 Received 21-08-2024; Accepted 26-12-2024; Published 02-01-2025

Overview of the Code-Reuse Attacks Mitigations, and Evaluation using SMAA-2 Approach

Ayman M. El-Zoghby ayman.m.elzoghby@gmail.com
School of Information Technology and Computer Science,
Nile University, Cairo, Egypt

Mahmoud Said ElSayed eng.mahmoud101@gmail.com
School of Computer Science, University College Dublin, Belfield, Dublin, Ireland

Anca Jurcut
School of Computer Science, University College Dublin, Belfield, Dublin, Ireland

Marianne A. Azer
School of Information Technology and Computer Science,
Nile University, Cairo, Egypt
National Telecommunication Institute,
Cairo, Egypt

Corresponding Author: Mahmoud Said ElSayed

Abstract

Exploiting modern software requires sophisticated attack vectors to bypass software protection mechanisms. Code-reuse Attacks (CRAs) are a widely used approach to attack modern software, even after applying memory protection defenses. The underlying vulnerabilities in the software codes or design enable the use of the program's own code and manipulation of data and code. This paper covers the foundation of memory-based attacks and provides an extensive overview of memory safety issues and exploitation methods, as well as the foundation of control flow attacks and different categories of code-reuse attacks. The focus is on the differences between the various methods employed to mitigate code-reuse attacks. We apply an analysis technique to the covered CRAs to assist in the ranking and evaluation process. The chosen decision-making technique is SMAA-2, which is employed to analyze the mitigation defenses and techniques. This novel approach to evaluating CRAs mitigations helps Decision-makers (DM) in selecting specific CRA techniques over others.

Keywords: Evaluation, CRA, Exploit mitigation, MCDA, Memory-based attacks, SMAA-2, Software security

Citation: Ayman M. El-Zoghby et al. Overview of the Code-Reuse Attacks Mitigations, and Evaluation using SMAA-2 Approach. Advances in Knowledge-Based Systems, Data Science, and Cybersecurity. Research 2024;2(1):6. Pages: 108-148

1. INTRODUCTION

Protecting the software memory is a core defense against memory manipulation techniques [1], used by attackers to exploit software codes. The corruption of memory space was first demonstrated in [2], regarding stack smashing. A series of attacks and defenses followed later, based on the same concept of protecting and manipulating stack memory space [3]. Novel methods, such as code-reuse attacks [4], and control-flow attacks [5], were introduced to bypass memory protection defenses like DEP [6], and ASLR [7]. Relying on loosely-typed programming languages such as (C, C+) and error-prone coding practices, along with the absence of code audits [8], are the main reasons modern software is vulnerable to memory-based attacks.

Software protection is mandated by well-known regulation bodies and mentioned explicitly in professional cyber security standards such as the EU Cyber Resilience Act, NIS2 Directive, and GDPR. Moreover, control objectives of the ISO/IEC 27002:2022 [9], cover the application security requirements from the Software Development Life Cycle (SDLC) point of view and put a strong emphasis on software protection against explication by putting sound SDLC practices into effect when developing new software. The bad design by-product is manifested in a vulnerability, such as CVE-2018-5392 [10], that can be exploited using one of the techniques of ROP.

Code-reuse attacks are extensively discussed in this paper, with a special focus on the mitigations proposed by researchers to limit or prevent the impact of code-reuse attacks. An in-depth analysis of the different mitigations is needed to provide an informed decision when selecting between them. The analysis methods are built upon evaluation criteria and a weighting system, in order to lay out a methodical approach for decision-making. Data-oriented memory attacks are out of the scope of this paper; our main focus is control-flow attacks. The background section of this paper covers the memory safety issues that make control-flow manipulation possible. Data-oriented attacks are possible after manipulating the non-control data (e.g., non-protected variables or code pointers) in the vulnerable code memory segment [11]. Discussing memory protection and defenses while presenting the differences between code-reuse attacks and code-injection attacks are core parts of our research. This will lay out the basic foundational blocks for the concepts related to attack mitigations and also highlight the challenges faced by researchers when they introduce a new protection mechanism.

This research provides an overview of the memory safety issues that provide attackers ways of manipulating codes or data stored inside the designated memory of the vulnerable software. The core memory-protection defenses will also be highlighted in this paper. A brief introduction to the computer memory architecture is mentioned in this paper, followed by the concerned memory attack categories, code injection, and code reuse mechanisms.

The main contribution of this paper is to lay out the foundation and comprehensive overview of the topic of code-reuse attacks, organize the different types of code-reuse attacks. For a better selection approach for the appropriate code-reuse technique after analyzing their internal working mechanism and its results, a Multi-Criteria Decision Making (MCDM) method called Stochastic Multicriteria Acceptability Analysis (SMAA) [12], is being used. The main challenge when comparing the mitigation techniques of code-reuse attacks is the lack of one or more of the experimental results in terms of technique universality or effectiveness. This led us to adopt a stochastic approach to overcome some of the unknown values of the experimental results.

The contributions of this paper are as follows:

- Novel application of a MCDA approach named SMAA-2 on the CRAs mitigations in order to provide accurate ranking and evaluation of the different CRAs mitigation techniques.
- Laying the foundational memory safety issues and attacks and linking them to code-reuse attacks and code-reuse mitigation techniques.
- A detailed comparison of the code injection attacks and mitigation.
- A structured organization of the defenses of the code-reuse attacks.
- In-depth analysis and discussion of twenty code-reuse attack mitigation techniques.

The remainder of this paper is organized as follows. In section 2, the computer memory architecture, memory safety issues and attacks are presented. This Section also covers the categorization of the code-reuse attacks. Section 3 provides a comprehensive analysis of the code-reuse mitigations. In Section 4, the decision-making technique, SMAA-2 is discussed and linked to the selected evaluation criteria of the presented code-reuse defenses. Section 5 discusses the SMAA-2 analysis outcome results. Finally, conclusions and future work are presented in section 6.

2. BACKGROUND

This section covers memory safety issues, attacks, and protection techniques. It provides a brief description of the computer memory structure. Additionally, comprehensive details about the code injection attacks and the different mitigation techniques proposed in the literature, such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), are presented. The computer memory structure, memory safety issues, memory safety enhancements, and memory control flow integrity attacks are presented in sections 2.1, 2.2, 2.3, and 2.4, respectively.

2.1 Computer Memory Structure

In this section, we discuss the stack frame architecture since most code-reuse attacks take advantage of the function's stack frame. We also touch on the function creation and destruction procedure, as it reflects on how attackers are using those procedures to hijack the control flow of the running program. This subsection highlights the structure of a typical stack in a binary. Understanding stack operations is vital to mount successful code-reuse attacks."

2.1.1 Stack frame

The stack is a part of the program memory where each function saves its own parameters and arguments. The stack works in a Last-In-First-Out (LIFO) style, with data being pushed into the stack using the assembly instruction PUSH and removed from the stack using the assembly instruction POP. The stack's design and implementation can vary depending on the CPU architecture and

compiler design. Each function or subroutine inside the program has its own allocated stack area, which is called the "Stack Frame" [13].

A general-purpose register is assigned by the CPU to address the stack space. The Extended Stack Pointer (ESP) is used to point to the upper point of the stack, while the Extended Base Pointer (EBP) is utilized to point to the base address of each stack frame, acting as an anchor for that frame. The stack is divided into three main sections:

1. **Arguments Section:** where all the arguments used by the calling function are saved.
2. **Return Address Section:** where the return address of each function is saved. This is used to direct the control flow to the next section of the program after the current function finishes its execution. Additionally, it holds the base pointer address, which each function uses to refer to local variables and arguments on the stack frame.
3. **Local Variables:** this section holds the local variables used by the calling function, such as integer values or local buffers. The base pointer and stack pointer are saved in the general-purpose registers of the CPU named EBP and ESP, respectively.

FIGURE 1, depicts the stack layout and how different stack elements are arranged. This figure shows two stack frames: the first one has a local buffer segment to hold the function's local variables, the saved base pointer of the previous frame, and some arguments passed to that function.

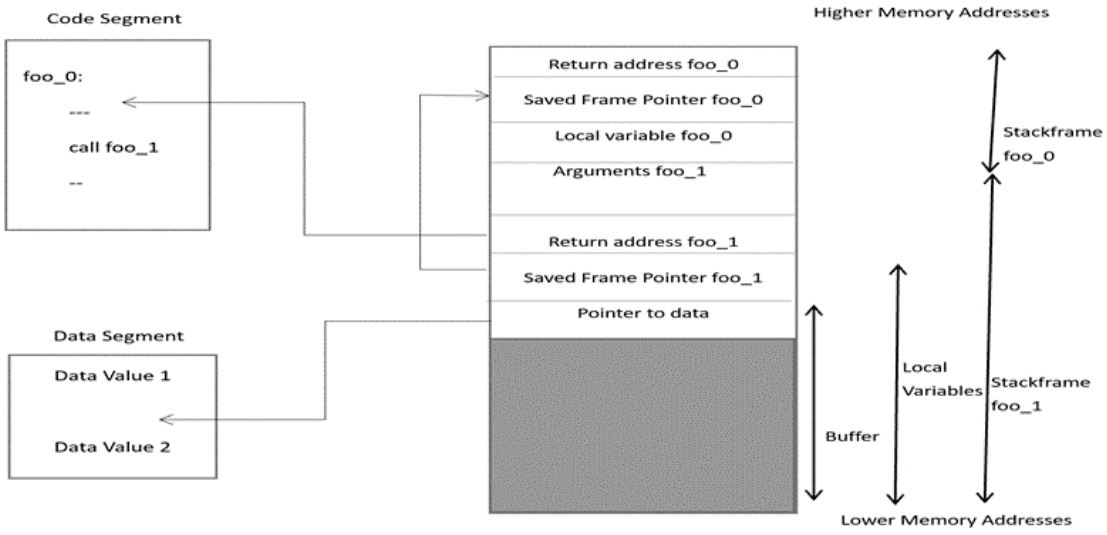

Figure 1: Stack frame layout

2.1.2 Function Epilogue and Function Prologue

Calling conventions are standardized strategies for compilers to implement in order to facilitate function calls by the machine. A calling convention designates the plan that a compiler sets up to invoke a subroutine. Calling conventions indicate how parameters are passed to a specific function

and how return values are retrieved. They also determine how a function is called and how the function manages the stack. In brief, the calling convention indicates how a call in C or C++ is translated into proper assembly instructions. The way to construct and destroy a function is known as an epilogue and prologue action. By default, the C compiler uses the CDECL calling convention, where the calling function needs to clean the stack after being called.

FIGURE 2, shows the stack of a simple function and the building and destroying routines taken by the compiler to load functions. When the below function is compiled by the GNU GCC compiler, the generated assembly code will have an epilogue and prologue routine to clean the stack after the function finishes its execution

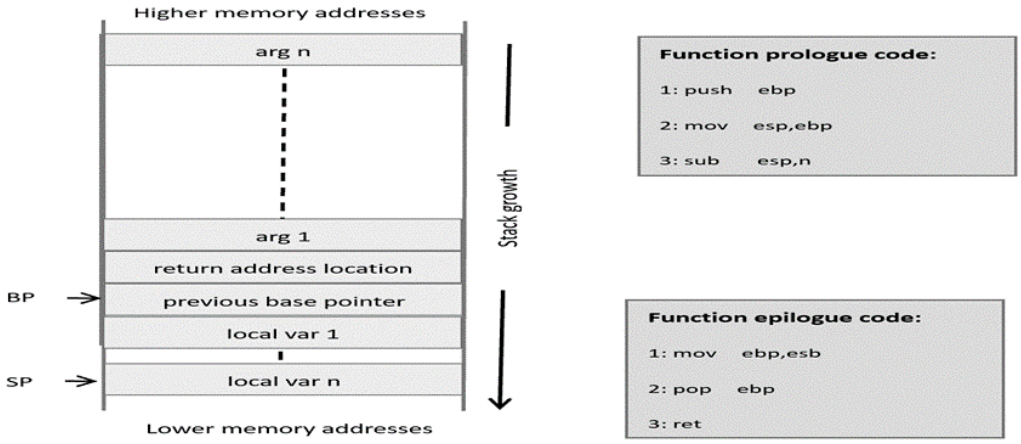

Figure 2: Function prologue and epilogue.

2.2 Memory Safety Issues

In this section, we will discuss the major memory safety issues and the resulting control-flow attacks. There are three main groups of different memory safety issues. These issues are the main root cause of attacks against the program's code, which resides in different memory sections such as the stack, heap, or unallocated space.

2.2.1 Buffer overflow (Spatial memory safety)

Buffer overflow is a popular and common security vulnerability; its popularity arises from the fact that it can give remote attackers the ability to take over a remote host [14]. It allows the offender to inject arbitrary code inside a privileged process and then execute that code to take control of the process and subsequently the host. Code injection is initially used to perform buffer overflow attacks. However, if the attacker's malicious code can be found in the program's address space, then they can only change a pointer in the stack or heap to point to this code (e.g., code reuse). Defenses against buffer overflow attacks range from writing proper code with input checks and validating those parts of the buffer as non-executable ($W \oplus E$). Several defenses have been proposed

to protect the integrity of pointers while being dereferenced; nevertheless, these defenses will not stop the attack completely, as seen in stack guard [15].

Algorithm 1 is an example of a Pseudocode algorithm of a vulenrable code to BoF.

Algorithm 1: Buffer Overflow Example

Input: argc, argv
Output: None
if *argc < 2* **then**
    Print “strcpy() NOT executed....”;
    Print “Syntax: <program_name><characters>”;
    Exit;
Define buffer as an array of 5 characters;
Copy argv[1] to buffer using strcpy(buffer, argv[1]);
Print “buffer content= ” + buffer;
Print “strcpy() executed...”;

2.2.2 Use-after-free (Temporal memory safety

Programmers use pointers in low-level languages to manipulate memory efficiently; however, this can lead to security compromises [16]. Some pointers can be dereferenced after they have been freed from their object, which leads to undefined behaviors; these pointers are called ”Dangling pointers,” which can be employed in a type of attack known as use-after-free. The attacker needs to know the exact location of the dereferenced pointer, the location of the object it was pointing to, and the instruction to read or write in that pointer location. Defenses against such attacks mainly depend on run-time detection of the invalid dereferencing of an expired pointer.

2.2.3 Type-confusion

System and application programming languages such as C and C++ don’t enforce type-safety practices; they depend solely on the developer’s intent to be able to change the type of a variable whenever needed (i.e., typecasting). In C++, developers can convert a pointer of a base class to a derived class. This unsafe practice leads to using data inside the subclass as a pointer to different memory contents, such as virtual table (vtables) contents in C++[17]. This problem is called ”type-conversion” and requires constant checking of the object types, which in turn introduces high-performance overhead and low code coverage[18].

