A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks

Edward J. Schwartz
Carnegie Mellon University
Software Engineering Institute
eschwartz@cert.org

Cory F. Cohen
Carnegie Mellon University
Software Engineering Institute
cfc@cert.org

Jeffrey S. Gennari
Carnegie Mellon University
Software Engineering Institute
jsg@cert.org

Stephanie M. Schwartz
Millersville University
Computer Science Department
stephanie.schwartz@millersville.edu

ABSTRACT

Code reuse attacks have been the subject of a substantial amount of research during the past decade. This research largely resulted from early work on Return-Oriented Programming (ROP), which showed that the then newly proposed Non-Executable Memory (NX) defense could be bypassed. More recently, the research community has been simultaneously investigating new defenses that are believed to thwart code reuse attacks, such as Control Flow Integrity (CFI), and *defense-aware* techniques for attacking these defenses, such as Data-Oriented Programming (DOP). Unfortunately, the feasibility of defense-aware attacks are very dependent on the behaviors of the attacked program, which makes it difficult for defenders to understand how much protection a defense such as CFI may provide. To better understand this, researchers have introduced *automated* defense-aware code reuse attack systems. Unfortunately, the handful of existing systems implement a single fixed, defense-specific strategy that is complex and cannot be used to consider other defenses.

In this paper, we propose a *generic* framework for automatically discovering *defense-aware* code reuse attacks in executables. Unlike existing work, which utilizes hard-coded strategies for specific defenses, our framework can produce attacks for multiple defenses by analyzing the runtime behavior of the defense. The high-level insight behind our framework is that code reuse attacks can be defined as a state reachability problem, and that defenses prevent some transitions between states. We implement our framework as a tool named Limbo, which employs an existing binary concolic executor to solve the reachability problem. We evaluate Limbo and show that it excels when there is little code available for reuse, making it complementary to existing techniques. We show that, in such scenarios, Limbo outperforms existing systems that automate ROP attacks, as well as systems that automate DOP attacks in the presence of fine-grained CFI, despite having no special knowledge about ROP or DOP attacks.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

code reuse attacks; software defenses

ACM Reference Format:
Edward J. Schwartz, Cory F. Cohen, Jeffrey S. Gennari, and Stephanie M. Schwartz. 2020. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3372297.3417234

1 INTRODUCTION

Over the past decade, there has been a flurry of research on techniques for constructing *code reuse attacks*, in which attackers hijack control of a program by leveraging code that was *intended to be there* as part of the original program. Before code reuse attacks, the preferred exploitation strategy was to inject new executable code into memory and execute it. However, this strategy was blocked by the Non-Executable memory (NX) [23] defense. Attackers eventually discovered that they could coerce the program into desired states and bypass NX by *reusing* code fragments that were already part of the program, and code reuse attacks were born. One of the best-known techniques for constructing such attacks is Return-Oriented Programming (ROP) [29], so named because attackers construct attacks by identifying code fragments called *gadgets* that perform useful functions and end in `ret` instructions.

Since the development of ROP attacks, defenders have developed newer and more sophisticated defenses explicitly designed to stop such attacks. Many of these new defenses are variants of *Control Flow Integrity* (CFI) [1, 2, 12, 22]. CFI employs a static analysis to over-approximate the intended targets of each indirect jump in a program. At runtime, CFI checks that each jump only transfers to one of these intended targets. If not, CFI terminates the program, since this is a sign that control flow has been hijacked. Since many code reuse attacks, including ROP attacks, rely on the ability to jump to an arbitrary location at an indirect jump, this greatly increases the difficulty of constructing attacks.

Although defenses such as CFI break many code reuse attacks, there are still avenues to attack them, and researchers have developed a variety of *defense-aware* strategies for targeting defenses.

For instance, early implementations of CFI purposefully favored performance over precision [35, 36], but attackers demonstrated that this deliberate imprecision was enough to permit so-called Call-Oriented Programming (COP) attacks [6, 10, 16]. Several years later, researchers showed that even with an ideal CFI implementation, attackers can still achieve Turing-complete control in at least some programs [17]. These attacks are called *Data-Oriented Programming* (DOP) attacks because they exploit the attacker's control over the program's data rather than its control flow.

Although these results are somewhat negative for defenders, one silver lining is that defense-aware attacks are more reliant upon the behaviors of the original program, and this means that the feasibility of such attacks varies on a per-program basis. Unfortunately, because these attacks have historically been identified and crafted by hand, it can be difficult for a defender to determine if a specific program would be protected by CFI or a similar defense. To address this problem, researchers have proposed automated systems that find code reuse attacks [18, 19, 24, 28, 34].

Unfortunately, the vast majority of automated code reuse attack systems are not *defense-aware*, which means that they will completely fail in the presence of defenses such as CFI. A handful of these systems are aware of CFI [18, 34], but these systems are complex and only target a single, hard-coded CFI variant. This makes them unsuitable for defenders who are trying to select a defense that provides adequate protection for their software.

In this paper, we propose a *generic* framework for automatically discovering *defense-aware* code reuse attacks in executables. Unlike existing work, our framework can produce attacks for defenses by analyzing their runtime behavior. As a result, our framework does not require hard-coded strategies or models of defenses, and can even be used with unknown defenses for which a defense-aware attack strategy has not yet been proposed.

The high-level idea underlying our framework is that code reuse attacks can be reduced to a state reachability (e.g., software model checking) problem, and defenses naturally restrict the state transitions that an attacker can employ. For example, when CFI detects a jump to an unintended target, it halts the program, which obviously prevents further state transitions. This is vastly different than prior work on automating code reuse attacks, which has always focused on automating a specific defense-aware technique for constructing code reuse attacks (e.g., DOP) that was manually discovered through human insight [18, 34].

Because our framework reasons about code reuse attacks in a *generic* way, it can naturally find attacks corresponding to currently known techniques including ROP [29], JOP [4, 8], COP [6, 10, 16], and DOP [17]. Although our framework can generate these attacks, it is not *limited* to them. Our approach can (and does!) discover attacks that do not correspond to known techniques for code reuse attacks. Because of this, our framework can often find attacks when other, less flexible techniques fail. For example, in our evaluation, we show that when considering CFI, our technique outperforms a system that was explicitly designed to find DOP attacks [18], even though our technique has no special knowledge about CFI. We also show that, in the absence of defenses other than the standard ASLR and NX defenses (see Section 2), our technique can find code reuse attacks using extremely small amounts of code (less than 10 KiB), which to the best of our knowledge is outside the ability of any existing technique.

We demonstrate our techniques by implementing them in a tool called Limbo. We designed Limbo to leverage existing work on binary concolic execution [7]. Our framework defines code reuse attacks as a state reachability problem, and Limbo uses concolic execution to solve this problem by exploring the target program's state space. Unlike existing automated approaches for constructing code reuse attacks [18, 28, 34], which are quite complicated, our proposed approach is conceptually *simple* and involves relatively few modifications to an existing concolic executor. Specifically, we modified a concolic executor to (1) guide execution using a heuristic-based search strategy, and (2) detect when a code reuse attack has been found.

We evaluate Limbo against existing techniques and show that it complements their strengths. Specifically, Limbo excels when there is little code available to reuse, which challenges existing techniques because there is not enough code to match their built-in strategies. We first evaluate Limbo's ability to automatically find CFI-aware attacks for 10 network and system utilities such as `apache` and `opensshd`. We show that Limbo outperforms BOPC [18], the state-of-the-art system for automatically discovering CFI-aware DOP attacks, even though Limbo contains no special logic for reasoning about CFI or DOP attacks. Second, we test how well Limbo performs in the absence of defenses when run on very small programs. We specifically evaluate Limbo on a random sample of twenty small utility programs in Ubuntu. We show that Limbo outperforms angrop, a popular ROP automation tool that is part of the angr binary analysis framework [3, 31], and that Limbo is even able to produce attacks when given access to less than 10 KiB of code.

Overall, we make the following contributions:

- We propose a *generic* framework for automatically identifying *defense-aware* code reuse attacks. Unlike the current state of the art [18, 34], which is to implement a hard-coded strategy that is specific to a single type of defense, our framework can identify attacks for multiple defenses by considering their runtime behaviors.
- We implement our framework in a tool called Limbo. We show how existing binary concolic executors can be leveraged to implement our framework using a small set of modifications.
- Finally, we show that Limbo outperforms both angrop [31] and BOPC [18], state-of-the-art tools for automatically constructing ROP and DOP attacks respectively, when there is little code available for reuse.

2 BACKGROUND

In this section, we briefly review the background and history of code reuse attacks. We also introduce some of the notable defenses designed to protect against code reuse attacks, and the defense-aware strategies that have been proposed to attack them.

2.1 Non-Executable Memory

Early control flow attacks consisted of two steps. The attacker would first inject *shellcode*, executable code that performs an attacker's desired computation such as spawning a shell, into the target process's memory space. She would then use a vulnerability to transfer control flow to her injected code. At this point, she could execute arbitrary instructions in the context of the vulnerable program.

