UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
INFORMATICS

SECURITY AND APPLIED LOGIC

Master's Thesis

TECHNIQUES FOR REVERSE ENGINEERING VM-BASED OBFUSCATION

Graduate
Radu Ștefan-Octavian

Scientific Coordinator
Conf.Univ.Dr.Ing. Paul Irofti

București, September 2024

Rezumat

Atacurile cibernetice s-au întețit în ultima perioadă, iar unul dintre pericolele principale, atât pentru companii, cât și pentru utilizatori, este infectarea cu malware. Ingineria inversă este una dintre ariile principale in care activează cei care asigură protecția împotriva atacurilor malware. Profesioniștii în domeniu, trebuie să studieze și să înțeleagă comportamentul instanțelor de malware pe care le descoperă, pentru a putea îmbunătății sistemele de apărare și detecție, prin integrarea noilor informații. Această sarcină este foarte dificilă, din cauza multiplelor straturi de obfuscare care sunt introduse de dezvoltatorii de malware, pentru a înlesni înțelegerea programelor. Deși tehnicile de obfuscare au evoluat considerabil în ultimii ani, una dintre cele mai greu de contracarat rămâne obfuscarea bazată pe virtualizare.

În această lucrare, ne propunem să integrăm, în procesul de inginerie inversă a malware-urilor obfuscate prin virtualizare, un framework pentru analiza binarelor, numit angr. Pentru a realiza acest lucru, am creat `arch-genesis`, un utilitar care simplifică procesul de construire a unor plugin-uri pentru angr, care permit utilizarea functionalităților de analiză ale framework-ului, direct pe bytecode-ul unei mașini virtuale non-standard. Vom descrie arhitectura și funcționalitățile utilitarului nostru, precum și cum îl putem folosi pentru contracararea obfuscării bazate pe virtualizare.

Abstract

Cybercrime is on the rise, and one of the main threats to companies as well as end users is being infected with malware. Reverse engineers are one of the core pillars of the defending side, being tasked with studying and documenting these malicious pieces of software, with the goal of improving detection and prevention systems. Their job is very difficult, because they often have to bypass multiple complex layers of obfuscation, which make understanding the behaviour of the program require a lot more effort. Despite obfuscation techniques continuing to evolve, virtualisation-based obfuscation remains one of the most difficult to overcome techniques.

In this thesis, we work on integrating angr, a popular binary analysis framework, in the current workflow for reverse engineering virtualisation-based obfuscation. As part of our work, we propose `arch-genesis` a tool which streamlines the process of building angr plugins for a custom Virtual Machine architecture. Such plugins bridge the gap between the unknown architecture and the suite of features that angr offers to its users. We cover the architecture of our tool, its functionalities, as well as the analysis process while using it.

Keywords: Virtualisation-based Obfuscation, Reverse Engineering, angr, VEX, Symbolic Execution

Contents

Acronyms **5**

1 Introduction **7**

1.1 Context 7

1.2 Contribution 8

1.3 Outline 8

2 Background **10**

2.1 Malware 10

2.1.1 Classification 10

2.2 Reverse Engineering 12

2.3 Static Analysis 13

2.4 Dynamic Analysis 15

2.4.1 Debugging 16

2.4.2 Function Call Analysis 16

2.4.3 Dynamic Taint Analysis 17

2.5 Mixed Analysis Techniques 18

2.5.1 Symbolic Execution 18

2.5.2 Concolic Execution 20

2.6 Obfuscation Techniques 20

2.7 VEX - Intermediate Representation 21

3 State of the Art **24**

3.1 Virtualization-based Obfuscation 24

3.1.1 Internals 24

3.2 Academic Work 25

3.2.1 Semi-Manual Approaches 26

3.2.2 Trace Simplification 26

3.2.3 Program Synthesis 27

3.3 Discussion 28

4 Our approach 29
4.1 Overview 29
4.1.1 Assumptions 30
4.2 Understanding the VM 30
4.2.1 Static Analysis 30
4.2.2 Automatic Code Summarisation using Miasm 33
4.3 Summary of Analysis 35
4.4 Building an angr Architecture Plugin 37
4.4.1 Extending the Arch Database 37
4.4.2 Writing a Loader 39
4.4.3 Writing a Lifter 39
4.4.4 Writing a SimOS 41
4.5 Plugin Generation - arch-genesis 41
4.5.1 Disassembler 43
4.6 Further Analysis 44
4.6.1 Solving the Challenges 44
4.7 Discussion 49
4.7.1 angr vs Miasm 49
4.7.2 Difficulties and Shortcomings 49
4.7.3 Future Directions 51
5 Conclusions 53
List of Listings 54
List of Figures 55
Glossary 58
Bibliography 60
Appendices 66

Acronyms

BB Basic Block. 14, 43, 51, 58, *Glossary:* BB

BF Brainf*ck. 37, *Glossary:* BF

BNF Backus-Naur Form. 44, 47, 52, *Glossary:* BNF

CC Command and Control. 12

CE Concrete Execution. 18

CFG Control Flow Graph. 7, 14, 20, 21, 26, 27, 29–32, 44–46, 53, 56, 57, *Glossary:* CFG

CLE CLE Loads Everything. 39

CPU Central Processing Unit. 14, 21, 24

CTF Capture the Flag. 29, 35, 49, *Glossary:* CTF

DSE Dynamic Symbolic Execution. 20

ELF Executable and Linkable Format. 29, *Glossary:* ELF

IP Instruction Pointer. 24, 26, 32

IR Intermediate Representation. 10, 21, 22, 26–28, 38–40, 49, 52, 54, 56, *Glossary:* IR

IRSB Intermediate Representation (IR) Super Block. 22

ISA Instruction Set Architecture. 24, 30, 35–37, 41, 44, 49–53, *Glossary:* ISA

MBA Mixed Boolean-Arithmetic. 20, 28, 35

NSA National Security Agency. 14

OS Operating System. 14, 36, 41

RE Reverse Engineering. 10, 12–14, 20, 29–31, 56

SE Symbolic Execution. 13, 18–20, 54

SMT Satisfiability Modulo Theories. 19, 28, 38, 47, 56

SP Stack Pointer. 24, 32

stdin standard input. 30

syscall System Call. 36, 38, 41

UI User Interface. 14

VM Virtual Machine. 7, 8, 21, 24–36, 42, 44, 49–53, 55, 56

Chapter 1

Introduction

1.1 Context

In the past years, the internet has become a significantly more hostile space. With cybercrime on the rise, the subject of malware, and its detrimental impact on businesses and society as whole, is more relevant than ever before.

Malware attacks, which are one of the most damaging types of cybercrime, can target governments, corporations, public figures, or individuals. Multiple sources, including a report from the World Economic Forum [1], suggest that cybercrime continues to rise both in numbers as well as in damage. The estimated costs of cybercrime in 2023, at a global scale, are of $11.5 trillion, and these numbers are expected to more than double in the next five years. Because of this, malware, and particularly the subject of malware analysis, are evermore relevant. Analysing malware, and in particular the protection schemes built around malware is a very important topic, and the main subject of this work.

In order to analyse, understand and mitigate malware, professionals use various reverse engineering techniques and tools. However, obfuscation is often used by cybercriminals that develop malware, in order to make reverse engineering as hard as possible. There have been a wide range of obfuscation techniques developed over the years. The target of this thesis is a technique called *virtualization-based* obfuscation, which has been regarded for over a decade as one of the most difficult to reverse engineer obfuscation schemes available [19].

Various approaches have been proposed over the years, on the topic of defeating virtualisation-based obfuscation. Some of these include manual analysis of the Virtual Machine (VM) [46], and others include automated or semi-automated approaches based on dynamic taint analysis, symbolic execution, compiler optimisation techniques, etc. [52] [48]. The goals of the proposals differ as well. Some of them manage to recover a semantically equivalent version of the protected code [48], while others claim to recover the Control Flow Graph (CFG) of the original program [63]. Only a few of the approaches work directly on the bytecode [50], most of them aiming to strip away the obfuscation layer in an automated manner.

1.2 Contribution

We believe that there is a significant need for new techniques integrating modern and promising frameworks in the process of analysing obfuscated code. We propose using angr in order to run a wide range of analyses directly on the bytecode of the custom VM. To achieve this, we need an intermediate layer that bridges the gap between the powerful execution engine in angr and the unknown architecture that we are targeting. In this work, we describe how we approached building this intermediate layer, in the form of angr plugins. We also propose a tool that simplifies the process of building such a plugin. We apply our tools in order to analyse two samples obfuscated using virtualisation-based obfuscation.

The main contributions in this dissertation are:

- Providing a concise, yet comprehensive, overview of the current state of the art on the topic of virtualisation-based obfuscation;
- Describing a new approach to reverse engineering bytecode of software obfuscated through virtualisation, based on the angr framework;
- Developing the `arch-genesis` tool, which streamlines the generation of angr plugins, and which can automatically generate a disassembler for the target bytecode;
- Describing the analysis process on two samples, using the developed tool.

1.3 Outline

The remainder of the dissertation is structured as follows:

- In Chapter 2 we give a technical background for the topics covered in the dissertation. We cover malware, reverse engineering techniques including static, dynamic and mixed analysis approaches, obfuscation techniques and the VEX intermediate representation;
- In Chapter 3, we start by giving a technical overview of the virtualisation-based obfuscation technique. We then present the state of the art on the subject, and compare our work with previous approaches;
- In Chapter 4, we present our contribution. We discuss our idea, the analysis techniques we employed, the implementation, and results. We end with a discussion on shortcomings and future directions;

- We express our conclusions in Chapter 5.

Chapter 2

Background

In this chapter we will cover various concepts which will be relevant throughout this dissertation. We start with defining what malware is, types of malware and their relevance. We continue by discussing Reverse Engineering (RE) and the different approaches to program analysis. We discuss obfuscation techniques, which are meant to make RE harder, and introduce the main topic of this work: virtualisation-based obfuscation. We end the chapter with a short overview of the VEX Intermediate Representation (IR), a core aspect of our proposed solution.

2.1 Malware

The word *malware* is a blend word shortening the phrase "**mal**icious soft**ware**". It is an umbrella term encompassing any type of software that is intentionally designed to disturb the intended use or operation of a computer system, without the explicit permission of its user(s) or owners. Malware can be written to achieve a wide array of goals, including, but not limited to: leakage or collection of private information, restriction of access to data with the goal of monetary gains, network overload, device hijacking, espionage, serving of targeted adds [14].

2.1.1 Classification

Malware is typically grouped by its behaviour and/or purpose and could fall under one or more of the labels in the following non-exhaustive list: backdoor, bot(net), dropper, fileless, ransomware, rootkit, spyware, trojan, virus, worm, etc. [14], [65]. We will shortly cover the most relevant ones as follows.

Backdoor

A Backdoor is an umbrella term for software that enables bad actors to obtain persistent, unauthorised access to a victim's computer, typically with them being unaware of the situation. Backdoor software is interesting because they can either be delivered through a Trojan, a worm, or another similarly purposed malware, but they can also be the result of vulnerabilities existing in legitimate software, that already exist on the victim's computer.

What is more, there is also a combined scenario, where a backdoor is intentionally inserted into legitimate software. This can be done, for instance, by a bad actor hiding their intention. A prominent example of this comes from the 2024 discovery of the XZ Utils backdoor [3].

Trojan

A Trojan, or a Trojan Horse, is a type of malware that conceals itself inside another program that appears benign. It typically misinforms the target about its behaviour in order to persuade a victim to install it on their computer. Trojans are usually delivered to the victim by some form of social engineering.

The payload of a Trojan can be anything. It often is another type of malware, case in which, the Trojan is considered a Dropper. It can also be the case that the Trojan deploys a backdoor which can enable unauthorised control of the infected system to a third party actor [14].

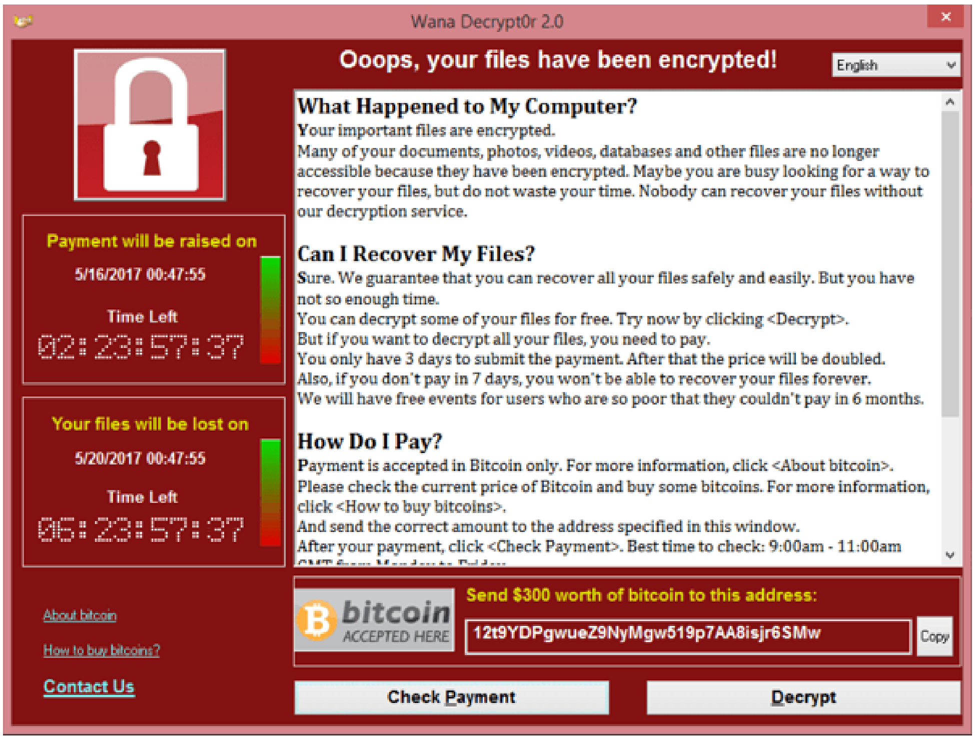
Worm / Virus

Worms and Viruses are similar in the sense that these are both stand-alone malware that have the capacity to spread through a network, and to infect other victims. A Virus (inspired by the biological term) will inject itself into seemingly harmless programs, that upon execution will further spread the infection.

Worms differ from Viruses in the sense that a virus requires the victim to execute the infected software in order for it to spread, whereas a Worm does not. Worms can spread without user intervention and without modifying other files on the system [14]. An example of such a piece of malware is the infamous Stuxnet virus [31].

Ransomware

Ransomware is a type of malware that, once it has infected a computer, restricts access to information on that particular machine, and then asks for a ransom from the victim, in exchange for the locked up information. Most commonly, a form of encryption is applied to the restricted data, which in properly executed attacks cannot be recovered without the encryption key. Typically, ransomware is delivered via a Trojan, but this is not always the case. The infamous *WannaCry* ransomware was a *worm* which spread through the network without user intervention [17], [15]. As it can be seen in Figure 2.1, the attackers typically ask for hard to trace digital currencies, such as Bitcoin in this case.