Attackers need to inject different codes into the victim’s program memory space to take control of the victim’s program. Code injection can take place either by writing in a code segment or overwriting a pointer to a code segment. If attackers want to change the control flow, then a code pointer gets dereferenced (*&C). We can observe from our discussion about memory attacks that they greatly depend on pointer manipulation. Pointers are a way to access certain memory spaces or memory objects [19]. Pointers are used to address code or data sections. If the pointer is used to point to a control-data structure, such as (switch code), then its integrity needs to be protected [20].

Pointers have type, validity, and specific size assigned to them; they can assign these features to other pointers during the program's lifetime. Each of the capabilities of the pointers can be secured using relevant safety measures. Examples include type safety to check the validity of pointer dereferencing while the object stays allocated and special memory safety to ensure the pointer is only allowed to access data inside the assigned memory object. One method to protect against injection is done via protecting the target program's memory pages (e.g., virtual memory) by assigning different permissions such as "writable, executable, readable" to the virtual memory pages. The program's data segment should be mapped to a read-only virtual memory page, and the code segment should be mapped to executable and read-only virtual memory pages.

Code integrity can be preserved by combining non-executable permission with an XOR'd writable permission [21]. To protect against the threat of executing data as code, Andersen [21] suggested the Data Execution Prevention (DEP) mechanism to prevent an attacker from running their injected data as code from the data memory segment. Code injection protection was proposed by [22] and employs the same concepts used to protect against data injection, such as Software Fault Isolation (SFI). Code injection protection is not sufficient to protect against attackers or software that uses dynamic loading of their associated libraries or applications to change their code in runtime, such as Just-In-Time (JIT) Compilers. This implies that mitigation against code injection cannot be applied to dynamically generated code.

The most targeted data structures by attackers are the stack and the heap segments of a vulnerable program. The stack and heap segments have multiple code and data pointers that could be manipulated and overwritten to hijack the program's execution. An interesting phenomenon is the "weird machine"[23]. It takes place at the level of code translation because the bits can be interpreted as code, data, pointers to code, or pointers to data. Thus, translation relies on the semantics of the code that is defined by the coder. This concept makes exploiting code and data possible since attackers may use a data value of a function argument as an instruction. Attackers use the concept of gadgets to be able to exploit memory safety issues. Gadgets are collections of assembly instructions that usually contain indirect control-flow instructions, such as RET, CALL, JMP. Gadgets exist inside the program code space and can be stitched[20] together to form a Turing-complete program. Turing-complete code is able to perform any arithmetic or logic operations. Gadgets' existence inside any code is possible due to the variable length of instructions that are enabled by different processor architectures, such as Intel x86.

2.3 Memory Safety Enhancement

Mitigations against the mentioned memory safety issues are proposed to protect sensitive data inside the binary, such as flags or code pointers, by keeping them confidential. Randomizing certain memory segments can deter attackers by hiding the location of different regions of the program's binary, such as shared libraries, dynamically-linked libraries, code segments, or data segments. Attackers may resort to side-channel attacks, such as information leakage, to know the address of certain libraries, which allows them to calculate the relative addresses of heap or stack segments [24]. Randomization can take place on a memory-block level (e.g., ASLR) or instruction-level [25]. Many researchers focused on randomizing the code at a fine-grained level by combining permutation concepts on top of per-processor per-system randomization [26], [27].

Another memory protection approach is to focus on protecting the integrity of the code data flow. Data-flow integrity checks every time data is read from memory and verifies if it's corrupted or malicious data. The same applies for control-flow integrity, with the aim of limiting the control transfer to only benign targets (e.g., pre-approved). The targets can be called by indirect function call (e.g., forward edge) or by direct function call (e.g., backward edge).

The main challenge is that memory defenses such as immutable code, immutable jump table, or immutable vtables in C++ can only hold true during code compile-time. During code run-time, many components interact with the software, such as the compiler, assembler, loader, or linkers. This interaction can weaken the software security policy and make it vulnerable. D'silva [28] discussed the negative impact of compiler optimization on protected code. The compiler tends to remove added security checks to improve program performance without changing the semantics of the program.

2.4 Memory Control-Flow Attacks

The memory safety issues may lead to different types of attacks. A buffer overflow error discussed earlier is considered a control-flow attack because it changes the originally intended control flow of the binary to an unintended malicious control flow branch. Control-flow attacks normally exploit memory error vulnerabilities due to bad coding practices or poor run-time security of the binary itself. The control-flow attack can be considered a 'Run-time Exploit' because it takes place at run-time to give the attacker instant exploiting capability in order to abuse a program's error and divert the control flow to an arbitrary code block. These types of attacks can be utilized as a preamble for mounting a post-exploitation activity such as malware infection or host takeover. Here we lay the foundation of two control-flow attacks that are based on code manipulation. Data manipulation can also lead to control-flow attacks; however, we don't cover attacks that result from manipulating data in our paper.

2.4.1 Control-flow attacks foundation

Control-flow attacks can be categorized into two distinct classes: (1) code injection and (2) code-reuse attacks. Code injection works by injecting malicious code directly into the vulnerable memory space in order to perform a malicious activity. On the other hand, code-reuse works by utilizing existing bits of code (i.e., code snippets) to form a sequence of malicious commands that can be used to perform arbitrary computations. To understand how control-flow attacks take place, the program binary is presented as code blocks; these code blocks are named Basic Blocks (BBLs). The BBLs are sequences of assembly instructions with unique entry and exit instructions. Each BBL is connected to the previous and the following block with forward and backward edges. The forward edges are normally represented by exit instruction functions, which can be direct jumps, returns, indirect jumps, or indirect call instructions. The backward edges are represented by the entry instruction of each BBL. This setup of code blocks and their way of functioning form a control-flow graph (CFG). In the following, we present code injection attacks and code-reuse attacks.

A) Code Injection Attacks: FIGURE 3, illustrates the CFG of a simple code injection attack. In this figure, the original CFG has 6 nodes starting from N1 to N6, each node has an exit instruction

that creates a forward edge to the next connected node. For an attacker to successfully inject the code and run it, two conditions need to be satisfied. The first condition is ensuring the malicious code can fit into the buffer area of the program's memory space. The second condition is exploiting a vulnerability to hijack the control flow of one block and redirect it to the malicious code nodes. The code injection attack can be considered a subclass of the control-flow attacks since it changes the benign control flow to the attacker-injected nodes. Code injection can be prevented by allowing only code to be executed rather than allowing data or code to be executed. If data execution is prevented due to this CPU architecture security feature by design, then attackers resort to a different method when trying to compile a code to be executed in the victim process space [3]

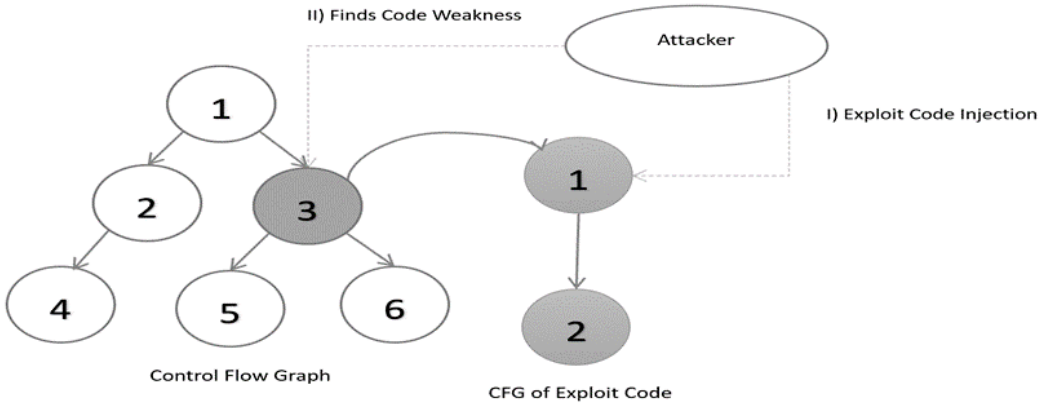

Figure 3: Code injection attack

B) Code-Reuse Attacks: A new approach has been introduced to overcome the data execution prevention mechanism implemented in nearly all processor CPU architectures [29]. It can be considered a subclass of the control-flow attacks category, where the attacker's focus is to re-use some parts of the program codes, which mainly reside inside shared libraries, to mount a successful attack without the need to inject new code into the victim process image. Code-reuse attacks are sophisticated because they are dependent on the CPU architecture and require special knowledge of useful assembly sequences to create functional code sequences. In the next section, we describe in detail the CPU layout, memory structure, and assembly sequences that are fundamental for successful code-reuse attacks

3. RELATED WORK

This section covers the attacks on code either using code injection or code reuse. The mitigations proposed to defend against code injections are discussed in detail. Moreover, we explain in detail how the control-flow integrity concept can be used to protect against the code reuse type of attacks.

3.1 Code Injection Attacks and Mitigation Overview

This section covers the different basic mitigations proposed to prevent code injection attacks. It also discusses the evolution of code injection attacks and their progression to attackers reusing existing code to mount attacks on code protected against code injection.

To explain code injection, we describe the vanilla-style buffer overflow attack. In this type of attack, the attacker uses techniques to determine the available buffer size inside the stack and uses this buffer to store the code they will inject for later execution. The attacker can stream or copy their malicious code using different types of inputs, such as network traffic packets, file contents, or a stream of bits inside a string. This input needs to be transferred to the selected vulnerable buffer; hence the attacker needs to know the stack layout and the values of ESP, EBP, and the overall stack size. In a buffer overflow, the attacker overwrites the content of the local buffer with their malicious code until they reach the ”return address” word and overwrite it with an arbitrary new address that points to their malicious code. Conveniently, this malicious code is often designed to spawn a command prompt, i.e., a shell, such as a command shell in Windows or a BASH or SH shell in Linux. The attacker can use various tools to obtain information about different register contents and memory layout details. These tools can range from a simple debugger, such as GNU GDB or Immunity Debugger.

FIGURE 4, depicts how a vanilla-style buffer overflow attack works, as it is the simplest form of code injection [15]. The main idea of this code injection is to redirect the control flow to the newly injected code. This is possible in our example because there is no memory protection in place, and the program code itself has a vulnerability that allows any input data to be accepted and processed without filtration.

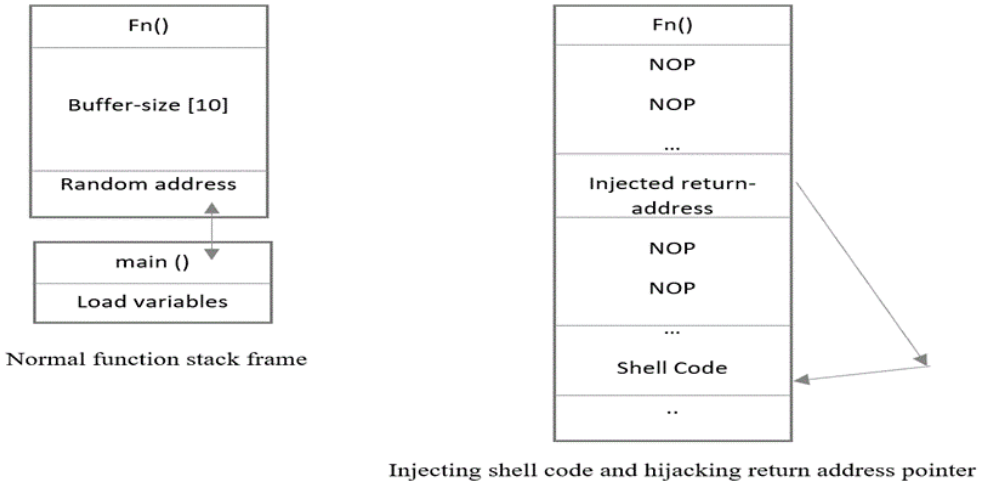

Figure 4: Buffer control flow attack using stack smashing technique.

3.1.1 Data execution prevention

What makes exploitation possible is the concept of weird machines [30]. Since code and data are intermixed in the computer memory as per the Harvard-based CPU architecture, the attacker can take advantage of this feature. Data segments such as Heap and Stack are marked by the CPU as readable, writable, and executable (RWX). This makes the stack exploitable since its intended use case is to store data, not code. Because the stack area is writable, the attacker can use this space to store code instead of data. This code is to be executed later by the CPU due to the mis-distinction between code and data. A mitigation for this threat vector is to introduce a mechanism to make the buffer areas that hold data only readable and writable but not executable. This was possible after patching the OS kernel in the Windows XP SP2 operating system and the Solaris 2.6 operating system [13]. The non-executable enforcement needs support from the hardware, e.g., CPU support, by introducing a special bit to mark certain memory pages as non-executable, and this bit is called a no-execute bit (NX).

Data Execution Prevention (DEP) has been introduced by Andersen [31] to enforce the security concepts of non-executable data areas. This makes the memory pages that contain data both writable and readable (RW) and memory pages that contain data marked as code to be readable and executable (RX). This approach is commonly referred to as Writeable XOR eXecutable ($W \oplus E$) and it effectively prevents code injection attacks. The introduction of software-based DEP and Hardware support for DEP [6] didn't stop attackers from devising more complex attacks in order to execute arbitrary code. Instead of relying on injecting code, they resorted to using existing code in program-shared libraries or code segments to stitch some code sequences to mount a successful memory corruption attack. Code-reuse attacks can be considered a generic version of the so-called Return-into-Libc (RILC) attack, which was the foundation of what became known as a return-oriented programming attack

3.1.2 Return-into-libc

Alexander Peslyak, A.K.A. solar designer, introduced the first implementation of a Return-Into-LibC (RILC) attack in [32]. In this attack, the exploit code overwrites the original return address with the address of a commonly used shared library function inside GNU/Linux libc. Libc is the standard UNIX C library that contains many functions that can be useful in exploitation, such as system(), open(), and exit(). In code injection, the attacker's main goal is to spawn a new administrator shell from within the vulnerable program on the target machine to take over the whole machine. By utilizing the code stored in return-into-libc, this can be achieved without code injection, only by pointing to the function call system("'/bin/bash'") to open the BASH shell on the victim's machine.

The main premise of RILC is the ability to redirect the original control flow of the vulnerable program to a critical function inside a shared library; hence, it's dependent on redirection to a full library, not a sequence of codes as we will explain in the return-oriented attacks technique.