In response, researchers developed the Non-Executable memory (NX) defense [23], which is also called DEP and W⊕X, to prevent such attacks. NX is very widely deployed: if you run a program on a conventional PC, tablet, or mobile phone, it is very likely that your program will be protected by NX. At a high level, NX prevents attackers from executing code that they have injected into the program's memory space. It does this by ensuring that certain memory regions, such as the stack, are not executable. On most platforms, this is now implemented efficiently using hardware support.

2.2 Code Reuse Attacks and ROP

With the introduction of NX, attackers realized that while they could not introduce *new* code into a process, they could still *reuse* code that was already there for other purposes, and such attacks are called *code reuse attacks*. One of the earliest types of code reuse attacks was *return-to-libc* attacks, which allowed attackers to reuse entire functions from libc by calling them. For example, by returning to `system("/bin/sh")`, the attacker could spawn a shell and bypass NX. Although powerful, the downside to return-to-libc attacks is that the attacker is limited to functions available in the vulnerable program's address space.

Return-Oriented Programming (ROP) [29] is a generalization of return-to-libc attacks. In a ROP attack, the attacker searches memory for *gadgets*, or instruction sequences ending with `ret`, and chains the gadgets together to implement the computation she wants. ROP attacks are desirable because they allow the attacker to perform computations beyond the functions available in the vulnerable program's address space. The earliest work on ROP used gadgets from large shared libraries such as libc, and focused on Turing-completeness.

2.3 Modern Code Reuse Attacks

Two external forces have caused attackers to develop new techniques for producing code reuse attacks. The first of these is the introduction of Address Space Layout Randomization (ASLR), which forced attackers to build strategies for producing code reuse attacks using less code. The second force consists of defenses that were designed to protect against code reuse attacks. In response, attackers developed new defense-aware techniques to attack them.

2.3.1 ASLR and Smaller Code Reuse Opportunities. ASLR [30] is another widely deployed defense that was proposed in response to traditional control flow attacks. Unlike NX, which stops the attacker from executing shellcode, ASLR stops the attacker from directly referring to objects in memory by randomizing their locations. Although the original goal of ASLR was to make it challenging for the attacker to reference their shellcode in memory, it also broke *return-to-libc* and early ROP attacks as a side effect. Because the location of libc was now randomized, it made it difficult for an attacker to reference ROP gadgets in libc since she would not know their addresses.

Because of compatibility and performance issues, early ASLR implementations had serious limitations and most programs would have at least some executable code that was *not* randomized [25, 28], although it was often smaller than libc by orders of magnitude. In practice, attackers began to build code reuse attacks from these smaller code bases. However, unlike in traditional academic ROP research, attackers did not focus on Turing-completeness. Instead, they used ROP as a stepping stone to disable NX by calling functions such as `mprotect` on Linux or `VirtualProtect` on Windows.

More recently, modern systems have been slowly evolving to fully randomize all executable code addresses by default. For example, to enable full randomization, Ubuntu Linux began compiling programs as Position Independent Executables (PIEs) on all architectures by default, starting with Ubuntu 17.10 [32]. It is important to note, however, that some programs are still not compiled as PIEs. One of the authors noted that gcc, emacs, Mendeley, docker, and python were not PIEs on his workstation, for example. When attacking fully randomized programs, attackers generally have two options. If the attacker wants an attack that will succeed deterministically (i.e., regardless of the memory layout), she can try to discover an *information leak* vulnerability that will reveal the address of a code module in memory. Once she knows this address, she can customize her attack for that particular memory layout. If the attacker does not mind if her attack only succeeds non-deterministically, she can simply enumerate the possible memory layouts and attempt the attack for each one [30]. In this paper, we assume that the attacker knows the address of the program image, either from an information leak or brute forcing. This is a much weaker assumption than other work in this area, which assumes the ability to read and write at arbitrary locations [18, 33].

2.3.2 Code Reuse Defenses and Attacks. The second evolution of code reuse attacks has been in response to defenses designed to stop code reuse attacks. Since the development of ROP, there have been several cycles of code reuse attacks inspired by advances in defenses, and vice versa.

Some of the earliest defenses focused exclusively on `ret` instructions [9, 11, 20], which seemed promising until attackers proposed a new technique that did not need to use `ret` instructions. The idea behind this new technique, which is called Jump Oriented Programming (JOP) [4, 8], is to find gadgets that end in instruction sequences besides `ret` but are semantically equivalent, such as `pop %eax; jmp *%eax` [8]. A more sophisticated approach is to identify a *dispatcher gadget* that replaces the behavior of the `ret` instruction [4]. In either case, `ret` instructions are no longer needed, which allows the above defenses to be bypassed.

Many of the other techniques for code reuse attacks are in response to Control Flow Integrity (CFI) [1] defenses. The idea behind CFI is to detect when a program's execution path deviates from a statically computed Control Flow Graph (CFG) of the program. When a program takes an execution path that is *not* allowed by the program's CFG, CFI terminates the program. CFI implementations are very broadly categorized as either *coarse-grained* [35, 36] or *fine-grained* [12, 22] CFI. As its name implies, coarse-grained CFI is less precise, and thus is more likely to allow a control flow transition that is not possible in the original program's CFG.

In response to these defenses, attackers developed a variety of defense-aware strategies that target both coarse-grained [6, 10, 16] and fine-grained [5, 13, 17, 26] CFI. These techniques include Call-Oriented Programming (COP) [6, 10, 16], Data-Oriented Programming (DOP) [17], and Counterfeit Object-Oriented Programming (COOP) [26]. At a very high level, all of these are techniques for crafting code reuse attacks that are aware of different types of CFI defenses. The existing systems that automate defense-aware code reuse attacks automate one of these existing strategies [18, 34].

3 IMPLEMENTATION

A *code reuse attack* can be defined as a program execution from a vulnerability to an attacker's desired goal state. The vulnerability and the goal state in this definition are usually known. Thus, the primary challenge is determining whether such an execution exists, and if so, how to trigger it. Fortunately, this type of problem has another name: it is a software model checking (SMC) problem. A typical SMC problem is to determine whether an error state (such as an assertion failure) can be reached from the program entry point. Analogously, a code reuse problem is to determine whether a goal state can be reached from a vulnerability.

In this section, we demonstrate our proposed techniques in Limbo, a practical system for automatically discovering defense-aware code reuse attacks. At a high level, Limbo reduces the problem of finding a code reuse attack to a SMC problem, which it solves using executable-level concolic execution (Section 3.1). As we describe in the remainder of this section, concolic execution provides a balance between soundness, completeness, and scalability that is well suited for automating the search for code reuse attacks. Concolic execution allows Limbo to take a state and enumerate some of the states that are reachable from it by executing the program symbolically. Limbo uses this ability to iteratively compute a set of states that are reachable from the vulnerability. As soon as this set contains a goal state, Limbo outputs the discovered attack as an executable test case. The test case contains inputs that, when given to the program, will cause it to reach a goal state.

3.1 Background: Concolic Execution

Concolic execution [7, 14, 15] is a combination of $\underline{\text{conc}}$rete and symb$\underline{\text{olic}}$ execution [27] that can reason about executable code very precisely. As the name suggests, concolic execution starts with a concrete (e.g., an ordinary) execution. A symbolic execution then executes the program symbolically along the same program path as the concrete execution.

Symbolic execution is similar to concrete execution, except that instead of mapping registers and memory locations to concrete values such as 42, symbolic execution maps them to *symbolic expressions*, which express values in terms of the symbolic input to the program. Just like algebraic expressions, symbolic expressions can represent multiple program values depending on the values of the input variables. For example, the symbolic expression $s4 + 8$ intuitively represents eight plus the 4th byte of the symbolic input to the program.

The other major difference is that symbolic execution maintains a path predicate Π that represents the constraints required to follow the execution path that has been symbolically executed so far. Π starts as true, and each time a conditional branch is executed, that condition is conjoined to Π. For example, if a branch is conditionally taken when `%eax` > 42, and `%eax` currently has the symbolic value s_{eax}, Π would be updated to $\Pi \wedge s_{eax} > 42$.

Concolic execution uses the value of Π to create new test cases. For each conditional branch taken in the followed concrete execution, concolic execution attempts to flip the branch and produce a new test case. For example, before encountering the above conditional branch, concolic execution would use an SMT solver to check if $\Pi \wedge \neg(s_{eax} > 42)$ is satisfiable. If it is, the resulting model yields a test case (i.e., program input) that will cause the program to execute the branch *not* taken in the concrete execution. If the formula is not satisfiable, this means that the path is *infeasible* and cannot be taken.

3.2 Why Concolic Execution?

Concolic execution has been employed for a variety of applications, but in Limbo, it serves two purposes. The first purpose is to detect when a goal state is reachable on the current path. As concolic execution analyzes a path, it builds the set of constraints Π that are required for the program to follow the same path. This makes it very easy to see if any execution on the same path has a particular property, by querying an SMT constraint solver.