Ooops, your files have been encrypted!

Payment will be raised on
5/16/2017 00:47:55
Time Left
02:23:57:37

Your files will be lost on
5/20/2017 00:47:55
Time Left
06:23:57:37

What Happened to My Computer?
Your important files are encrypted.
Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?
Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.
You can decrypt some of your files for free. Try now by clicking <Decrypt>.
But if you want to decrypt all your files, you need to pay.
You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don't pay in 7 days, you won't be able to recover your files forever.
We will have free events for users who are so poor that they couldn't pay in 6 months.

How Do I Pay?
Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.
Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.
And send the correct amount to the address specified in this window.
After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am

Send $300 worth of bitcoin to this address:
12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw

Figure 2.1: An infamous screenshot of the ransom pop-up which would show up on a system infected by the WannaCry worm [17].

Bots and Botnets

A Bot is a computer infected by a specific type of malware which enables its victim to be remotely controlled. Such malware is spread with the goal of infecting as many targets as possible. The infected machines are added to a common pool, called a *botnet*, which can be orchestrated from a Command and Control (CC) centre to perform other malicious activities on a bigger scale. The Andromeda botnet is an example of such malware [47] [61].

2.2 Reverse Engineering

As stated on *Wikipedia*: *"RE is a process or method through which one attempts to understand through deductive reasoning how a previously made device, process, system, or piece of software accomplishes a task, with very little (if any) insight into exactly how it does so"* [16]. It is analogous to scientific research performed on man-made products.

In this work we will focus on reverse engineering pieces of software in the context of security research or malware analysis. Typically, in such contexts, the goal is to understand the behaviour of a piece of software, without having access to its source code. However, depending on the type of analysis, a reverse engineer might have different approaches.

They might focus on gaining a comprehensive understanding of specific parts of the software in order to identify weaknesses, or more commonly named *vulnerabilities*. This analysis could be restricted to specific parts for multiple reasons which include, but are not limited to: the full program being too big to justify performing a full analysis, or the existence of prior knowledge which gives higher priority to the analysis of certain code regions. With the knowledge obtained from RE, the researcher can identify and prove the existence of attack vectors on a system that is running this software. They might then write a report which covers the risks that the entities running the software are exposed to, describing the findings in detail, and exemplifying how an attacker might abuse the discovered vulnerabilities. The end goal of this sequence of steps is to enhance the security of the analysed product.

In other instances, the engineers might perform a full and comprehensive analysis of piece of software. This is typically done when dealing with malware. The malware analyst will first try to determine if the piece of software is in fact malicious or not. If the code is malicious, it is important to determine its behaviour, how it interacts with the system, or with outside entities (possibly by creating network traffic). During this process, analysts might study and document novel techniques employed by attackers. They might also integrate the newly found malware into a detection system to prevent future uses of the respective malware [36].

Regardless of the goal, RE falls into, or somewhere in between two broad categories which determine the typical approach and the tooling used: **static analysis** and **dynamic analysis**.

2.3 Static Analysis

Static analysis represents the multitude of techniques used to analyse a program without executing it. These techniques range in difficulty and complexity, starting with source code auditing, to reading assembly, attempting to decompile binaries, and ultimately using very advanced tools and theoretical knowledge such as Symbolic Execution (SE) [1] engines, SMT solvers [64] or formal methods.

In its most basic form, static analysis is equivalent with reading the source code in order to understand what the program does. However, in the context of this dissertation, we are dealing with binary files, resulted from compiling code to machine code. We assume that the source code is not available to us, so we must resort to other analysis techniques. One option is to convert the machine code into the human readable form, called assembly. The process is known as machine code disassembly. Assembly code is typically very hard to understand for humans, but given enough time and effort, the logic of the program can be successfully recovered from it.

One advantage of static analysis through reading assembly is that the entry barrier

[1] Albeit, it is debatable if SE can be considered static or dynamic analysis. We will, consider it a mixed approach, and discuss it accordingly.

is not high in terms of the tooling required. A very basic tool such as `objdump` [41] can be enough for simple programs, but most likely a more feature rich tool such as `radare2` /`cutter` [18] are more suitable, as these can display Basic Block (BB)s and how all such BB relate to each other and form the CFG.

Reverse engineers default to advanced static analysis tools, such as IDA or Ghidra [25], [24]. On one hand, IDA is the industry standard of RE and as a complete package it is an expensive piece of software. On the other hand, Ghidra is a RE framework, developed by the National Security Agency (NSA), which was open-sourced in March 2019 [40]. Despite its dated looks and non-friendly User Interface (UI), Ghidra's powerful set of features and the fact that it is free to use by anyone, made it gain considerable popularity in the reverse engineering community.

Both IDA and Ghidra feature a large suite of functionalities, out of which, probably the most prominent is the *decompiler*. Compilation is the process of converting source code into machine code. Decompilation is the opposite: the process of converting machine code, back into source code. Compared to the disassembly process, which is deterministic and corresponds exactly to the assembly process, the decompilation process will almost never yield back the original source code. This is the case because a lot of information useful to programmers, but useless for the Operating System (OS) or the Central Processing Unit (CPU) is lost during the compilation process. This information includes, but is not limited to: variable and function names, type information, custom defined data types such as structures, or specific language features. This is why decompilers will always output an approximation of the original code.

Let us consider a concrete example in Listings 2.1, 2.2 and 2.3. Listing 2.1 contains the implementation of the function `add`, which takes the end of linked list and a value, and creates a new node in the list with that value, also taking the necessary steps to update the list accordingly. The code is part of a slightly larger `C` program, which we compiled and imported into Ghidra. Looking at Listing 2.2 we can see the decompilation of the code referenced in the previously mentioned Listing.

It is immediately obvious that the type information related to `node` structure is completely lost and that the original function was inlined by the compiler. As a result, the decompilation is an obfuscated version of the original code. This is a well known fact and advanced tools such as Ghidra offer various features, which a reverse engineer can utilise in order to remove part of the obfuscation. Listing 2.3 contains the same segment of decompiled code after a minimal amount of manual intervention, which includes: variable renaming, custom type creation and type updates. Clearly, it is a lot more human readable and bears a closer resemblance to the original code in Listing 2.1.

Decompilers and disassemblers are very powerful tools, which aid significantly in the process of static analysis. However, these tools have their shortcomings as highlighted above. Moreover, there are certain program behaviours which cannot, or are significantly harder to understand only by *looking* at the code. One such scenario, is when the code is heavily obfuscated. We will cover obfuscation later, in Section 2.6.

```c
void add(lnode** node, int v) {
    // allocate memory for a new node in the liked list
    lnode* new_node = (lnode*) malloc(sizeof(lnode));
    new_node->val = v; // set the value
    if (*node != NULL) {
        (*node)->nxt = new_node; // link to the new node from the end of the list
    }
    *node = new_node; // move the list end to the new node
}
```

Listing 2.1: A function which adds an integer value `v` to the end of a linked list.

```c
piVar1 = (int *)malloc(0x10);
*piVar1 = iVar3;
if (piVar5 != (int *)0x0) {
    *(int **)(piVar5 + 2) = piVar1;
}
piVar5 = piVar1;
if (piVar4 == (int *)0x0) {
    piVar4 = piVar1;
}
```

Listing 2.2: Ghidra decompilation of the code presented in Listing 2.1. The decompilation is take as-is and has not modified in any way.

```c
new_node = (node *)malloc(0x10);
new_node->val = v;
if (last_node != (node *)0x0) {
    last_node->nxt = new_node;
}
last_node = new_node;
if (root == (node *)0x0) {
    root = new_node;
}
```

Listing 2.3: Ghidra decompilation of the code presented in Listing 2.1. The decompilation has been modified by renaming variabled and changing data types, based on educated guesses.

2.4 Dynamic Analysis

As described by T. Ball in his 1999 paper [5], *"dynamic analysis is the analysis of a running program"*. This type of analysis is desirable in different situations where static analysis could not extract sufficient information, or when acquiring extra information depends the program to be running.

Dynamic analysis is an umbrella term which covers many powerful techniques used for program analysis. A taxonomy of these techniques has been presented in a comprehensive survey by Ori et al. in 2019 [37]. We will briefly cover a selection of these.

2.4.1 Debugging

Debugging is a very well known technique, especially popular among developers who use it mainly to identify bugs or errors in their code. However, it is also a very effective and reliable form of analysing unknown programs (e.g. malware). Also called *single stepping*, it involves using a tool called a *debugger*, in order to run the program one instruction at a time. After each instruction, the analyst can inspect the state of the registers, the memory and what instructions follow. This process can also help in determining any relevant changes in the operating system itself, caused or related to the debugged program.

Debuggers use the CPU's trap flag in order to trigger an interrupt after each instruction, or certain desired instructions. The interrupt causes a context switch from the execution of the debugged program to the debugger. To continue the execution, the trap flag is set again and the context is switched back to the program. The high number of context switches means that debugging is a very resource intensive analysis technique. It is also very easy to detect by the running malware, which can check the state of the trap flag and hide its behaviour in case it is being debugged [37].

2.4.2 Function Call Analysis

Any type of meaningful action that a program can make, will ultimately rely on System Calls (syscalls) [9]. It can be the case that these syscalls are performed through function calls from an external library, such as the standard `libc` library, or from an internally defined function. Analysing function calls, the state of the program before, during and after the function call, as well as the parameters used can provide valuable information about the behaviour of the analysed program.

Techniques for approaching this goal vary. For instance, we could use command line programs such as `strace`, or `ltrace`, which track syscalls and library calls respectively.

We could also use more advanced techniques, such as function hooking. An analyst can extract more information from a function call by *hooking* (i.e. linking) a piece of code to the targeted function. What will happen is that upon the function call, the *hooked* code will also run. The hooked code can simply print debugging messages to inform the analyst that the function has just been called, or access the state of the program at that time and save it for further inspection. [37]

Function calls can also be used very effectively as a side-channel. More specifically, one can monitor the amount of `calls` which have been made since the reference point in order to determine if progress was made (or not) in the execution.

Let's consider Listing 2.4. We're running the crackme [2] through `ltrace` to monitor function calls, with a randomly chosen input string. We notice a length check with `strlen`

[2] A crackme is a program designed to test a reverse engineer's skill [13].

at Line 4, after which the program crashes. By selecting the correct input length of 70 bytes, we can pass the length check at Line 13. This `crackme` is a particularly good example for applying this technique, because it is heavily obfuscated. We cannot effectively use static analysis on this binary, so employing dynamic analysis techniques enables us to make progress and recover the secret [45].

```
 1 >_ python -c "print('a'␣*␣42)" | ltrace ./crackme
 2 memset(0x8625ae8, '\0', 10000)                          = 0x8625ae8
 3 fgets("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"..., 10000, 0xf22e9700) = 0x8625ae8
 4 strlen("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"...)           = 43
 5 puts("WROOONG!"WROOONG!)                                = 9
 6 exit(1 <no return ...>)
 7 +++ exited (status 1) +++
 8
 9 >_ python -c "print('a'␣*␣70)" | ltrace ./crackme
10 memset(0x8625ae8, '\0', 10000)                          = 0x8625ae8
11 fgets("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"..., 10000, 0xedcf4700) = 0x8625ae8
12 strlen("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"...)           = 71
13 strstr("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"..., "zihldazjcn") = nil
14 puts("WROOONG!"WROOONG!)
15 exit(1 <no return ...>)
```

Listing 2.4: ltrace ("a library call tracer") output of an obfuscated crackme. One can observe a length check in the first execution, and different output when an input of the expected length is provided.

2.4.3 Dynamic Taint Analysis

Dynamic Taint Analysis is a technique used to track data flow from sources to sinks. In order to achieve this goal, data considered important is given a label (*a taint*), based on a *taint introduction policy*. Typically, we would taint untrusted user input or data arriving over the network. This *tainted* data is propagated through the system based on execution and how the code interacts with the data at the opcode level. When an operation is performed on tainted data, memory locations used during the respective operation are also tainted, based on a *taint propagation policy*. Some memory areas, or code sections are also marked as *sinks*. When tainted data arrives at a sink, the path it took through the code can be traced back.

In the context of malware analysis, the flow of tainted data is valuable because it gives valuable insights about the ways the malware interacts with the user and the operating system. Taint analysis is also valuable for exploit detection, and was initially used specifically for this goal. By tainting untrusted user input one can detect unusual data flows and detect attempts at exploiting a system. In such cases, a *taint checking policy* might be used to determine further behaviour (e.g. halting execution) [37] [49].

2.5 Mixed Analysis Techniques

2.5.1 Symbolic Execution

SE is a powerful program analysis technique, and one of the core techniques which the idea of this dissertation is based on. As such, we will cover SE in more detail compared to the other analysis approaches.

SE is typically discussed in relation with *Concrete Execution (CE)*. CE is the formal term for what we refer to as normal program execution. That is, executing a program with a concrete input until the end of a single execution path. When every possible external value (user input, response from a system call, return value of a function), or internal value (memory and registers) has a concrete value, we're dealing with CE. Let us consider Listing 2.5. The value of the argument `c` is given by the caller of the function `fizzbuzz`. If we consider a concrete value of 7 for `c`, we expect the program to print the same value 7 at the standard output, as consequence of executing Line 12. We can test this hypothesis by running the program, passing the value 7 to the function and inspecting the result. This is concrete execution.

```
1  void fizzbuzz(int8_t c) {
2      if (c < 1) {
3          print("too␣small");
4      } else if (c > 15) {
5          print("too␣big");
6      } else if (c % 3 == 0 || c % 5 == 0) {
7          if (c % 3 == 0)
8              print("fizz");
9          if (c % 5 == 0)
10             print("buzz");
11     } else {
12         print(c);
13     }
14 }
```

Listing 2.5: A trivial code example of a function taking a one-byte argument and giving different outputs to `stdout`, based on that argument. The example is meant to showcase SE. A visual representation of symbolically executing this piece of code can be seen in Figure 2.2.

In contrast with CE, with SE we can explore all possible paths of execution (or part of them). Moreover, for each path there will be an associated logical formula, which precisely describes the set of inputs which would lead the execution on that specific path.

Formulas associated with paths are obtained by applying *constraints*, to what are known as symbolic values, which replace certain, or all concrete values in SE.

Initially, the symbolic values are unconstrained, meaning that they can represent any possible input value associated with their designated type. The program is emulated in a controlled environment by a SE engine. The SE engine keeps track of symbols (variables and memory) and adds constraints to these, based on what conditionals are encountered. Execution starts with an initial symbolic state. As the program executes, each conditional branch splits the symbolic state into two and adds new constraints to the state associated with each branch. Figure 2.2 is a visual representation of what symbolically executing Listing 2.5 would look like. One can notice each execution path, the results and the constraints applied to argument c that lead on that respective path.