FIGURE 5, depicts two states of the program's memory space. Before the attack, the local buffer has a size of 100 bytes, and the Stack Pointer (SP) is pointing to the upper point of the stack. First, the attacker takes control of the environment variables area and injects a new variable with the value

of /bin/bash to pass this value later to the system() function. It is easier to pass the values of the variables instead of scanning the program code space looking for this value, as per the code-reuse attack's first instinct. However, scanning for strings is cumbersome and may fail if the program code space doesn't contain such strings. A more dependable exploitation approach is injecting the string directly into the data memory pages and pointing to the address of this data later when needing to use it. The problem when the attacker needs to inject strings is that strings are NULL-terminated. This means a NULL byte should be sent along with the string to the program to terminate the string. Sending NULL bytes is usually not permitted and hence it breaks the exploit. One way to work around this is to send the string inside an 'environment variable' or to encode the string NULL byte and filter it after the string passes to the victim process buffer.

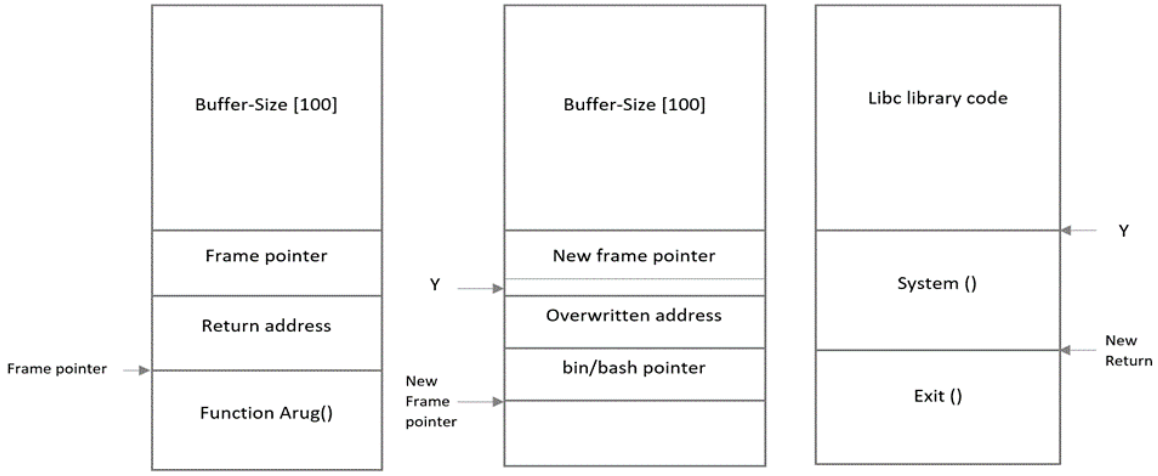

Figure 5: Typical return-into-libc attack.

Now, since the stack is protected by $(W \oplus E)$, it is readable and writable. The attacker overflows the local buffer and overwrites the addresses of the Saved Base Pointer (EBP) and then the return address with the runtime address of the system() function. Knowing the runtime address of the system() and exit() function is mandatory for the attacker; if those addresses are changed or randomized, then the attacker needs to figure out a way to know the new addresses right before he injects his data. Then, the attacker overwrites the function arguments with two addresses: one is the address of the exit() function to close the shell after the attacker is done with his attack, and the other is the address of the environmental variable $SHELL$, which has the string needed for the system() function to load the shell terminal

One of the limitations of the original return-into-libc attack is the lack of chaining function calls together to perform complex computations. Another limitation is the need to load a whole class such as libc, which contains certain useful functions. If those classes or functions are removed or obfuscated, it can render this attack useless. In [11], researchers proposed two techniques to allow command chaining, which eventually led to the concept of return-oriented programming in later stages. If the attacker wants to utilize return-into-libc in Intel 64 Bit CPU architecture, arguments will be passed using processor registers, not through the stack, which is not supported by the original implementation of return-into-libc.

There are several ways to mitigate the risk of return-into-libc attacks. One way is to instrument or eliminate sensitive functions. Another way is to map the shared libraries, such as libc, to memory addresses that hold NULL bytes [33].

3.1.3 Return-oriented programming

Return-into-libc has many limitations, as shown earlier; however, the most crippling limitation is the inability to perform unconditional branching. Only sequential function calling can be done via the return-into-libc technique. Return-Oriented Programming (ROP) was first introduced by Shacham [34] in 2007. ROP was devised to address several limitations of return-into-libc and allow more flexibility in the code-reuse sequence selection and arrangement.

The main idea of ROP is to chain together a series of short code sequences instead of using the whole class or library. These chained code sequences are conventionally called ”gadgets,” which are mainly sequences to perform specific functions, such as load, branch, move, or add operations. The difference between the code sequences used in return-into-libc attacks and ROP code sequences is that ROP gadgets are proven Turing-complete. This allows the attacker to form a full computation machine to perform any arbitrary malicious function using code sequences extracted solely from the vulnerable process image. ROP has seen wide adoption by both security researchers and attackers on many platforms, including x86 and ARM.

The name ”return-oriented” resulted from the way this attack works; because it depends on the RET instruction to invoke indirect branching in the original control flow to jump/move to the next instruction sequence (i.e., gadget). Indirect branching can also be done using instructions such as jump (JMP) and call (CALL). These types of ROP are called Jump-oriented programming and call-oriented programming.

FIGURE 6, depicts the ROP attack that takes place inside the vulnerable program’s heap space instead of the stack space. First, the attacker writes all the address pointers to all the code sequences he would like to chain together and puts those addresses inside the readable and writable memory segment like the heap. Next, the attacker uses some vulnerability to hijack the control flow to the first address of the first code sequence instead of the normal execution flow. The attacker must have a way to execute the code sequence as desired, and this means that a way is needed to slide through the addresses of the code sequences. An example of a simple gadget is provided as follows.

1. Gadget Creation

   Gadget creation is feasible due to some inherent features in Intel X86 architecture. This CPU is RISC-based and supports variable-length instructions and unaligned memory access, which allows accessing instructions from the middle or at any memory address. This creates a side-effect of unintended instruction sequences, which can be used later to form many useful gadgets.

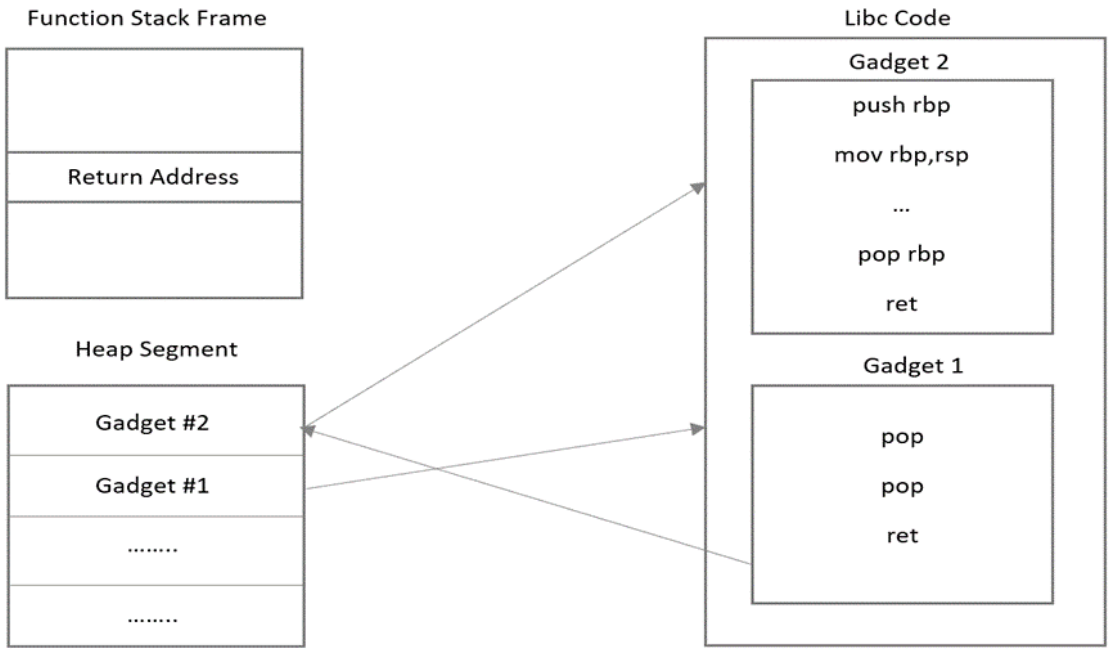

Figure 6: Code-reuse attack using ROP.

3.2 Real-World Exploits

The introduction of memory protection mechanisms, such as $(W \oplus E)$, to mitigate code injection attacks enforces attackers to use more sophisticated and combined techniques to bypass this code injection protection. An attacker can resort to a code-reuse attack to bypass the $(W \oplus E)$ protection and then use code injection, as it is much simpler than code-reuse. The processor usually alternates the marking of the memory page as writable and then later executable.

FIGURE 7, depicts an example of an attacker who employs code reuse and then code injection to control a program's memory. The memory area marked as code (non-writable) holds the program's own code and other needed system libraries. Modern OSs use the concept of lazy linking as described earlier; hence, the needed functions and libraries, as a whole, are linked to the program during run-time, and they have a reserved place in the program's memory. This means the full library, e.g., libc, is linked for a program to run a simple function like printf(), which can be misused by the attacker because other libc functions can be executed, such as system(), memcpy().

The attack steps are as follows.

(a) The attacker exploits the program and changes the control flow to the linked libraries segment.

(b) The attacker creates a new data space for his shellcode and knows its run-time address.

(c) The attacker copies the injected shellcode to the newly created data space.

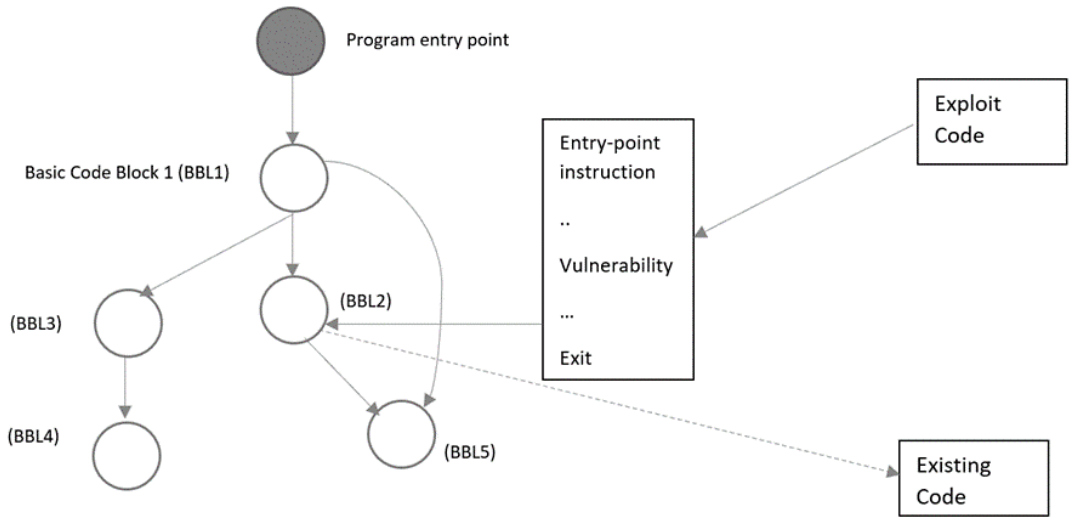

Figure 7: Basic control-flow hijacking using code-reuse and code-injection.

(d) The attacker changes the permission of the newly created data space to be executable. Calling the different functions inside the linked libraries segment is done via the ROP mechanism, while the rest of the attack is concerned with injecting the malicious code. The flowchart in Fig. 8 illustrates the attacker's steps.

3.2.1 Address space layout randomization (ASLR)

Code-reuse attacks' success is tightly linked to the attacker's knowledge of the vulnerable program's memory layout. If the memory addresses of the code or data memory space are randomized, the attacker cannot reuse existing code snippets or link them together because the addresses of the code snippets change frequently. The attacker needs to guess or leak the location of the functions and instruction sequences to bypass ASLR protection. The ASLR idea was introduced initially by Forrest et al.[35] as a method to randomize stack objects larger than 16 bytes, such as the base addresses of the stack, heap, linked libraries, and system's functions memory segment. ASLR has seen wide adoption by mainstream operating systems such as Linux[21] and Windows [36].

FIGURE 9, shows the memory space of the same program that is protected by ASLR during two different executions. The second time the program runs the start addresses of heap, stack, shared library, and executable is changed in a random manner. The operating system randomizes the program libraries during the restart of the OS, not each time the program gets loaded to the memory.

3.1.4.1 aslr limitations

32-bit systems, such as Intel x86 architecture, which are protected by enforcing ASLR, have a low entropy randomization limitation [37]. Applying brute-force attacks has proven to be useful in

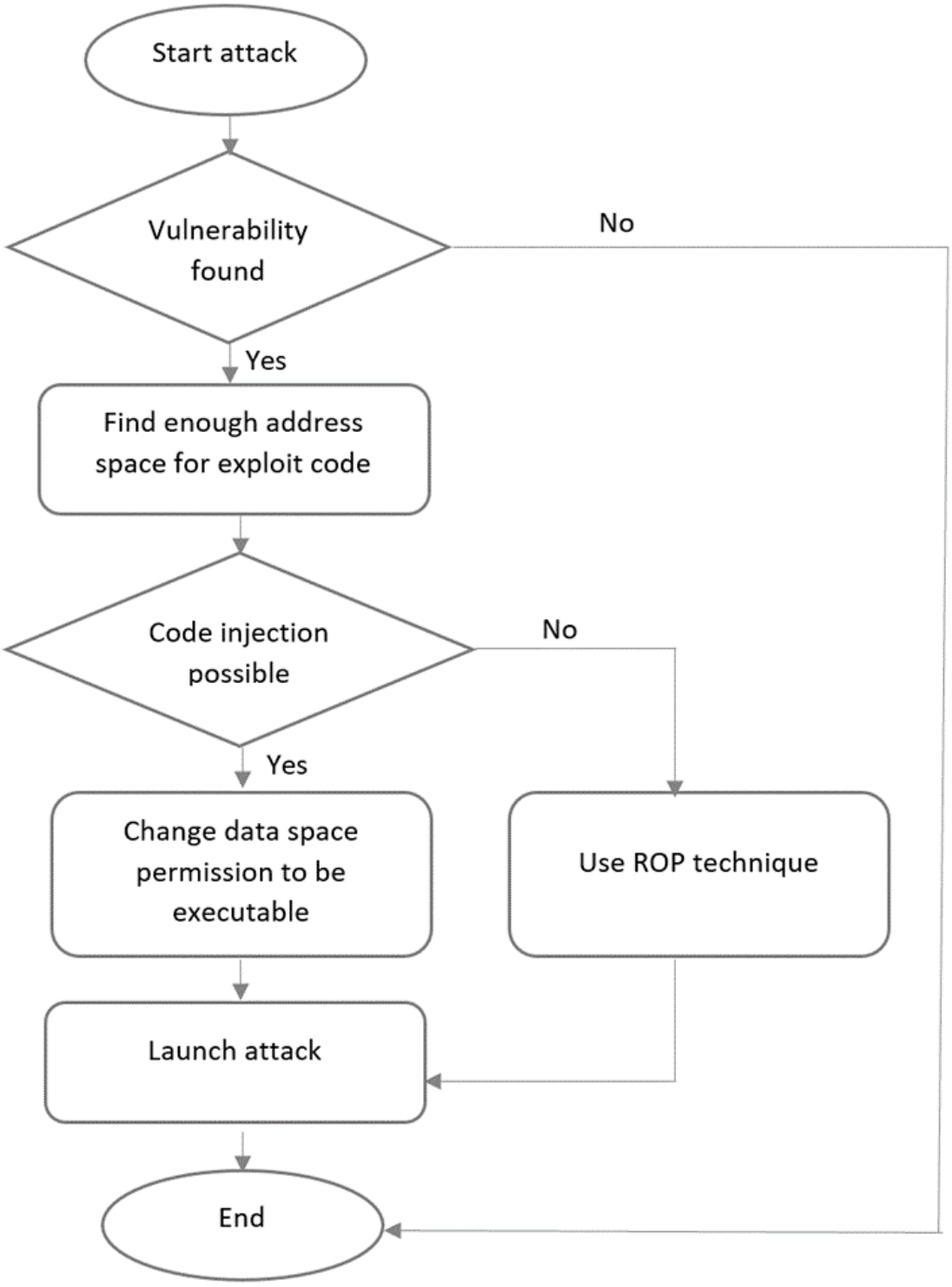

Figure 8: Simple code-injection and code-reuse attack flowchart.