The second and more important purpose is to approximate the program states that are reachable from the vulnerability. At a high level, a single concolic execution takes a concrete test case (e.g., console, file, or network input) as input, and outputs new test cases that explore adjacent execution paths. Thus, by running concolic execution to produce new test cases, and then concolically executing those test cases, and so on, it is possible to explore the symbolic state space of the program that is reachable from the original starting test case (or test cases).

More succinctly, we say that a program state Δ' is reachable from state Δ, or $\Delta \rightsquigarrow \Delta'$, if there exists a program execution in which Δ is reached before Δ'. As described above, concolic execution defines its own reachability relation $\overset{\text{exec}}{\rightsquigarrow}$ which approximates $\rightsquigarrow$. We believe that concolic execution is well suited for this task because it is sound, relatively complete, and relatively scalable, as we discuss below.

3.2.1 Soundness. We say $\overset{\text{exec}}{\rightsquigarrow}$ is *sound* if $\Delta \overset{\text{exec}}{\rightsquigarrow} \Delta'$ implies $\Delta \rightsquigarrow \Delta'$, or less formally, a state is reachable if the analysis says it is. Concolic execution is based on symbolic execution, which is a sound analysis. As a result, concolic execution can accurately reason about the executions that are adjacent to the current path. At a high level, this means that if concolic execution says an execution is reachable, then it is. Furthermore, concolic execution can also provide the *inputs* to an example execution to serve as a validation.

3.2.2 Completeness. Completeness is the converse of soundness. We say an analysis is *complete* if $\Delta \rightsquigarrow \Delta'$ implies $\Delta \overset{\text{exec}}{\rightsquigarrow} \Delta'$, or if a state is reachable *only if* the analysis says it is. Concolic execution is not complete in practice, although it comes close. The primary cause of incompleteness is when the analysis times out or needs too many resources. We discuss this problem in more detail in the next section.

3.2.3 Scalability. Many dynamic analyses, which are analyses that execute the program, can only reason about one program execution at a time. Since there is an astronomical number of concrete executions, this can make finding a particular execution infeasible. On the other hand, static analyses, which do not execute the program, can reason about many program executions at once and tend to scale much better than dynamic analyses. Because static analyses often abstract some of the complexities of binary analysis away, however, this also means that they often sacrifice precision in favor of scalability.

Concolic execution lies in an interesting middle ground. It is a dynamic, sound analysis. But it also scales much better than most dynamic analyses because it reasons about one *program path* at a time instead of one *program execution* at a time. We say it is *relatively* scalable because there are still many program paths, and for most programs, it is not practical to expect to be able to reason about each one. We address this in Limbo by using a heuristic search strategy that prioritizes states more likely to lead to a code reuse attack (Section 3.4.1). However, even with our heuristics, Limbo fails to find some attacks, especially those with complex goals (Section 5).

3.3 Searching for Code Reuse Attacks Using Concolic Execution

In this section, we explain how Limbo leverages concolic execution to search for code reuse attacks by searching for program executions from a vulnerability to an attacker's desired goal state.

3.3.1 Vulnerable Starting State. Since code reuse attacks leverage an attacker's control over the program, and different vulnerabilities yield different types of control, it is important for the user to be able to specify the starting state that corresponds to the vulnerability that they are trying to exploit. Limbo allows the user to specify the vulnerability and its corresponding starting state in three ways.

The primary method is to provide a concrete test case (i.e., a program input) that triggers a control flow vulnerability. Limbo will automatically detect the vulnerability's starting state by looking for an indirect jump to a symbolic location that can be directed to an unmapped address, which indicates the attacker has enough direct control over the program counter to make the program crash.

The user can also specify a starting state that is a concrete test case that does *not* trigger a vulnerability. In this case, Limbo will act as a regular concolic executor [7] and will explore the state space of the program to search for a vulnerability. Once it finds a vulnerability, it will act as if the user provided that vulnerability as its starting point, and will attempt to reach a goal state from it.

Finally, the user may elect to employ a synthetic buffer overflow vulnerability. This allows Limbo to look for code reuse attacks in programs that do not contain known vulnerabilities, and is similar to most automated ROP tools, which assume the attacker controls the stack. We use this capability in Section 4.3 to study the relationship between the amount of code available for reuse and Limbo's ability to produce code reuse attacks. To simulate a vulnerability, immediately after `__libc_start_main` is reached, Limbo calls a function that reads data from a symbolic file and deliberately uses it to overflow its stack frame. This enables Limbo to determine the level of control over the program state that can be achieved by an ideal stack buffer overflow. The vulnerability is ideal in the sense that it does not constrain the attacker's input at all; as long as the attacker supplies enough bytes, the overflow will be triggered. Of course, real vulnerabilities do not always have this property.

3.3.2 Goal States. Limbo allows the user to specify goal states by providing a *goal expression*, which may reference register and memory values. A state is a goal state if and only if the goal expression evaluates to true in that state. For example, if the attacker's goal is to set the `%eax` register to `0xd34db33f`, her goal expression might be `%eax == 0xd34db33f`. On the other hand, if she wanted to write `0xd34db33f` into memory at address `0x12345678`, she might choose the goal condition `M[0x12345678] == 0xd34db33f`. Goal conditions are a simple, flexible, and powerful way to define the attacker's goal.

Because concolic execution maintains the path predicate Π (Section 3.1), which collects the constraints required for an execution to follow the path of the current program execution, it is easy to check if any execution reaches a goal. To see if the goal expression *expr* can be true, Limbo queries an SMT constraint solver to see if $\Pi \wedge expr$ is satisfiable. If it is, Limbo has found an execution that reaches a goal state, and will create the corresponding test case.

This declarative approach, in which the attacker describes the program states which she would like the program to reach, rather than commands to put the program into the desired state, is different from the traditional academic approach to code reuse attacks, which utilize a Turing-complete language [8, 10, 17, 26, 29]. However, in practice, attackers only use code reuse attacks for relatively simple goals, such as disabling NX. After disabling NX, the attacker can then execute shellcode, which is also Turing-complete, and gives them full control in the context of the hijacked program. We believe this is simpler and more practical than trying to implement Turing-complete behavior using code reuse techniques alone (although we do find that type of work fascinating!)

3.3.3 Exploring Reachable States. To find a code reuse attack, Limbo searches for an execution from a vulnerable starting state Δ to a goal state Δ'. Concolic execution represents states *symbolically*. As a consequence, each concolic execution state Δ^{SYM} corresponds to one or more concrete execution states Δ. Reachability is checked with concolic execution by initializing a queue of pending states with the vulnerable starting state Δ. Limbo then iteratively selects a symbolic state from the queue of pending states, concolically executes the selected state, and adds all of its successors to the queue. The check is complete when the queue contains the goal state, demonstrating that it is reachable, or is empty.

A very real possibility is that the procedure never terminates, which can happen for programs that contain an infinite number of paths. We discuss methods for dealing with this in Section 3.4.1, and the consequences of this limitation in Section 5.

3.4 Customizing Concolic Execution for Code Reuse Attacks

Limbo is built on top of the Mayhem binary concolic executor [7]. Limbo uses a version of Mayhem that only supports concolic execution of 32-bit Linux binaries, and as a result Limbo is limited to 32-bit Linux binaries as well (see Section 5).

In the rest of this section, we describe the changes that we made to Mayhem to enable it to search for code reuse attacks.

3.4.1 Heuristics. One of the most important changes that we made to Mayhem was adding a heuristic to encourage exploration of states more likely to lead to a code reuse attack. Before we created this heuristic, Limbo would start exploring reachable paths in no deliberate order, and more often than not would get "stuck" exploring complex functions. Limbo still worked, but it was spending too much of its time in unpromising parts of the code. In response, we created a heuristic that prioritizes more promising executions.

We wanted to be very careful about how we created a heuristic, however. Naturally, one type of heuristic would be to reuse existing techniques to look for code reuse attacks (e.g., ROP), which would defeat the purpose of Limbo. Instead, we wanted a general heuristic for code reuse attacks.

Storage Considerations One of the complicating factors is that real computers have a finite amount of disk space, and thus can only store a finite number of states. Our first attempt was simply to use Breadth First Search (BFS) over sequences of symbolic indirect jump targets. In other words, Limbo would start by considering each sequence of indirect jump targets of length one, followed by sequences of length two, and so on. Exploring attacks in this order is natural, as it prefers simpler, shallower executions that are likely to have fewer constraints and use less processing power. Unfortunately, this approach uses a lot of disk space. As an example, consider a scenario in which at least three symbolic indirect jumps are needed to reach a goal state (which happens in Section 4). BFS would first exhaustively explore sequences of length one. This would produce at least as many states as there are targets of the indirect jump. There is generally at least one page worth of targets, or 4096 targets. A Mayhem test case utilizes about 300 KiB of disk space. After exhaustively exploring states of depth one, BFS would require $4096 * 300\text{KiB} = 1.2\text{GiB}$ of disk space. After exhaustively exploring depth two, it would need a whopping $4096^2 * 300\text{KiB} = 5TiB$ of disk space to contain all 4096^2 intermediate test cases. Needless to say, available disk space naturally constrains the order in which states can be explored.