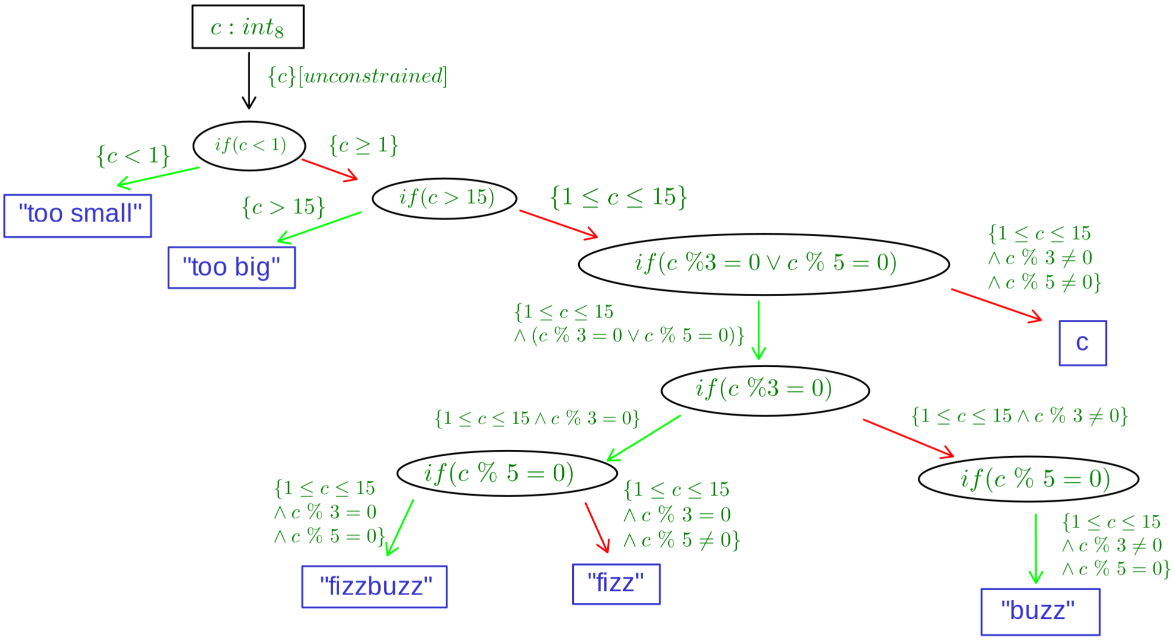

$c : int_8$
$\{c\}[unconstrained]$
$if(c < 1)$
$\{c < 1\}$
$\{c \geq 1\}$
"too small"
$\{c > 15\}$
$if(c > 15)$
$\{1 \leq c \leq 15\}$
"too big"
$if(c \,\%3 = 0 \lor c \,\%\, 5 = 0)$
$\{1 \leq c \leq 15 \land c \,\%\, 3 \neq 0 \land c \,\%\, 5 \neq 0\}$
$\{1 \leq c \leq 15 \land (c \,\%\, 3 = 0 \lor c \,\%\, 5 = 0)\}$
c
$if(c \,\%3 = 0)$
$\{1 \leq c \leq 15 \land c \,\%\, 3 = 0\}$
$\{1 \leq c \leq 15 \land c \,\%\, 3 \neq 0\}$
$if(c \,\%\, 5 = 0)$
$\{1 \leq c \leq 15 \land c \,\%\, 3 = 0 \land c \,\%\, 5 = 0\}$
$\{1 \leq c \leq 15 \land c \,\%\, 3 = 0 \land c \,\%\, 5 \neq 0\}$
$if(c \,\%\, 5 = 0)$
$\{1 \leq c \leq 15 \land c \,\%\, 3 \neq 0 \land c \,\%\, 5 = 0\}$
"fizzbuzz"
"fizz"
"buzz"

Figure 2.2: Visual representation of the path tree resulting from symbolically executing the code in Listing 2.5. Each node in the tree represents a conditional. Each edge has a weight associated with the constraint on argument c, which would result in taking the respective branch. Blue rectangles show the output for the associated execution path.

When analysing a binary with SE, one will often want to know what values will result in a specific path being executed. This is where Satisfiability Modulo Theories (SMT) [51] solving comes into play, also known in this context as *constraint solving*. SMT solvers are powerful tools which aim to prove if a given mathematical formula is satisfiable or not (i.e. if there is a viable configuration of the variables which result in the formula being true). One can feed in the constraints, associated with a path, to an SMT solver (e.g. Z3 [64]) and determine if that path is reachable. If positive, the SMT solver can also provide a model – a valid input which leads execution down the path in question. SMT solvers are a very complex topic and discussing their inner workings is outside the scope of this work [49].

2.5.2 Concolic Execution

One of the bigger downsides of SE is path explosion. Path explosion happens when too many paths coexist at once, which leads to an exponential growth in memory usage. Because of that, by itself, SE typically reaches very shallow depths in the CFG of the program. Often, when working on real-world software, symbolically executing all the possible control flow paths is simply not feasible.

This issue can be addressed through a compromise between coverage (high in pure SE) and depth (shallow in pure SE), in the form of concolic execution, also known as Dynamic Symbolic Execution (DSE). In this context, the term *concolic* is a blend word from the words "**con**crete" and "symb**olic**". The concept is straight forward. We can concretely execute the code, and obtain an execution trace. We can then use SE to mutate the original path in order to obtain new paths and improve the code coverage. In this way, SE becomes a very powerful method to enhance other analysis techniques such as fuzzing [22], taint analysis and program slicing [4].

2.6 Obfuscation Techniques

Obfuscation is an umbrella term for a set of techniques that are very widely employed in the field of software engineering for protecting source code. Obfuscation makes the code harder to understand without affecting its semantics. Obfuscation is common, for instance, when the code contains intellectual property, which the owner(s) want to protected against RE. Not surprisingly, obfuscation is also very common in malware. In this case, obfuscation is used to evade automatic malware detection systems, and to prevent, or, at least, dramatically slow down, the process of RE the malware.

There are a number of well documented obfuscation techniques, which have been observed in the wild time and time again. Obfuscation can be categorised into data-flow obfuscation, control-flow obfuscation and mixed obfuscation [19]. We will shortly cover some obfuscation schemes which are relevant in the context of this work.

Constant unfolding is applied to a constant value, replacing it in the obfuscated code with a complex sequence of operations that compute the original value at runtime. Similarly, *pattern-based* obfuscation consists of replacing co-located instructions with a more complex set of instructions that bear the same semantics. *Dead code insertion* is another common technique which is based on the insertion of sequences of instructions with no semantic effect to the state of the program. Executing those does effectively nothing. Mixed Boolean-Arithmetic (MBA), which is one of the most sophisticated obfuscation techniques, is used on both constants and expressions. MBAs encodes expressions into semantically equivalent ones, using a combination of logical and arithmetic operations, and are notoriously hard to simplify [67] [43]. All techniques mentioned above fall in the data-flow obfuscation category [19].

Reverse engineers rely strongly on assumptions about how compilers translate source code into machine code, and learn to recognize certain patterns: `call` instructions are used only for switching control to a function, indirect jumps are used only in specific and predictable situations, such as `switch` statements, temporally related code is grouped together, etc. Control flow obfuscation aims to invalidate these assumptions. Function *inlining* and *outlining* are used to degenerate the original CFG and turn it into a nonsensical structure. The locality of temporally related code can be affected by reordering instruction blocks and introducing unconditional jumps (in order to preserve semantics). *Junk code*, which is code that is never executed, can be included in order to pollute the CFG. Building on this idea, *opaque predicates* [12], which are boolean expressions that can only evaluate to either true or false, can be used to add branches leading to *junk code*, into the CFG [19].

The most complex obfuscation schemes uses a mix of data-flow and control-flow obfuscation. CFG flattening is a technique that completely alters the control flow of a program. It is typically applied locally, on the body of a function, and works by replacing all control structures with a switch statement, called a *dispatcher*. Virtualisation, or the use of embedded VMs, is one of the most complex and difficult to reverse engineer types of obfuscation. It is conceptually similar with the previously mentioned CFG flattening technique (see Figure 4.1), but differs in the sense that virtualisation is applied to both control flow and the data-flow of the program. Because they come with a significant computational overhead, virtualisation is typically applied only to certain parts of the program. Virtualisation-based obfuscation is the main topic of this work and will be covered in more detail in the following chapter.

2.7 VEX - Intermediate Representation

An IR is a generalised and structured representation of all possible actions that can be performed via CPU instructions. Typically, IRs are intermediaries between hardware-specific or platform-specific source code, and further processing that needs to be done on that source code.

VEX is the IR developed and used for the Valgrind [57] binary instrumentation framework, which specialises in detecting memory corruption issues, profiling, etc. VEX was also adopted for the `angr` [60] binary analysis toolkit, which is one of the core elements of this dissertation. The main reasoning behind using an IR in the context of `angr` was to easily support a multitude of architectures. If the source code to be analysed can be converted (lifted) into the IR, any analysis developed for the IR can be transitively applied to the source code. Not only does VEX supports all major architectures, but it also offers an open-source library to interface with it, is well documented and intended by design to be used for program analysis [21].

An aspect of VEX that makes it particularly suited for program analysis is that it is semantically transparent, in the sense that it does not hide any side effects that instructions might have (e.g. implicit modifications to flags made explicit). It also abstracts away register naming conventions which differ among architectures, and the manner in which memory is accessed. [27]

VEX IR code is grouped into Intermediate Representation (IR) Super Block (IRSB)s, which correspond to extended basic blocks in the target architecture. Each block can have multiple exits and is comprised of IR Statements, which describe changes to the program state, such as changes to memory or registers. State changes are performed by using IR expressions, which can be constants or results of computations (arithmetic computations, but also values read from memory or registers). IR operations are applied to modify IR expressions, which in turn result in other IR expressions. These include arithmetic, logical, or bitwise transformations, etc. Lastly, what ties everything together are the temporary values, which are used as internal registers in VEX. Temporary values are unlimited, numbered starting from `t0` and strongly typed. [27] [33]

To exemplify how typical assembly looks like when lifted into VEX, let us consider the following snippet from a function prologue in Intel x86_64: `push rbp; mov rbp, rsp`. The value of old base pointer is pushed onto the stack and the stack pointer's value is saved as the current base pointer. These two assembly instructions are transformed into VEX IR statements and shown in Listing 2.6.

```
------ IMark(0x525cc4, 1, 0) ------
// load value of rbp as int_64 into t0
t0 = GET:I64(rbp)
// load value of rsp as int_64 into t20
t20 = GET:I64(rsp)
// subtract 8 out of t19 (side effect of push)
t19 = Sub64(t20,0x0000000000000008)
// store value of t19 back into rsp
PUT(rsp) = t19
// store t0 (previous rbp) on the new address on the stack
STle(t19) = t0
------ IMark(0x525cc5, 3, 0) ------
// store the value in t19 (new rsp) into rbp
PUT(rbp) = t19
// advance the instruction pointer
PUT(rip) = 0x0000000000525cc8
```

Listing 2.6: VEX IR corresponding to the `push rbp; mov rbp, rsp` instruction sequence

in Intel x86_64. Each machine instruction is split into muliple, transparent, VEX statements. Each original machine instruction is delimited by an IMark, which specifies the address of the original instruction, and its length in bytes.

In Chapter 4, we show how custom bytecode can be lifted into VEX and analysed through the angr framework.

Chapter 3

State of the Art

In this chapter, we discuss various approaches to recovering code obfuscated through virtualisation that were proposed in academia. We start by discussing what virtualisation-based obfuscation entails. We continue with presenting state of the art approaches to bypassing this obfuscation scheme. We end the chapter with a discussion on previous approaches to the problem and how our proposal differs from those.

3.1 Virtualization-based Obfuscation

Virtualisation-based Obfuscation is one of the strongest protections employed by current obfuscation tools. Analogous to system-level VMs, used to emulate the multitude of hardware components that comprise an entire system, in the context of obfuscation, VMs only emulate a custom-made Instruction Set Architecture (ISA).

Due to its complexity, virtualisation adds a significant runtime overhead. Thus, it is common that only very specific and sensitive functionality of the original program (e.g. a hashing function) is virtualised [30].

3.1.1 Internals

The end goal of virtualisation-based obfuscation is to transpile the targeted segment of code, from its native architecture (e.g. ARM64) into a non-standard custom-made architecture. The translated code is encoded as *bytecode* according to the VM ISA specification, and is embedded in the target binary file. The resulting bytecode can obviously not be executed by the CPU. Thus, a custom interpretor for the bytecode must also be embedded alongside the bytecode into the original binary.

The embedded interpretor is made-up of multiple components. The first component, called the *VM entry*, performs the context switch from normal execution to virtualised execution. In this step, virtual registers (e.g. the virtual Instruction Pointer (IP) or Stack Pointer (SP)) and memory locations used in the VM context are initialised. The next

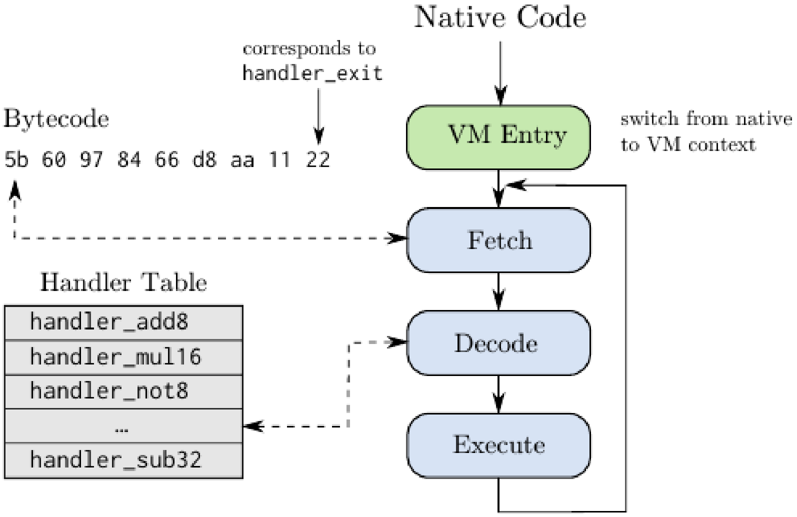


Figure 3.1: A high-level abstraction of the inner workings of an embedded VM interpretor. Execution begins in the VM entry, where context is switched from the native context to the virtualised context. Then a fetch-decode-execute loop follows. Instructions encoded as bytecode are fetched from memory and then decoded. Control is switched to a corresponding function handler, identified based on the decoded information. This loop is commonly known as the dispatcher, which continues execution until all the bytecode is processed. [7]

component is the *dispatcher* which performs a fetch-decode-execute loop, as depicted in Figure 3.1. Each iteration, a new bytecode sequence is fetched from memory and decoded. Afterwards, control is passed to the corresponding function handler, which holds the logic which should execute for the respective bytecode sequence. Control is passed either directly, through an indirect jump, or through a dispatch table stored in memory. The jump destination is determined based on information extracted in the decoding process.