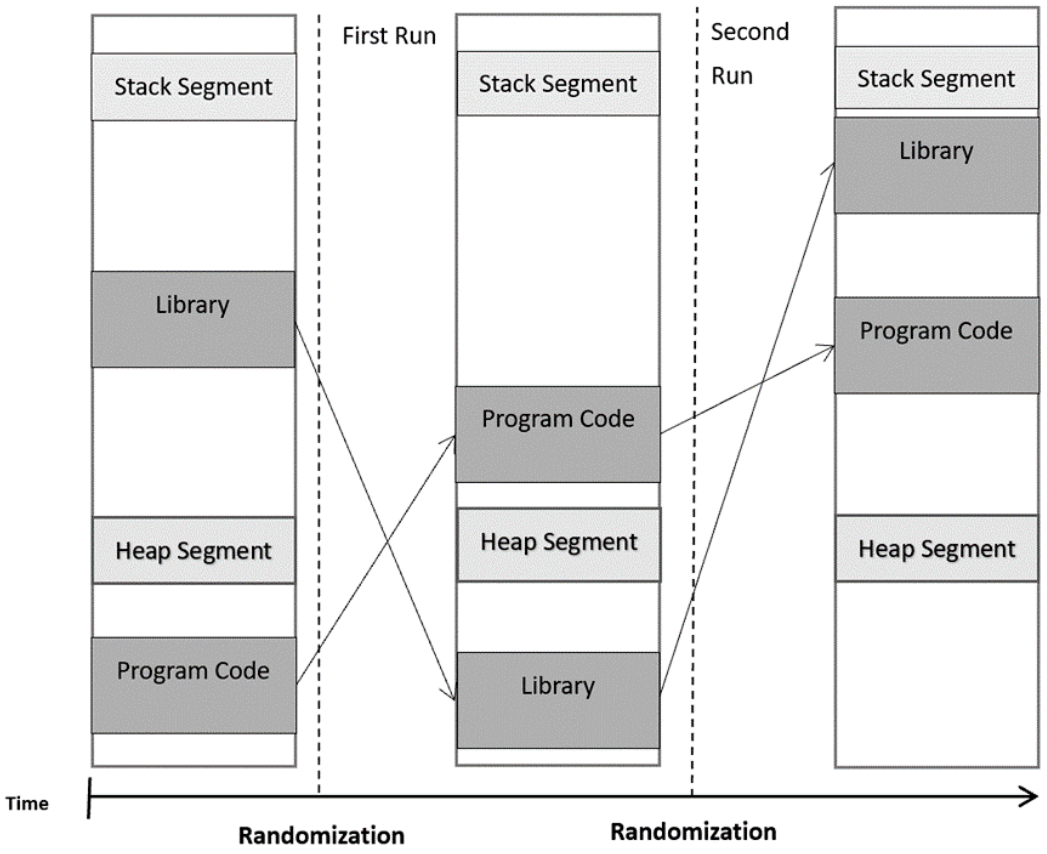

Figure 9: Address Space Layout Randomization (ASLR).

bypassing the randomization and knowing the address of those segments [38]. Another limitation is that ASLR can only randomize the first 16 bits of the base address of different segments, such as heap and stack. This means that the different memory segments have the same offset. If the attacker uses some memory leakage mechanism, such as Use-after-free or Integer Overflow, the memory location of those segments can be leaked, and then a code-reuse attack can be mounted because now the addresses of critical segments, such as Heap and Linked libraries, are known.

Memory leakage attacks are effective in allowing attackers to read information directly from program memory during run-time. For a program that runs on Linux, the memory information of the run-time linked libraries is saved in the Global Offset Table (GOT). This allows the application to resolve the address of a needed function, such as scanf(), during run-time. The address resolution can be dynamic or on-demand, i.e., lazy linking, through a table called Procedure Linkage Table (PLT). Once the address of one function is known, the attacker can adjust his entire set of pointers to libc or other libraries in real-time to mount a code-reuse attack.

3.2.2 Control-flow integrity (CFI)

Control-reuse attacks rely on the attacker's ability to change the intended program control flow to unintended control flow. The hijacked or diverted control flow allows the attacker to point to malicious code or payload. Control-Flow Integrity (CFI) is a mitigation mechanism for code-reuse attacks. It was originally introduced by Abadi et al. in their seminal paper [39], [40]. CFI is a way to enforce the program's control flow to adhere directly to the legitimate intended path as per the software control-flow graph (CFG). FIGURE 10, demonstrates a simple CFG with imposed CFI checks. The benign control flow graph has 6 nodes, and all nodes have output links to the next nodes as drawn below. What CFI introduces is a layer of run-time checks against a pre-defined policy to check if the attacker violated the intended CFG.

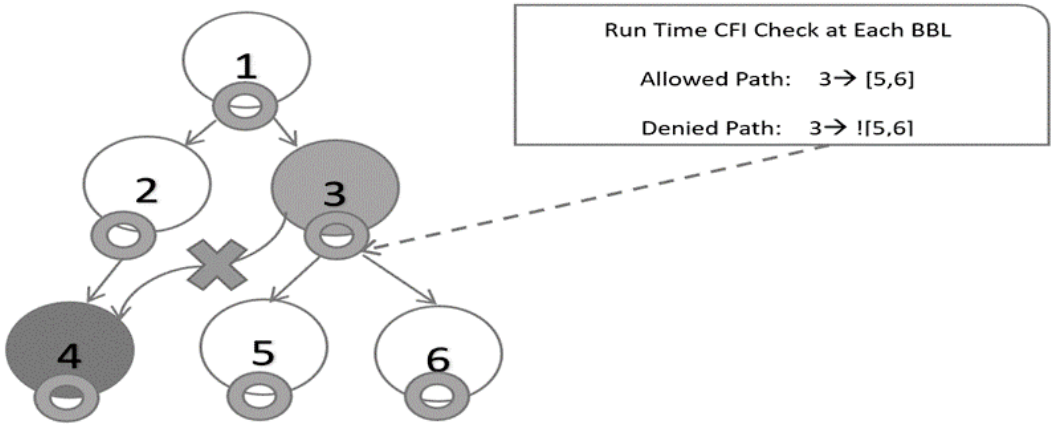

Figure 10: The original CFI concept [41].

The CFI checks in the example above allow node 3 to have exit instructions to nodes 5 or 6 only. Any other exit instructions to nodes other than 5 or 6 will be prohibited as per the CFI policy. The CFI checks are produced by multiple mechanisms that will be discussed later. However, all the checks need to be executed during runtime. This is to prevent attackers from diverting the control-flow during the execution of the program.

The original CFI proposed by Abadi et al.[42] is a label-based CFI. It relies on putting a label at the beginning of each Basic Block-Level [BBL]. This label is either programmed as a simple data word or as an address offset into x86's cache pre-fetch command. The label is added before each "jump/call or load/store" instruction, which mandates reordering the memory offsets of these instructions. A binary instrumentation framework is needed to do all the instruction insertions and monitoring, along with the binary CFG creation. Abadi et al.'s approach depended on Vulcan[41] instrumentation framework. CFI has a prerequisite to work correctly and protect the CFG with high accuracy. The first prerequisite is the need for a protected code and data memory space to be maintained during binary execution; this requires $(W \oplus E)$ enforcement. The second prerequisite is that the code will not modify itself at each runtime; hence if the code is generated by a virtual machine such as Java Virtual Machine (JVM), the CFI will not be effective. CFI introduces high-performance toll because of the offset changes and binary re-write; thus, a more optimized CFI is

being considered by Abadi et al., which enforces CFI checks only when the node terminates with indirect branch instructions such as jump or call. For direct branch instructions, the offsets are written statically after the binary is instrumented, hence cannot be changed by the attacker

3.1.5.1 cfi for indirect jumps

CFI can be used to protect against control-flow hijacking that utilizes indirect jump techniques. Indirect jumps are produced when the coder utilizes a switch-case statement or calls a certain subroutine. Switch-case statements allow the programmer to check against pre-defined variable content in order to execute the corresponding statement. The example mentioned below in Fig. 11 represents the assembly code structure that corresponds to the switch-case source code. Each case value triggers indirect branching to some memory location.

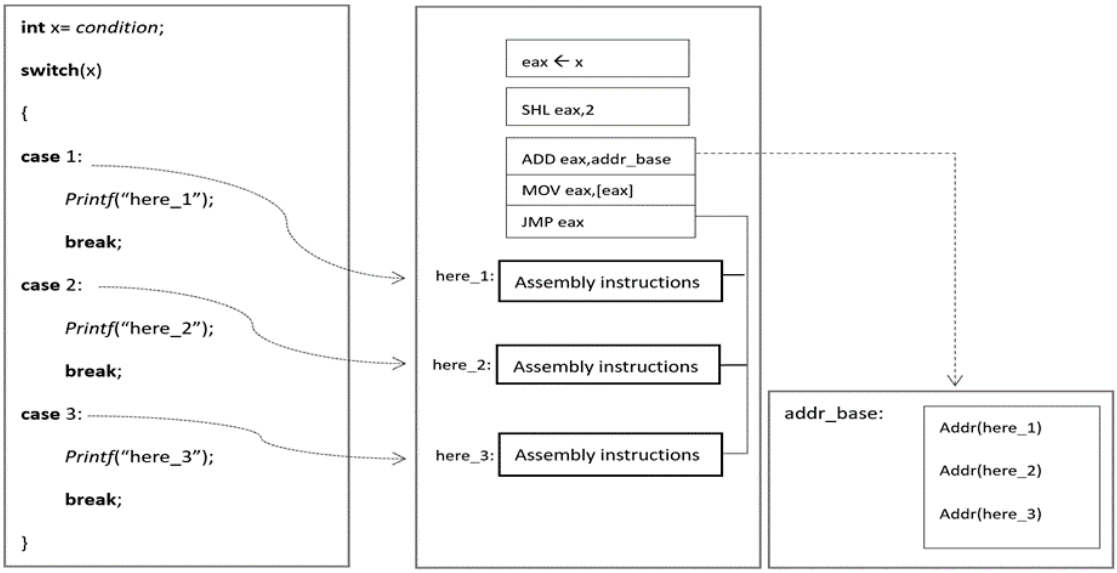

Figure 11: Switch-case as indirect jump example.

As discussed earlier, the EAX register is usually used by the attacker as a placeholder for the memory address that will be loaded into the EIP register to redirect the control flow. In the switch-case statement, the value of the variable named 'x' is saved in EAX. To hijack the control flow, the attacker needs to load the needed jump address into the EAX register or change the content of the "x" variable to hold that address. Each content stored in the 'x' register corresponds to a memory address to load that content. The indirect branching occurs when content is loaded into the EAX through the following steps:

1. Shift left EAX by 2 to align it with the correct address contained in the content of the 'x' variable.
2. Load the base address from the jump table that holds all addresses of each case statement into the EAX register.

3. Dereference the EAX value and jump to that correct indirect branching

Another way for attackers to hijack the control flow is by controlling the Global Offset Table (GOT), which contains the runtime addresses of all functions within shared libraries such as printf() of libc. This address is called through a code stub stored within the Procedure Linkage Table (PLT).

The CFI checks must take place during run-time before the indirect jump. Fig. 12 illustrates how CFI can protect against indirect branching. CFI emits a unique label before node 2 and node 3; in this example, the label is "0000ffff". The code of node 1 is instrumented to check whether EAX targets the label "0000ffff"; only then, the transition is allowed. The labels in the [43] method are 4 bytes each, and the jump target is updated to skip those four bytes to reach the next node. This is what LEA and EAX+4 do.

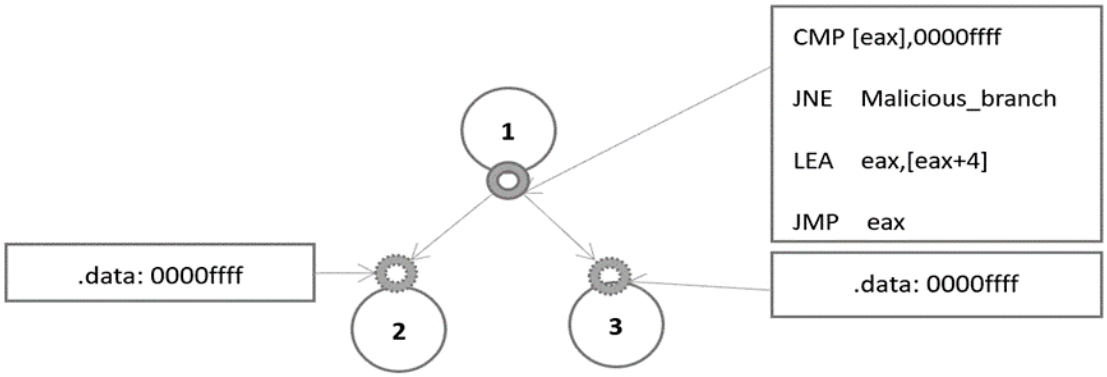

Figure 12: Indirect jump checking using CFI.