Heuristic Principles At a high level, our final heuristic is based on two principles. The first is to *prefer states with more symbolic control.* To see why this is helpful, assume that there are two states available to explore. The first state is completely concrete, but in the second state all the registers are symbolic. If the goal is to write to memory, almost any instruction that writes to memory will work for the symbolic state. For example, `mov %eax, (%ebx)` could be used, or even `addl %eax, (%ebx)` if the prior value in memory was known. Having symbolic control over registers allows the semantics of instructions to be reused in very powerful ways. In contrast, the concrete state is likely to crash when executing those instructions unless `%ebx` happens to be a valid pointer.

For this reason, Limbo awards a state +1 point for each symbolic bit in a register. However, it does not award any points for registers that are wider than the architecture's pointer type. For similar reasons, Limbo also awards +1 point for each symbolic memory byte located in the scratch storage memory area (Section 3.4.2).

The second principle of our heuristic is to *prefer a smaller state space.* The rationale behind this is that each indirect jump increases the amount of state space to explore significantly. Thus, if all else is equal, it's better to search a smaller state space first. As a consequence, Limbo "charges" states for each symbolic indirect jump they execute. The first symbolic indirect jump in an execution is charged a high price — $3*n$ where n is the number of bits in the pointer type. In other words, when Limbo first starts searching, it will prefer to explore targets for the first indirect jump until a state is found that symbolically controls at least three registers. After the first, the cost for subsequent symbolic indirect jumps drops down to n.

When Is the Heuristic Evaluated? One counter-intuitive aspect of the heuristic is that it cannot be evaluated at the time a new test case is created. For example, if Limbo is exploring the targets of a symbolic indirect jump, it will produce test cases that trigger execution of different targets of that jump. Let's say that Limbo produces a test case for the jump target `0x12345678`, and that this code executes `pop %eax`, which allows Limbo to gain symbolic control over `%eax`. At the time the test case for `0x12345678` is created, Limbo is *not* executing the code at `0x12345678`. Instead, it executes the existing path up to the indirect jump, and ensures that the new test case will take the indirect jump to `0x12345678`. Crucially, Limbo does *not* know the effects of executing that code yet. It will not know the effects of that code until the new test case is concolically executed. Although it might seem too late at that point, the heuristic is still useful because `%eax` will remain symbolic in most of the new test case's children, and thus they are promising too. We also modify our heuristic slightly to reflect this delayed reaction. In particular, we delay the "charge" for an indirect jump until the new code for that jump is evaluated. Otherwise, we would penalize the test case for an indirect jump without allowing it to gain symbolic control.

Limiting Symbolic Branches We have also found pruning certain subtrees of the state space to be helpful, although it introduces incompleteness (Section 3.2.2). For example, an earlier version of Limbo often discovered that it could use a function to perform a symbolic write, but some type of safety check would always terminate the state before it could escape the function. If that function was complex, Limbo would generate many seemingly promising states that took a long time to explore but which would ultimately lead nowhere.

Therefore, Limbo includes the following limits (with default values in parentheses) to address this concern (also see Section 4.4):

- `max-branches`: Maximum number of symbolic branches since the last symbolic indirect jump (0)
- `max-forks`: Maximum number of concolic executor forks since the last symbolic indirect jump (25)
- `max-indjumps`: Maximum number of symbolic indirect jumps (3)

3.4.2 *Tweaks to Concolic Execution.*

Mayhem already contained some level of support for the following binary execution behaviors, but we modified their behavior in Limbo.

Indirect Jumps Mayhem originally handled symbolic indirect jumps by creating test cases for each possible target up to a hard-coded limit. There are two problems with this approach. First, in vulnerable executions, indirect jumps are often completely unconstrained. They can jump to any address (although most of these targets would be unmapped and cause a crash). Rather than allowing these jumps to go anywhere, Limbo allows the user to designate a set of code modules which symbolic indirect jumps can target. This effectively sets the code modules from which code will be reused. By default, Limbo assumes this code module is not randomized. If it is, the user can either provide the address of the code module as an argument, or Limbo will produce a non-deterministic attack (Section 2.3.1).

The second problem is that throwing away targets above the hard-coded limit is a source of incompleteness (Section 3.2.2), and is especially bad for code reuse attacks. A code module contains at least a page worth of executable addresses. It is important to be able to explore all of them. We tried multiple approaches, including just removing the limit. Writing thousands of test cases was slow, and also tended to fill the disk up with unpromising executions. The approach we found to work best is to enumerate targets up to a fixed number (64), and to produce test cases for each of those targets. We then produce a "continuation" test case that replays the indirect jump with a constraint that disallows the previously enumerated targets. When this other test case is executed, it will produce 64 more test cases, and another "continuation" test case. This will continue until no more feasible targets exist. As a minor improvement, we also found that splitting the continuation space in half and producing two continuation test cases worked better to ensure that we never ran out of test cases. This intentionally causes exponential division of test cases.

Symbolic Memory Another challenging situation occurs when a symbolic address is dereferenced in a memory read or memory write. By default, Mayhem handles symbolic memory operations by redirecting them to a memory region so they do not crash. It will prefer memory regions that are symbolically controlled, which is in line with our heuristic (Section 3.4.1).

Limbo uses a slight variation of this: it tries to redirect all symbolic memory reads and writes to the same scratch space area. Reading and writing at the same location provides an opportunity to control additional program state. For example, if Limbo has symbolic control of the scratch area, and can symbolically load from the scratch area into a new register, it will gain symbolic control of that new register.

There are downsides to constraining symbolic memory operations in this way. Attackers have used symbolic memory loads on code regions to copy data from one location to another [21]. For example, this can be useful when it is not possible to include a specific character in a payload. Likewise, there could be code that, when reused, behaves differently depending on values in memory. By not allowing symbolic memory writes to change these locations, we could theoretically miss out on some behaviors. Unfortunately, the cost of considering all possible memory regions is high. We feel that, by default, the cost of exploring all these extra executions is not worth the benefit of slightly improving completeness.

4 EVALUATION

In this section, we evaluate how effectively Limbo can discover code reuse attacks using our state reachability approach. We are specifically interested in addressing the following research questions:

- **RQ1: Can Limbo discover code reuse attacks in the presence of fine-grained CFI? (Section 4.2)**
  One of Limbo's advantages over existing work is that it is *generic* and thus can find defense-aware attacks without any special knowledge about a defense. As one of the most popular types of defenses, we test whether Limbo can identify attacks in the presence of fine-grained CFI.

- **RQ2: How much executable code does Limbo require to produce code reuse attacks? (Section 4.3)**
  Whether because of ASLR or newer defenses such as CFI, attackers have less code from which to produce modern code reuse attacks. Thus, it is important to understand the relationship between the amount of executable code that the attacker can leverage and the amount of control over the program that this affords the attacker.

- **RQ3: How sensitive is Limbo to its heuristics? (Section 4.4)**
  Limbo employs several heuristics in order to achieve scalability. We study how much these heuristics help, and identify the parameter values that have the largest impact.

- **RQ4: Can Limbo discover new techniques? (Section 4.5)**
  Because Limbo searches for code reuse attacks in a very general way, it can "discover" new techniques to construct code reuse attacks. We conduct a case study of one such technique.

Name	Description
`execv`	Execute a command using execv
`s-mem`	Store an arbitrary byte into an arbitrary address
`l-mem`	Read a byte from an arbitrary address into a register
`l-ref`*n*	Load *n* registers with pointers to controlled memory
`l-reg`*n*	Load *n* arbitrary full-width values into registers
`l-`*reg*	Load an arbitrary full-width value into `%`*reg*

Table 1: Goals evaluated by Limbo

4.1 Experiment Setup

Hardware Each experiment is run inside a virtual machine hosted by a machine with 4 AMD Opteron 6386 SE processors and 256 GiB of RAM. Each processor has a total of 16 cores for a total of 64 cores per host. Each virtual machine is assigned 32 vCPUs, 64 GiB of RAM, and 2 TiB of disk space, and runs Ubuntu Linux 18.04.3. Limbo uses all 32 vCPUs, but the other tools tested in this section (angrop [3, 31] and BOPC [18]) are single threaded.

Goals There is no universal goal that makes sense for all attackers and all programs. Instead, an attacker's goal depends on both her higher level objectives (e.g., data infiltration, subversion, or denial of service) and the nature of the program she is attacking. In this paper, we address this by evaluating a number of fairly generic goals, such as writing an arbitrary value to an arbitrary location in memory and executing functions in libc. Some of these goals may be of direct interest to attackers (e.g., calling `system("/bin/sh")`). Others serve as a measurement for how much control the attacker has over the program's environment.