After an instruction finishes execution, control is passed back to the dispatcher and the process repeats with the next instruction. After all instructions encoded in the bytecode have been executed, the dispatcher loop terminates, typically after executing an *exit* instruction. Finally, in the *VM exit* step, the native context is restored, and control is passed back to the native execution. Software protection solutions that use virtualisation-based obfuscation include, but are not limited to: VMProtect [58], Themida [42] and The Tigress C Obfuscator [11].

3.2 Academic Work

There is a decent body of academic work tackling the subject of virtualisation-based obfuscation, as also highlighted in a recent survey by Kochberger et al. in 2021 [30]. Going forward, we will cover what we consider to be the most important pieces of work on this subject.

3.2.1 Semi-Manual Approaches

One of the first academic works on the topic roots back to 2009, when Rolles [46] published a systematic approach to tackling virtualisation-based obfuscation. The authors describe a multi-step strategy to recover the original source code from the protected program. They start with analysing the VM interpretor in order to recover the semantics of the handlers. They use this information in order to design an IR to lift the VM bytecode into. As mentioned by the authors, the analysis step depends on the work of a professional reverse engineer, and must be performed once for each analysed virtualisation scheme. In the next step, they identify the exact location in the code where control is passed from native execution to the VM interpretor – the *VM entry*. In the third step, the authors cover the process of building a disassembler for the bytecode. They use this disassembler to automatically lift the bytecode into the previously chosen IR. Multiple rounds of compiler optimizations can be applied on the resulting IR, in order to simplify it and strip away obfuscation. Finally, they generate native `Intel x86` code from the simplified intermediate code. The result of this process is native code, semantically equivalent with the virtualised bytecode. The strategy suggested in this work is very effective, but requires significant reverse engineering effort.

3.2.2 Trace Simplification

Sharif et al. introduce in 2009 the Rotalumé framework [50], which they claim to be the first work aiming to automatically deobfuscate virtualised malware. In their prototype, they start by executing the chosen samples using QEMU [44]. In this way, they capture a bytecode execution trace. They lift the obtained trace into a chosen IR and apply clustering on memory reads. Finally, they apply behavioural analysis based on the clustered data, in order to identify essential artefacts of the VM, such as the dispatcher loop, changes in the virtual IP, etc. The Rotalumé framework was successfully used in order to automatically detect the bytecode buffers and extract the syntax and semantics of the bytecode. They were also able to recover the CFG of the protected program.

In 2015, Yadegari et al. proposed a generic approach to deobfuscate programs [63]. They explicitly do not make any assumptions about the nature of the obfuscation. The authors claim that their method is effective against a wide array of obfuscation techniques, including virtualisation. They define the obfuscation process as a series of semantic-preserving transformation which make the original code harder to understand. From this, it immediately follows that the opposite process, the deobfuscation, involves applying a series of semantic-preserving transformation that simplify the code and make it easier to understand. In their work, the authors apply bit-level taint analysis in order to identify the relevant code. They further use dependency analysis in order to identify the relationships between pieces of data in the program. In this way, they isolate the code controlling the flow of data from input to output. Finally, they simplify the resulting trace, recover the CFG and apply further graph simplification algorithms in order to further reduce the complexity of the CFG.

Taking a similar approach to [63], SEEAD [52] was proposed by Tang et al. in 2017. SEEAD is also intended to be a generic deobfuscation tool, so the authors make little to no assumptions about the type of obfuscation applied to their targets. This approach aims to address a common shortcoming of previous attempts based on dynamic analysis: the lack of sufficient code coverage. To achieve better code coverage, they employ a carefully designed code exploration technique, which ensures that execution takes different paths in the CFG across multiple execution rounds. The authors use taint analysis and control dependency analysis in order to guide the execution of only the relevant parts of the code that are related to the input data. This increased code coverage achieved by SEEAD has the potential to expose hidden behaviours that previous approaches might have missed. After optimizing the resulting traces, their experiments showed that SEEAD is capable of recovering the CFG from the protected sections.

Salwan et al. introduce at the end of 2018 a deobfuscation tool targeting the Tigress C Obfuscator [48] [11]. They suggest using a mix of taint analysis, symbolic execution, as well as compiler optimizations through the LLVM [53] tool-set. Their approach involves a multi-step process. In the first step, they identify an input source in the program, which they call a seed. Next, they use dynamic taint analysis in order to filter out code that does not interact, either directly or indirectly, with the seed. The sub-trace resulted in the previous step is then used in conjunction with symbolic execution in order to obtain a comprehensive symbolic representation of the trace. In the forth step, the authors apply path coverage analysis, in order to identify a way to guide execution onto the tainted path. Since there exists the possibility of discovering new seeds, the previous steps are repeated until there is no seed left to be processed. The result of the previous steps is a symbolic path tree. In the final step, the result is converted into LLVM IR. The IR is optimised through the LLVM tool-chain, and then compiled into native code. The result of this approach is native, deobfuscated code.

3.2.3 Program Synthesis

Blazytko et al. came up in 2017 with Synthia [7], another generic deobfuscation tool. As indicated by a paper by Banescu et al. [6] published in 2016, in which the authors present ways to circumvent the underlying techniques (e.g. symbolic execution) employed by previous deobfuscation proposals, the authors of Synthia approach the problem from a different angle. They propose eliminating the virtualization-based obfuscation through semantic synthesis of the VM handlers. Similar to previous approaches, they extract execution traces from the obfuscated program, but take a different approach to simplifying them. Their approach uses SMT solvers and Monte Carlo Tree Search (MCTS) for trace simplification. They split the resulting traces into *windows*, and then fuzz each window. The result of the fuzzing step is a set of input-output pairs, which represent the semantics of the trace window. They use the resulted MCTS in order to synthesize semantically-equivalent expressions with the input-output pairs resulted from the fuzzing process. Their work suggests that Synthia can extract semantics from arithmetic VM instruction handlers, and can also simplify MBA expressions.

3.3 Discussion

As suggested by a survey by Kochberger et al. in 2021 [30], out of the numerous projects proposed by the academia, very few of them are publicly available. The authors analysed four publicly available projects and encountered difficulties when running the tools on their custom sample set. Their claim is that the evaluated tools work either on very specific samples, or need various amounts of manual intervention in order to achieve results. They conclude stating that further work is needed in order to achieve more robust deobfuscation solutions.

Until better automatic solutions are developed, reverse engineers in the industry still heavily rely on manual approaches. Our proposal resembles the techniques described by Rolles [46], yet shifts attention to the bytecode. We use a mix of static analysis and symbolic function summarisation in order to extract the function handlers' semantics. Then, in our proposed approach, we lift the custom VM bytecode into the VEX IR (see Section 2.7). We then use angr [60], a very popular tool in the binary analysis community, in order to directly analyse the lifted bytecode. As far as we are concerned, there is no other proposal of using angr for reasoning directly about bytecode in the context of virtualisation-based obfuscation.

Chapter 4

Our approach

4.1 Overview

In this thesis we propose a new approach for reverse engineering programs obfuscated through the use of virtualisation. Our approach resembles in some aspects previous methodologies [46], but introduces some new tools and techniques into the process. We aim to utilise the popular angr framework and its features in order to run automated analyses directly on the embedded bytecode. To get there we propose using a mix of static analysis and function summarisation via symbolic execution (see Section 2.5.1), as also seen in [48] and [32], in order to extract the semantics of the function handlers. We use the extracted information and build architecture-specific plugins for angr. Such a plugin contains the missing information which the framework requires, in order for its tools and features to function as expected.

Among our contributions we highlight `arch-genesis`, a tool which simplifies the process of creating a new architecture-specific plugin for angr. It abstracts away most of the boilerplate code, and allows the user to focus only on lifting the bytecode and on analysing it in angr. We also show how this tool will generate a disassembler for the VM bytecode, given the structure and names of the virtual instructions. Furthermore, we demonstrate how a CFG can be recovered from the bytecode, using primitives exposed through the resulted plugin.

To exemplify the use of our proposed methodology, we use `arch-genesis` in order to solve two crackme-style Capture the Flag (CTF) challenges of varying difficulty. Both challenges propose as RE targets Executable and Linkable Format (ELF) binaries, compiled for the `Intel x86_64` architecture, obfuscated using virtualisation. Although the chosen samples do not accurately resemble real-world malware, these are solid examples for virtualisation-based obfuscation, and serve as good preliminary targets for testing our tool.

In the following sections we present the process of reverse engineering the mentioned binary files and their associated embedded bytecode. We end this chapter with a discussion on the advantages and shortcomings of our approach, based on our empiric experience using it. We end with future directions.

The two analysed binary files were part of the 2023 UIU CTF (the `vmwhere` challenge) [55] and the 2023 Imaginary CTF [26] (the `vmcastle` challenge), in the RE category. The challenges were solved during the competition by the top 8% and top 2% of the teams, respectively.

All the source code and related materials for this work are available on Github [1].

4.1.1 Assumptions

We make some notable assumptions about the targets that our project can be used for in its current iteration. We expect it to work with binaries obfuscated with, but not limited to, the embedded VM obfuscation technique, that present an easy to medium level of complexity. As such, we assume that the VM, and specifically its function handlers can be reverse engineered in a reasonable amount of time with common RE techniques. We further assume that the number of function handlers is reasonable enough to implement, and that each function handler's semantics can be translated into simple primitives.

4.2 Understanding the VM

In order to integrate a new architecture into angr, we need to understand the implementation and the structure of the VM ISA, and what each of the VM handlers does. For this, we rely both on static analysis using Ghidra [24], as well as a symbolic execution (see Section 2.5.1) for simplifying and summarising the semantics of the VM function handlers, using the *Miasm* framework [38].

4.2.1 Static Analysis

We begin by opening the binary file in Ghidra [24], which was previously covered in 2.3. We use it to quickly identify the source of the bytecode, which is the standard input (stdin): the file path of a file containing the VM bytecode is passed to stdin in both cases, after which it is parsed and passed onwards to the bytecode interpretor. We continue the analysis with this interpretor, as it is the most interesting part of the program.

We first consult the CFG view in order to identify the relevant components of the bytecode interpretor. We are looking for a generic and well known structure of a VM, which consists of:

[1] https://github.com/Stefan-Radu/master/tree/master/thesis/proj

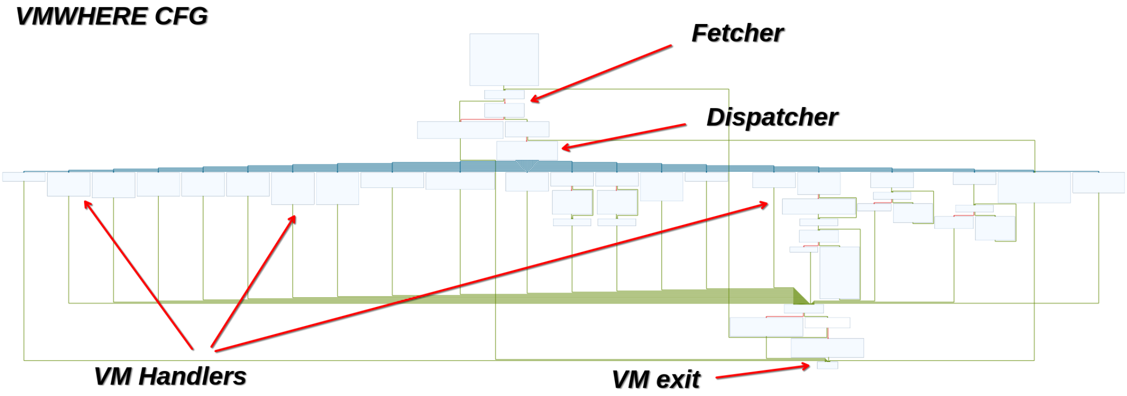


Figure 4.1: CFG of the `vmwhere` VM interpretor. The major components, such as the instruction fetcher, the dispatcher, VM handers, as well as the VM exit are clearly labelled. The image is generated with the help of the Cutter RE tool [18].

1. A VM entry where the context switch from *normal* execution to virtualised execution takes place;

2. A fetch-decode-execute loop, known as a dispatcher, which extracts each encoded operation from the bytecode, processes it, and then passes control to the corresponding function handler. After execution in the handler is finished, control is passed back to the dispatcher, where, either a new iteration begins, or the loop is broken out of;

3. A VM exit, which restores the state and passes control back to native execution;

We identify these components in the high level overview, as we can see in Figure 4.1, in one of the two analysed targets. We can visibly see the individual handlers, as well as some handler functions which are more shallow in complexity (mainly on the left side of the CFG), and some that are visibly more complex (mainly on the right side of the CFG).

```
switch(*IP) {
case 0:
    return 0;
case 1:
                        /* add */
    SP[-2] = SP[-2] + SP[-1];
    SP = SP + -1;
    IP = rip_next;
    break; //...
case 0xb:
                        /* jlz bb */
    if ((char)SP[-1] < 0) {
    rip_next = rip_next + CONCAT11(*rip_next,IP[2]);
    }
    IP = rip_next;
    IP = IP + 2;
    break; //...
case 0xf:
                        /* push top() */
    *SP = SP[-1];
    SP = SP + 1;
    IP = rip_next;
    break;
}
```

Listing 4.1: Decompilation section of the `vmwhere` dispatcher, after variable renaming and retyping. We notice the implementation of the `add`, `jlz` and `push_top` instructions.

The *dispatch* tree is clearly visible in the case of the `vmwhere` challenge, because function handlers are inlined inside a big switch statement, which handles each individual VM instruction. We can identify the respective switch statement in the Ghidra decompiler. Moreover, a high-level CFG of the dispatcher can be inspected in Listing 4.1.

In the cease of the `vmcastle` binary, the handlers are not visible in the CFG at all, because the dispatching mechanism used is different. Each handler routine is accessed via an indirect jump, more precisely a call instruction on the `RDX` register. The `RDX` register indexes into a function dispatch table, based on the current opcode. Listing 4.2 contains the relevant disassembled code for this example.

```
...
0x10281c      LEA       RDX, [RAX*0x8]
0x102824      LEA       RAX, [G_MNEMONICS]
        // Compute offset in the dispatch table
0x10282b      MOV       RDX, qword ptr [RDX + RAX*0x1]
0x10282f      MOVSX     EAX, byte ptr [RBP + -0x11a]
0x102836      MOV       EDI, EAX
        // Call the corresponding handler function
0x102838      CALL      RDX
...
```

Listing 4.2: x86_64 disassembly of the `vmcastle` dispatcher. The function handler corresponding to the current opcode is indirectly called through the register `RDX`.