3.1.5.2 cfi for indirect calls

Indirect call guidelines are used by compilers in the following cases: function calls through its pointers, callbacks, and C++ virtual functions. A processor register or memory space can hold the address of the target for the indirect calls on x86 architecture. Control-flow hijacking using C++ virtual tables is a very common technique to attack state-of-the-art software, such as browsers or Microsoft Office applications, during runtime.

A virtual function is a foundational concept for the idea of polymorphism in object-oriented programming. An example of how virtual functions work is the C++ method draw(), which is a virtual method that is part of the base class shape. To draw a rectangle or square, the method draw() can be utilized, and its parameters are defined to be used within the child class, such as oval or rectangle. The pointers to every method are saved in the virtual table 'vtables' that is created specifically for this method. vtables are accessible during run-time and through a pointer that is stored in the object's data structure, which sits in writable memory space. The vtable is stored in read-only memory. The offender uses the vtable pointer to insert a fake vtable and changes the vtable pointer to point to the injected fake vtable. Thus, during the next execution, the program will dereference the overwritten vtable pointer and redirect the control flow to the address stored in the fake vtable.

3.1.5.3 cfi for function returns

There are two ways to change the return address inside the stack memory space to direct the control flow. The first way discussed above is through indirectly changing the address by calling a jump or switch-case statement. The second way is using a direct method to call the function address, which modifies the return address to be the new return address of that function, thus overwriting the return address directly. Directly calling a function gives an advantage to the attacker because there is more flexibility to use any of the functions within the program code. However, it also represents a challenge for protecting those functions or subroutines because function calls take place at run-time.

Applying the ”label-based CFI” approach proposed by Abadi et al. [43], [44] is a coarse-grained defense method due to the lack of total control of the exit node of each called function. FIGURE 13, explains this challenge; only one single CFI label is placed before the intermediary node, and only the check will be instrumented before the return instruction of every calling function.

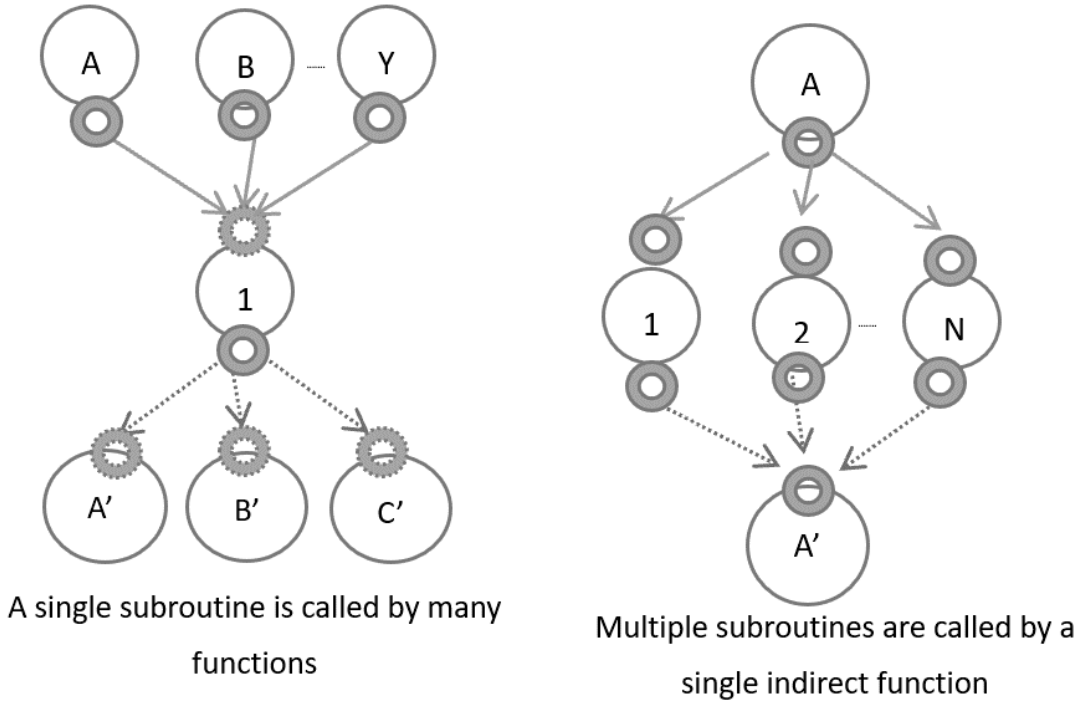

Figure 13: label-based static checking for return of functions using CFI.

One of the solutions to keep track of which function called which node is to use a shadow stack to save all the memory addresses of the calling function and the called nodes, and compare these values when a direct function call is executed during run-time. The authors in [43],[44] suggested this solution to allow fine-grained integrity checks for function return addresses. The shadow stack area must be protected against unauthorized writing. Therefore, a technique such as Software Fault Isolation (SFI)[45] can be used to segment the memory area that contains the shadow stack.

To explain the shadow stack, Fig. 14 provides a simple instruction instrumentation flow chart. By checking every call and every return instruction, a table inside the protected memory area can be used to store all the return addresses for each legitimate function call. These values will be used during run-time to compare each executed function's return address to its stored return address. The shadow stack introduces performance overhead due to the load and compare operations needed for validating function return addresses. Additionally, the instruction instrumentation for each direct call instruction during run-time adds additional overhead, and there is another overhead from the need to have a dedicated shadow stack for the program thread.

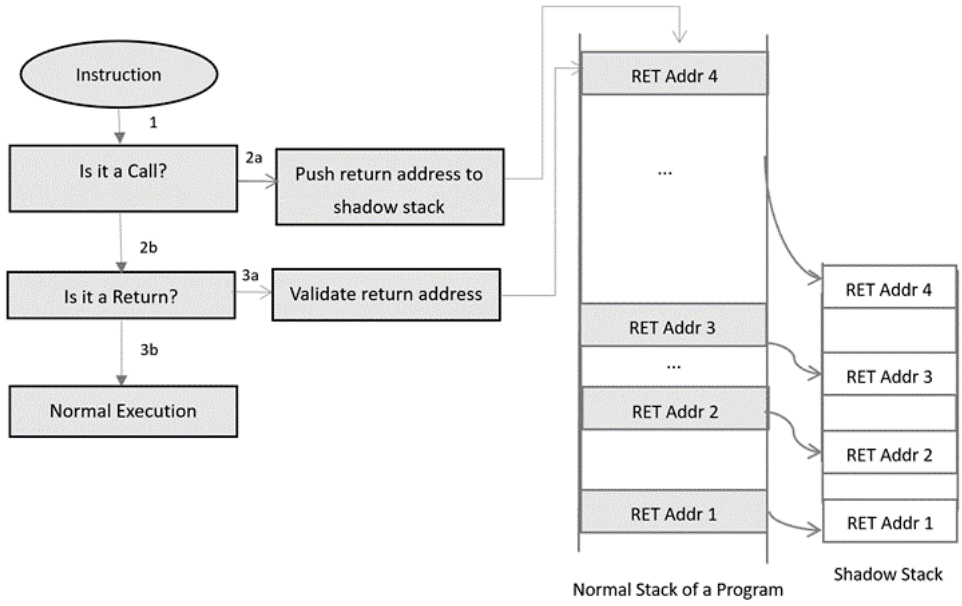

Figure 14: Basic shadow stack operation.

The shadow stack approach is not useful when the called function does not return to the original caller by design, such as with setjmp or longjmp functions used in C++. Additionally, if the attacker uses Position Independent Code (PIC) in their malicious direct call to control the value of the EIP register directly, they can know the current value of the program counter.

The ROP defender [46] approach uses dynamic binary instrumentation to overcome the unique return address situations mentioned in earlier return cases. Position Independent Code (PIC) represents a challenge to ROP defenses because of the way it operates. Some instructions, such as those contained inside Linux shared libraries (.so), need to calculate their addresses during run-time to avoid overlapping with other instructions in the calling program's code space. PIC hinders both attackers and defenders since the memory pointers to the PIC change frequently and cannot be easily found. Some defense methods opt to use the shadow stack to store the new memory locations of the function calls. If the defense technique uses a randomization pattern for instructions or memory blocks, then the shadow stack needs to be modified by adding memory padding to the new random addresses.

3.3 Code Reuse Attacks Categorization, Mitigations, and Defenses

In this section, we are laying out the different Control-Flow Attacks (CRAs) in terms of mitigations and defenses. We are presenting the attacks and mitigations in an organized fashion based on the similarity of internal workings or design mechanisms of the attacks and mitigations. FIGURE 15, lays out the memory control flow attacks in a concise manner.

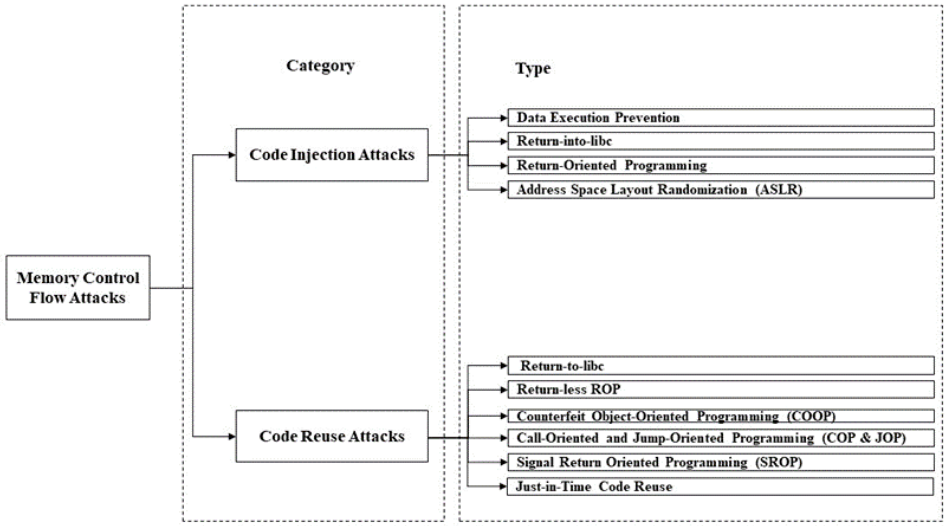

Figure 15: Memory Control Flow Attacks.

3.3.1 Code-reuse attacks categorization

Our emphasis is on the investigation of the Control-Flow Attack (CRA) mitigation methods. These protection methods are developed to safeguard against specific categories of CRAs. Our discussion will focus on the diverse categories of code-reuse attacks to provide an introduction to our analysis of the mitigation methods. TABLE 1 mentions the attacks that can be mounted in the code-reuse fashion against software binaries and the basic protection concepts, along with their limitations.

1. **Return-to-libc:** The original Return-to-libc attack [34] demonstrates the reuse of some existing core functions in the Linux Libc system library. These functions enable attackers to change the normal code control flow and introduce indirect branching to jump to exploit code. The functions used for reuse need to run as privileged or system functions, such as mprotect [47]. One of the most proposed techniques recommended for mitigation is [48], which is built around controlling indirect branching to only pre-filtered sets of destinations for each branching direction. This is called a fine-grained approach, which solves the problem of obscure function calls for indirect branching and their supposed targets. The fine-grained approach relies on labeling each basic code block (BBL) and its calling block, storing the

Table 1: CRAs Attacks and Limitations

Attack	Reference	Description	Mitigation	Mitigation Limitations
Return-to-libc	50	Reusing parts of Linux Libc	Randomizing library location. E.g, ASLR	The Randomization is not truly random.
Return-less ROP	53	Reusing code parts that are similar in function to return instructions	Removing parts of the code that has the same function as return	It can introduce overhead and side-effect gadgets.
Counterfeit Object Oriented Programming (COOP)	54	Using fake objects as code pointer	Protecting the access to virtual pointers or not using virtual pointers	Very hard to remove virtual pointers completely
Call Oriented and Jump-Oriented Programming (COP JOP)	55	Using call and jump instructions instead of return instructions	Provide mitigations that works on backward edge detection.	Backward edge detection requires code disassembly and code instrumentation
Signal Return Oriented Programming (SROP)	56	Send fake signaling to redirect code blocks	Blocking fake signals or protecting signal's location. "Signals Canary"	Canaries can be overwritten or bypassed.
Just-in-Time Code Reuse	57	Generating code redirection instruction in real-time	Enforce code-flow integrity checks	The CFG of the code is different each time the code gets complied in real-time.

information using a data structure within a protected memory location. This approach is called "Forward-Edge" detection. "Backward-Edge" detection is possible using a technique similar to the shadow stack [48]. The shadow stack is a table with all the locations of the return addresses of all the normally called functions in the control flow graph.

2. **Return-less ROP:** As a defense mechanism, all return-like assembly commands can be filtered out from software code. However, this doesn't prevent attackers from using some assembly instructions in a certain sequence to have the same impact as using the "ret" instruction. The downside of these sequences is the functional side effects on the code integrity. Attackers can still use "jmp" or "call" instructions to invoke indirect branching.

   Return-less Kernel [49] was implemented in FreeBSD as a protection approach against ROP attacks. After removing all "ret" instructions from the software assembly code, a set of assembly instructions needs to be written in the same place with the same effect as the removed "ret" instructions.

3. **Counterfeit Object-Oriented Programming (COOP):** Some pointers in C++ can be used to change the existing control flow [50], such as vptrs. Vptrs construct a fake code object with a similar memory structure to the current memory structure used to store the dynamic dispatch information. This pointer construct will

be used at a later stage by the attacker to change the code flow. Detecting and preventing such an attack is challenging due to the subtle nature of the legitimate vptrs.

4. **Call-Oriented and Jump-Oriented Programming (COP & JOP):** The return (ret) instruction provides an easy way to compute the location of the calling function; however, the destinations of call and jmp instructions are typically unknown until the program is executed (run-time). It is understandably complex to detect backward edges created by call and jmp instructions. Attackers use the jmp and call instructions to control code branching in the same way they did with ret instructions [40].

5. **Signal Return Oriented Programming (SROP):** Most modern OSes use some sort of signaling mechanism to protect, manage, and interact with computer memory space. A POSIX signaling framework and similar frameworks can be misused by attackers to send fake malicious signals to any memory space they can control in order to execute their malicious code from a specific location. A sigreturn system call can be used for such an attack. Protecting current OSes from such attacks can be cumbersome due to the weakness of the proposed defenses. One defense that can be used to achieve minimum protection is ”Signaling Canary,” which works like Stack Canaries but at the signaling level instead of the stack level.

6. **Just-in-Time Code Reuse:** In [51], the authors implemented a mechanism for the Just-In-Time (JIT) compiler to generate a ROP compiler that provides a sequence of gadgets during run-time. The developed ROP compiler requires the attacker to perform some instructions to mount the attack, then it interfaces with the leaked memory space data and searches within the software system calls and APIs for a set of useful gadgets. The JIT compiler might modify the CFG of the code during runtime to improve the code, which introduces complexity when generating an accurate CFG.

