access = 0, i = 0;
8
9       while((c = getchar()) != EOF) buf[i++] = c;
10      if (!strcmp(buf, secret)) access = 1;
11      if (access == 1) grant_access();
12  }
```

Listing 2.1.1 is a simple example of code vulnerable to buffer overflow. The `read` function call writes to a buffer allocated on the stack, but is not bounds checked. This means that someone could (accidentally or maliciously) overflow the buffer by sending more data to be read than the buffer can handle. Other variables allocated on the stack are subject to be overwritten since they are adjacent in memory to the overflowed buffer.

With Data Execution Prevention (DEP) [16], these types of code-injection attacks are no longer feasible. Since code on the stack (and other regions like the heap) is no longer executable, setting the program counter to point to memory on the stack will likely just cause the program to abort or perform some other type of control flow recovery.

Now, this doesn't mean that buffer overflows are no longer a means of memory corruption,

we just can't simply attach code with malicious payloads and expect them to execute. This led to a new class of attacks where attackers instead use existing code to perform malicious attacks. Instead of corrupting stack memory to jump to injected code, an attacker could redirect control flow to another executable part of the already-running program.

2.1.2 Code-Reuse Attacks

Return-into-libc [46] is one of the earliest manifestations of code-reuse attacks [11, 12, 24, 25, 45]. It uses buffer overflow vulnerabilities to overwrite the return address on the stack to point to some other function, often a libc function like system or mprotect. Along with the modified return address, the attacker would attach function arguments to spill onto the stack, thus causing the program to deviate from its intended control flow.

Naturally, these types of attacks circumvent DEP since they are only executing code in memory which is already marked executable. However, with Address Space Layout Randomization (ASLR) [22], this attack becomes less practical, as the location of libc is no longer predictable. In addition, this attack is tightly coupled to the ABI of the machine. In x86-64 systems, this Return-into-libc will not work directly since arguments are first stored in registers before being spilled onto the stack [37].

Now arrives Return Oriented Programming (ROP) [43] - a well-known attack involving code-reuse. ROP is a more general, modern version of the Return-to-libc attack. There exist many extensions and variations of ROP which provide more advanced and robust ways of hijacking the control flow of a program through memory corruption, but they all follow similar principles to ROP. In ROP, an attacker will search for *gadgets*: small fragments of existing code which share some characteristics. In traditional ROP, these gadgets are short instruction sequences ended by a control transfer instruction (e.g. ret, jmp, call, etc.).

These gadgets may be chained together through memory corruption to perform arbitrary computations (It has been shown that ROP is Turing-complete [43], which basically means ROP is able to perform any computation). By carefully chaining ROP gadgets together, an attacker could load registers used for function arguments (as defined by and ABI), then return from a gadget to a targeted function (including those in libc). These gadgets do not have to consist only of existing instructions in an executable. By feeding the instruction pointer with a unaligned address, an attacker could run an *unintended gadget* [41].

Recall that these attacks are still coordinated through memory corruption. The attacker must identify gadgets and carefully modify memory through techniques like buffer overflows such that the malicious gadget chain will be executed. Also, traditional ROP attacks rely on return instructions (hence the name), however, there exists other variations of ROP attacks that do not need return instructions to hijack a program's control flow such as Jump-Oriented Programming (JOP) [12].

2.1.3 Non-Control Data Attacks

A more recent class of code-reuse attacks have emerged, and are far more threatening than previous methods. These attacks are dangerous because, unlike ROP, they do not deviate from the program's intended control flow (i.e. they are harder to detect). These are called *non-control data* attacks (as opposed to control-data attacks like ROP) [14, 28]. Non-control data attacks do not alter a program's control data (function pointers and return addresses). Instead, attackers target the program's data plane to extract data or gain control of the program or system.

Naturally, these attacks are much more difficult to orchestrate. However, work by Dr. Hu *et al.* shows that non-control data attacks can be Turing-complete through Data-Oriented

Programming (DOP), and are a very real threat to computer systems [28]. In some cases, just one memory error can be used to form a chain of DOP gadgets, allowing the attacker to execute arbitrary operations without breaking the intended control flow of a program. These DOP gadgets are small fragments of code used to perform a number of operations (logical, arithmetic, memory accesses, and control transfer operations). DOP is particularly effective in loops where the loop condition is controlled by a stack-allocated variable. Once that condition is hijacked by memory corruption (i.e. stack buffer overflow), the attacker could potentially chain together an infinite number of DOP gadgets.

2.2 AMD64 System V Stack Layout

DOP attacks and other stack-related memory corruption techniques often take advantage of the stack's predictability. In order to understand how this happens, we need to understand how stack frames are organized. This organization differs between architectures and ABI's, but we will be looking at the AMD64 System V ABI [37].

Whenever a function is called (generally, with the exception of red zones [37]), memory is allocated onto a special region of memory called the *stack*. This region of memory grows downward (from a high address to a low address). As a function is called, it places a unique block of memory onto the runtime stack called a frame. Stack frames are used to preserve function-specific information, like shared register contents. In the x86-64 System V specifications, each stack frame will begin with a return address (an address pointing to the caller function). Following that, the contents of the frame base pointer is stored, which is then proceeded by callee-saved registers and function-local variables. The stack pointer will (in the absence of a red zone) always point to the top of the stack.

These registers (the stack pointer, frame base pointer, and callee-saved registers) are all

owned by currently executing function. With the frame base pointer, programs can easily trace back execution, and thus return to the caller function. The other registers (%rbx, %r12, %r13, %r14, %r15), must be preserved as the ABI states that these registers are free to use - if not saved onto the stack, their contents would be clobbered, causing parent functions' states to change unintentionally. If the frame pointer is omitted (an optimization used to reduce the number of instructions used to setup and navigate stack frames), then it is also considered a function-owned register and must be saved to the stack in callee-preservation code. However, in this work, we do not omit the frame pointer since it provides a safe and stable way to unwind the stack (further explained in section 2.3.1).

In our ABI, call instructions push the return address of the calling function onto the stack, then increment the stack pointer - this begins the allocation of a new stack frame. Two more instructions are used to preserve the old frame base pointer and update the register with the address of the new frame (again, the stack pointer is incremented). Finally, function-owned registers are pushed onto the stack for preservation. This makes up the *prologue* of the function.

Once the program is ready to return from a function, it enters the *epilogue*. It first pops all values off the stack back into the function-owned registers, restores the old frame pointer, then, by executing the return instruction, pops the return address off the stack and sets the instruction pointer equal to that value.

2.3 ELF Binaries

The Executable and Linkable Format [2] is the de-facto standard for binaries on Unix-like systems. These files house converted source code (generated by a compiler) along with metadata describing what the source code is used for, and how it is to be handled. There

are different types of ELF object files, each with different use cases: normal executable files, relocatable object files, core files, and shared libraries.

At the beginning of the file resides the ELF header, which provides information like what type of file it is (an executable or shared object for example), what architecture this code was compiled for, and metadata describing the rest of the file. It is followed by the program header table and/or the section header table (these may or may not be present depending on the file).

ELF files are made up of segments and sections. Segments describe how an executable is to be loaded into memory (e.g. alignment, memory access permissions, dynamic linking information, etc.). Segments can contain sections, which can contain information like code (typically labeled the .text section), relocation entries for static and/or dynamic linking, or debug information. Figure 2.1 provides a visual overview of what an ELF might look like.