We are also interested in the internal structures of the VM, namely, the registers used, the stack, the memory addressing scheme, calling convention, etc. The most important registers to look out for are the virtual IP and the virtual SP, which we were able to identify. We also notice the location of the stack and the fact that it *grows* upwards in both cases, a clear distinction from native `x64` architecture. A notable distinction between the `vmwhere` and the `vmcastle` architectures is that the former performs all its operations directly on the stack, while the latter uses four extra registers for its operations: three general purpose registers and a flag register. This is clearly highlighted in Listings 4.3 and 4.4, where the implementation of similar add operations is displayed. The way operations are performed and how data moves inside the VM is important information which we need throughout the rest of the analysis.

```
SP[-2] = SP[-2] + SP[-1];
SP = SP + -1;
IP = rip_next;
```

Listing 4.3: Stack-based implementation of a simple add instruction in the vmwhere architecture.

```
MEM.AC = MEM.R2 + MEM.R1;
```

Listing 4.4: Register-based implementation of a simple add instruction in the vmcastle architecture.

After getting a high-level overview, we continue the analysis with a mix of static analysis and symbolic execution, in order to better understand the details of each of VM handler function.

4.2.2 Automatic Code Summarisation using Miasm

We present here a technique utilising symbolic execution through the Miasm [38] reverse engineering framework, in order to summarise function semantics. We have seen this approach used in multiple places including the Miasm blog in an article about reverse engineering the ZeusVM malware [66], as well as in multiple workshops by Tim Blazytko [62]. There are also academic publications such as [32] which use this exact method, or [48] which employ a similar enough technique. We will apply this technique during our analysis, in order to automatically parse all VM handlers and output their semantics in a simplified and structured way.

In order to get started, we followed the previously mentioned article [66]. Since the article is 8 years old at the point of writing this dissertation, it was not surprising to figure out that the code presented it outdated, and no longer compatible with recent versions of the framework. We made a small contribution by updating the script from the article to python3 and to an up-to-date version of the Miasm framework.

Miasm and angr, at their core, are emulators, and will function just like a normal emulator when computing strictly on concrete (non-symbolic) data. The beauty happens when we introduce and allow symbolic values in the state. In that case, by executing several blocks of code, we register the state change across instructions, through the symbolic values. As such, from the high-level static analysis performed previously, we collect the addresses of each individual handler, and execute them symbolically. For scope limiting purposes, we also identify the address where the control switches back to the dispatch loop: 001018fd | eb 7b | JMP | DISPATCH_LOOP.

Let's look into an example featuring a more interesting handler from the `vmwhere` VM (handler number 17), and execute it symbolically. We can see a partial result in Listing 4.5.

```
IRDst = 0x18CF
cf = ((((@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0x7) ^ (@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF)) & ((@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF) ^ 0xFFFFFFFFFFFFFFF7)) ^ (@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0x7) ^ (@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF) ^ 0x8)[63:64]
zf = @64[RBP + 0xFFFFFFFFFFFFFFF0] == 0xFFFFFFFFFFFFFFF9
RDX = {{@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF] >> 0x7 0 8, 0x0 8 32} & 0x1 0 32, 0x0 32 64}
RIP = 0x18CF
...
@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0x4] = (@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF] >> 0x5) & 0x1
@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0x5] = (@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF] >> 0x6) & 0x1
@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0x6] = (@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF] >> 0x7) & 0x1
@8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF] = @8[@64[RBP + 0xFFFFFFFFFFFFFFF0] + 0xFFFFFFFFFFFFFFFF] & 0x1
```

Listing 4.5: Partial result of symbolically executing a function handler in Miasm. We notice the state change in core registers such as `RDX`, flag changes, as well as changes in memory. The output is clearly convoluted and not easy to read.

This output is complete, but not very useful in this form for two reasons. Firstly, the output is polluted with changes in flags and other side effects of the instruction execution that we are not particularly interested in. Secondly, the data that we are interested in is presented as memory offsets from `RBP`, which are particularly difficult to read. We tackle both points by ignoring the changes to the memory that we are not interested in and by substituting memory addresses that we are interested in with explicit labels. The result of this transformation can be seen in Listing 4.6.

```
*************** | Mnemonic 17 | addr: 0x17DE | ***************
VM_SP = VM_SP + 0x7
VM_STACK_TOP = VM_STACK_TOP & 0x1
@8[VM_SP] = (VM_STACK_TOP >> 0x1) & 0x1
@8[VM_SP + 0x1] = (VM_STACK_TOP >> 0x2) & 0x1
@8[VM_SP + 0x2] = (VM_STACK_TOP >> 0x3) & 0x1
@8[VM_SP + 0x3] = (VM_STACK_TOP >> 0x4) & 0x1
@8[VM_SP + 0x4] = (VM_STACK_TOP >> 0x5) & 0x1
@8[VM_SP + 0x5] = (VM_STACK_TOP >> 0x6) & 0x1
```

```
@8[VM_SP + 0x6] = (VM_STACK_TOP >> 0x7) & 0x1
```

Listing 4.6: A cleaned-up result of symbolically executing the same function handler as in Listing 4.5. We only chose to display the change in relevant registers and memory locations. Additionally, we introduced labels for better clarity.

After cleaning up the output and applying more intuitive labelling, the semantics of the handler are very easy to understand. It pop a 1-byte value from the top of the stack, and pushes each of its bits back onto the stack, as individual entries. For example, if we had the value $0x1e$ on top of the stack, we would end up, instead, with the following values on top of the stack $0, 1, 1, 1, 1, 0, 0, 0$. We confirm this conclusion through static analysis in Ghidra, where we observe a loop which pushes each individual bit from the respective byte to the stack.

By studying both the decompiled code and the output from the Miasm automated analysis, we were able to recover the semantics of all the handlers.

It is important to note that the true power of this analysis technique is not fully highlighted in this scenario with the two CTF challenges, as the samples lack more complex obfuscation. The blog post from Miasm about ZeusVM [66] is a better example, as there are considerably more function handlers to analyse, with some of them being very similar. This technique can be further used to compare symbolic states, in order to identify what the exact differences between certain handlers are, through a simple difference operation. When dealing with heavier obfuscation, Miasm's symbolic execution engine can deal with common data-flow obfuscation such as *constant unfolding*, *dead code*, *pattern based-obfuscation*, etc. by simplifying the mathematical expressions that arise. Even more, in the case of MBA, which is one of the most difficult to deal with obfuscation schemes, we can use msynth [39], a framework built on top of Miasm which can simplify MBA expressions, based on one of the most generic and powerful attacks on MBA, called QSynth [28].

4.3 Summary of Analysis

Manual static analysis in Ghidra, as well as symbolic analysis using Miasm, which were previously discussed, are the main techniques which we used in order to reverse engineer the custom ISAs of the embedded VMs. We were able to identify the main data types used, the registers, as well as reverse engineer each of the individual instruction implementations. We summarise below the types of instructions we encountered:

1. arithmetic and logic operations: We identifies a number of function handlers which perform simple arithmetic operations, resembling instructions such as `add`, `sub`, `xor` , `shl`, `shr`, `mul`, `div`, from well known architectures. The implementations in the

two targeted VMs are not the same. In the `vmwhere` binary, the operations are computed using values from the top of the stack, whereas in the case of `vmcastle`, the computation results are achieved through the use of registers;

2. stack operations: Both ISAs have similar implementations of the `push` and `pop` instructions. We could also identify variations where the value of a register is pushed onto the stack, a value is popped into a register (or not), the popped values are also printed to `stdout`, etc.;

3. conditionals and jumps: We identified function handlers which resemble in behaviour instructions such as `cmp`, `jlz`, `jz`, `jmp`. In the case of `vmwhere`, all conditional checks are made on the value on top of the stack, and the jumps are all *direct*, relying on immediate values as offsets from the current position. In the case of `vmcastle` the implementations are slightly more complex. The `cmp` operation updates the flag register `ac`. Subsequently, a conditional jump will perform an indirect jump based on the `ac` flag, taking the value stored in one of the three general purpose registers (`r1`, `r2`, `r3`) as an offset;

4. System Call (syscall): In both cases we identified handlers that perform reads and writes. Both operations result in a context switch to the kernel as a result of syscalls in the host OS. Their implementation is straight forward. In the case of `vmwhere` the value is read to, or written from, the stack, whilst in the `vmcastle` case, data is read to or written from registers. Since we are dealing with syscalls, in the angr plugin implementation that follows, special measures will have to be taken for correctly implementing these instructions;

5. `nop` and exits: In both cases we identified `exit` instructions that simply signal to the embedded interpreter that the execution of the bytecode should cease and context should be switched back to the native execution. In the case of `vmcastle` we also encountered 101 entries in the dispatch table pointing to the same instruction handler. The implementation of this instruction does nothing, and constitutes the equivalent of a `nop` from `x64`;

6. adhoc and complex instructions: In both cases, but especially in the case of `vmwhere`, we identified function handlers that do not resemble any well known instructions from common architectures, but rather a combination of multiple instructions, leading to a more complex set of transformations that are applied to the state of the program. In the case of `vmcastle` there are two conditional arithmetic operations applied to the top of the stack: a `shl` and an `add` operation that will be executed (or not) having 1 as the argument, based on an immediate value. In the case of `vmwhere`, the handlers which stand out are handler number 17, also mentioned in

Section 4.2.2, and handler number 18, which performs the corresponding reverse operation. Moreover, handler number 16 reverses the order of the elements on the stack in a given range, specified through an immediate value;

Other relevant information is the size of the stack, which in both cases is of 4096 bytes. An unusual piece of information, is that in the case of `vmcastle`, the stack is cyclic, meaning that after pushing 4096 elements on the stack, the 4097th will override the first element. All the information acquired during this stage is crucial for the proper implementation of the `angr` plugin.

4.4 Building an angr Architecture Plugin

In this section, we will cover `angr` [60], a popular binary analysis framework, which has been growing in popularity in the binary analysis scene. It is built in python and designed to be modular and extensible. Thus, each of its modules can be easily extended. Figure 4.2 clearly depicts angr's components and the way in which those interact.

Our goal is to be able to write a simple angr program (Listing 4.7 shows the equivalent of *hello world* in angr) which loads a binary file containing bytecode written for a custom ISA. We expect to have angr be able to load it, parse it, run analysis tools on it and symbolically execute it. We essentially want angr to treat our binary file just like any other binary file based on a common architecture, such as `x86_64`. We will now go over the components which we extended in our plugins. We took a lot of inspiration from a tutorial on building an angr plugin for Brainf*ck (BF) [8], and other examples showcased by the angr team on their Github page [2].

```python
import angr
p = angr.Project("program") # program for our custom architecture
sm = p.factory.simulation_manager()
sm.step(until=lambda s: s.addr == 0xcafebabe) # symbolic execution
print(f'input:␣{sm.active[0].posix.dumps(0)}') # retrieving the input
```

Listing 4.7: A minimal angr code sample. We load a program into `p`, create a simulation manager, symbolically execute the program until we reach the desired address `0xcafebabe`, and finally print the input which determined this execution path.

4.4.1 Extending the Arch Database

`Arch`, short for `Archinfo`, is an architecture database, where each entry holds all structural details regarding an architecture. In our case, this entailed the list of registers, the bit-width, endian-ness, alignment and name. Listing 4.8 shows how we inherit from the `Arch` class and add relevant information about our particular architecture, in this case, `vmwhere`.

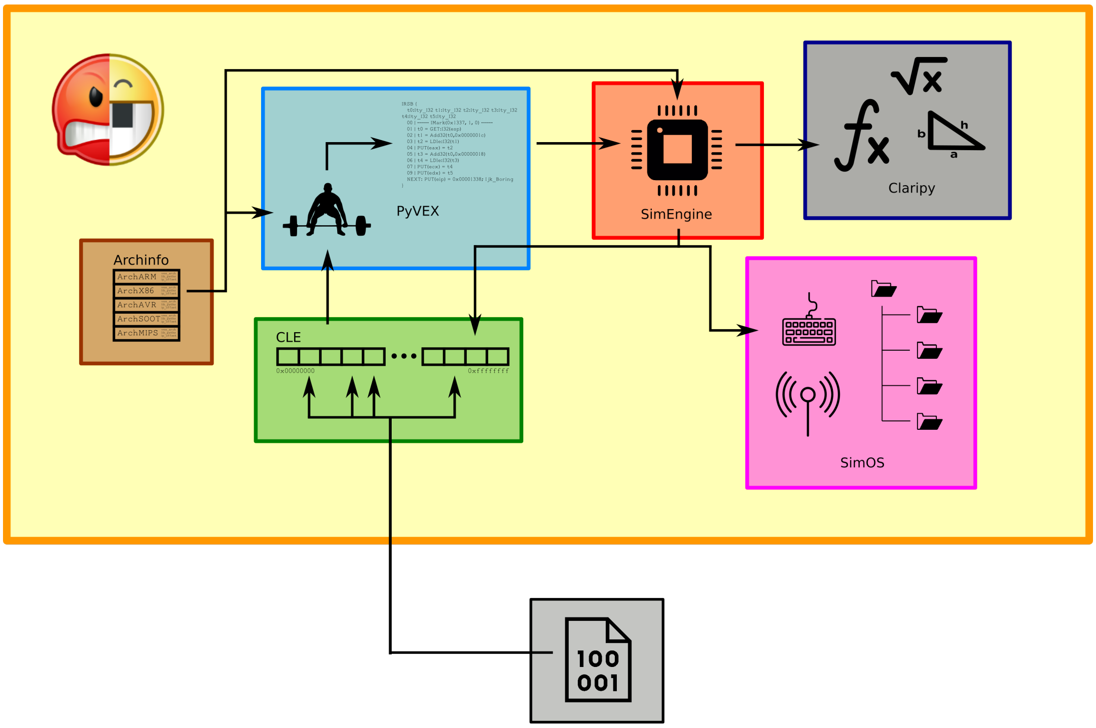


Figure 4.2: Diagram describing the components of the angr framework and how they interact. Binary code is loaded by the CLE module. The loaded information is lifted into VEX IR through the PyVEX module. Symbolic execution is achieved through the SimEngine. Syscalls and external library calls are relayed through the SimOS module. Claripy takes care of modeling the symbolic state and constraint solving during execution, using an SMT solver. Information about the target's architecture is provided to all mentioned components by the Archinfo module. [60]

Line 15 from the same code Listing, containing the `register_arch` function call, is very important, because it *lets angr know* that a new architecture has been added, which should be considered during program loading. One will notice two registers which we did not mention before: `sysnum` and `ip_at_syscall`. These two registers are not part of the architecture, but are relevant in angr's protocol for dealing with syscalls. As a matter of fact, `bp` is also not part of the architecture, but we added it for convenience and easier debugging. Although it is not a lot, this information is a crucial part of our package.

```
1  class ArchVMWHERE(Arch):
2      bits = 64
3      instruction_alignment = 1
4      memory_endness = Endness.LE
5      name = "vmwhere"
6
7      register_list = [
8          Register(name="ip", size=8, vex_offset=0, alias_names=['pc'],),
9          Register(name="bp", size=8, vex_offset=8,),
10         Register(name="sp", size=8, vex_offset=16,),
        Register(name="sysnum", size=8, vex_offset=24,),
        Register(name="ip_at_syscall", size=8, vex_offset=32,),
    ]

register_arch(['vmwhere|VMWHERE'], 64, Endness.LE, ArchVMWHERE)
```

Listing 4.8: Implementation of an Archinfo module extension for the `vmwhere` sample. A list of registers, bit-width, endian-ness and module name are explicitly specified.