3.3.2 Code-reuse attacks mitigations and defenses

Code-reuse attacks present a challenge to software engineers, security researchers, and systems defenders [8]. Over the past 20 years, numerous mitigation techniques have been introduced to defend against the different types of code-reuse attacks discussed in the previous section. The mitigation techniques can be categorized in different ways. Here, we categorize the defenses under five different categories based on their internal way of working, which falls within the same category. FIGURE 16, shows the layout of the proposed classification of CRA mitigation techniques.

3.2.2.1 mitigations based on control-flow integrity

1. **The Original Control-Flow Integrity (CFI):** was introduced by Abadi et at. in [46]. The main idea of CFI is to re-write the binary instruction which applies certain policies to ensure that the original control-flow graph (CFG) is followed. The detection mechanisms of the technique rely on checking upon any indirect control flow branching at the run-time and do a comparison between the intended target of indirection branching is different from the one created by CFG then that branch is marked malicious. The CFI technique can be applied at different granularity levels such as Basic Block-level (BBL) or instruction level. CFI was introduced mainly to enhance the detection of malicious code branching due to injection or

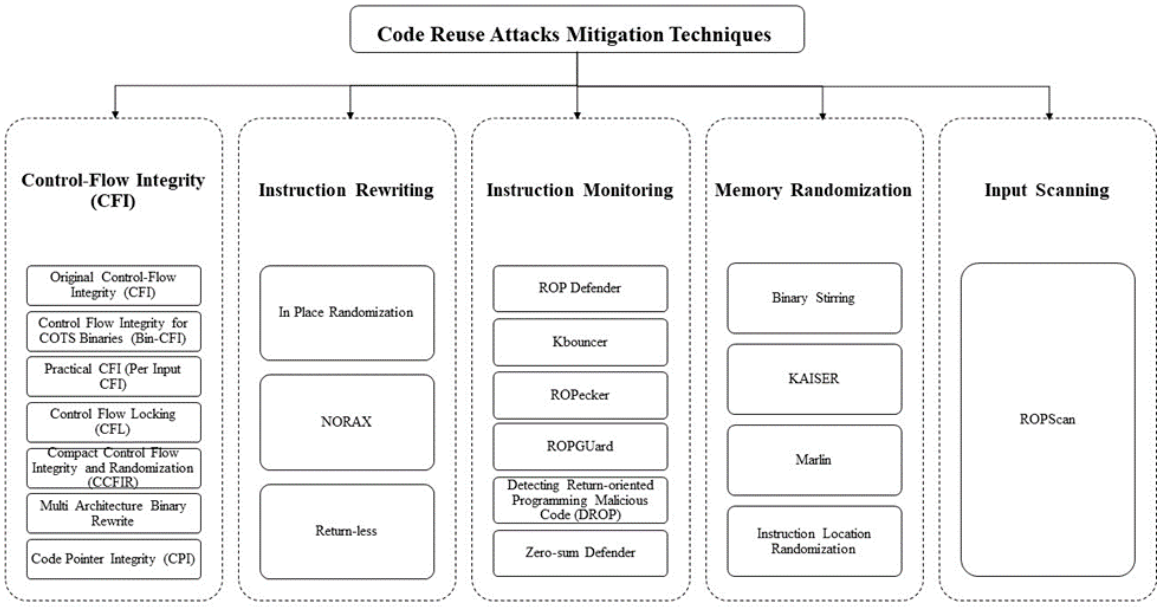

Figure 16: Classification of CRA Mitigation Techniques.

code-rescue attacks. The proposed defenses against CRA before the introduction of CFI had high-performance overhead [16].

2. **Control Flow Integrity for COTS Binaries (Bin-CFI):** This technique was proposed by Zhang
[52] to solve a fundamental problem that existed in the original CFI. The original CFI needs debug symbols and relocation information in order to generate an accurate CFG. This Bin-CFI technique works on stripped binaries directly without needing compiler support information. In this technique, the researchers have proposed a numeric variable to calculate the efficiency of CFI-based protection approaches. The value is a functional way to indicate the number of indirect branching target that gets removed by a CFI approach. The metric ideal value should be 1. The majority of CFI-based protections depend on one out of five techniques to eliminate malicious branching. Bin-CFI: utilize arithmetic operations to enforce CFI on a program that contains specific indirect branching commands such as indirect jumps, long jumps, C++ exception returns, and Jumps using return commands.

3. **Practical CFI (per-Input CFI):** has been proposed by Ni [53]. This approach adopts the per-input CFI (PICFI or πCFI) techniques to handle all unnecessary edges in the program's CFG per each input. This technique solves the vulnerability of adding malicious edges to the program's CFI. This approach was proposed to provide a solution to the limitation of both coarse-grained and fine-grained CFI methodology where an accurate CFG is needed; which means a static analysis should be done to the binary's code to accurately generates all possible inputs.

4. **Control-Flow Locking [54]:** was introduced by Bletsch. It is an assembly and binary rewriting technique based on CFI. It depends on Code Locking (to control all indirect calls). Control

flow locking depends on analyzing the software control-flow graph to determine all indirect calls and set a lock in the memory each time an indirect call takes place. This lock is unlocked when the original function return address is executed

5. **Compact Control Flow Integrity and Randomization (CCFIR):** was proposed by Zhang et al. [55] as an enhanced CFI protection mechanism to solve the problem of enumeration of all le-gal indirect branching targets and allowing transfer to those targets using only a trampoline. This technique employs the relocation table in any binary to generate all legal indirect branching without the need for the program's source code the utilization of "Springboard" or "code trampoline" is helpful to whitelist only authorized code jumps or call via code alignment enforcement.
6. **Multi Architecture Binary Rewrite:** This technique is an extension to CFI for COTS protection. Valensi et al. [56] This technique is an extension of the original prototype designed by Wang et al. named Uroboros [57]. The main goal of binary rewrite is to cover more processor architecture such as ARM. The reconstruction of program binary proposed by Wang et. Al. [57] had limitations when it comes to covering multiple architectures such as ARM 32 bit. The MADRAS technique has rebuilt the Uroboros tool using Python instead of OCaml language.
7. **Code Pointer Integrity (CPI):** was introduced by Kuznetzov et al. [20] as an improved CFI protection that focuses on pointer protection to prevent indirect control flow. The goal of the researchers was protection against all types of CFI attacks and to fix the shortcomings of coarse-grained CFI and fine-grained CFI mechanisms. The main idea of CPI is to divide the memory of any thread into two parts, safe region, and regular region. The safe region will contain only all sensitive pointers and protect access to those pointers using hardware segmentation.

3.2.2.2 mitigations based on instruction rewriting

In this section, we discuss the mitigation techniques which are based on rewriting sensitive instructions inside the binary. Rewriting the instructions are valuable for removing malicious content such as potential gadgets. Instructions can be written directly in the binary file or on the disassembled binary file.

1. **In Place Randomization [58]:** has been introduced by Pappas, like IRL it focuses on rewriting the program binary code in order to randomize the instruction and eliminate any possible gadget. The focus of this technique to is re-write the binary each time the binary runs, hence producing different randomized instructions per instance.
2. **NORAX [59]:** was proposed by Chen et al. to build upon the work of ASLR protection by enabling executing only (XOM). It solves the problem of COTS binary since they are not protected by XOM techniques. It works by retrofitting the protection concepts used in Bin-CFI to the Android binary system by supporting the AArch64 CPU architecture. To enable the COTS binaries to be protected by hardware protection features such as (XOM), this system retrofits the Execute-only Memory (XOM) protection on the binary during load time using the hardware features of AMD64 bit processor.

3. **Return-less kernels [49]:** focused on removing one of the most important instructions that is getting used by the attackers to mount gadget chains, which is the 'RET' instruction. This technique was demonstrated on Linux kernels. It combines three different approaches to remove intended and unintended returns.

3.2.2.3 mitigations based on instruction monitoring

In this section, we discuss the mitigation techniques which are based on monitoring binary instructions for the protected file. Safe instructions will be marked by the monitoring technique using some sort of label-insertion mechanism or through an instrumentation virtual machine.

1. **ROPDefender [46]:** has been introduced by Davi et al., it's a method to protect against ROP attacks by monitoring the dynamic code transformations Just-In-Time (JIT) at the binary runtime. It utilizes novel techniques to detect call-return pairs and to check if those pairs are broken by the ROP attackers
2. **kBouncer [60]:** has been introduced by Pappas to detect the ROP-based attacks at run time. The technique is based on monitoring a special hardware register in intel x86 CPU called Last Branch Record (LBR) using an added kernel module.
3. **ROPecker [61]:** has been introduced by Cheng et al. As the counterpart of kBouncer but on the Linux platform. It's a run-time heuristic detection method that monitors the LBR stack to detect unauthorized indirect branching.
4. **ROPGuard:** has been introduced in [62]. Itis a run-time detection approach to protect against 6 different ROP characteristics.
5. **Detecting return-oriented programming malicious code (DROP):** The DROP technique [63] has been proposed to be one of the early protection against ROP attacks. The technique depends on binary instrumentation during program run-time in order to detect ROP.
6. **Zero-sum Defender:** has been introduced by Kim et al [64]. It's a compilation-time protection-based technique since it focuses on instrumenting the complied code to count the number of the call-return pairs and makes sure that the total difference between the pairs is zero.

3.2.2.4 mitigations based on memory randomization

In this section, we are going to discuss the mitigation techniques which are based on randomizing the memory space of the running applications. Randomization can take place on the memory block level or instructions per instruction level.

1. **Binary Stirring:** has been introduced by Wartell et al [25]. Its main contribution is to rewrite the code of the binary to randomize the location of the basic blocks. It was introduced to address one of the common shortcomings of different CRA mitigations, which is the need for the program source code. Typically, all Commercial-off-the-shelf programs (COTS) don't have their source code available.

2. **Kernel-space Address Layout Randomization (KASLR):** is the Linux implementation of ASLR. Gruss et al. [65] have introduced KAISER, which is an address isolation framework based on Ubuntu Linux. The goal of KAISER is to close the information leakage vulnerability by strictly enforcing the separation between kernel-space and user-space, mainly to mitigate side-channel attacks such as memory page faults and prefetch attacks that can occur when ASLR splits the program memory into kernel space and user-space.
3. **Marlin:** has been introduced by Gupta et al. [66]. It focuses on randomizing the binary functions using a random distribution after program loads.
4. **Instruction Location Randomization:** was introduced by Hiser et al. [27], it implements the randomization on the instruction level instead of the block level as in the Binary Stirring technique. The randomization takes place post-deployment to eliminate all possibilities of gadget creation.
5. **Mitigations Based on Input Scanning:** In this section, we discuss the mitigation techniques which are based on scanning the input sent to the protected application. Code-reuse attacks can be mounted using gadgets; hence scanning for the address of the gadgets in the input of the application can prevent such ad-dress injection attacks.
6. **ROPScan:** has been introduced by Polychronakis [67], it depends on scanning the input data either from network or memory buffer and run it through a CPU emulator to detect shellcode.

4. CODE REUSE ATTACKS EVALUATION AND RANKING USING SMAA-2 APPROACH

As discussed in the previous section, each defense mechanism has its advantages and disadvantages. However, the performance overhead value can be used as nominal information regarding the mechanism to be selected for future improvement or adoption by OS manufacturers. Performance overhead is critical for several reasons, such as expected delays in run-time for core program functions after employing the defense mechanism [68]. It also impacts the resources needed from the CPU and computer memory to run the modified binary. Other less important factors include the level of changes needed to be done on the code to make it immune against CRA and the coverage of the CRA defense against different forms of code (e.g., binary only, source code only, or both).

From the three factors mentioned above, we have devised a simple way to rank CRA mitigations in a previous paper [68], by applying an average weighted sum approach to the three selected criteria (Performance, Universality, and Effectiveness). The challenge in that approach is that published researchers don't typically include information on all three criteria in their experimental results, so we had to omit some numbers related to missing information about one of the three criteria. In order to provide a better evaluation and more accurate guidance, we are adopting a different evaluation technique in this paper. The selected technique is Stochastic multicriteria acceptability analysis (SMAA) [12], which is part of many techniques that can be employed to solve a multiple criteria decision-aiding problem. This approach can be tuned to aid in ranking, sorting, or choosing certain alternatives based on putting a complex weight model to their associated criteria. The complex stochastic model for each criterion will solve the problem of the missing or incomplete values of some of the criteria variables.

SMAA is a relatively new family of methods proposed to support the decision-aiding process for problems with multiple criteria. Multiple-criteria Decision-Aiding (MCDA) methodology provides assistance in the selection process among multiple alternatives by aiding in the process of sorting, ranking, or choosing among evaluated alternatives. We have adopted the SMAA approach to provide an advanced mechanism that can help in the evaluation and ranking of different CRA mitigations under our evaluation. The challenge we are trying to overcome is the lack of some evaluation parameters, such as binary size overhead or the absence of some information, such as performance overhead. This makes the SMAA approach suitable for our ranking purpose because it can work with missing or incomplete information. Additionally, we aim to reduce selection bias due to the existing knowledge of each technique's weakness points. The SMAA family of methods includes multiple approaches. We are going to use SMAA-2 because it is an extension of the original SMAA model, which extends the Decision Making (DM) generated by SMAA into feasible ranking and ordering of alternatives. The indices introduced by SMAA-2 are as follows:

1. **Ranking Acceptability Index:** is an integral part of multidimensional weight over the distribution of the criteria and the favorable rank weights
2. **K-Best Rank Indices:** it's a way of grouping alternatives in order to make the selection process between alternative group instead of weak alternatives.
3. **Holistic Acceptability Index:** This represents a confidence factor assigned to the alternatives as a measure of the efficiency of the alternatives. If this index is high, this means that the alternative is not worth of being evaluated

The selection of the evaluation criteria is an extension of the previously selected criteria in our previous research. The old research had three main criteria that were used to evaluate the discussed CRA mitigations:

1. **Performance overhead:** This is the overhead added to the performance of the binary after applying the CRA mitigation mechanism.
2. **Efficiency:** This is a measure of the level of details needed to be altered by the protection mechanism to safeguard the software. If the software needs to be disassembled or doesn't need to be, and only the source code can be altered to apply protection, this makes the protection less efficient than applying it directly to the off-the-shelf binary.
3. **Universality:** If the protection can defend against more than one type of CRA, such as JUMP-Based or CALL-Based, it can be considered more universal than protection that only defends against ROP-based attacks.

We added two more evaluation criteria to the previously selected three criteria. The newly selected criteria are:

1. **Binary size overhead:** This is the change to the binary size after applying the protection. Since protection involves changing the code of the original binary, it can add significant overhead to the original binary size.