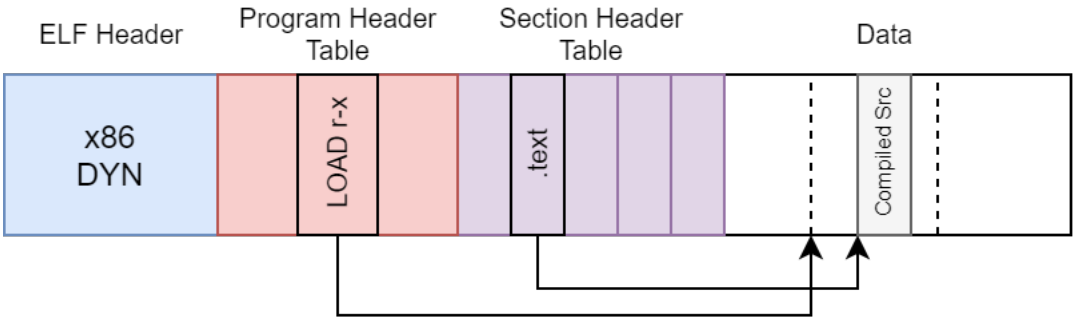

Figure 2.1: Overview of the Layout of an ELF Executable

In an executable (or dynamically linked executable, which are labeled as shared object files), there will be a loadable segment containing all (local) executable code. This segment will have read and execute permissions, and will likely request to be page-aligned. When an ELF is being prepared for execution, the contents of this segment (which will contain sections like the .text section), will be copied into a memory region marked with matching permissions. Any remaining memory, not initialized by the file, is zero-filled.

In our system, we replace that executable segment (containing the .text section among other executable sections) with randomized code.

2.3.1 DWARF Metadata

Binaries (not limited to ELF), are often accompanied with extra metadata. This metadata can give us all sorts of hints about the compiled code including function boundaries, debug information, and more. With this metadata, we can more safely and confidently perform analysis (and even binary rewriting). DWARF [15] is a popular debugging format specification which defines how metadata emitted by compilers should look. Its most obvious use is in software debugging, but with the information it provides, we can use it to augment binary analysis. It's also used to perform stack unwinding, as it emits metadata necessary to locate callee-saved registers preserved on the stack (this is especially important when the frame pointer is omitted).

If the frame pointer is used as a stack element, then we can use it to traverse up the call stack, frame by frame. This process of walking through stack frames is called *stack unwinding* [9]. In our work, we do not omit the frame pointer as a stack element since it helps us reliably unwind the stack during a rewriting phase. However, it is possible to omit the frame pointer during program compilation. In other words, it can be left to use as another function-owned, general-purpose register [37]. However, without the frame pointer, any generated code will now use offsets from the stack pointer (which is more difficult to track) to access and unwind the stack. DWARF emits a debug section (commonly labeled .eh_frame), which is used to gather information about how stack frames are organized. This is usually included as a section in the binary, and contains information for every instruction in the binary about how to find stack slots (specifically, callee-saved registers and the return address).

DWARF unwind information may sometimes be unreliable [9], especially as a program is processed through optimization passes during compilation. Keeping this information up-to-date is no simple task, and it is constantly improving. Nevertheless, to ensure stability, this work does *not* omit the frame pointer (i.e. we do *not* rely on DWARF unwind information to unwind/rewrite the stack).

2.4 Linux Process Migration with CRIU

Process migration [8, 39, 42] involves saving the state of a process (register data, opened files, memory layout, etc.) and moving it somewhere else (temporally or spatially). In cloud-based scenarios, the ability to move programs (or even virtual machines) from one machine to another (without shutting down that service) promotes high availability in dynamic load balancing. Other benefits of process migration include fault tolerance, flexible administration, and better data locality. An example scenario: new machines are getting installed or old machines are beginning to fail from age. Programs running on the machines are expected to maintain high levels of availability, and they cannot be allowed to shut down. So, through process migration, it is possible to relocate those processes to another machine with minimal downtime. In the case of works like H-Container [8], process migration is used to better serve clients by moving processes to edge nodes which are physically closer to the client (cutting down on latency).

Checkpoint/Restore In Userspace or CRIU [1] is a hardened checkpoint and restore utility built for Linux. When a user wants to migrate a process, they can invoke CRIU to dump the process' state into a set of image files, then, from those files, restore the process (either on the same machine, or on another machine).

At the beginning of the dumping process, CRIU attaches to a process and all its children

using ptrace [3]. To stay as true to the current state of the process, CRIU does not use ptrace to signal the process to stop. Instead, they use an in-kernel facility to freeze the process [1] before collecting and saving to disk information about the running process(es). Information about the process is mostly gathered from Linux's /proc filesystem [5].

When its time to restore a process, CRIU will analyze the dumped process image, then morph itself into the target to be restored. For every restoree, CRIU will fork itself then continue per-process restoration. Files are re-opened, memory is remapped and filled with dumped data, thread's executions are resumed, and the process gets restored. This can happen on the same or different machine, but there are some restrictions: the filesystems must match (or else things like open files cannot be restored). Any kernel features that exist in the source node, must also be available on the target node. Unlike virtual machines, process migration is not as flexible since they must still be fully supported by the OS that runs them.

CRIU also has additional methods for restoring a process. In some cases, like live migration, it may be undesirable to copy all the dumped process data to another machine before restoring that process (since this data may contain heap data, it could be very large). So, CRIU provides a way to lazily-load memory pages. Processes are restored like normal for the most part, but instead of reading and copying all dumped memory from the files into the restorees' memory, some pages are marked as lazy loadable and registered with a userfaultfd [4] file descriptor. Userfaultfd is a Linux kernel facility that allows users to handle page faults in user space. Basically, this allows us to register regions of memory with a file descriptor, then when a page fault happens (e.g. when some memory has not been loaded into memory), we are notified of it and are able to serve the fault.

This allows a second process, the lazy-pages process, to provide the restoree with data only when it needs it (both locally or over the network). In this work, we leverage lazy-pages

restoration to perform randomization outside of the target process' context, making the randomization tooling transparent to the target.

Chapter 3

Related Works

In this chapter, we explore several works which aim to mitigate memory corruption and code-reuse attacks discussed in section 2.1. By no means is this an exhaustive list, but by enumerating a few of these works and by studying their underlying concepts, we can come to understand methods of defense against aforementioned attacks.

3.1 Control-Flow Integrity

While Control-flow integrity may not prevent memory corruption, it is still useful in defending against some code-reuse attacks because it can verify that the program has not deviated from its intended control flow [6, 20, 26]. There are a number of ways we can use CFI to maintain the integrity of a program's execution: In general, CFI involves detecting anomalies in a program's control flow. Some implementations might monitor a process, making sure it follows a pre-generated control-flow graph (CFG) [26], while others may insert artifacts into the code which help verify the intended control-flow. Naturally, there are trade offs for different designs of CFI - some may incur too high an overhead as they monitor a process in its runtime; some may require complex offline processing (which in of itself may be imperfect, generating false positives).

3.1.1 Works Using Control-Flow Integrity

One reason why memory corruption vulnerabilities are so good at granting attackers control of program execution is because the data-plane and control-planes of applications are often interleaved [13]. For example, the stack stores return addresses, which, under memory disclosure, can allow an attacker to hijack control flow (these are called backward control-flow attacks). Because of this vulnerability, several works employ shadow stacks or stack canaries to maintain the integrity of program execution.

Stack canaries [13, 17, 18] are essentially tags that sit in stack memory along with saved return addresses in function frames. During runtime, these tags (often random values) are verified to make sure a continuous buffer overflow hasn't occurred. If this value is different than what the runtime expects, it's likely that some time of memory corruption has occurred, so the code can terminate before any damage is done. However, these are still possible to bypass if the attacker can acquire a pointer to the return address. For example, instead of directly overwriting the return address in a stack frame, an attacker could overwrite some other memory to point to the return address, effectively skipping over that tag.

Shadow stacks [13, 18], as opposed to stack canaries, do not inject the defense into the target space. Instead, shadow stacks are parallel data structures which allow applications to store function return addresses somewhere else in memory. When a function is prepared to return from execution, the return address saved on the stack is compared to a matching value in the shadow stack. If these values do not match, then the application has likely detected a memory corruption attack. By separating the control and data planes, shadow stacks enforce stronger isolation compared to stack canaries. Of course, these do not defend against non-control data attacks like DOP.