4.4.2 Writing a Loader

When we first load a program in `angr`, by default it will try to determine the architecture of the file, and find an instance of an `Arch` class which matches that guess (see Line 2 in Listing 4.7). The component that takes care of this step is the loader: CLE Loads Everything (CLE).

In order to build a custom loader for our architecture we must inherit from the `Blob` class, explicitly state the expected architecture, and add some relevant information on how we want the binary file to be loaded: the number of bytes which should be skipped from the beginning (offset), where the entry point of the program is, and the base address. We must also override the `is_compatible` method, which checks if an input binary file matches the expected architecture. In our implementation, we artificially introduced a 3-bytes header into the bytecode for easier identification, but our experiments show that this it is not necessary.

If we register only one custom architecture during analysis, all the loaders for the well known architectures will fail, and, as a consequence, our loader will be selected. However, in the unlikely scenario in which we would instrument simultaneously two different programs, with different custom architectures, we would have to make sure that we take the necessary measures for each of them to be correctly identified. In the end, we register the new loader with the CLE backend.

4.4.3 Writing a Lifter

angr is a multiplatform binary analysis platform and can be easily extend to also support other architectures. The main reason behind this powerful mode of operation is the following: every type of analysis that is performed through angr is done on the VEX IR (see Section 2.7). Thus, in order to instrument code for a custom architecture, we do not need to implement a new custom analysis engine, but only to provide a translation layer between the bytecode we provide and the VEX IR.

This translation layer is called a lifter, because it *lifts* the bytecode into a higher level representation. angr provides a framework called `gymrat`, which offers a clear and concise interface for building a lifter. gymrat is an abstraction over pyvex – a pythonic interface over the VEX IR objects.

In order to build a lifter for a custom architecture we must define the format for the syntax and the semantics of each instruction. Typically, the syntax is given by the length of the instruction, a fixed *magic number* which identifies it, called the opcode, and the embedded arguments. In order to express the semantics of the instructions, we have to describe the state change that the execution of the instruction would produce during execution, by utilising the primitives exposed through the pyvex framework.

These primitives consist of operations which enable using immediate values in the form of constants, reading/writing data from/to pointers, reading/writing data from/to memory, and jumping. They are sufficient for expressing the vast majority of transformations, but are sometimes lacking, as we will discuss in the later section on shortcomings 4.7.2.

```
1  class Instruction_JZ(Instruction):
2      bin_format = '00001100xxxxxxxxxxxxxxxx'
3      name = 'jz'
4
5      def compute_result(self, *args):
6          jump_offset = int(self.data['x'], 2)
7          dst = self.constant(self.addr + self.bitwidth // 8 + jump_offset,
       Type.int_16).signed
8
9          sp = self.get(SP_REG, PTR_TYPE)
10         top = self.load(sp - 1, STACK_ENTRY_TYPE).signed
11         zero = self.constant(0, STACK_ENTRY_TYPE)
12
13         self.jump(top == zero, dst)
```

Listing 4.9: Implementation of a conditional jump instruction, as part of the lifter module.

To give an example, let us consider an implementation of a jz (jump when zero) instruction as seen in Listing 4.9. We define a class inheriting the base Instruction class exposed by pyvex. We define the syntax as a binary string in bin_format. The first eight bits represent the opcode (number 12), and the 16 that follow represent an immediate value of two bytes. We mark it with 16 x characters, in order to represent the sequence of bits as a wildcard, and also to more easily retrieve the data from the bit-sequence at lift time.

The logic of the instruction is given by overriding the computer_result method. We start by retrieving the jump offset from the immediate value x, we compute the destination address on Line 7, load the value in the stack pointer on Line 9, load the value on top of the stack on Line 10, and finally perform a conditional jump to the computed address, when the value on top of the stack is equal with the constant value zero, on Line 13.

Similarly, we provide an implementation for the rest of the instructions. To finalize the lifter, we define a class which inherits from the base class GymratLifter, and give it a list containing all possible instructions that might be encountered when analysing a binary file for the target custom ISA. We must provide the instructions in the exact order that they should be parsed, knowing that more generic instruction should be left at the end. We then register the new lifter with angr.

4.4.4 Writing a SimOS

The last component which we must extend is SimOS, an abstraction layer for the OS. An instance of a SimOS class is created based on the architecture identified by the loader. It exposes simplified symbolic abstractions of OS-specific entities, such as files or network components, syscalls, or common standard library functions in the form of SymProcedures [2]. The SimOS component exists to eliminate the very complex task of interpreting complex code that is not directly related to the main target of the analysis. As also stated by the angr team, "symbolically executing libc itself is, to say the least, insanely painful" [2].

In our case, we wrote a minimal SimOS implementation in order to deal with the read and write handlers. We implemented two SymProcedures, one for each of the syscalls. An example implementation of the write syscall can be seen in Listing 4.10. We created a class, which inherits from SimProcedure, and overridden the run method. The parameter sp is used as a result of the calling convention which we also had to define as part of the same extension.

```python
class WriteByte(SimProcedure):
    def run(self, sp):
        # Write a byte to stdout
        self.state.posix.fd[1].write(sp, 1)
```

Listing 4.10: A short example of SimProcedure implementing a write instruction.

4.5 Plugin Generation - arch-genesis

After writing the necessary extensions, we grouped them in a python package. By importing the newly created package (e.g from vmwhere import *), the example provided in Listing 4.7 will function accordingly. Even more, all of angr's analyses will function accordingly, because all of these operate on VEX, which any bytecode samples wrote for the respective ISA can be lifted into. Our work is, however, not yet complete. Despite the high level abstractions that angr exposes, writing such a custom architecture plugin still requires a considerable amount of effort to be spent on implementation details that are not related to the analysis of our target bytecode.

To mitigate this, we propose `arch-genesis`, a tool which generates all the necessary boilerplate code necessary for a plugin, so that the analyst can focus only on writing code for the relevant parts of the engine: the logic for the instructions in the lifter, and the necessary SymProcedures. The generator is built in python, using the jinja2 [29] template engine. All information regarding the VM that is relevant for generating the plugin (e.g. name, registers, endian-ness, list of instructions, etc.) is provided via a carefully and intuitively structured config file. We chose `TOML` [54] as the format of choice for the config files, because of its ease of writing compared to `json` and its lack of ambiguity compared to `YAML`. Listing 4.11 contains a section of the config file we used in order to generate the `vmcastle` plugin.

```
...
[lifter.opcodes.stack_top_itshl]
bin_format = [ 115, ['r', 8] ]

[lifter.opcodes.stack_top_itadd]
bin_format = [ 116, ['r', 8] ]

[loader]
offset = 0
entry_point = 0
base_addr = 0x000000
header = ''

[simos]
syscall_args = [ 'reg_no' ]
return_addr = [ 'ip_at_syscall', 8 ]
syscall_addr_alignment = 8

[[simos.syscalls]]
syscall_no = 0
name = 'ReadByte'
...
```

Listing 4.11: Section of the a config file used as input for the `arch-genesis` tool. The section contains a partial list of instructions (opcodes), loader details, as well as information essential for the syscall calling convention.

4.5.1 Disassembler

One of the most important aspects of binary analysis is being able to read code in order to understand what it does, as we have described for instance in Section 4.2.1. Using a disassembler is still essential, even when attempting automatic analysis with a tool such as angr, in order to determine and analyse relevant BB and to determine strategies of guiding symbolic execution. As such, at the very least, we would need a disassembler for the bytecode that we are analysing. When analysing *classic* programs in angr, we can inspect the disassembly of the machine code through the well integrated capstone disassembly framework [10]. We could theoretically write a capstone module for our custom architecture, but to achieve such a thing would require an unreasonable amount of effort, which is outside the scope of this work.

```
0x17c4:  push_imm b'?'
0x17c6:  pop_reg ac
0x17c8:  print_reg ac
0x17ca:  push_imm b'\xff'
0x17cc:  pop_reg r3
0x17ce:  push_imm b'\n'
0x17d0:  pop_reg r2
0x17d2:  read_reg r1
0x17d4:  push_reg r1 | push the value in the register on the stack
0x17d6:  cmp  | compare R1 and R2; set AC
0x17d8:  push_imm b'\xf4'
0x17da:  pop_reg r1
0x17dc:  push_imm b'\x00'
0x17de:  pop_reg r2
0x17e0:  push_imm b'\xf4'
0x17e2:  pop_reg r3
0x17e4:  jmp_cond  | jump with reg offset, based on AC | AC < 0: +R1 | AC == 0: +R2 | AC > 0: +R3
0x17e6:  pop_reg r2
0x17e8:  pop_reg r2
0x17ea:  push_imm b'\t'
0x17ec:  pop_reg r1
0x17ee:  add  | ac = r1 + r2
```

Listing 4.12: Disassembly from a section of bytecode extracted from the `vmcastle` sample. The disassembler itself is generated with the `arch-genesis` tool.

Luckily, there is enough information available to the lifer in order to also perform disassembly of the bytecode. In fact, the angr team took care of this: there is *hidden* functionality for disassembly, which we were able to extend and customize. Thus, the `arch-genesis` program will generate a disassembly executable, which we can run with any file containing architecture-specific bytecode as input, and will output its corresponding disassembly. An example disassembly sample can be inspected in Listing 4.12. As we can see on Line 9, the disassembly can be enriched with descriptions of the instruction behaviour, or by changing the formatting of the respective output. All of these enrichments can easily be customized from the previously mentioned `.toml` configuration file.

4.6 Further Analysis

While working with the disassembler and inspecting the implementation of the `gymrat` lifter, we had the idea of also building a CFG generator. This might seem strange, knowing that angr has CFG building capabilities. In fact, these analyses work, but our goal was to create a visualisation in the `.dot` format and transforming angr's output into dot is not a trivial task. Even more, we were interested in having the disassembly also included in the graph, which would have added another layer of complexity, since the system we were working on for acquiring the disassembly is not fully integrated with the main analyses. Considering the fact that the `vmwhere` ISA is only capable of performing direct jumps, we wrote a custom CFG building algorithm, using common graph theory techniques. The result can be seen in Figure 4.3.

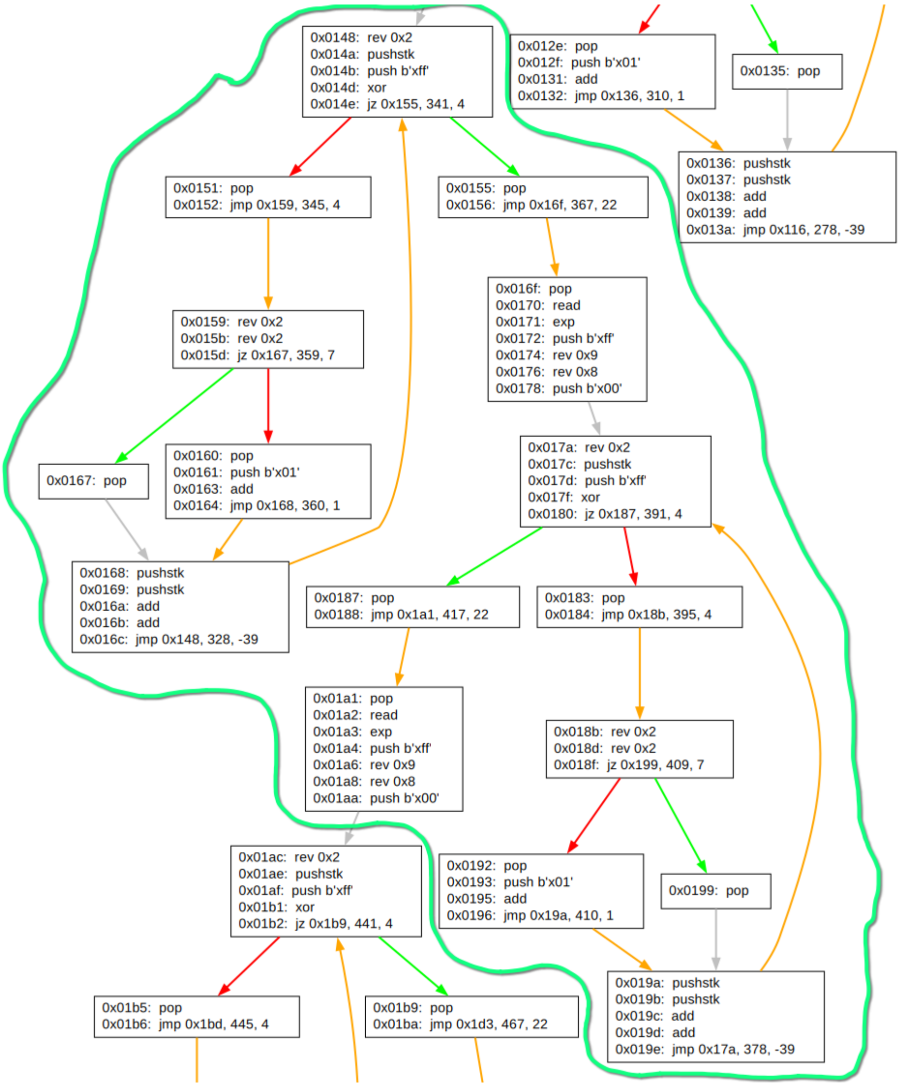
The section circled in green represents a sequence of operations on stack data, which repeats multiple times in the sample bytecode. In fact, this corresponds to a form of obfuscation, commonly known as *function inlining*, which we mentioned in Section 2.6. Visualising the CFG clearly helps in identifying this kind of obfuscation, as well as in better understanding the flow of information and how the state of the program changes during execution. Building a CFG is, however, generally not a trivial task and depends directly on the difficulty of computing jump targets. Because of indirect jumps, we weren't able to apply the same strategy for the `vmcastle` binary.

4.6.1 Solving the Challenges

VMWHERE

Listing 4.14 holds the corresponding grammar of the `vmwhere` bytecode, in Backus-Naur Form (BNF), with comments on semantics.