Table 1 lists each goal and gives a short description. We also describe each goal in more detail below, starting with one of the most desirable goals, `execv`, which allows the attacker to execute an arbitrary command. This goal can be accomplished by finding an existing call to `execv` in the program, or by computing the address of the function inside libc using a Global Offset Table (GOT) pointer [25]. For example, if the GOT has an entry for `printf`, and the attacker knows that `execv` occurs `0x3bf8` bytes after `printf`, this goal would ensure that the goal state jumps to `M[GOT] + 0x3bf8`. The next two goals, `s-mem` and `l-mem`, represent storing an arbitrary constant byte to an arbitrary address in memory, and loading a byte from an arbitrary address in memory to any general purpose register, respectively. The `l-reg`*n* goal measures the ability to concurrently set *n* general purpose registers to arbitrary full-width values. The `l-ref`*n* goal concurrently initializes *n* general purpose registers with *pointers* to arbitrary data controlled by the attacker. Finally, `l-`*reg* goals exist for each general purpose register, and represent the ability to store an arbitrary full-width value in that register.

4.2 RQ1: Can Limbo discover code reuse attacks in the presence of fine-grained CFI?

One of Limbo's advantages is that it is *generic* and can theoretically find defense-aware attacks without any special knowledge about a defense. In this section, we put this theory to the test by evaluating whether Limbo can find find attacks in one of the most common defenses, Control Flow Integrity (CFI) [1, 2, 12, 22]. We compare Limbo's performance to BOPC, a state-of-the-art tool for identifying

Program	Size (KiB)	Goals													
		execv		s-mem		l-mem		l-ref4		l-ref5		l-reg4		l-reg5	
		Limbo	BOPC	Limbo	BOPC	Limbo	BOPC	Limbo	BOPC	Limbo	BOPC	Limbo	BOPC	Limbo	BOPC
orzhttpd	37.9	~~3,750.4~~	~~1.4~~	23.4	~~1.4~~	1.9	~~1.4~~	~~3,750.4~~	~~1.4~~	~~3,750.4~~	~~1.3~~	31.9	1.8	~~3,750.4~~	1.9
nullhttpd	97.2	~~93.6~~	~~1.5~~	25.6	~~1.5~~	2.8	~~1.5~~	~~93.6~~	~~1.5~~	~~93.6~~	~~1.4~~	33.0	1.5	~~93.6~~	1.5
sudo	182.8	160.1	~~3.6~~	56.9	~~3.6~~	0.2	~~3.6~~	276.2	~~3.8~~	345.6	~~3.5~~	1.7	7.3	1.7	251.9
opensshd	320.7	157.0	~~5.4~~	263.4	~~5.4~~	7.5	~~5.4~~	~~∞~~	~~6.8~~	~~∞~~	~~5.2~~	17.7	6.9	39.9	6.9
wuftpd	404.5	~~∞~~	~~8.9~~	234.3	~~9.0~~	2.1	~~11.7~~	~~∞~~	~~9.8~~	~~∞~~	~~8.6~~	136.4	23.6	275.1	2,191.8
proftpd	1,306.9	~~∞~~	~~26.9~~	39.0	~~26.6~~	0.3	~~26.5~~	~~∞~~	4,291.7	~~∞~~	~~25.5~~	1.8	176.3	14.6	181.6
httpd	1,474.5	1,278.0	~~28.2~~	1.8	~~29.2~~	0.2	~~100.3~~	464.5	3,766.0†	~~∞~~	~~27.0~~	12.7	310.9†	12.7	586.0†
nginx	3,351.2	151.5	~~37.5~~	16.9	~~37.7~~	0.1	57.4†	~~∞~~	~~417.7~~	~~∞~~	~~35.9~~	4.2	209.5†	~~∞~~	~~∞~~
wireshark	7,638.6	~~∞~~	~~67.3~~	212.5	~~71.3~~	28.5	1,174.0	~~∞~~	~~∞~~	~~∞~~	~~63.0~~	314.0	1,597.8	375.3	1,897.7
smbclient	10,378.8	~~∞~~	~~179.0~~	2.3	~~1,790.8~~	0.5	1,829.9	~~∞~~	~~1,722.3~~	~~∞~~	~~171.6~~	9.5	~~3,138.0~~	~~∞~~	~~2,089.3~~

Table 2: A comparison of the attacks discovered by Limbo and BOPC [18] for the listed programs and goals. CFI is enforced on both forward- and backward-edges based on the CFG recovered by angr as implemented in BOPC [18]. A ~~red~~, struck out field indicates that the tool failed to produce an attack. A yellow† field with a cross† indicates that BOPC only found attacks that required either an arbitrary write primitive (AWP) or a pre-initialized register. The duration in minutes is also reported, but note that BOPC is single threaded. If either tool ran out of memory or took longer than 72 hours, ∞ is reported instead.

Data Oriented Programming (DOP) attacks [18], which are designed to operate in the presence of CFI.

We adopt many of the evaluation parameters from BOPC's evaluation [18]. Specifically, we simulate fine-grained CFI on both forward- and backward-edges based on the CFG recovered by angr [31], and do not enforce CFI for shared libraries. We also evaluate the same set of 10 network and system utilities (such as `apache` and `opensshd`), as these programs are representative targets of code reuse attacks. We chose to simulate CFI in order to have a fair comparison with BOPC, which, unlike Limbo, cannot be used with real CFI implementations.

For each program, we ran Limbo and limited it to explore executions containing no more than three symbolic indirect jumps for 72 hours. Limbo searches for all goals simultaneously. BOPC executes in two stages, which are both single threaded. In the first stage, BOPC identifies and saves abstractions that represent the behavior of each basic block in the program. BOPC is then invoked once per goal to determine if that goal can be met using the saved abstractions. Both stages of BOPC were limited to 72 hours.

Table 2 shows the results of this experiment. For each program and goal tested, we report whether Limbo and BOPC found an attack for that goal (shown in green), or report that no attack was found (shown in struck-out ~~red~~). Times are also reported in minutes for context. If either tool ran out of memory or took longer than 72 hours, ∞ is reported instead. In a few cases, BOPC was able to find an attack, but only by making assumptions about the attacker's control over the environment (shown in yellow†). Specifically, BOPC assumes that it may use an arbitrary write primitive (AWP) to arbitrarily write to memory, or that it can initialize registers to arbitrary values. Since Limbo does not make such assumptions, we modified BOPC to report when it had to use either assumption to create its attack.

The goals in Table 2 are ordered so that the goals indicating more control over the program are on the left side, and less control to the right. We can quickly compare the performance of Limbo and BOPC by summarizing the outcome of each goal as a *record* such as W-L-T, where W is the number of scenarios in which Limbo won (e.g., was able to produce an attack but BOPC was not); L is the number of losses (e.g., BOPC found an attack but Limbo did not); and T is the number of ties (e.g., both systems found an attack). Starting with the most difficult goal, `execv`, Limbo has a record of 4-0-0, indicating that it found attacks for 4 (out of 10) programs, compared to 0 for BOPC. This is one of the more desirable goals for an attacker, since it allows her to execute arbitrary commands. Limbo also outperformed BOPC at memory operations, yielding records of 10-0-0 and 7-0-3 for storing to and loading from memory respectively. Both systems struggled to initialize references to controlled memory, with records of 1-1-1 and 1-0-0 for `l-ref4` and `l-ref5` respectively. When controlling registers, both systems performed roughly equally. When setting four and five registers in tandem (`l-reg4` and `l-reg5`), Limbo earned records of 1-0-9 and 0-2-6 respectively. When setting individual registers in isolation, Limbo earned a record of 1-0-59 (not shown in Table 2).

We investigated why BOPC failed to produce any attacks for the `s-mem` and `execv` goals. At a high level, BOPC attempts to map basic blocks to statements in its SPloit Language (SPL) payloads, which are analogous to what we call goals. For memory writes, BOPC specifically looks for a block that contains a write akin to `mov %eax, (%ebx)` where both registers have not been modified since the start of the block. It is noteworthy that an instruction such as `mov %eax, 12(%ebx)` would *not* match. This strategy makes sense considering that, unlike Limbo, BOPC's broader goal is to enable Turing-complete computation, and in the general case, the value in a register may be coming from a previous computation and thus not constant. This design decision is restrictive, and as these results show, it rules out many attacks that could otherwise be found. Because Limbo is designed to be generic, it does not make (explicit) assumptions that prevent the discovery of attacks in this way. The trade-off, however, is that Limbo can only be used to search for attacks that put the program into a designated goal state; it cannot, for instance, execute an infinite loop, which is an attack BOPC can search for. Since the vast majority of code reuse attacks in practice do not need this ability, we argue that this trade-off is desirable.

We made minor modifications to both Limbo and BOPC to perform our experiments. To allow Limbo to consider the same CFI policy (i.e., the set of allowed control flow transitions) as BOPC, we added functionality that extracts the CFI policy used by BOPC, and simulates a crash in Limbo whenever a disallowed control-flow transition is taken. In BOPC, we first added an option to disable the use of an Arbitrary Write Primitive (AWP) and arbitrary control of registers.[1] We

[1]The modified version of BOPC is available at https://github.com/sei-eschwartz/BOPC.