In CFI, there are two directions of control we need to protect. The aforementioned stack-

based CFI methods only defend backward control-flow, where as works like BBB-CFI [27] also operate on forward control-flow (e.g. function calls or use of function pointers outside of invoking a return instruction). BBB-CFI works on a basic block level (as opposed to function granularity). This creates a more fine-grained method of CFI. Basic blocks are streams of instructions separated by control transfer instructions (blocks have one pair of entry and exit points). Code-reuse attacks might often chain basic blocks together since many of them can be equivalent to gadgets found in an application. BBB-CFI enforces CFI by ensuring basic blocks are not accessed abnormally. That is, they should only be entered through their entry point and exited through the exit point (not in between). It does this without the need for a CFG or source code. Unfortunately, like many CFI works, this does not defend against non-control data attacks like DOP since the stack is not protected.

3.2 Randomization

Parallel to CFI, another way to defend against code-reuse (and memory corruption more directly in some cases), is to employ some form of randomization. The basic idea is to play hide-and-seek with the attacker. By shuffling locations of targets, or by reducing the predictability of vulnerable components in a program, we can significantly increase the difficulty of attack. Some of these works attempt to disrupt code-reuse attacks specifically, while others fight memory corruption directly. There are different types of randomization: some are more coarse-grained, like ASLR [22], do not defend well against more advanced types of code-reuse attacks. Others, more fine-grained like Chameleon [35], are more effective in defending against even DOP as the level of entropy introduced into the program is much higher. Randomization is defense through diversification [31].

3.2.1 Works Using Randomization

Address Space Layout Randomization [20, 22] is worth expanding upon because it is a staple in modern computer systems. There is no reason not to have this active. However, it does *not* sufficiently defend against certain types of code-reuse attacks. Simply put, ASLR loads various application components into random places in memory (e.g. the base address of libc's executable region will be random). However, in some attacks, these address can be leaked. There are even other variations of ROP that don't need leaked addresses like Position-Independent ROP (PIROP) [25] which leverages left-over data on the stack (i.e. it corrupts memory in stale function frames). Thus, this coarse-grained form of randomization does not introduce enough unpredictability to defend against more sophisticated code-reuse attacks.

Diving into more fine-grained randomization, Pappas *et al.* [41] attempt to defend against ROP attacks with in-place code randomization. This paper makes a few interesting contributions to application diversification. Firstly, any randomization transformations are done in-place. That is, the size of the code is not changed. Instructions are only appropriately shuffled and replaced with equally-size instructions. This is an important restriction - if new instructions are added or instructions removed, any further binary rewriting would have to find and fix all code references. That is, arbitrary modification to code size would require us to update an transfer control instruction targets (jumps between basic blocks, returns, etc.). This is not known to be statically solvable [47].

Despite operating within this restriction, they are still able to make an effective defense against ROP. By reordering instructions, performing register reassignment, and equivalent instruction substitution, they are able to break or eliminate ROP gadgets while maintaining program correctness. This is an important distinction which is eloquently introduced by

this work. In some cases, it may be possible to completely eliminate gadgets (for ROP, this means getting rid of unintended return instructions). However, most of the time, gadgets are simply broken. That is to say, the same ROP chain may not work between two version of the binary. Nevertheless, this work is susceptible to other types of code-reuse attacks, in particular, Blind ROP [11], in which ROP payloads are generated remotely (as opposed through static, offline analysis which requires a copy of the target binary).

Shuffler [48] is a type of fine-grained ASLR where functions in the binary are continuously relocated. By continuously reorganizing code locations, attackers will have a more difficult time trying to reuse gadget strings. Shuffler does not require any source code, making it fairly flexible - it only needs the binary with symbols still attached. It rewrites the binary to abstract code pointers, meaning, instructions are modified to (instead of calling directly into a function) reach out to a table in memory which contains a function address. In other words, function addresses are replaced with table indexes. This allows Shuffler to randomize the location of functions simply by changing entries in this indirection table. A separate thread continuously relocates these code pages, however, the stack remains untouched (the contents of each function remain constant). The stack, left unmodified, is still vulnerable to exploitation, thus making DOP a viable attack on applications protected by Shuffler.

Isomeron [19] features randomization designed to take down JIT-ROP code-reuse attacks. These differ from traditional ROP as they are designed to work against fine-grained ASLR. By leaking a runtime memory address (again, through memory disclosure techniques), JIT-ROP can craft gadget chains on the fly. Isomeron battles these types of code-reuse attacks by invoking fine-grained randomization along with continuous randomization. This work keeps multiple copies of diversified code in a process' address space, then decides which version to execute during runtime. This does not inherently prevent memory corruption, but it can disrupt gadget chains built for are particular execution path.

Smokestack [7] is an interesting work where stack frames are randomized as a part of the binary's runtime. Using a modified version of LLVM [32], Aga *et al.* instrument binaries such that several permutations of functions' stack allocations are available at runtime. The randomization instrumentation randomly chooses among these permutations when a function is called, thus introducing a different stack layout each time a function is called. They have even taken steps to protect the randomization instrumentation where return addresses are obfuscated to defend against control-flow attacks. The authors have shown that this method is effective in defending against DOP attacks, but it incurs a non-trivial runtime overhead in some cases (although, sometimes a performance hit is necessary to obtain better security while maintaining stability).

There is another, more conservative, method of randomization described by Kumar and Kisore [30]. In their work, the researchers develop a (theoretical) method of adding padding between stack slots, specifically, between function-local variables allocated on the stack. Their design, requiring the modification of GCC, inserts instrumentation into the binary which adds random amounts of padding between stack-allocated variables. This makes more difficult for an attacker to corrupt memory via buffer overflow. Another, similar work, developed by Liang *et al.*, performs a minimal rewriting of ARM binaries (running on Android). These researchers take advantage of ARM push and pop instructions but adding additional registers without modifying code size [33]. By pushing more registers onto the stack (and by shuffling existing registers), they induce the same padding effect, making it more difficult to trigger memory corruptions. Still, these, like other works, cannot guarantee that an attacker will not be able to locate a stack slot for memory corruption.

Chameleon [35] is one of the more aggressive code diversifying frameworks out there. Like Shuffler [48], randomization is transparent to the application. It tries to defend against memory corruption by reorganizing all stack slots in a function frame through binary rewrit-

ing. That is, any callee-saved registers, function-local variables, and even the return address saved to the stack gets shuffled and relocated. In addition to this shuffling, it has the ability to add random amounts of padding between stack slots, further increasing the entropy of the application. With this aggressive type of randomization, it is already more difficult to cause memory corruption through buffer overflow (and by extension defends against several code-reuse attack vectors). Chameleon takes this randomization a step further and continuously re-randomizes applications during runtime (both the code and the live stack are rewritten periodically). This way, even if the attacker is somehow able to guess the location of a stack slot through memory corruption (buffer overflow), that location will likely change before an effective attack can be executed. Additionally, the randomization happens in a separate process, providing a strong isolation between the vulnerable process and itself. The downside is that Chameleon is limited to running in a largely experimental setting - enabling this level of randomization required significant modifications to LLVM [32] and additions to a binary's runtime. One of the larger drawbacks is that it only works with statically compiled executables, which is not an option for some applications (since they may call functions like dlopen which invoke the dynamic loader).

Chapter 4

Design and Implementation

This chapter covers the design and implementation of the rave library as well as its integration into the checkpoint-restore tool for Linux called CRIU.

In chapter 3, we explored several existing techniques in attack mitigation. Often, these works target a single type of memory corruption or code-reuse attack to defend against, then fail to be effective against another variation. Since some works address the weaknesses in others, it would be interesting to combine several, non-interfering techniques to defend against multiple variations of memory corruption and code-reuse attacks. Altogether, rave seeks to be a more flexible, extensible framework to code transformations which can defend against aforementioned attacks.