Static analysis of the bytecode disassembly, in conjunction with the CFG, enabled us to identify the previously mentioned inlined function, which takes a byte of data from `stdin` and transforms it through a series of operations. We are not particularly interested in what the exact operations are, because we can symbolically execute them in angr. We are rather interested in how the result of the computation is used. The result of each round of computations is stored on the VM's stack. Listing 4.13 shows another sequence of operations that repeatedly takes place at the end of the program. Being on the shorter side, it is not so difficult to understand that it takes exactly one value from the top of



```
0x0148:  rev 0x2
0x014a:  pushstk
0x014b:  push b'xff'
0x014d:  xor
0x014e:  jz 0x155, 341, 4
```

```
0x012e:  pop
0x012f:  push b'x01'
0x0131:  add
0x0132:  jmp 0x136, 310, 1
```

```
0x0135:  pop
```

```
0x0136:  pushstk
0x0137:  pushstk
0x0138:  add
0x0139:  add
0x013a:  jmp 0x116, 278, -39
```

```
0x0151:  pop
0x0152:  jmp 0x159, 345, 4
```

```
0x0155:  pop
0x0156:  jmp 0x16f, 367, 22
```

```
0x016f:  pop
0x0170:  read
0x0171:  exp
0x0172:  push b'xff'
0x0174:  rev 0x9
0x0176:  rev 0x8
0x0178:  push b'x00'
```

```
0x0159:  rev 0x2
0x015b:  rev 0x2
0x015d:  jz 0x167, 359, 7
```

```
0x0167:  pop
```

```
0x0160:  pop
0x0161:  push b'x01'
0x0163:  add
0x0164:  jmp 0x168, 360, 1
```

```
0x017a:  rev 0x2
0x017c:  pushstk
0x017d:  push b'xff'
0x017f:  xor
0x0180:  jz 0x187, 391, 4
```

```
0x0168:  pushstk
0x0169:  pushstk
0x016a:  add
0x016b:  add
0x016c:  jmp 0x148, 328, -39
```

```
0x0187:  pop
0x0188:  jmp 0x1a1, 417, 22
```

```
0x0183:  pop
0x0184:  jmp 0x18b, 395, 4
```

```
0x01a1:  pop
0x01a2:  read
0x01a3:  exp
0x01a4:  push b'xff'
0x01a6:  rev 0x9
0x01a8:  rev 0x8
0x01aa:  push b'x00'
```

```
0x018b:  rev 0x2
0x018d:  rev 0x2
0x018f:  jz 0x199, 409, 7
```

```
0x01ac:  rev 0x2
0x01ae:  pushstk
0x01af:  push b'xff'
0x01b1:  xor
0x01b2:  jz 0x1b9, 441, 4
```

```
0x0192:  pop
0x0193:  push b'x01'
0x0195:  add
0x0196:  jmp 0x19a, 410, 1
```

```
0x0199:  pop
```

```
0x019a:  pushstk
0x019b:  pushstk
0x019c:  add
0x019d:  add
0x019e:  jmp 0x17a, 378, -39
```

```
0x01b5:  pop
0x01b6:  jmp 0x1bd, 445, 4
```

```
0x01b9:  pop
0x01ba:  jmp 0x1d3, 467, 22
```

Figure 4.3: A portion of the recovered CFG of the `vmwhere` sample. We notice circled in green an inlined function which applies a number of transformation to an input value read from `stdin`.

the stack, performs a *xor* logical operation on it with the byte `\xZZ`, and then performs an *or* logical operation on the result, with the value at the bottom of the stack, which is initially 0.

```
0x0972:  push b'\xZZ'
0x0974:  xor
0x0975:  rev (sizeof(stack) - 1)
0x0977:  rev (sizeof(stack))
0x0979:  or
0x097a:  rev (sizeof(stack) - 1)
0x097c:  rev (sizeof(stack))
```

Listing 4.13: Disassembly of `vmwhere` bytecode, which performs a bitwise-xor operation on the top two values on top of the stack, and then a bitwise-or operation on the previous

result, and the value at the bottom of the stack.

We are dealing with a *crackme* type of program. What is does is that it accepts an input, performs a series of checks on it, and returns a message saying if the input string was accepted or not. Our goal is to find a correct input that bypasses the checker. In this case, the final check that we are interested in, can be found at the very end of the bytecode: `0x0b9a:  jz 0xba0, 2976, 3`. After performing all the rounds which resemble Listing 4.13, the only value left on the stack, which is the bottom of the stack, is checked to be equal to zero. The response is considered correct if that check passes. We can extrapolate, and deduce that each result of the computation we observed in the CFG 4.3 is checked for equality against the immediate value from Line 1 in the previously mentioned code Listing. By symbolically executing the inlined function, giving it as input a symbolic value, and finally constraining the output to be equal to the immediate value `\xZZ` from Line 1, we will obtain the corresponding input byte that produces the respective output. We automated this process in angr and repeated it for each of the identified constants in the bytecode in order to recover the full secret.

```
<program> ::= <instruction> | <instruction> <program>

<instruction> ::= <exit> | <add> | <sub> | <and> | <or> | <xor>
                | <shl> | <shr> | <read> | <write> | <push>
                | <jlz> | <jz> | <jmp> | <pop> | <pushstk>
                | <rev> | <exp> | <sqz>

<exit>     ::= "00000000" ; exit the program
<add>      ::= "00000001" ; add the two values on top of the stack
<sub>      ::= "00000010" ; sub the two values on top of the stack
<and>      ::= "00000011" ; and the two values on top of the stack
<or>       ::= "00000100" ; or  the two values on top of the stack
<xor>      ::= "00000101" : xor the two values on top of the stack
<shl>      ::= "00000110" ; shl the value on top of the stack
<shr>      ::= "00000111" ; shr the value on top of the stack
<read>     ::= "00001000" ; read from stdin a byte on top of the stack
<write>    ::= "00001001" ; write to stdout the value on top of the stack
<push>     ::= "00001010" <bit_8> ; push one immediate byte on the stack
<jlz>      ::= "00001011" <bit_16> ; conditional jump to 16-bit immediate
                                     address if the value on top of the
                                     stack is lower than 0
<jz>       ::= "00001100" <bit_16> ; conditional jump to 16-bit immediate
                                     if the value on top of the stack is
                                     equal to 0
<jmp>      ::= "00001101" <bit_16> ; unconditional jump to 16-bit immediate
                                     address
<pop>      ::= "00001110" ; pop the value on top of the stack
```

```
<pushstk>  ::= "00001111" ; push the value on top of the stack on the stack
<rev>      ::= "00010000" <bit_8> ; reverse a range given by an 8-bit
                                    immediate value from the stack
<exp>      ::= "00010001" ; expand the value on top of the stack to 8 bits
                            and push each individually on the stack
<sqz>      ::= "00010010" ; pop the top eight values on top of the stack
                            and compress them to an 8-bit value and push
                            it on the stack

<bit_8>   ::= <bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit>
<bit_16>  ::= <bit_8> <bit_8>
<bit>      ::= "0" | "1"
```

Listing 4.14: BNF Grammar of the vmwhere bytecode syntax.

VMCASTLE

Listing 4.15 holds the corresponding grammar of the `vmcastle` bytecode, in BNF, with comments on semantics.

The second challenge is also a *crackme*. It is, however, a lot more difficult to statically analyse. The bytecode itself is obfuscated with techniques such as *function inlining*, *dead code inserting*, *constant unfolding*, etc. We can bypass these obfuscation techniques, by executing the code symbolically with angr, using our plugin for `vmcastle`. We only needed a small, but crucial, amount of information before automating the whole cracking process:

1. In this case, the whole secret is read at once. We found out that it is stored on the stack, and the exact address in the bytecode where input reading ends and the checking process begins.

2. We determined a range of addresses that the code would reach if the input was deemed incorrect.

3. We determined the first address that would be reached if the input was deemed correct.

With the above information, we created a symbolic string, inserted it on the stack, and started executing at the address determined in (1). We then start a guided symbolic execution of the code: we specifically avoid the range of addresses from (2) and try to reach the address from (3). In the end we end up with one state found and many more avoided. At this point we focus our attention on the found state. Its constraint solver has built an expression, which guides the execution to follow the exact path which leads to the found state. By finding a model which satisfies the constraints, using the integrated Z3 [64] SMT solver, we essentially crack the checker. The full source code of the solution can be found in Appendix 5.

```
<program> ::= <instruction> | <instruction> <program>

<instruction> ::= <exit> | <pop_reg> | <push_reg> | <push_imm>
                | <add> | <sub> | <mul> | <div> | <mod>
                | <jmp_reg> | <jmp_cond> | <cmp> | <print_reg>
                | <read_reg> | <stack_top_itshl> | <stack_top_itadd>

<exit>       ::= "01110101" <byte>      ; halt execution
<pop_reg>    ::= "01100110" <reg>       ; pop the stack into the register
<push_reg>   ::= "01100111" <reg>       ; push the value from the register,
                                        ; on top of the stack
<push_imm>   ::= "01101000" <byte>      ; push the 8-bit immediate value
                                        ; top of the stack
<add>        ::= "01101001" <byte>      ; ac = r1 + r2
<sub>        ::= "01101010" <byte>      ; ac = r1 - r2
<mul>        ::= "01101011" <byte>      ; ac = r1 * r2
<div>        ::= "01101100" <byte>      ; ac = r1 / r2
<mod>        ::= "01101101" <byte>      ; ac = r1 % r2
<jmp_reg>    ::= "01101110" <reg>       ; unconditional jump to the
                                        ; address from the register
<jmp_cond>   ::= "01101111" <reg>       ; conditional jump to the
                                        ; address from the register
                                        ; based on the value of ac
<cmp>        ::= "01110000" <byte>      ; compare the values stored
                                        ; in registers r1 and r2 and
                                        ; update ac with the result
<print_reg>  ::= "01110001" <reg>       ; write to stdout the value
                                        ; from the register
<read_reg>   ::= "01110010" <reg>       ; read from stdin a value
                                        ; and store it in the register
<stack_top_itshl> ::= "01110011" <byte> ; shl the value on top of the stack
                                        ; with 1 bit if the immediate
                                        ; value is not zero
<stack_top_itadd> ::= "01110000" <byte> ; add one to the value on top of
                                        ; the stack if the immediate value
                                        ; is not zero

<byte>      ::= <bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit>
<bit>       ::= "0" | "1"
<reg>       ::= <r1> | <r2> | <r3> | <ac>
<r1>        ::= "00000000"
<r2>        ::= "00000001"
<r3>        ::= "00000010"
<ac>        ::= "00000011"
```

Listing 4.15: BNF Grammar of the vmcastle bytecode syntax.

4.7 Discussion

We presented our approach on reverse engineering and analysing binaries obfuscated through virtualisation, by building an angr package which enables running various analyses techniques directly on the bytecode. We will further discuss our framework choice, difficulties that we encountered, advantages and shortcomings of the technique, as well as future directions.

4.7.1 angr vs Miasm

Throughout this chapter we mainly focused on building an angr plugin, its inner workings, and how it can be used in order to solve two CTF challenges. We also discussed automated reverse engineering techniques for recovering the semantics of function handlers from the embedded VM. For that respective process, we used another framework capable of symbolic execution, Miasm. A natural question would be: "why choose one over the other?"

A very big difference between angr and Miasm is the ease of use and out of the box functionality. angr is, generally, easier to use than Miasm. It exposes higher level abstractions and a lot of functionality out of the box. Moreover, it was designed to be modular, and it was very clear that adding an architecture plugin would enable us to use this big suite of already implemented tools. Miasm, on the other hand, has similar functionality at its core, but doesn't have as much abstraction and requires some scaffolding in order to do anything useful with it. Because of that, despite being harder to customize later, we chose angr for the vast array of functionality that it provides by default.

A follow-up question would be: "why did we not use angr instead of Miasm, for function summarisation, in order to maintain consistency?". Well, we tried. We quickly realised that angr simply does not offer this functionality in its default tool-set. There exists a conceptually similar analysis named the *Congruency Checker*, which exposes a method that compare two states. The output was however rather far from our expectations. In cases such as this one, angr is hard to enhance with new functionality and doing so was outside the scope of this work. Because of that, we stuck with using Miasm, just like the author of the referenced blog post [66].

4.7.2 Difficulties and Shortcomings

While developing our solution we encountered several difficulties, which we will discuss in this section. We will cover difficulties with regards to the implementation of a plugin, as well as general shortcomings of our proposed technique.

The `pyvex` library, in particular the `Instruction` class exposes a number of primitives which we used in order to *translate* the VM ISA semantics into the VEX IR. As previously mentioned these are more or less limited to interacting with registers and memory, the use of constant values (i.e. immediate values), and conditional jumps. This set of primitives is perfectly reasonable for *normal* architectures, where an instructions performs a very specific and limited number of changes to the state of the program. This is not the case when we are dealing with custom VMs. The function handlers of each instruction can happen to be complex and have very convoluted logic. Listing 4.6 is an example of such a handler, but the complexity cap can be a lot higher.

We have faced difficulties in implementation, especially with of handlers which deal with conditionals. This happened because there is no method which the `Instruction` class exposes, for reliably describing conditionals. The `ite(condition, a, b)` method suggests that it does exactly what we desire: it takes a conditional expression, and based on its truth value it will return `a`, or `b`. We have had several problems with this method. Firstly, we were not able to chain conditionals into more complex logical expression. Secondly, the result of a call to `ite` does not have the expected type of `VexValue`, but is instead a `rdt`. Listing 4.17 holds the implementation of the function handler number 111 from `vmcastle`. In order to properly implement the conditional jump, we had to first wrap all `ite` return values back into `VexValues`, as can be seen for example at Line 8. Our intuition is that the `ite` method is not supposed to be directly used, considering the fact that it is used as part of the `jump` method implementation, and is has no documentation for it, whereas all other primitives are properly documented.

We also encountered some generic difficulties with regards to angr, and its symbolic execution engine, while working on cracking the sample binaries. More explicitly, we encountered problems when we performed a read operation of a symbolic value. What we expected was that after the read, we would have a single state with the symbolic value in its expected location. However, the result was that the execution engine *split* the state and concretized the symbolic value, based on the constraints applied to it. If, for instance, we were dealing with a symbolic byte, with its value constrained between 50 and 100, instead of a single state, there would be 50 resulting states. Not surprisingly, this quickly leads to the common problem of path explosion. We were not able to figure out a way to avoid this issue, so we ended up manually putting the symbolic values in their corresponding memory locations, as a workaround.

Lastly, our implementation could be limited performance-wise when instrumenting bigger programs. We claim this, because we are not able to use the `unicorn` emulator [56], which angr nicely integrates with, for the same reasons that we are not able to use capstone disassembly: lack of support for our custom ISA.

```c
if (AC == 0) {
    PC = R2 + PC;
}
else if (AC < 0) {
    PC = R1 + PC;
}
else if (0 < AC) {
    PC = R3 + PC;
}
```

Listing 4.16: Cleaned-up decompilation, extracted from Ghidra, of the instruction handler number 111 from the `vmwhere` embedded VM. It performs a conditional jump, based on the value of the `AC` register.

```python
ac = self.get(AC_REG, PTR_TYPE).signed
dst_r1 = self.get(R1_REG, PTR_TYPE).signed
dst_r2 = self.get(R2_REG, PTR_TYPE).signed
dst_r3 = self.get(R3_REG, PTR_TYPE).signed

dst = dst_r1
dst = self.ite(ac == 0, dst_r2, dst)
dst = VexValue(self.irsb_c, dst)

dst = self.ite(ac > 0, dst_r3, dst)
dst = VexValue(self.irsb_c, dst)

dst = dst * self.constant(2, PTR_TYPE)
      + self.addr + 2
self.jump(None, dst)
```

Listing 4.17: Python code which lifts bytecode encoding the conditional jump instruction from Listing 4.16 into VEX IR.