2. **Acceptability:** This is a hybrid criterion that links the protection mechanism to the holistic acceptability index of SMAA-2. It is a dynamic criterion that can have different values at different times, depending on the usage of certain protection mechanisms in mainstream research or adoption by OS OEMs. Some protection mechanisms can be widely adopted at some points in time and can be overlooked or neglected at different points in time, depending on the advancement of research or the appearance of new defenses.

The SMAA-2 criteria can have a cardinal value (i.e., different value types depending on the nature of the criteria) or an ordinal value (a descending or ascending ranking of all the techniques under evaluation). The mapping between the selected variables and criteria is as follows:

1. **Performance criterion:** The selection measurement method is an exact value of the performance of each protection mechanism normalized to a value between (1-15). The selection of those values is driven directly from the SMAA-2 formula.
2. **The Binary size over-head Criterion:** The selection measurement of this approach is a normal distribution value of 5 and standard deviation of (0.5). This is because the numbers of the binary size overhead are not mentioned in most of the experiment results that are published by the researchers and it can be mapped using a normal distribution function as per SMAA-2 approach recommended for treating unknown values for certain criteria.
3. **The Universality Criterion:** This is mapped to an exact value of either (10 or 5). If the protection mechanism is covering all types of CRA at-tacks then it's given the value of 10 otherwise it's given the value of 5.
4. **The Efficiency Criterion:** This was represented as a gaussian normal distribution with the same values of the Binary size overhead.
5. **The Acceptability Criterion:** is selected at an arbitrary value between (1-20) as it represents the subjective amount of adaptability of such protection mechanism in mainstream OS or research. This criterion will help in the ranking between different options. We are evaluating in this paper twenty different CRA protection mechanisms

5. EXPERIMENTAL SETUP

We used the published tool JSMAA v1.0.3 (LINK) in our analysis as it was built by the researchers as a proof-of-concept for their proposed MCDA mechanism. The benchmarking tool runs on a computer with 64GB of RAM and an Intel i9-9980HK x64 CPU at 2.40GHz processor. All numbers related to the selected criteria are extracted from their primary sources of published research.

5.0.1 Experimental results

Here, we are discussing the two main outcomes of the SMAA-2 technique: the central weight vector and rank acceptability indices. The central weight vector represents the weight center of gravity, which shows the preferences that made this decision favorable over other alternatives. The rank

acceptability describes the weight of each parameter value that makes an alternative have a certain rank over other ranks. TABLE 2, displays the analysis results of the CRA techniques against the selected criteria. The confidence factor value is the central weight vector that represents the probability for a specific alternative to be selected as a preferred one. The nominal factor of ”1” in the confidence vector represents the neutrality of the alternative; hence, it can be out of the DM selection process. TABLE 1, results are the application of relative weights of the different criteria on all alternatives with no cardinal preference bias application.

Table 2: SMAA-2 Analysis results of the CRA techniques against the selected criteria.

Alternatives	Confidence Factor	Performance	Overhead	Universality	Acceptability	Efficiency
Binary Stirring	0.6916	0.278893806	0.110230261	0.201251363	0.219262505	0.190362
BIN-CFI	0.1004	0.042853512	0.092635139	0.007968427	0.580510582	0.276032
CCFIR	0.4677	0.065535764	0.049343885	0.198348967	0.485579788	0.201192
CFI Locking	0.1198	0.061719449	0.056862104	0.033772538	0.270566079	0.57708
CFI Original	1	NaN	NaN	NaN	NaN	NaN
CPI	1	NaN	NaN	NaN	NaN	NaN
G-Free	0.0353	0.001213693	0.011335477	0.26608556	0.046730133	0.674635
ILR	0.0394	0.085017685	0.050568817	0.222819204	0.219010348	0.422584
In Place Rand	1	NaN	NaN	NaN	NaN	NaN
KAISER	0.0441	0.086165967	0.039092478	0.233560329	0.15693121	0.48425
kBouncer	0.2585	0.301945727	0.07591299	0.205095484	0.154975894	0.26207
Marlin	0.0117	0.021433432	0.006900578	0.208672257	0.160461234	0.602532
Multi-Arch. Binary Rewrite	0.0049	0.034158935	0.055916713	0.127605921	0.087286657	0.695032
PICFI	0.0127	0.020134753	0.0128884	0.516012015	0.032348225	0.418617
Return-Less Kernel	1	NaN	NaN	NaN	NaN	NaN
ROP Defender	0.9981	0.113268079	0.328251009	0.2010939	0.174416533	0.18297
ROPecker	0.1825	0.315171433	0.073347282	0.214714991	0.100315174	0.296451
ROPGuaaaard	0.0671	0.350636667	0.101256192	0.005376151	0.155814968	0.386916
ROPScan	1	NaN	NaN	NaN	NaN	NaN
Zero-Sum Defender	1	NaN	NaN	NaN	NaN	NaN

FIGURE 17, displays the weight distribution of each weight factor that makes a specific alternative (e.g., technique CRA) favorable. For example, when discussing the universality results, we find

out that PI-CFI and G-Free have the highest weights of universality. Hence, they can aid DM when universality is given the highest importance over the other four criteria.

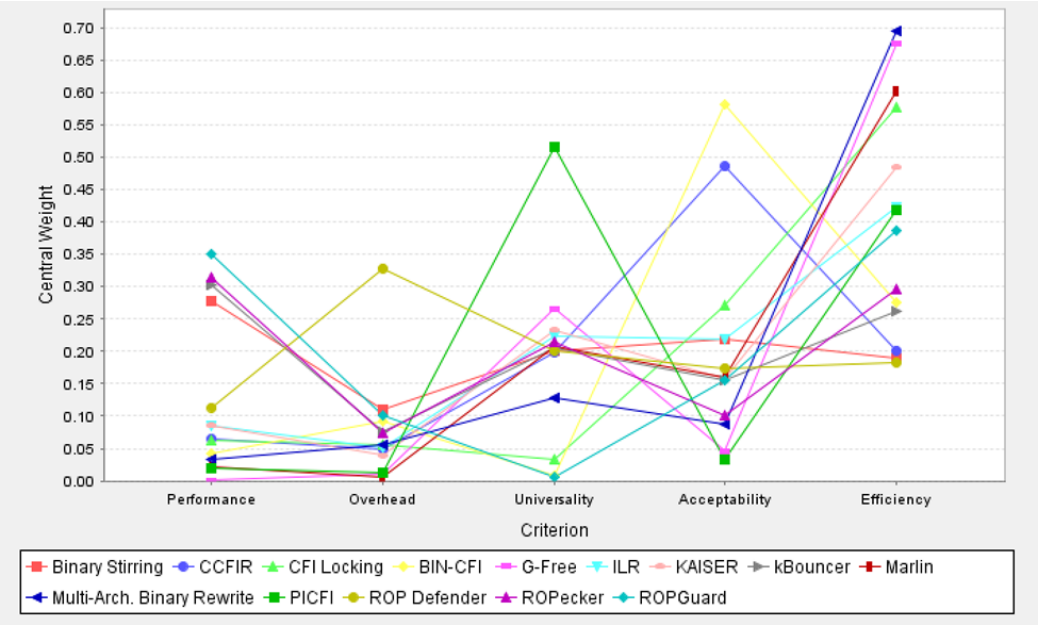

Figure 17: The central weight analysis of the After feeding the JSMAA model with the values discussed earlier across different criteria.

One of the added features to SMAA-2 over SMAA is the integration of DM preference as a part of the selection bias process. The preference parameter can represent the bias of the DM depending on prior experience with the alternatives or the situation where some criteria are favored over others. For our experiment, we have added preference information to the second experiment to analyze the results after the selection bias is introduced. The preference information is an ordinal ascending ranking of the five criteria. We put performance as the first selection preference, the second preference is Binary size overhead, the third preference is Universality, the fourth is acceptability, and the last one is efficiency.

FIGURE. 18, represents the analysis results after applying the selected preference information. We can see that when performance is given the first preference when ranking alternatives, the ROPGuard mechanism has the highest central weight due to the best performance results. The other alternatives can be ranked as per our selection of performance. The preference for the performance results in five favored CRAs (ROPGuard, ROP Defender, ROPecker, kBouncer, and Binary Stirring). Other CRAs are omitted as DM options since our selected preference information will not impact the ranking of those alternatives.

The ranking acceptability incidence diagram, FIGURE. 19, below, shows the ranking of the alternatives based on the analysis of the criteria and weight information. The ranking information is represented as a weight distribution. The color legends in the picture are used to represent the ranking information. For example, rank 1 is colored whitish red, and you will find that the alternative with the lightest red color index is Binary stirring. Therefore, binary stirring is ranked first because it has the highest weight distribution of criteria that makes this alternative rank number 1. The main

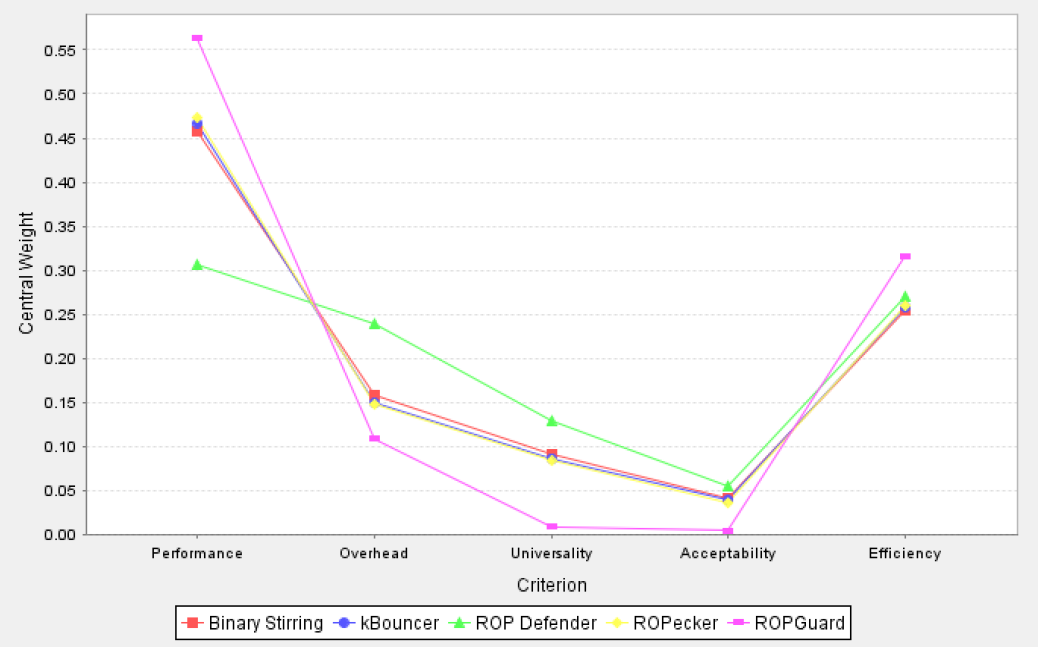

Figure 18: The analysis results after applying the selected preference information.

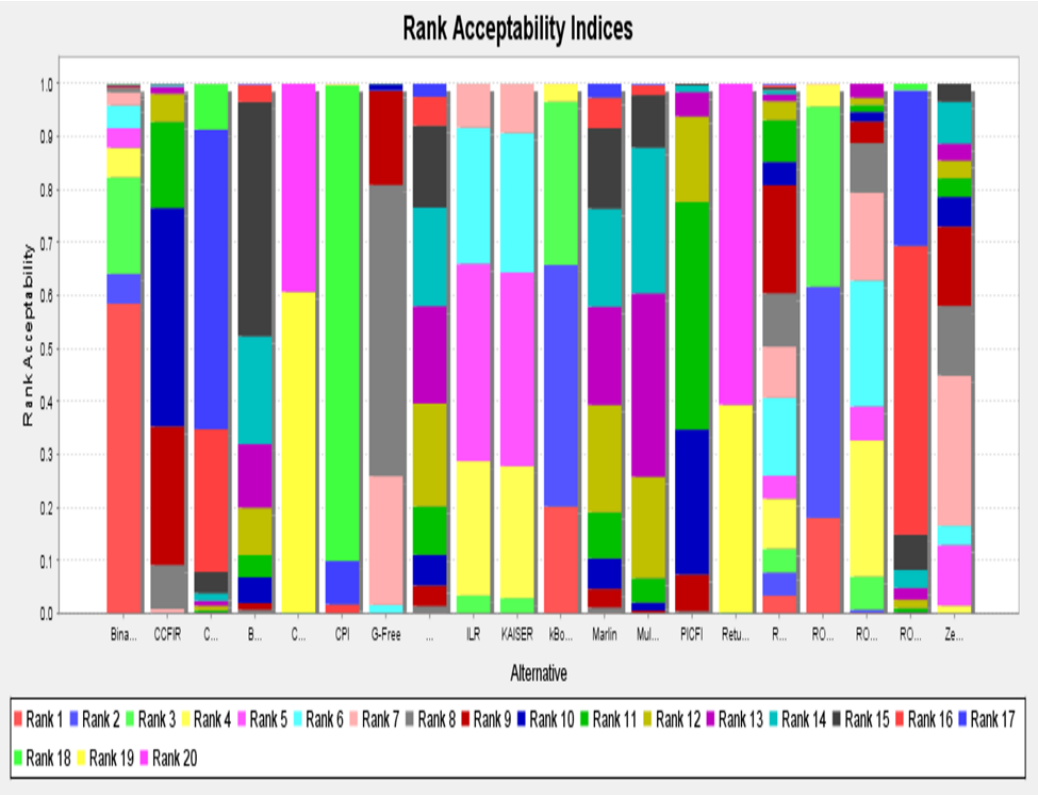

Figure 19: The ranking of the alternatives based on the analysis of the criteria and weight information

advantage of SMAA-2 is a more methodical mechanism to rank alternatives based on the weight distribution of the selected criteria. The remaining ranks from rank number 2 to rank number 20 can be explained in the same manner.

TABLE 3, below shows the ranking of the alternatives under evaluation. This ranking information is extracted from the rank acceptability indices calculations. The rank with the highest weight for each alternative is considered the ordering ranking of that alternative, aiding in the sorting and ranking of multiple alternatives.