Rave is split up into two components: a library and an extended version of CRIU. Both are written entirely in user space. In an effort to defend against memory corruption-based attacks, rave rewrites code and live stacks spaces to confuse attackers who are trying to take advantage of stack predictability. It leverages the rich environment of CRIU's process migration, which allows it to take advantage of several existing features while bolstering features like live migration through the additional security techniques it provides. Section 4.1 covers librave in more detail, while section 4.2 discusses the extended version of CRIU.

4.1 Librave

One draw back of previous works (discussed in chapter 3) is that the driver code is often either a part of the target binary or is tightly coupled to the randomization code. That is, it would be a non-trivial engineering task to disassociate randomization techniques in previous works from the applications that apply those techniques. In order to avoid this type of behavior, rave was built as a library so that no one program was tied to its capabilities. Librave is also coded in C to avoid adding unnecessary weight (e.g. the C++ standard library) to existing applications like CRIU, which is also written in C.

This library's goal is to provide the basic tools with which code transformations can be applied (somewhat following LLVM's [32] modular philosophy). This includes abstractions for reading and navigating binary files (in this case ELF files), abstractions for reading and using binary metadata like DWARF [15] debug information, binary rewriting (disassembly and reassembly), and methods for maintaining records of code transformation (so that live processes can be adjusted to match re-written code).

librave can be logically broken into two phases of execution: an analysis phase, and a transformation phase. These are discussed further in subsections 4.1.1 and 4.1.2 respectively.

4.1.1 Analysis Phase

The analysis phase of librave consists of any setup required to being transforming code. This includes setting up pages for serving transformed code, parsing metadata, and creating internal representations of transformable functions.

The first step in rave analysis is to prepare the binary. When given an executable ELF, librave parses the program headers and section headers to find the .text section (the section

containing the user's compiled code) as well as the segment containing the .text section. This segment is artificially loaded into librave's address space for further analysis and transformation. That is, a region of memory is prepared for transformation so that it can be readily served via page faults or written back to a randomized executable (which one is up to the discretion of the driver code). Note that in order to serve valid code pages, we cannot *just* read in the .text section since it may not be the only section included in an executable's executable segment. For drivers like CRIU (which use facilities like userfaultfd to serve code pages), we cannot omit other executable sections like .plt or initialization code. These often share pages with the .text section and must be served along with it.

Next, librave parses any metadata available to it through a metadata abstraction class. The prototype of librave written for this paper uses DWARF debug information as the backing structure for this metadata class. This class exposes an interface through which we can interact with information about the code we just mapped (regardless of the backing source of metadata). For example, there is a `foreach_function` method which we can used to iterate over all functions defined by the backing metadata. This function takes a callback as an argument through which we can interpret common function information, like function boundaries, and apply them to further analysis and transformation.

This leads us to the final step in analysis where we iterate through and process each function defined by the included DWARF metadata. Using the metadata and mapped code region, rave disassembles each function to discover and record information about randomizable functions. We use DynamoRIO [34] to perform disassembly - this is a dynamic binary instrumentation tool, but they have a standalone disassembler which is robust and easy to use.

What constitutes a randomizable function? In this prototype of librave, we focus on permuting callee-preservation code. That is, stack slots containing the contents of function-owned

registers. By permuting the locations of these stack slots, we can make it harder for attackers to guess where target slots are located (i.e. an increase in entropy can make memory corruption more difficult to execute). So, a randomizable function is one that features a prologue that pushes two or more callee-saved registers onto the stack. Several push instructions construct function prologues and are accompanied by one or more matching epilogues.

In our analysis (to find prologues and epilogues), we break down functions into logical instruction sets. To locate these sets, we record groups of sequential instructions that pass a particular test. For example, to find function prologues, the test expects there to be only push instructions (preceded by frame base pointer manipulation) for function-owned registers at the beginning of the function. Once we encounter an instruction that does not pass the test, we close off the instruction set. Subsequently, when searching for epilogues, we search for any instruction sets that pass the epilogue test: sets will only contain pop instructions for function-owned registers, and they will mirror the order of push instructions in the prologue.

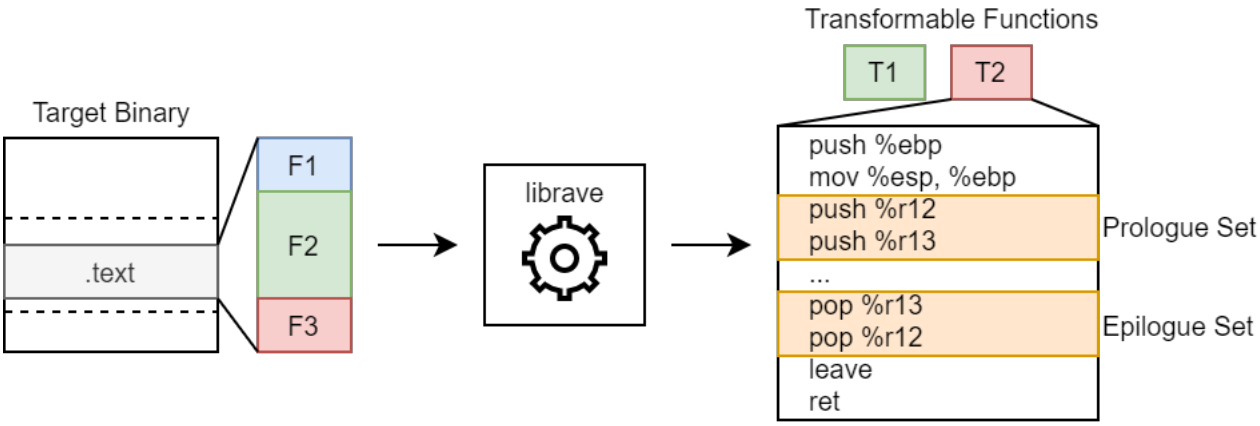

Figure 4.1: Rave Analysis Phase Visulaized. The binary is loaded into librave's address space and analyzed. Librave searches for transformable functions and records metadata about those functions (e.g. locations of prologues and epilogues).

For any randomizable functions found, we record them as a *transformable*, which is librave's

internal representation of a function that is... transformable. In this prototype, we record any relevant information about the function, including the locations of any prologues and epilogues along with an indexed mapping of stack slots (referring to the order in which callee-saved registers are saved onto the stack). These transformables are available for the transformation phase.

Librave also exposes a way to artificially relocate the randomized code. For dynamically linked executables, the executable segment may not be located at the address given in the ELF program header. Thus, for live programs, rave is designed to be able to interpret a new base address for these sections. This is a mirrored in stack rewriting, since the base address and offsets of the stack space could vary.

4.1.2 Transformation Phase

Once librave has finished analysis, the driver program can trigger a transformation. This transformation is applied to each function captured by the analysis phase, then re-encoded back into the locally loaded text segment (using the same disassembler used to decode instructions during analysis).

In the current prototype, rave transforms application code by permuting register preservation code. That is, the stack slots containing callee-saved register data and their respective push/pop instructions are permuted. The transformables generated from the analysis phase contain metadata relating the order of callee-saved registers on the stack. This information is stored in an array which we can shuffle to logically reorder instructions in the prologue and epilogues of these transformables. Once the new order is determined, we re-encode preservation instructions for each function according to the shuffled order.

Finally, once transformations have been completed, the driver code is responsible for taking

the modified code and serving it. This could mean saving it to a new, randomized binary, or serving code pages through page faults like CRIU-rave does.