Program	Size (KiB)	Goals												
		`execv`		`s-mem`		`l-mem`	`l-ref4`		`l-ref5`		`l-reg4`		`l-reg5`	
		Limbo	angr	Limbo	angr	Limbo	Limbo	angr	Limbo	angr	Limbo	angr	Limbo	angr
`arch`	33.5	1,910.8	~~1.6~~	13.2	~~1.6~~	17.6	13.2	~~∞~~	13.2	~~∞~~	12.9	1.6	42.1	1.6
`b2sum`	73.5	2,902.3	~~2.5~~	994.6	~~2.5~~	753.7	994.6	~~∞~~	994.6	~~∞~~	2.5	2.5	148.9	2.5
`bzcat`	33.4	240.7	~~0.9~~	289.9	~~0.9~~	112.2	289.9	~~∞~~	289.9	~~∞~~	2.7	0.9	15.5	0.9
`chattr`	9.4	71.9	~~0.3~~	~~∞~~	~~0.3~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	7.3	0.3	25.1	0.3
`cut`	41.5	252.9	~~2.0~~	38.8	~~2.0~~	39.9	38.8	~~∞~~	80.8	~~∞~~	6.1	2.0	10.2	2.0
`dmesg`	69.6	646.8	~~2.6~~	652.2	~~2.6~~	191.2	652.2	~~∞~~	~~∞~~	~~∞~~	0.4	2.6	1.8	2.6
`getopt`	13.4	~~∞~~	~~0.4~~	71.7	~~0.4~~	75.3	71.6	~~∞~~	~~∞~~	~~∞~~	23.3	0.4	64.5	0.4
`md5sum`	45.5	~~∞~~	~~2.2~~	1.8	~~2.2~~	622.1	1.8	~~∞~~	1.8	~~∞~~	1.8	2.2	1.8	2.2
`nawk`	127.9	213.1	~~1.6~~	213.1	~~1.6~~	79.8	213.2	~~∞~~	279.6	~~∞~~	3.8	1.6	3.9	1.6
`numfmt`	65.5	~~∞~~	~~2.2~~	1,575.8	~~2.1~~	1,499.8	1,576.9	~~∞~~	1,584.3	~~∞~~	0.4	2.1	17.1	2.1
`pathchk`	33.5	504.6	~~1.6~~	6.5	~~1.6~~	325.4	6.5	~~∞~~	77.2	~~∞~~	6.5	1.6	14.5	1.6
`pgrep`	25.4	~~∞~~	~~0.9~~	1,571.9	~~0.9~~	2,366.5	1,571.9	~~∞~~	1,571.9	~~∞~~	3.7	0.9	24.6	0.9
`renice`	9.4	~~∞~~	~~0.4~~	~~∞~~	~~0.4~~	32.3	~~∞~~	~~∞~~	~~∞~~	~~∞~~	4.1	0.3	182.8	0.3
`sh.distrib`	125.7	1,401.1	~~1.6~~	150.8	~~1.6~~	21.1	150.8	~~∞~~	150.8	~~∞~~	0.5	1.6	4.4	1.6
`sha512sum`	101.5	~~∞~~	~~1.6~~	24.9	~~1.6~~	3,263.8	24.9	~~∞~~	2,838.5	~~∞~~	24.7	1.6	24.9	1.6
`sum`	45.5	454.9	~~2.2~~	164.4	~~2.2~~	523.7	164.4	~~∞~~	164.4	~~∞~~	3.4	2.2	3.3	2.2
`umount`	25.4	~~∞~~	~~0.8~~	~~∞~~	~~0.8~~	104.0	~~∞~~	~~∞~~	~~∞~~	~~∞~~	0.2	0.7	19.7	0.7
`vdir`	141.7	401.8	~~1.7~~	225.9	~~1.7~~	203.1	225.9	~~∞~~	243.6	~~∞~~	5.3	1.7	13.3	1.7
`xargs`	73.5	~~∞~~	~~3.4~~	5.4	~~3.4~~	51.7	5.4	~~∞~~	38.4	~~∞~~	2.6	3.4	7.5	3.4
`zdump`	21.4	647.2	~~1.0~~	~~∞~~	~~1.0~~	1,897.8	~~∞~~	~~∞~~	~~∞~~	~~∞~~	0.5	0.8	0.2	0.8

Table 3: A comparison of the attacks discovered by Limbo and angrop [3, 31] for the listed programs and goals using a synthetic stack buffer overflow vulnerability with only ASLR and NX. A ~~red~~, struck out field indicates that the tool failed to produce an attack. The duration in minutes is also reported, but note that angrop is single threaded. If either tool ran out of memory or took longer than 72 hours, ∞ is reported instead.

also added support for x86 executables, so that BOPC could operate on the same executables as Limbo, which was straight-forward.

The fact that Limbo is able to bypass CFI is not surprising or new; researchers have documented that CFI does not prevent code reuse attacks in all programs [17], and that some of these attacks can be automated [18]. However, what is notable is that we did not implement any strategy in Limbo for attacking CFI-protected binaries. Limbo simply tried to follow its directive to reach a goal state. If it encountered a CFI check that terminated the program because of an illegal jump, that state could obviously not reach a goal state, and Limbo would try other executions.

4.3 RQ2: How much executable code does Limbo require to produce code reuse attacks?

In this section, we examine the relationship between the amount of code available for reuse and Limbo's ability to find code reuse attacks. As we discussed in Section 2.3, advances in two dimensions have restricted the amount of code available for modern code reuse attacks. In some cases, ASLR may randomize all but a small amount of executable code, forcing the attacker to use the unrandomized code, or determine the address of a randomized code module. In other scenarios, defenses such as CFI may restrict transitions to some code addresses, effectively limiting the code available to the attacker. As defenses advance, attackers will have less code available for them to reuse. Thus, it is important to understand the relationship between the amount of executable code that the attacker can leverage and the amount of control over the program that this affords the attacker.

Since the goal of this experiment is to evaluate Limbo when only a small amount of executable code is available, we selected test programs by randomly sampling twenty programs that are installed by default in the Ubuntu 18.04.3 Docker image. These programs are small, and thus provide a small amount of executable code for Limbo to reuse. The smallest and largest programs in our sample are 9.4 and 141.7 KiB respectively, with a mean and median size of 55.8 and 43.5 KiB respectively. Since these programs do not contain vulnerabilities that we know of, Limbo uses a synthetic stack buffer overflow vulnerability (Section 3.3.1). These programs are also all PIEs (Section 2.3.1), so Limbo assumes the attacker knows where in memory the executable is loaded.

To compare with existing work, we also tested a popular open-source tool for automating ROP attacks that is part of the angr binary analysis framework, angrop [3, 31]. At a high level, angrop uses symbolic analysis to identify gadgets in the target program, and then combines the gadgets to implement a goal specified by the user. We tested version 8.20.1.7 of angrop, which was the latest version at the time of writing. Both Limbo and angrop were allowed to run for 72 hours per program, although angrop is single threaded.

Table 3 shows the results of this experiment, and can be interpreted in the same way as Table 2 in Section 4.2. We will also summarize the results here using the Win-Loss-Tie record notation from that section. Starting with the two most difficult goals, `execv` and `s-mem`, Limbo earned records of 12-0-0 and 16-0-0 respectively, indicating that Limbo was able to construct `execv` and `s-mem` attacks for 12 and 16 programs respectively, but angrop did not find any attacks for either of these goals. Limbo was able to load arbitrary bytes from memory in 19 programs, but angrop does not support this type of goal. Because of angrop's inability to write to memory in these programs, it performs poorly at `l-ref4` and `l-ref5` (Limbo earns records of 16-0-0 and 14-0-0). Both systems were able to control the majority of registers, with Limbo earning a record of 0-0-20 for both `l-reg4` and `l-reg5`, and an overall record of 2-4-113 across all registers (not shown in Table 3).

Overall, these results suggest that Limbo requires *very* little code to be able to construct code reuse attacks. The smallest program that Limbo was able to attack was less than 10 KiB. In contrast, we ran angrop on a variety of executables from our system and it rarely found `s-mem` or `jmp-libc` attacks in executables smaller than 200 KiB.

Experiment	Heuristics	Goals						
		`execv`	`s-mem`	`l-mem`	`l-ref4`	`l-ref5`	`l-reg4`	`l-reg5`
Executables protected by CFI (RQ1/Section 4.2)	Enabled	4 (27,667.3)	10 (876.1)	10 (44.0)	2 (35,301.4)	1 (39,226.4)	10 (562.9)	6 (17,999.7)
	Disabled	3 (31,691.6)	9 (5,797.7)	10 (58.9)	2 (35,833.9)	1 (39,616.1)	10 (1,104.4)	5 (22,413.1)
Small Ubuntu executables (RQ2/Section 4.3)	Enabled	12 (44,209.3)	16 (23,281.5)	19 (16,501.1)	16 (23,282.6)	14 (34,249.9)	20 (112.4)	20 (626.0)
	Disabled	4 (71,443.5)	10 (45,275.9)	12 (39,097.8)	10 (45,328.6)	6 (62,851.2)	20 (455.4)	19 (6,265.0)

Table 4: A summary of the code reuse attacks that Limbo was able to find conditioned on whether all heuristics were enabled or disabled. For each configuration and goal, the number of programs in which an attack for that goal was found is reported (more is better). The parenthesized number represents the total number of minutes across all programs spent searching for that goal; if the goal was not found in a program, the maximum experiment duration is used instead (72 hours). Green values highlight the better performing configuration, whereas red shows the inferior one. If either configuration ran out of memory or took longer than 72 hours, ∞ is reported instead.