Table 3: Ranking of the CRA Mitigations using SMAA-2 Approach

Alternative	Rank
Binary Stirring	1
kBouncer	2
ROPecker	3
KAISER	4
In Place Rand	5
ROPGuard	6
Zero-Sum Defender	7
G-Free	8
ROP Defender	9
CCFIR	10
PICFI	11
Marlin	12
ILR	13
Multi-Arch. Binary Rewrite	14
BIN-CFI	15
ROPScan	16
CFI Locking	17
CPI	18
CFI Original	19
Return-Less Kernel	20

5.1 Discussion and Limitations

This experimental result provides better insight into the ranking and evaluation of different code-reuse attack mitigations. It addresses some analysis challenges, such as the lack of published results and selection bias. Applying SMAA-2 to different CRAs has proven to be useful in ranking mitigation techniques and visualizing the weight distribution of the selected criteria. The freedom given by SMAA-2 in choosing the preference model and the acceptability index can be utilized by researchers to enhance the CRA mitigation comparison results. Moreover, it allows fine-tuning of criteria parameters to better fit different applications and use-cases in future comparisons.

The SMAA-2 approach can provide guidance to decision-makers in the ranking and ordering process of different alternatives; however, it mainly depends on the input fed to the algorithm. Changing input parameters from one type to another (ordinal to cardinal and vice versa) will impact the overall

results. Therefore, it is crucial for the evaluator to carefully choose the type of parameter and align the parameter nature with the criteria type and use case.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel approach to evaluate and rank CRAs mitigation using the SMAA-2 model. This approach enables researchers to conduct a more thorough analysis of the evaluation criteria and adapt these criteria to different situations, depending on the evaluator's preferences or desired use case. The ranking results of the analysis are subjective due to the applied weighting model; however, they provide a clear indication of the weight factors that may make certain code-reuse mitigation techniques more suitable for specific use cases or implementation scenarios.

Moreover, this paper provides a detailed analysis of memory safety issues, attacks, and protection mechanisms. Code re-use attacks and code re-use mitigations are extensively discussed and categorized, making it easier for analysis and collective evaluation. Mitigating code-reuse attacks is a complex topic that requires a deep understanding of the details and know-how as a prerequisite for this discipline. It is recommended to adopt flexible and dependable approaches when evaluating or presenting attack scenarios and mitigation mechanisms. Future efforts could focus on gaining a better understanding of the internals of advancing mitigations and proposing attack models to bypass the proposed mitigations. Additionally, a formal model could be introduced to aid in the evaluation of CRA mitigations against various types of memory attacks, not just code-reuse attacks.

Author contribution

(1) AME made substantial contributions to the design of the work and drafted it. (2) MSE revised it critically for important intellectual content; (3) AJ and MAA approved the version to be published; (4) all the authors agree to be accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved.

Funding

This research is funded by the University College Dublin, School of Computer Science.

Data availability statement

Not applicable.

Declarations

Conflict of interest

The authors declare that they have no conflict of interest.

Ethical approval

Not applicable. This article does not contain any studies with animals performed by any of the authors.

Informed consent

Not applicable.

References

[1] Gross M, Jacob N, Zankl A, Sigl G. Breaking Trust Zone Memory Isolation and Secure Boot Through Malicious Hardware on a Modern Fpga-Soc. J. Cryptogr. Eng. 2022:1–16.

[2] Calatayud BM, Meany L. A Comparative Analysis of Buffer Overflow Vulnerabilities in High-End Iot Devices. In: 12th Annual Computing and Communication Workshop and Conference (CCWC). Vol. 2022. IEEE PUBLICATIONS. 2022:694-701.

[3] Meer H, et al. Memory Corruption Attacks: The (Almost) Complete History. Blackhat USA. 2010.

[4] Schloegel M, Blazytko T, Basler J, Hemmer F, Holz T. Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains. In: Comput Sec–ESORICS 26th European Symposium on Research in Computer Security .2021;26:218-239.

[5] Schilling R, Nasahl P, Mangard S. Fipac: Thwarting Fault- And Software-Induced Control-Flow Attacks With Arm Pointer Authentication. In: International Workshop on Constructive Side-Channel Analysis and Secure Design. 2022:100-124.

[6] Gao Yc, Zhou Am, Liu L. Data-Execution Prevention Technology in Windows System, Information Security Communications Privacy. 2013.

[7] Hund R, Willems C, Holz T. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In 2013: IEEE Symposium on Security and Privacy. IEEE PUBLICATIONS. 20131:191-205.

[8] Szekeres L, Payer M, Wei T, Song D. Sok: Eternal War in Memory. In: 2013 IEEE Symposium on Security and Privacy. IEEE PUBLICATIONS. 2013:48-62.

[9] Available from:https://www.isms.online/iso-27002/control-8-26-application-security-requirements/,accessed.

[10] Available from:https://nvd.nist.gov/vuln/detail/CVE-2018-5392.

[11] Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, et al. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In 2016: IEEE Symposium on Security and Privacy (SP). IEEE PUBLICATIONS. 2016:969-986.

[12] Lahdelma R, Salminen P. Smaa-2: Stochastic Multicriteria Acceptability Analysis for Group Decision Making. Operations research. 2001;49:444-454.

[13] Etoh H. Propolice: Improved Stack-Smashing Attack Detection. IEICE Technical Report; 2001:ISEC2001-43.

[14] Cowan C, Beattie S, Johansen J, Wagle P, Pointguard™: Protecting Pointers From Buffer Overflow Vulnerabilities. In: 12th USENIX Security Symposium (USENIX Security 03). 2003.

[15] Cowan C, Pu C, Maier D, Walpole J, Bakke P, et al. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: USENIX security symposium. 1998;98:63-78.

[16] Dang TH, Maniatis P, Wagner D. The Performance Cost of Shadow Stacks and Stack Canaries. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. 2015:555-566.

[17] Zhang C, Song C, Chen KZ, Chen Z, Song D. Vtint: Protecting Virtual Function Tables' Integrity. In Proceedings of 2015 Network and Distributed System Security Symposium. NATIONAL DOWN SYNDROME SOCIETY. 2015.

[18] Lee B, Song C, Kim T, Lee W. Type Casting Verification: Stopping an Emerging Attack Vector. In: 24th USENIX Security Symposium (USENIX Security 15). 2015:81-96.

[19] Cox B, Evans D, Filipi A, Rowanhill J, Hu W, et al. N-Variant Systems: A Secretless Framework for Security Through Diversity. In: USENIX Security Symposium. 2006;114:114.

[20] Evans I, Fingeret S, Gonzalez J, Otgonbaatar U, Tang T, et al. Missing the Point (ER): On the Effectiveness of Code Pointer Integrity. In2015 IEEE Symposium on Security and Privacy. IEEE PUBLICATIONS. 2015:781-796.

[21] P. Team, Pax address space layout randomization (aslr), http://pax. grsecurity. net/docs/aslr. txt (2003).

[22] Van der Veen V, Dutt-Sharma N, Cavallaro L, Bos H. Memory Errors: The Past, the Present, and the Future. In: Research in Attacks, Intrusions, and Defenses. Proceedings 15: 15th International Symposium. 2012:86-106.

[23] Castro M, Costa M, Martin JP, Peinado M, Akri Ptidis, et al. Fast Bytegranularity Software Fault Isolation. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. 2009:45-58.

[24] Snow KZ, Rogowski R, Werner J, Koo H, Monrose F, et al. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In 2016 IEEE Symposium on Security and Privacy (SP). IEEE PUBLICATIONS. 2016:954-968.

[25] Wartell R, Mohan V, Hamlen KW, Lin Z. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy X86 Binary Code. In: ACM Conference on Computer and Communications Security. 2012:157-168.

[26] Kil C, Jun J, Bookholt C, Xu J, Ning P. Address Space Layout Permutation (Aslp): Towards Fine-Grained Randomization of Commodity Software. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC'06). IEEE PUBLICATIONS. 2006:339-348.

[27] Hiser J, Nguyen-Tuong A, Co M, Hall M, David JW. Ilr: Where'd My Gadgets Go?In: 2012 IEEE Symposium on Security and Privacy. IEEE PUBLICATIONS. 2012:571-585.

[28] D'Silva V, Payer M, Song D. The Correctness-Security Gap in Compiler Optimization. In: 2015 IEEE Security and Privacy Workshops. IEEE PUBLICATIONS. 2015:73-87.

[29] Wojtczuk R. The Advanced Return-Into-Lib (C) Exploits: pax case study, Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile 0x04 of 0x0e. 2001;70:227-242.

[30] Bratus S, Locasto ME, Patterson ML, Sassaman L, Shubina A. Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation. USENIX; login. 2011;36:13-21.

[31] Backes M, Holz T, Kollenda B, Koppe P, Nürnberger S, et al. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014:1342-1353.

[32] S. Designer, return-to-libc" attack, Bugtraq, Aug (1997).

[33] Novark G, Berger ED. Dieharder: Securing the Heap. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. 2010:573-584.

[34] Shacham H. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc Without Function Calls (On the X86). In: Proceedings of the 14th ACM conference on computer and communications security. 2007:552-561.

[35] Forrest S, Somayaji A, Ackley DH. Building Diverse Computer Systems. In: The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133). IEEE PUBLICATIONS; 1997:67-72.

[36] Rudd R, Skowyra R, Bigelow D, Dedhia V, Hobson T, et al. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. Proceedings 2017 Network and Distributed System Security Symposium. NATIONAL DOWN SYNDROME SOCIETY. 2017.

[37] Shacham H, Page M, Pfaff B, Goh EJ, Modadugu N, et al. On the Effectiveness of Address-Space Randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security; 2004:298-307.

[38] Liu L, Han J, Gao D, Jing J, Zha D. Launching Return-Oriented Programming Attacks Against Randomized Relocatable Executables. In: 10th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE PUBLICATIONS. 2011:37-44.

[39] Bhatkar S, DuVarney DC, Sekar R. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: 12th USENIX Security Symposium (USENIX Security 03).2003.

[40] Carlini N, Wagner D, Rop Is Still Dangerous: Breaking Modern Defenses. In: 23rd USENIX Security Symposium (USENIX Security 14). 2014:385-399.

[41] Srivastava A, Edwards A, Vo H. Vulcan: Binary Transformation in a Distributed Environment. Technical report msr-tr-2001-50. microsoft research. 2001.

[42] Erlingsson M, Jigatti J. Control-Flow Integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. 2005:340-353.

[43] Burow N, Carr SA, Nash J, Larsen P, Franz M, et al. Control-Flow Integrity: Precision, Security, and Performance. ACM Computing Surveys (CSUR). 2017;50:1-33.

[44] Abadi M, Budiu M, Erlingsson U, Ligatti J. Control Flow Integrity Principles, Implementations, and Applications. ACM transactions on information and system (TISSEC). 2009;1:1-40.

[45] Wahbe R, Lucco S, Anderson TE, Graham SL. Efficient Software-Based Fault Isolation. In: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles. 1993:203-216.

[46] Davi L, Sadeghi AR, Winandy M. Ropdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. 2011:40-51.

[47] Homescu A, Stewart M, Larsen P, Brunthaler S, Franz M. Microgadgets: Size Does Matter in Turing Complete Return-Oriented Programming. WOOT. 2012;12:64-76.

[48] Diatchki I, Pike L, Erkök L. Practical Considerations in Control-Flow Integrity Monitoring. In 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. Vol. 2011. IEEE PUBLICATIONS. 2011:537-544.

[49] Li J, Wang Z, Jiang X, Grace M, Bahram S. Defeating Return-Oriented Rootkits With Return-Less Kernels. In: Proceedings of the 5th European Conference on Computer Systems. 2010:195-208.

[50] Schuster F, Tendyck T, Liebchen C, Davi L, Sadeghi AR, et al. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In 2015 IEEE Symposium on Security and Privacy. IEEE PUBLICATIONS. 2015:745-762.

[51] Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, et al. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In 2013 IEEE symposium on security and privacy. IEEE PUBLICATIONS. 2013:574-588.

[52] Zhang M, Sekar R. Control Flow and Code Integrity for Cots Binaries: An Effective Defense Against Real-World Rop Attacks. In: Proceedings of the 31st Annual Computer Security Applications Conference. 2015:91-100.

[53] Niu B. Practical Control-Glow Integrity, lehigh University. 2016.

[54] Bletsch T, Jiang X, Freeh V. Mitigating Code-Reuse Attacks With Control-Flow Locking. In: Proceedings of the 27th Annual Computer Security Applications Conference. 2011:353-362.

[55] Zhang C, Wei T, Chen Z, Duan L, Szekeres L, et al. Practical Control Flow Integrity and Randomization for Binary Executables. In 2013 IEEE Symposium on Security and Privacy. IEEE PUBLICATIONS. 2013:559-573.

[56] Valensi C. Madras: Multi-Architecture Binary Rewriting Tool Technical Report. 2013.

[57] Wang S, Wang P, Wu D. Uroboros: Instrumenting Stripped Binaries With Static Reassembling. In 2016 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE PUBLICATIONS. 2016:236-247.

[58] Pappas V, Polychronakis M, Keromytis AD. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In: IEEE Symposium on Security and Privacy. IEEE PUBLICATIONS .2012:601-615.

[59] Chen Y, Zhang D, Wang R, Qiao R, Azab AM, et al. Norax: Enabling Execute-Only Memory for Cots Binaries on AARCH64. In: IEEE Symposium on Security and Privacy (SP). IEEE PUBLICATIONS. 2017:304-319.

[60] Pappas V. Kbouncer: Efficient and Transparent Rop Mitigation. 2012:1-2.

[61] Cheng Y, Zhou Z, Yu M, Ding X, Deng RH. Ropecker: A Generic and Practical Approach for Defending Against Rop Attacks. 2014.

[62] https://www.ieee.hr/$_{d}ownload/repository/Ivan_{F}ratric.pdf$

[63] Chen P, Xiao H, Shen X, Yin X, Mao B. Drop: Detecting Return-Oriented Programming Malicious Code. In Information Systems Security: 5th International Conference, ICISS 2009. Proceedings 5 Springer. 2009:163-177.

[64] Kim J, Kim I, Min C, Eom YI. Zero-Sum Defender: Fast and Space-Efficient Defense Against Return-Oriented Programming Attacks, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences. 2014;97:303-305.

[65] Gruss D, Lipp M, Schwarz M, Fellner R, Maurice C, et al. Kaslr Is Dead: Long Live Kaslr. In: Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017. Proceedings 9 Springer. 2017:161-76.

[66] Gupta A, Kerr S, Kirkpatrick MS, Bertino E. Marlin: A Fine Grained Randomization Approach to Defend Against Rop Attacks. InNetwork and System Security: 7th International Conference. 2013:293-306.

[67] Polychronakis M, Keromytis AD. Rop Payload Detection Using Speculative Code Execution. In 2011 6th International Conference on Malicious and Unwanted Software. IEEE PUBLICATIONS. 2011:58-65.

[68] El-Zoghby AM, Azer MA. Survey of Code Reuse Attacks and Comparison of Mitigation Techniques. In: Proceedings of the 9th International Conference on Software and Information Engineering; 2020:88-96.