Transformations also support stack rewriting since rave as a whole was designed to support live-process transformations in systems like CRIU in addition to offline transformations. Given a stack space, the current instruction pointer, and the frame base pointer, rave can unwind a live application stack (using the frame pointer, we can traverse through each stack frame). Each stack frame is matched to its respective function (previously recorded in the analysis phase) by either the instruction pointer or the return address saved on the stack. If the function is found to be randomizable, rave rewrites the current frame to match the code layout. In the current prototype, this means that callee-saved registers (found just after the location of the frame pointer) are re-organized to match the randomized order in the transformable function's metadata. The driver program is responsible for providing librave with the stack space and relevant information

4.2 CRIU-rave

Rave is a library, and thus can be driven by a third party. CRIU is one such party which enables process migration in Linux. CRIU-rave is a fork of CRIU built to link with and drive librave. Upon restore, CRIU is able to invoke librave to randomize a process by rewriting its code and stack. We chose to build this randomization framework on top of CRIU since it is a battle tested process migration tool, and it offers a rich set of features we can use to apply code transformations made by CRIU.

While rave could be built in directly to the restore process, integration was instead performed within the lazy-pages co-process (a feature already existing in CRIU). Building directly into CRIU restore would offer a simpler approach, but it would only allow for one-shot migration.

That is, the process could not be checkpointed again unless the randomization metadata was stored somewhere else. Since transformation metadata could end up being too complicated to serialize in future iterations of librave, it was decided that maintaining that information in the rave runtime was more efficient. So, when it comes time to checkpoint a randomized process, we can restore the original layout of the code and stack spaces before dumping it with CRIU. This has an added effect of not having to update any debug information (i.e. DWARF metadata and any other information remains accurate). Of course, this was not the only justification in using lazy-pages.

In addition to maintaining transformation metadata, there are a few more reasons for piggy-backing the lazy-pages co-process: by keeping the randomization instrumentation in a separate process, we have much stronger isolation. It is much easier to make claims about the safety of tooling when it runs outside the target application. Also, this creates opportunities for continuous re-randomization. Similar to works like Chameleon [35], we can perform code randomizations while the target application runs, only interrupting it occasionally update the code and stack. The current prototype does *not* feature continuous re-randomization.

Figure 4.2 features an overview of CRIU-rave architecture (which essentially covers the design of this thesis' prototype). The target process is restored separately from where librave is invoked to randomize its layout. The stack and code pages are served from the lazy-pages co-process on demand through page faults. So, by the time the restoree is ready to resume execution, the randomized code pages and stack are ready to be delivered once accessed. Let us now explore the finer details of implementation:

To serve code and stack pages from the lazy-pages daemon, CRIU uses Linux's `userfaultfd` facility. This interface allows us to handle page faults from user space. During CRIU's restore process, we can register memory regions in the restoree's address space with the uffd (the file descriptor associated with the `userfaultfd` facility). Note that this interface only works on

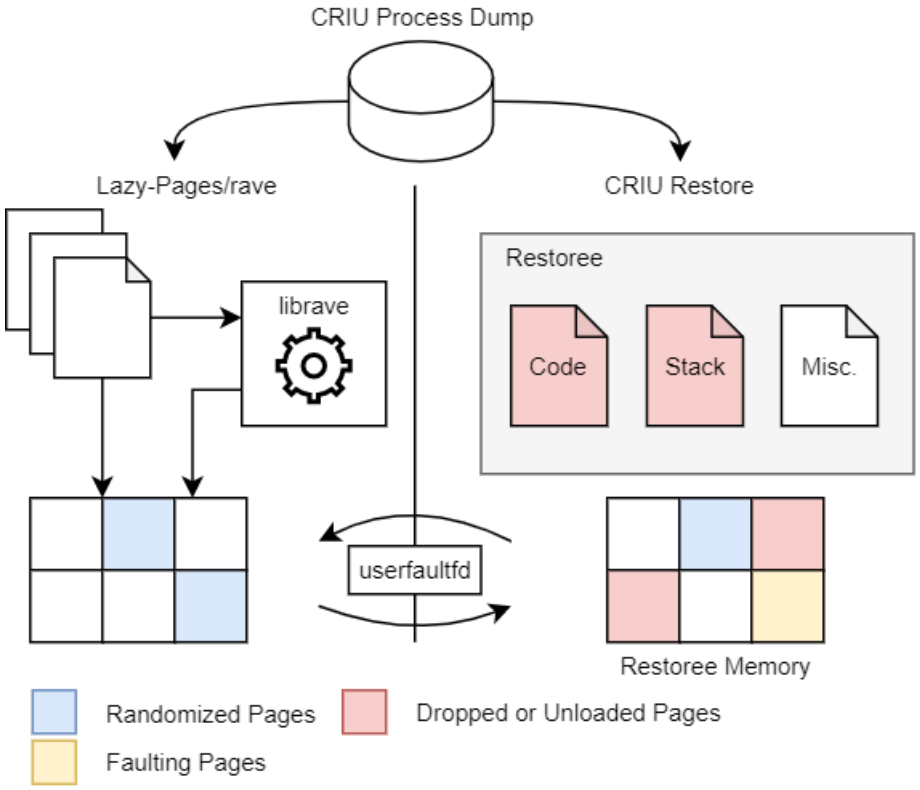

Figure 4.2: Overview of CRIU-rave Runtime. CRIU-rave runs the restore process and the lazy-pages process in parallel. Relevant pages are dropped or left unloaded by the restoree. Librave intercepts code and stack pages in the lazy-pages process to serve them out via `userfaultfd` whenever the restoree triggers a page fault by touching an unmapped page.

anonymous memory mappings. Trying to register a file backed mapping, like the executable region of the target binary, will fail. To work around this, we replace the file-backed mapping of the executable segment with an anonymous mapping (matching all original permissions of course). This is the only artifact in the target application that might suggest our tooling is active. Once this region is registered with `userfaultfd`, we make a call to `madvise` with the `DONT_NEED` flag, effectively telling the kernel that these pages can be dropped (thus triggering a page fault the next time they are accessed).

The stack region needs no special treatment as it is already marked for lazy loading. Once both the code and the stack regions are prepared in the restoree (and any other lazy-loadable

pages), CRIU sends the `userfaultfd` file descriptor to the lazy-pages process. Normally, under this facility, we could only serve pages from within the same process. However, by using a Unix socket to transfer the file descriptor to a listening process, we can continue to serve page faults in user space from outside the target process. Let us take a closer look at the lazy-pages process to see how it will handle these page faults:

CRIU lazy-pages will initialize itself in preparation to receive the uffd. It sets up a list of lazy-process structures which carry any relevant information necessary to serve page faults including structures which are prepared to read memory from the dumped process images. During this initialization, we can prowl the dumped images (on a per-process basis) to find the location of the executable file, as well as which region we should expect the code to reside in. At this time, we can also read the stack memory and register snapshot, which will be used to rewrite the stack.

The binary file is not saved in the dumped memory (it is re-opened on CRIU restore). So, we end up passing the location of this file to librave, triggering the analysis and transformation of the code. Once the code is transformed, we can send it the stack space we read from the dumped memory for rewriting. Once we have these components available, the co-process must wait for a page fault.

In unmodified CRIU, when a pagefault occurs, it will capture that event and serve memory directly from out the dumped memory images. In the rave-aware version of CRIU, we intercept this process and check to see if the page fault happened in a registered code or stack region. If this was the case, librave exposes the modified code or stack to CRIU so that it can serve the page fault, thus injecting the randomized memory into the target application.

Chapter 5

Evaluation

In this chapter, we evaluate both librave and CRIU-rave in terms of security and performance. Specifically, we will quantify the level of randomness introduced into the application. We also evaluate the time it takes to perform code analysis and transformations to get an idea of what types of overheads are induced through CRIU-rave's code modifications.