4.4 RQ3: How sensitive is Limbo to its heuristics?

Limbo contains several modifications and heuristics that are not standard in concolic executors (Section 3.4). In this section, we evaluate Limbo's performance while varying these modifications to understand their contribution to Limbo's overall results.

First, to broadly understand the implications of our modifications, we replicate the experiments for RQs 1 and 2 (Sections 4.2 and 4.3) but disable all of Limbo's heuristics, and use Mayhem's default heuristics instead. We do *not* disable the modifications to symbolic indirect jumps and memory that are described in Section 3.4.2. These can be considered the bare minimum changes to allow Mayhem to discover code reuse attacks; without these modifications, Mayhem is unable to find virtually any code reuse attacks.

Table 4 shows the overall effect of disabling heuristics in the experiments for RQs 1 and 2, and the results differ between the two experiments. Each row in the table represents a *configuration* of Limbo, and the columns represent a goal. Each cell reports the number of programs in which the corresponding configuration found an attack for that goal. In parentheses, the time required to find the attacks is reported, with any program missing an attack counted as the full experiment duration (72 hours). The first two rows show that for the RQ1 experiment in Section 4.2, Limbo performed slightly better with its heuristics enabled. Using the Win-Loss-Tie notation from Section 4.2, Limbo had a record of 4-1-39 across all goals in the table compared to the version with heuristics disabled. On the other hand, the last two rows show that heuristics had a more substantial effect on the RQ2 experiment in Section 4.3, with a record of 37-1-80 over all goals in the table. The salient difference between these two experiments is whether CFI is enabled. We hypothesize that because strong defenses such as CFI greatly restrict the state space, it is easier for Limbo to cover a large portion of the state space, and thus the order is not as important. Without a strong defense, the state space explodes very rapidly at indirect jumps, and the heuristics play a more important role in deciding which states are explored given a fixed computational budget. This also leads us to a counter-intuitive prediction: as defenses becomes stronger, our techniques will be able to search the state space for code reuse attacks more thoroughly. Thus, strong defenses counter-intuitively provide some benefit to attackers as well.

Config.	Goals						
	`execv`	`s-mem`	`l-mem`	`l-ref4`	`l-ref5`	`l-reg4`	`l-reg5`
`mb=0`†	1,986.2	63.5	569.4	63.5	1,271.7	1.6	63.5
`mb=1`	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	2.6	143.1
`mb=2`	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	1.8	143.3
`mb=4`	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	3.1	219.1
`mf=6`	1,962.4	50.7	490.2	50.7	1,156.7	1.6	50.7
`mf=12`	2,533.4	623.0	395.7	900.3	~~∞~~	5.0	62.2
`mf=25`†	1,986.2	63.5	569.4	63.5	1,271.7	1.6	63.5
`mf=50`	2,264.6	353.1	349.9	1,372.5	~~∞~~	1.9	76.0
`mf=100`	2,337.7	428.1	228.4	~~∞~~	~~∞~~	2.1	113.9
`mf=∞`	2,030.0	114.7	751.8	114.7	~~∞~~	1.9	114.6
`mi=1`	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	1.8	26.3
`mi=2`	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	1.7	105.0
`mi=3`†	1,986.2	63.5	569.4	63.5	1,271.7	1.6	63.5
`mi=4`	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	1.4	81.9
`mi=5`	2,003.4	93.9	~~∞~~	93.9	~~∞~~	1.9	93.9

Table 5: A comparison of the attacks discovered by Limbo under the listed configurations (`mb`: `max-branches`, `mf`:`max-forks`, and `mi`:`max-indjumps`) using the same experimental configuration as Section 4.3. A ~~red~~, struck out duration indicates that the configuration failed to produce an attack. † indicates a default Limbo configuration. The duration is also reported in minutes. If Limbo ran out of memory or took longer than 72 hours, ∞ is reported instead.

As discussed in Section 3.4.1, Limbo's heuristics depend on three parameters, `max-branches`, `max-forks`, and `max-indjumps`. To better understand the effect of these parameters, we repeatedly ran the experiment from Section 4.3 while varying these parameters. Because performing these tests is computationally expensive, we selected one program that Limbo performed well on, `b2sum`, and performed all tests on it. The results are not representative, but still provide insight into the parameters.

Each row in Table 5 represents a configuration where one of Limbo's parameters is varied from the default setting: `mb`=n denotes `max-branches` = n, *mf*=n denotes `max-forks` = n, and `mi`=n represents `max-indjumps` = n. In these experiments, Limbo clearly performs best when `max-branches` = 0, suggesting that exploring alternate branch configurations is not worth the cost. We hypothesize that this is because the opportunity cost for exploring a branch is often exploring a different indirect jump target, which can result in a wider variety of code behaviors. Varying `max-forks` did not consistently affect the results. We believe this is because Limbo's default configuration does not explore symbolic branches, which is the leading cause of forking. Finally, varying `max-indjumps` shows the tension between goal complexity and search space. When `max-indjumps` ≤ 2, Limbo fails to find more complicated goals such as `execv`, `s-mem` and `l-mem`. This is likely because there are few (or no) instruction sequences of that length in the program that can accomplish those goals. When `max-indjumps` is set to 3, it finds attacks for both `execv`, `s-mem` and `l-mem`. But as `max-indjumps` is increased again, it begins to struggle. This is because the number of executions scales exponentially with the number of indirect jumps. We have found max-indjumps = 3 to be a reasonable balance between being feasible to search, and long enough to find interesting goals.

4.5 RQ4: Can Limbo discover new techniques?

In this section, we investigate whether Limbo can discover new techniques for constructing code reuse attacks. To answer this question, we examine in detail an attack that Limbo found in tput (9.6 KiB) for the s-mem goal. In tput, this goal attempts to write 42 to address 0x804b000. Since Limbo is not constrained to known techniques or subclasses of code reuse attacks, it is reasonable to expect that it will use some unusual or novel strategies. Although Limbo can use code sequences that do not end in a ret instruction, it often uses ret instructions since they are common and make it easy to transfer control from one location to another. This attack uses three such sequences.

Limbo starts its attack by jumping into __libc_csu_init, which is responsible for invoking constructors on shared libraries. This function does a lot of extra work, including calling __init, but eventually it executes the following assembly code, which allows Limbo to symbolically control %ebx, %esi, %edi, and %ebp by loading values from the stack, which Limbo controls.

```
0x8049a5c: pop    %ebx
0x8049a5d: pop    %esi
0x8049a5e: pop    %edi
0x8049a5f: pop    %ebp
0x8049a60: ret
```

Limbo next uses the ret instruction to jump to use_env@plt, the Procedure Linkage Table (PLT) entry for use_env. PLT entries are stub functions inserted by the compiler to dynamically resolve symbols in shared libraries on demand. As expected, jumping into the PLT entry invokes the dynamic linker, which runs for several hundred instructions. Eventually, the address of use_env in libtinfo.so is resolved, and use_env starts to execute.

use_env is a relatively simple function that ends with the following instructions:

```
0xf7fa9285: add    $0x16d7b,%ecx
0xf7fa928b: mov    0x4(%esp),%eax
0xf7fa928f: mov    -0x1c(%ecx),%edx
0xf7fa9295: mov    %al,(%edx)
0xf7fa9297: ret
```

The second instruction reads a value from the stack, which Limbo controls, and puts it into %eax, giving Limbo symbolic control over %eax. This is the primary reason why Limbo called use_env.

Most code reuse attacks avoid reusing code from shared libraries because their locations are randomized by ASLR (Section 2.3.1). But Limbo figured out that it can safely reuse code from functions in shared libraries that are dynamically linked to the binary by calling the function's corresponding PLT stub. We believe that this is the first documented example of a code reuse attack that employs a gadget in a shared library by calling a PLT stub. We call this *reusing dynamically linked gadgets*. This is a good example of how Limbo's generic design allows it to discover code reuse attacks that employ creative means to reach the goal state. In this case, since there was very little code available in the tput binary, Limbo started using dynamically linked gadgets.

Last but not least, Limbo returns to a regular gadget in tput:

```
0x8048f7c: mov    %al,0xa4(%ebx)
0x8048f82: add    $0x14,%esp
0x8048f85: pop    %ebx
0x8048f86: pop    %esi
0x8048f87: ret
```

The first instruction takes the lower eight bits of %eax, which Limbo just gained control of, and writes them to memory at the address %ebx + 0xa4. Limbo controls %ebx and sets it to 0x804af5c, since 0x804a5fc + 0xa4 = 0x804b000, the address Limbo is trying to write to. Finally, Limbo also controls %eax and sets the lower eight bits to 42, which is what it wanted to write to memory.

After all is said and done, this attack executes 883 instructions which were reused from the tput binary and libtinfo.so, a shared library used by tput. The fact that Limbo automatically employed a new strategy (reusing dynamically linked gadgets) shows that the generic approach that Limbo takes can allow it to find new attacks and strategies that other automated systems may not find.

5 LIMITATIONS

The goal of our work is to discover any code reuse attack should one exist, but there are limitations of both our *technique* and *implementation* that can prevent it from doing so.

5.1 Limitations of Concolic Execution

Our technique builds upon concolic execution, which has several limitations. It can fail because constraints take too much memory or time to solve, or because it runs out of disk space when trying to store reachable states. The most fundamental limitation of concolic execution, however, is that it is a path-based analysis, and most programs have too many paths to exhaustively explore. Thus, in practice, our technique *searches* for an attack, and as a result, its effectiveness depends on how tractable this search problem is.

A smaller search space helps make the search problem tractable. Since the state space tends to explode at symbolic indirect jumps, Limbo performs best on small programs and programs protected by defenses such as CFI, which both limit the possible state transitions at indirect jumps. The other major factor in tractability is goal complexity. Limbo's heuristics help make the search problem more tractable in part by *bounding* the state space that is searched. As our evaluation shows (Section 4), many real-world attacks can still be found in the bounded space. Unfortunately, there are more complex goals that are unlikely to be found in this limited search space, such as a multi-stage computation that writes to five memory addresses. Bounding can also prevent us from finding entire classes of attacks, such as COOP [26], which requires the analysis of loop iterations that Limbo's default parameters exclude. In summary, Limbo works best with simple goals and very restricted code. Interestingly, this is very complementary to most existing work [18, 28, 34], which tends to perform better when given fewer restrictions. In the future, we plan to investigate a hybrid implementation of Limbo that is informed by static gadget discovery tools such as angrop, BOPC and Q for scalability [18, 28, 34].

5.2 Implementation Limitations

Limbo is limited to analyzing executables that its underlying concolic executor, Mayhem [7], supports, which are 32-bit x86 Linux executables. Our state reachability techniques are not inherently architecture specific, however. It should be possible to implement our techniques on other architectures and we are currently investigating the feasibility of porting Limbo to support AMD64 Linux executables. Because the code available for reuse may vary greatly on other architectures, we cannot predict how effective our technique will be at finding attacks in practice. For example, our heuristics may not be as effective at identifying promising states on another architecture.

Another limitation of Limbo is that it does not find information leak vulnerabilities on its own. As many systems are now randomizing all code addresses (Section 2.3.1), this poses an additional challenge for constructing code reuse attacks. On these systems, the attacker must either determine the memory layout using an information leak vulnerability, or ascertain it by brute forcing all possible memory layouts. This is true for manual attacks as well.

6 RELATED WORK

Much of the work related to this paper is about the development of code reuse attacks and defenses, which we summarized in Section 2. In this section, we focus on *automated* techniques for discovering and reasoning about code reuse attacks.

6.1 Automation of Code Reuse Attacks

There are several existing efforts to automate code reuse attacks. The earliest systems did not consider defenses beyond ASLR and NX, and attempted to automate the construction of ROP attacks by reusing code from large libraries such as libc [24] and mobile support libraries [19]. Roemer [24] built a system which allowed users to search for gadgets by issuing *pattern matching* queries over assembly code sequences. Kornau [19] also designed a system that identified gadgets by patterns, but instead of matching syntactic assembly instructions, his system matched patterns to a tree-based semantic representation. Q [28] was one of the earliest systems to automate the construction of ROP attacks using much smaller amounts of binary code. Unlike prior work, it was not based on pattern matching and used software verification techniques to test whether an instruction sequence implemented a useful semantic action.

Wollgast et al. [34] were the first to develop an automated system to find defense-aware code reuse attacks. Specifically, their system can produce code reuse attacks in the presence of coarse-grained CFI (Section 2.3.2) by automating the Call Oriented Programming (COP) technique that was proposed previously [6, 10, 16]. The basic idea of COP is to utilize *call-site* gadgets, which immediately follow a `call` instruction, and *entry-point* gadgets, which occur at the entry point of a function, since these gadgets are permitted by coarse-grained CFI. The actual mechanism to identify and combine gadgets is similar to the one used in Q [28].

BOPC [18] is a system to automate the construction of code reuse attacks that are aware of fine-grained CFI. BOPC programs are written in the SPL language. BOPC first identifies *functional* blocks, which are basic blocks in the target binary that can implement the behavior described in one of the SPL statements. Functional blocks are then connected by assigning *dispatcher* blocks, which is an NP-hard problem.

All of these systems can automatically discover code reuse attacks. However, unlike Limbo, they are not *generic*. Instead, they each automate a fixed strategy that was already known. For example, the system from Wollgast et al. can only find COP attacks, which bypass coarse-grained CFI, and BOPC only automates DOP attacks, which are aware of fine-grained CFI.

6.2 Evaluation of Code Reuse Defenses

Newton [33] is a tool for evaluating the effectiveness of code reuse defenses such as CFI. Like Limbo, Newton infers information about the possibility of code reuse attacks by leveraging dynamic binary analysis. Newton analyzes a single execution of the target program and outputs a list of *gadgets* that it believes are callable by the attacker. It contains a built-in language for defining constraints which allows it to model both static and dynamic defenses. Newton's authors employed Newton to count the number of gadgets that were accessible in several programs under various defenses, which serves as a metric for how effective those defenses are. They were able to use this output to manually construct attacks on `nginx` that would function with some of the strongest defenses available.

There are many differences between Newton and Limbo, however. First, Limbo is an automated tool that produces code reuse attacks. Newton can provide its user with information that will no doubt be helpful to construct a code reuse attack, but it cannot do so itself. Second, Limbo uses concolic execution, and Newton employs *taint analysis*. Taint analysis can be thought of as a simplified version of symbolic execution. Instead of representing each program location symbolically (e.g., `%eax` contains $s4 + 8$), it keeps track of which program locations are *tainted* by the attacker's symbolic input to the program (e.g., `%eax` is affected by $s4$). More importantly, taint analysis can only reason about the current execution path; unlike concolic execution, it cannot explore adjacent paths. This means that if a gadget was only accessible via a different path, Newton would not detect it, unlike Limbo. Nevertheless, we believe that Newton was the first system to reason about multiple code reuse defenses, even though it requires explicit models of those defenses.

7 CONCLUSION

In this paper, we proposed a generic framework for automatically identifying defense-aware code reuse attacks. Unlike existing work, which utilizes hard-coded strategies for specific defenses, our framework can produce attacks for multiple defenses by analyzing the runtime behavior of the defense. We implemented our framework in a tool called Limbo by making a small number of modifications to an existing binary concolic executor. We evaluated Limbo and demonstrated that it excels when there is little code available for reuse, making it complementary to existing techniques. We showed that, in such scenarios, Limbo outperforms state-of-the-art tools for both automatically identifying DOP attacks in the presence of fine-grained CFI, and automatically producing ROP attacks.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security* 13, Article 4 (November 2009), 40 pages. Issue 1. https://doi.org/10.1145/1609956.1609960

[3] angrop [n.d.]. angrop. https://github.com/salls/angrop. Checked 8/11/2020.

[4] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer, and Communication Security.*

[5] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the USENIX Security Symposium.*

[6] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the USENIX Security Symposium.*

[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[9] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. 2009. DROP: Detecting Return-Oriented Programming Malicious Code. In *Proceedings of the International Conference on Information Systems Security.*

[10] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the USENIX Security Symposium.*

[11] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2009. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the ACM Workshop on Scalable Trusted Computing.*

[12] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the USENIX Security Symposium.*

[13] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[14] Patrice Godefroid, Michael Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium.*

[15] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[16] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[17] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[18] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[19] Tim Kornau. 2009. *Return Oriented Programming for the ARM Architecture.* Master's thesis. Ruhr-Universität Bochum.

[20] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. 2010. Defeating return-oriented rootkits with "Return-Less" kernels. In *Proceedings of the European Conference on Computer Systems.*

[21] Le Dinh Long. 2010. Payload already inside: data re-use for ROP exploits. https://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf. Checked 8/11/2020.

[22] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[23] PaX Team [n.d.]. PaX non-executable (NX) pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt. Checked 8/11/2020.

[24] Ryan Glenn Roemer. 2009. *Finding the Bad in Good Code: Automated Return-Oriented Programming Exploit Discovery.* Master's thesis. University of California, San Diego.

[25] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. 2009. Surgically Returning to Randomized lib(c). In *Proceedings of the Annual Computer Security Applications Conference.*

[26] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[27] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy.*

[28] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the USENIX Security Symposium.*

[29] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security.*

[30] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Maril Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[32] Ubuntu security 2020. Ubuntu Security/Features. https://wiki.ubuntu.com/Security/Features?action=recall&rev=154. Checked 8/11/2020.

[33] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[34] Patrick Wollgast, Robert Gawlik, Behrad Garmany, and Benjamin Kollenda. 2016. Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets. In *Proceedings of the European Symposium on Research in Computer Security.*

[35] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[36] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX Security Symposium.*