5.1 Experimental Environment

CRIU-rave was evaluated on an x86-64 machine with an Intel i7-6500U CPU clocked at 2.5 GHz. This core has two physical cores, two thread per core (4 total threads). This machine has 16 GB of DDR4 RAM. For the OS, it is running Ubuntu 20.04 LTS (kernel version 5.8). To compile and link benchmarks and other test programs, we used GCC version 10.3.0 and binutils version 2.34. CRIU-rave was tested on several programs including: SPEC CPU 2017, SNU C versions of NPB (single-threaded) benchmarks, NGINX, Redis, Lighttpd, and MySQL server. All programs were compiled with flags `-fno-omit-frame-pointer` and `-mno-red-zone` due to limitations in librave (see section 6.1.2). They are also dynamically linked (only the target application code is modified - external libraries and libc are not touched).

Figure 5.1 contains a basic overview of librave's analysis of all programs tested. This tells us how many functions exist in each binary, and how many are randomizable by the current

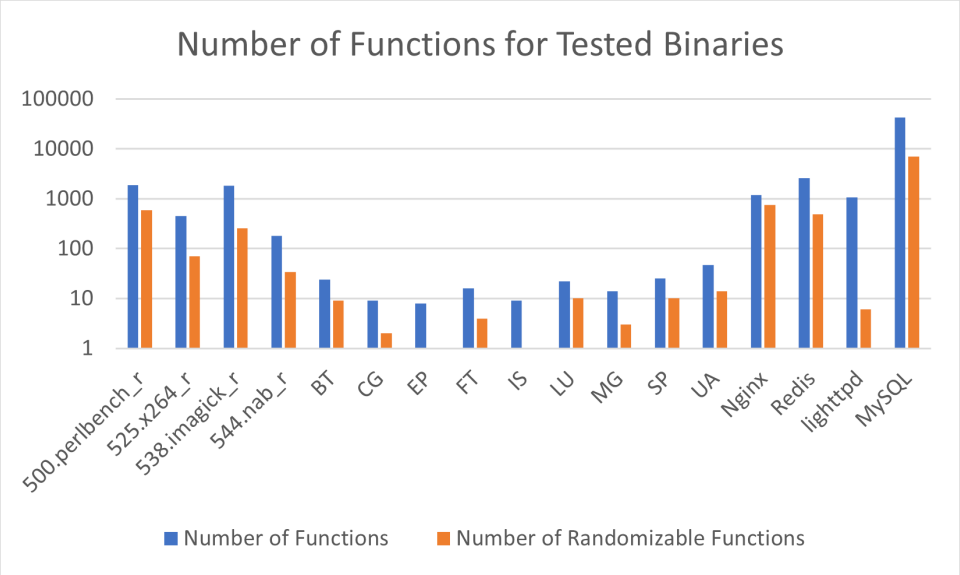

Figure 5.1: Number of Functions for Tested Binaries

prototype of librave. Naturally, as we will see in section 5.3, smaller applications (like NPB EP which has *no* randomizable functions), gain no security benefits through rave.

5.2 Performance Analysis

The primary point of focus for performance overhead lies in the time it takes to analyze and transform binaries. This overhead, for the current prototype of the rave framework, is incurred only once during process restoration (which in of itself is already littered with variability). Note that this overhead does *not* affect the runtime of the application because transformations happen out-of-band in the CRIU lazy-pages process.

Even for large applications like MySQL (which had 42470 functions to analyze, 7049 to transform), it takes less than two seconds from rave initialization of the unmodified binary to the complete transformation of the target. This is an acceptable performance hit since rave runs in a migration context where there is already much variability in the checkpoint/restore

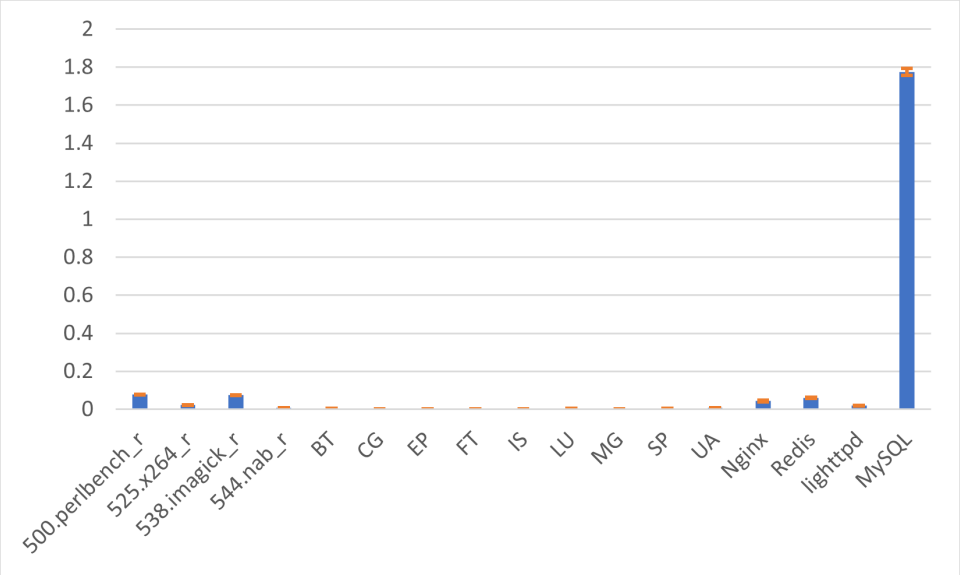

Figure 5.2: Average time taken to analyze and transform various binaries (seconds)

process. The geometric standard deviation for performance was about %7.74. Upon closer inspection, we see that this number (being a bit high) is skewed by the performance times of smaller applications (including ones like NPB's EP which has no randomizable functions). For these smaller applications, standard deviation is very high because the analysis and transformation runtimes are clouded by OS support (e.g. memory allocation). For larger applications (like MySQL), the standard deviation was only %1.05, which equates to about 1.8 ± 0.02 seconds.

5.3 Security Analysis

Librave cannot guarantee any attacks will not succeed, since it may be possible for an attacker guesses the location of a shuffled stack slot or accesses a stack slot not covered by rave transformations. In this work, rave's goal was primarily to show that transformations can be made within the migration context. Nevertheless, by shuffling stack slots, we can still

claim to disrupt memory corruption attacks since the predictability of the stack is partially broken.

We can quantify the quality of randomization by measuring the average *entropy* of an application. Functions that are randomizable will always have an entropy of two or higher. A function's entropy, in this case, is equal to the number of permutable stack slots e.g. if there are three stack slots, there are three possible locations a particular slot could be in, thus that function has an entropy of three. By collecting an average entropy across the entire application, we can get a general idea of how well librave transformations might protect against memory corruption attacks.

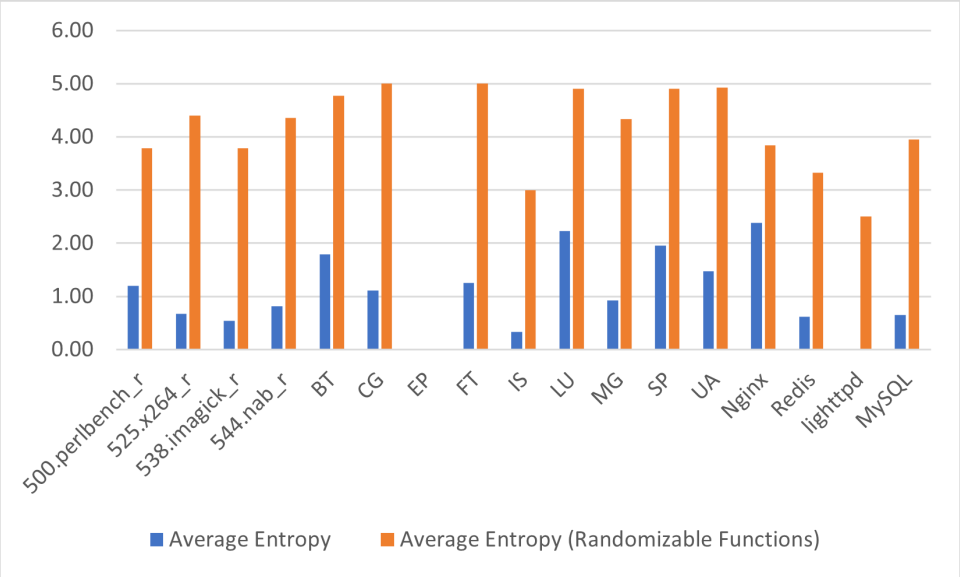

Figure 5.3: Quality of Randomization for Various Applications

Figure 5.3 shows the average entropy for an application assuming all functions are called with uniform probability. For particular workflows or attacks, the true entropy may vary and can be calculated given function call frequency. However, do to the nature of many code-reuse attacks, we can assume that any code in the application is vulnerable to memory corruption and/or code-reuse.

For most applications, the total average entropy is less than two, which implies that an attacker can generally guess where stack slots will be located regardless of randomization. The only application here that has a high enough entropy to qualify in disrupting memory corruption attacks is NGINX. This is because more than half the functions in NGINX are randomizable. With an average entropy of 2.39, an attacker will have an average probability of $\frac{1}{2^{2.39}} \approx 19.1\%$ in guessing the location of a stack slot. Do note that in some cases attackers will generally have to chain together multiple stack slots to execute an attack. In the case where three stack slots are required, there is an average probability of $19.2\%^3 \approx 0.7\%$ that the attacker will find all three slots.

Compared to other works, like Chameleon [35], librave does not introduce nearly as much randomization (Chameleon boasts over 9 bits of entropy for some applications). However, in these works, applications are statically linked, with self-compiled versions of libraries like libc. In that case, the code base is much larger, creating more opportunity for randomization. The drawback though is that certain applications, like NGINX, do not run under Chameleon's instrumentation (partially because NGINX requires access to the dynamic linker to call dlopen). In addition, we already knew that the types of transformations made in those works are far more aggressive compared to the ones made in rave (see chapter 2). This is discussed further in chapter 7.

Chapter 6

Conclusions

In this thesis we explored common security vulnerabilities relating to memory corruption and code-reuse attacks. We studied how these attacks are still prevalent today as they have evolved in the attack versus defense arms race of computer systems. Memory corruption still remains to be one of the leading program weaknesses which malicious actors are able to exploit. Attackers continue to formulate new and sophisticated ways of taking advantage of these weakness to gain control of the vulnerable program or steal information. More recent methods of attack, like Data-Oriented Programming (DOP), have proven troublesome as they are difficult to detect once they are active. Following this discussion, we surveyed several works which aim to fight against both code-reuse attacks and the root of these exploits, memory corruption. There were two classes of defense: control-flow integrity (CFI) and randomization. In CFI, applications are monitored or have checks to ensure they do not deviate from their intended execution paths, however they are still just as vulnerable to memory corruption. Randomization continues to prove successful against many variations of attacks, but is often met with instability or high runtime overhead. Modern computer systems would benefit from a combination of these works, yet many of them struggle to operate outside of an experimental setting or are difficult to deploy. Moreover, both CFI and randomization techniques are too tightly coupled to the implementation that drives them, making it difficult to use more than one technique at a time. To address this problem, we designed a framework with which programmers could develop and deploy defense instrumentation in a

real-world setting.

In summary, this thesis makes the following contributions: We developed librave, a library which assists in the static analysis and rewriting of binaries. The prototype of which is able to shuffle callee-saved registers in an attempt to disrupt attackers from corrupting those stack slots. For some applications, it is able to introduce non-trivial amounts of randomization which could disrupt attacks involving memory corruption. We built an extended version of CRIU which drives librave. We showed that it is possible to checkpoint a live process and resume it with a new code layout and matching stack space. Additionally, by piggy-backing on CRIU, we provide randomization tooling which is able to protect container migration by means of diversification.

In the next section, we discuss the limitations of librave and how it and other defenses might evolve to meet new, challenging methods of attack. Namely, we discuss how code patching techniques and coordination with custom binary compilation could enable us to perform more aggressive transformations, giving us an advantage in this game of memory corruption hide-and-seek.

6.1 Limitations

There are a few limitations that exist with the current prototype of rave (both the library and extended version of CRIU). This section highlights some of those issues and discusses previously unmentioned implementation details and potential solutions to these limitations.

6.1.1 Cross-Platform Support

Since rave was built with CRIU in mind, only ELF binaries are supported since the only platform CRIU runs on is Linux. However, librave principles could certainly be extended to support other platforms. Again, rave was built to be a foundation for flexible and extensible code transformation. It is not tied to CRIU in any way. If one so desired, they could write their own driver code which invokes librave for any platform.

Do note, however, that the transformations the current prototype of librave are performed on an assembly level. So, to support other architectures, transformations may look different [33].

6.1.2 Stack Unwinding

Working within CRIU (more generally, live processes) induced a limitation required for stable stack rewriting. While third party stack unwinders exist, they are often unreliable or unstable for the type of stack frame rewriting rave requires to be effective [9, 48]. Stack unwind information can also be unreliable when it comes to optimized binaries. Of course, this is a difficult problem to solve, but it is outside the scope of this work. So, to simplify the problem, binaries to be randomized with CRIU-rave are compiled with the frame pointer in tact (red zone is also omitted). This allows rave to more reliably unwind and rewrite a live stack when driven by CRIU.

There is another limitation in stack unwinding which manifests when working with dynamically linked executables. It is possible that CRIU will stop a process at an unknown location. That is, inside a function which is not tracked by librave. For example, if the process was trapped in the middle of a libc call, we would not have the necessary information required to being unwinding the stack (the frame pointer is likely omitted, and it is not reasonable to think we have the unwind metadata for this shared library). Of course, this extends to any

library, not just libc. Since it is not reasonable to require any applications intended for use with librave be statically linked (also requiring versions of libc and other libraries to retain the frame pointer), we take a different approach. To ensure we are able to unwind the stack, librave was extended to trap processes at well-defined locations.

We use `ptrace` [3] to attach to the target application before dumping it with CRIU. Librave then analyzes the target, locating all call sites. We again use `ptrace` to spray breakpoints (`int3` instructions in x86) so that the application will trap at a well-defined location where we can unwind the stack. Once the application is trapped, we can detach from the process and allow CRIU to dump it.

Under normal execution, most applications can have their stacks unwound by librave. However, there is still a corner case where a runtime stack could be interleaved with call frames from untracked libraries if callbacks are used. In which case, the stack transformation will fail.

For a more robust and flexible implementation, it may be necessary to locate any shared libraries loaded for the target application (all that information is stored by CRIU). In these cases, it may be possible for librave to also analyze those applications. At which point, it will have to rely more on DWARF frame data instead of the frame pointer for stack unwinding. However, in this approach, care must be taken to adjust relocation entries if necessary since all relocation entries will have already been resolved and saved by CRIU - updating this information may be difficult.

6.1.3 Towards Multi-Threaded Support

Multiple threads are not supported by the current prototype. It proves to be a non-trivial engineering problem, although not very complicated in design. As discussed in subsec-

tion 6.1.2, librave requires stacks to be stopped at well-defined locations. This must be true for all threads in an application. Since librave uses `int3` instructions to trap a target process at a known location, only the first thread to execute that instruction will be appropriately trapped since the trap signal will be sent to the whole process (not just the thread). One way to trap all threads at known locations is to trace each of them with `ptrace`, single stepping their execution until a valid call site is reached. After this, identifying stack regions in CRIU restore is fairly trivial, and they could just be passed to librave the same way the main thread's stack is passed.

Chapter 7

Future Work

This chapter covers related and future works with which rave can be extended. These include two methods of improving the quality of randomization librave offers (sections 7.1 and 7.2), and an additional work involving process migration.

As seen in chapter 5, librave is outperformed from a security standpoint compared to other works like Chameleon [35]. While beating these other works was not rave's only focus, this still raises the question about what makes those techniques more effective. Part of the reason is because rave operates in a very restricted mode. That is, it relies mostly on static assembly code analysis and binary re-writing. Because we are restricted to *not* alter the code size, or do not have control over external libraries (when working with dynamically linked executables), the types of transformations we can make are severely limited. In order to reach the same quality of randomization, transformation, and instrumentation other works present, there are a couple techniques we might employ to help.

7.1 Trampolining

One of the major restrictions in rave is not being able to change the code size. It is infeasible to recover control flow through just static binary rewriting [21, 47] (i.e. if we change code size, jump targets become mangled and very difficult to fix). This prevents us from making aggressive changes to applications that would otherwise allow us to more adequately randomize

the layout of memory on the stack. Other works suffer from this same restriction [34, 35, 41], and some offer some pretty crazy workarounds (which are sometimes impractical). There are, however, ways to circumvent this issue with code patching.

Duck *et al.* present a tool called e9patch [21] which enables you to insert arbitrary instrumentation into binaries without changing the set of jump targets. At the core of their design is a method called trampolining, where code for patches or other instrumentation can be arbitrarily sized and flexible. By inserting path points into existing functions (through several novel methods like instruction punning), they are able to jump from certain places to trampoline functions which execute the desired instrumentation and jump back to the original code. Rave could use similar methods to patch every function such that stack slot allocation and organization is fully controlled with these trampoline functions.

7.2 Compiler Assisted Randomization

Another way to expand on possible transformations is to modify binaries by changing the way they are compiled (LLVM [32] makes this a more friendly process). With pure rave, we are restricted to in-place code transformation or code-patching discussed in section 7.1. With compiler support, we could make much more aggressive changes to code while also eliminating restrictions such as these.

With the help of the compiler, you could tailor metadata designed to cooperate with some binary rewriting framework. One of the reasons why rave does not attempt to shuffle all stack slots in a function (as opposed to just callee-saved registers), is because there is no way to confidently identify them at arbitrary points of executions. LLVM can be extended to emit stackmaps [32] which contain information like this. With the help of the compiler, we could also generate new rules for stack frame allocation. You may even be able to link

metadata to code through the use of an indirection table - instead of having instructions directly access stack memory by using offsetting the stack or frame pointers, they could be assigned indexes specific to the running function. With these indexes, they could access the indirection table to figure out where relevant information is stored on the stack.

This coordination between the compiler and instrumentation is further discussed by Koo *et al.* [29]. However, this approach is not without burden. Some works, like Chameleon, struggle to work in a more practical environment because they rely heavily on custom compiler toolchains which add lots of complexity to an already complex process.

7.3 Cross-ISA Migration and Randomization

Another work related to CRIU-rave is H-Container [8] - another extension of CRIU which enables cross-architecture migration (e.g. a process or container can migrate from AMD64 to ARM architecture or vice versa). This work was motivated by the emergence of edge computing in recent years. Edge computing basically means that there might be servers at the edge of larger networks that are physically closer to clients. Naturally, clients will realize much lower latency when interacting with closer targets. So, H-Container provides a way to move services closer to clients regardless of system architecture.

This system would benefit from rave since it does not directly defend against memory corruption or code-reuse attacks. You could also imagine a scenario where an attacker might force a service to move to a node that is more susceptible to attack. Through rave's diversification built into migration, we could disrupt such malicious actors.

Bibliography

[1] Criu. URL https://criu.org/Main_Page.

[2] *elf(5) - Linux man page*, 2021. URL https://man7.org/linux/man-pages/man5/elf.5.html.

[3] *ptrace(2) - Linux man page*, 2021. URL https://man7.org/linux/man-pages/man2/ptrace.2.html.

[4] *userfaultfd(2) - Linux man page*, 2021. URL https://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[5] *proc(5) - Linux man page*, 2021. URL https://man7.org/linux/man-pages/man5/proc.5.html.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009. ISSN 1094-9224.

[7] M. T. Aga and T. 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. doi: 10.1109/CGO.2019.8661202.

[8] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: The case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 73–87, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375542. doi: 10.1145/3381052.3381321.

[9] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. Reliable and fast dwarf-based stack unwinding. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–24, 2019. ISSN 2475-1421.

[10] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014. ISSN 2325-6095.

[11] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. ISBN 2375-1207. doi: 10.1109/SP.2014.22.

[12] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40.

[13] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE. ISBN 153866660X.

[14] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5.

[15] DWARF Standards Committee. The dwarf debugging standard. URL http://dwarfstd.org/Home.php.

[16] corbet. x86 nx support. *Linux Weekly News*, 2004. URL https://lwn.net/Articles/87814/.

[17] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic

adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX.

[18] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566.

[19] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, .

[20] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, . ISBN 1931971153.

[21] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163.

[22] Jake Edge. Kernel addres space layout randomization. *Linux Weekly News*, 2013. URL https://lwn.net/Articles/569635/.

[23] Common Weakness Enumeration. Cwe top 25 most dangerous software weaknesses. Report, 2020. URL https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[24] Y. Guo, L. Chen, and G. Shi. Function-oriented programming: A new class of code reuse attack in c applications. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. doi: 10.1109/CNS.2018.8433189.

[25] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. 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 (EuroSP)*, pages 227–242, . doi: 10.1109/EuroSP.2018.00024.

[26] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, . ISBN 1479946869.

[27] Wenjian He, Sanjeev Das, Wei Zhang, and Yang Liu. Bbb-cfi: Lightweight cfi approach against code-reuse attacks using basic block information. *ACM Trans. Embed. Comput. Syst.*, 19(1):Article 7, 2020. ISSN 1539-9087. doi: 10.1145/3371151.

[28] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. ISBN 2375-1207. doi: 10.1109/SP.2016.62.

[29] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE. ISBN 1538643537.

[30] K. S. Kumar and N. R. Kisore. Protection against buffer overflow attacks through runtime memory layout randomization. In *2014 International Conference on Information Technology*, pages 184–189. doi: 10.1109/ICIT.2014.57.

[31] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. ISBN 2375-1207. doi: 10.1109/SP.2014.25.

[32] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. Thesis, 2002.

[33] Yu Liang, Xinjie Ma, Daoyuan Wu, Xiaoxiao Tang, Debin Gao, Guojun Peng, Chunfu Jia, and Huanguo Zhang. Stack layout randomization with minimal rewriting of android binaries. Information Security and Cryptology - ICISC 2015, pages 229–245. Springer International Publishing. ISBN 978-3-319-30840-1.

[34] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065034.

[35] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. Dynamic and secure memory transformation in userspace. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve Schneider, editors, *Computer Security – ESORICS 2020*, pages 237–256. Springer International Publishing. ISBN 978-3-030-58951-6.

[36] Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188.

[37] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99:57, 2013.

[38] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[39] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian

Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000. ISSN 0360-0300.

[40] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[41] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. ISBN 2375-1207. doi: 10.1109/SP.2012.41.

[42] Simon Pickartz, Niklas Eiling, Stefan Lankes, Lukas Razik, and Antonello Monti. Migrating linux containers using criu. High Performance Computing, pages 674–684. Springer International Publishing. ISBN 978-3-319-46079-6.

[43] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security Privacy*, 10(6):84–87, 2012. ISSN 1558-4046. doi: 10.1109/MSP.2012.152.

[44] Rami Rosen. Linux containers and the future cloud. *Linux J*, 240(4):86–95, 2014.

[45] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. ISBN 1081-6011. doi: 10.1109/SP.2013.45.

[46] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. Recent Advances in Intrusion Detection, pages 121–141. Springer Berlin Heidelberg. ISBN 978-3-642-23644-0.

[47] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*.

[48] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 367–382. ISBN 1931971331.