4.7.3 Future Directions

Our proposed approach can be a very effective way of tackling virtualisation-based obfuscation, when dealing with small to medium sized embedded VMs. There are, however, a number of ideas which could take this project further.

Considering the shortcomings mentioned in the previous Section 4.7.2, we would like to contribute directly to the angr project. We would like to propose some changes to the `pyvex` component, in particular to improve the `ite` method from the `Instruction` class. We consider that implementing such changes would contribute significantly to streamlining the process of writing the implementation of a non-trivial VM handler.

angr, in its current state, provides two decompilation routes for a BB: via capstone and via VEX. In our case, since there is obviously no capstone module for a custom ISA, we can only see the VEX assembly code, which is useful in many cases, but is rather verbose and hard to read. angr exposes a shallow, but sill very usable disassembly layer, separate from capstone. It is the exact layer that we tapped into in order to provide the disassembler through our `arch-genesis` tool. We would like to propose some changes to the angr engine, such that, when instrumenting a binary file that is not supported by capstone, a disassembly output is still provided via the previously mentioned, already existing layer.

This change would have a positive impact in the overall process of instrumenting a custom ISA with angr.

A more ambitious goal would be to think about recovering the source code in its original form before obfuscation. As we have previously seen when we discussed about decompilation 2.2, this is not entirely possible, due to the varying levels of information loss during the compilation process and obfuscation process. Instead, we can hope to achieve similar code with the same semantics. The advantages of a system which enables such a transformation are immediate: reverse engineers and analysts would be able to use the large tool-set which they are already used to in order to analyse bytecode written for a custom architecture. This is clearly not an easy task, as we would need a transpiler from the custom ISA to a mainstream architecture (e.g `Intel x64`). Coming up with a transpiler for every embedded VM that we encounter is obviously not reasonable. What we can do is to look again at IRs. LLVM [53] is an IR with a very powerful infrastructure supporting it. What we would need is a way to generate the semantically equivalent LLVM IR for our targeted bytecode. To get there, we could for instance use LLVM bindings in a language such as python. Generating LLVM is known to be difficult, so we could instead implement the custom instruction semantics in VEX as we already did, and build a generic VEX to LLVM transpiler instead. For this goal, we could build on previous work [48].

The part of the process which we could not automate in our project, is also the most difficult and time consuming: implementing the logic of each function handler. We would like to further build onto the ideas from Section 4.2.2, regarding the automatic analysis of the bytecode semantics via symbolic execution. We believe that a comprehensive and cleanly structured symbolic execution trace of each function handler could serve as the basis for an automatic VEX IR generator. We could integrate this generator with angr, and have a fully automated version of our current project. To achieve this, we could build on previous work such as [19], [32], [48].

A last consideration would be that of introducing grammars which describe the syntax of the target bytecode, into the flow of `arch-genesis`. Instead of a large portion of configuration file, which is arguable tedious to write and susceptible to errors, a grammar in BNF (as seen in Listings 4.14 and 4.15) could be provided. As a result, the outline of the lifter can be generated from it, reducing the risks of parsing errors. However, introducing grammars, would not be sufficient in order to replace the whole configuration file, because other details such as the endian-ness or memory address size cannot be determined from it.

Chapter 5

Conclusions

In this dissertation we presented a novel approach for reverse engineering virtualisation-based obfuscation using the angr binary analysis framework. We began with offering some background knowledge and context where we discussed what malware is, as well as what tools and techniques can be leveraged against it. We covered static analysis, dynamic analysis and mixed analysis, the latter being expanded with a focus on symbolic and concolic execution. We then covered obfuscation techniques that make reverse engineering a significantly more difficult process. Following that, we gave an overview on virtualisation-based obfuscation and what it entails. We then proceeded to discuss relevant work in academia on this topic, as well as explained how our approach stands out from previous proposals from the scientific community. We then took a deeper diver into our contribution. We offered an overview of our approach, presented our sample targets, and stated a series of assumptions we make about a generic target, as well as the reasons behind these assumptions. We continued by explaining the incipient process of acquiring information about the VMs embedded in our targets through static analysis, but also through symbolic execution. Based on the acquired information, we described the process of creating an angr plugin for a custom ISA, and introduced `arch-genesis`, our tool which automates a time-consuming part of the plugin building process. We further explained how we automatically generate a disassembler using `arch-genesis`, and how we can use angr's functionality and our plugin in order to recover the CFG from the bytecode. We presented the analysis process on the selected samples, using the previously generated plugins. We ended this work with a discussion on results, shortcomings and future directions.

Listings

2.1 A function which adds an integer value `v` to the end of a linked list. 15

2.2 Ghidra decompilation of the code presented in Listing 2.1. The decompilation is take as-is and has not modified in any way. 15

2.3 Ghidra decompilation of the code presented in Listing 2.1. The decompilation has been modified by renaming variabled and changing data types, based on educated guesses. 15

2.4 ltrace ("a library call tracer") output of an obfuscated crackme. One can observe a length check in the first execution, and different output when an input of the expected length is provided. 17

2.5 A trivial code example of a function taking a one-byte argument and giving different outputs to `stdout`, based on that argument. The example is meant to showcase SE. A visual representation of symbolically executing this piece of code can be seen in Figure 2.2. 18

2.6 VEX IR corresponding to the `push rbp; mov rbp, rsp` instruction sequence in Intel x86_64. Each machine instruction is split into muliple, transparent, VEX statements. Each original machine instruction is delimited by an IMark, which specifies the address of the original instruction, and its length in bytes. 22

4.1 Decompilation section of the `vmwhere` dispatcher, after variable renaming and retyping. We notice the implementation of the `add`, `jlz` and `push_top` instructions. 31

4.2 x86_64 disassembly of the `vmcastle` dispatcher. The function handler corresponding to the current opcode is indirectly called through the register `RDX`. 32

4.3 Stack-based implementation of a simple `add` instruction in the `vmwhere` architecture. 33

4.4 Register-based implementation of a simple `add` instruction in the `vmcastle` architecture. 33

4.5 Partial result of symbolically executing a function handler in Miasm. We notice the state change in core registers such as `RDX`, flag changes, as well as changes in memory. The output is clearly convoluted and not easy to read. 34

4.6 A cleaned-up result of symbolically executing the same function handler as in Listing 4.5. We only chose to display the change in relevant registers and memory locations. Additionally, we introduced labels for better clarity. 34

4.7 A minimal angr code sample. We load a program into `p`, create a simulation manager, symbolically execute the program until we reach the desired address `0xcafebabe`, and finally print the input which determined this execution path. 37

4.8 Implementation of an Archinfo module extension for the `vmwhere` sample. A list of registers, bit-width, endian-ness and module name are explicitly specified. 38

4.9 Implementation of a conditional jump instruction, as part of the lifter module. 40

4.10 A short example of SimProcedure implementing a write instruction. 41

4.11 Section of the a config file used as input for the `arch-genesis` tool. The section contains a partial list of instructions (opcodes), loader details, as well as information essential for the syscall calling convention. 42

4.12 Disassembly from a section of bytecode extracted from the `vmcastle` sample. The disassembler itself is generated with the `arch-genesis` tool. 43

4.13 Disassembly of `vmwhere` bytecode, which performs a bitwise-xor operation on the top two values on top of the stack, and then a bitwise-or operation on the previous result, and the value at the bottom of the stack. 45

4.14 BNF Grammar of the vmwhere bytecode syntax. 46

4.15 BNF Grammar of the vmcastle bytecode syntax. 48

4.16 Cleaned-up decompilation, extracted from Ghidra, of the instruction handler number 111 from the `vmwhere` embedded VM. It performs a conditional jump, based on the value of the `AC` register. 51

4.17 Python code which lifts bytecode encoding the conditional jump instruction from Listing 4.16 into VEX IR. 51

List of Figures

2.1 An infamous screenshot of the ransom pop-up which would show up on a system infected by the WannaCry worm [17]. 12

2.2 Visual representation of the path tree resulting from symbolically executing the code in Listing 2.5. Each node in the tree represents a conditional. Each edge has a weight associated with the constraint on argument *c*, which would result in taking the respective branch. Blue rectangles show the output for the associated execution path. 19

3.1 A high-level abstraction of the inner workings of an embedded VM interpretor. Execution begins in the VM entry, where context is switched from the native context to the virtualised context. Then a fetch-decode-execute loop follows. Instructions encoded as bytecode are fetched from memory and then decoded. Control is switched to a corresponding function handler, identified based on the decoded information. This loop is commonly known as the dispatcher, which continues execution until all the bytecode is processed. [7] 25

4.1 CFG of the `vmwhere` VM interpretor. The major components, such as the instruction fetcher, the dispatcher, VM handers, as well as the VM exit are clearly labelled. The image is generated with the help of the Cutter RE tool [18]. 31

4.2 Diagram describing the components of the angr framework and how they interact. Binary code is loaded by the CLE module. The loaded information is lifted into VEX IR through the PyVEX module. Symbolic execution is achieved through the SimEngine. Syscalls and external library calls are relayed through the SimOS module. Claripy takes care of modeling the symbolic state and constraint solving during execution, using an SMT solver. Information about the target's architecture is provided to all mentioned components by the Archinfo module. [60] 38

4.3 A portion of the recovered CFG of the `vmwhere` sample. We notice circled in green an inlined function which applies a number of transformation to an input value read from `stdin`. 45

Glossary

BB A basic block is a (typically) maximal sequence of instructions that “are guaranteed to execute together”. Any branch incoming to a basic block will end at its entry point. Any branch outgoing from a basic block will start from its exit point [34] . 14

BF Brainf*ck is a very famous esoteric programming language, known for its minimalis syntax which consists of only the following eight instructions: `+ _ < > . ; [ ]` [8] . 37

BNF A style of notation for context-free grammars, used in Computer Science to formally describe the syntax of programming languages or formal languages. [23] . 44

CFG A CFG is a directed graph, modeling potential execution of a computer program. A set of maximal basic blocks (see BB) constitutes the set of vertices. There is an edge in the graph between vertex A and vertex B, if it is possible for the code associated with B to be executed right after the execution of the code associated with vertex A [34] . 7

CTF In the context of security, CTFs are competitions where the participating teams attempt to exploit purpusefully vulnerable applications, in order to find text strings commonly known as “flags” and earn points . 29

ELF ELF is a standardised file format, which originated in the Unix echosystem, used for binary executable files. It is adopted by multiple operating systems including: Linux, Solatris, BSDs, and many others [20] . 29

IR An IR is an abstract, fully encompassing, structure capable of expressing the operations which can be performed on a target (virtual) machine. IRs are commonly used during the compilation process, in order to facilitate further transformations [59] . 10

ISA “An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software. The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.” [35] . 24

Bibliography

[1] *2023 was a big year for cybercrime – here's how we prepare for the future.* [Online; accessed 10. Jun. 2024]. 2024. URL: https://www.weforum.org/agenda/2024/01/cybersecurity-cybercrime-system-safety.

[2] *angr-platforms/angr_platforms/msp430/instrs_msp430.py at master · angr/angr-platforms.* [Online; accessed 12. Jun. 2024]. 2024. URL: https://github.com/angr/angr-platforms/blob/master/tutorial/1_basics.md.

[3] *Backdoor in XZ Utils That Almost Happened - Schneier on Security.* [Online; accessed 10. Jun. 2024]. 2024. URL: https://www.schneier.com/blog/archives/2024/04/backdoor-in-xz-utils-that-almost-happened.html.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques." In: *ACM Comput. Surv.* 51.3 (2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[5] Thoms Ball. "The concept of dynamic analysis." In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ESEC/FSE-7. Toulouse, France: Springer-Verlag, 1999, pp. 216–234. ISBN: 3540665382.

[6] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. "Code obfuscation against symbolic execution attacks." In: *Proceedings of the 32nd Annual Conference on Computer Security Applications.* 2016, pp. 189–200.

[7] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. "Syntia: Synthesizing the semantics of obfuscated code." In: *26th USENIX Security Symposium (USENIX Security 17).* 2017, pp. 643–659.

[8] *brainfuck - Esolang.* [Online; accessed 18. Jun. 2024]. 2024. URL: https://esolangs.org/wiki/Brainfuck.

[9] Stephen Brennan. *Tutorial - Write a System Call - Stephen Brennan.* [Online; accessed 16. Jun. 2024]. 2016. URL: https://brennan.io/2016/11/14/kernel-dev-ep3.

[10] Capstone. *The Ultimate Disassembly Framework*. [Online; accessed 13. Jun. 2024]. 2023. URL: https://www.capstone-engine.org.

[11] Christian Collberg. *The Tigress C Obfuscator*. [Online; accessed 15. Jun. 2024]. 2023. URL: https://tigress.wtf.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[13] Contributors to Wikimedia projects. *Crackme - Wikipedia*. [Online; accessed 16. Jun. 2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Crackme&oldid=1220283524.

[14] Contributors to Wikimedia projects. *Malware - Wikipedia*. [Online; accessed 10. Jun. 2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Malware&oldid=1223843083.

[15] Contributors to Wikimedia projects. *Ransomware - Wikipedia*. [Online; accessed 10. Jun. 2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Ransomware&oldid=1224625270.

[16] Contributors to Wikimedia projects. *Reverse engineering - Wikipedia*. [Online; accessed 21. May 2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Reverse_engineering&oldid=1221145420.

[17] Contributors to Wikimedia projects. *WannaCry ransomware attack - Wikipedia*. [Online; accessed 10. Jun. 2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=WannaCry_ransomware_attack&oldid=1227800240.

[18] *Cutter*. [Online; accessed 15. Jun. 2024]. 2024. URL: https://cutter.re.

[19] Bruce Dang, Alexandre Gazet, and Elias Bachaalany. *Practical reverse engineering: x86, x64, ARM, Windows kernel, reversing tools, and obfuscation*. John Wiley & Sons, 2014.

[20] *ELF 101 - a Linux executable walkthrough*. [Online; accessed 18. Jun. 2024]. 2024. URL: https://github.com/corkami/pics/blob/28cb0226093ed57b348723bc473cea0162dad36
binary/elf101/elf101-64.svg.

[21] *Frequently Asked Questions - angr documentation*. [Online; accessed 31. Aug. 2024]. Aug. 2024. URL: https://docs.angr.io/en/latest/faq.html#why-did-you-choose-vex-instead-of-another-ir-such-as-llvm-reil-bap-etc.

[22] *Fuzzing | OWASP Foundation*. [Online; accessed 16. Jun. 2024]. 2024. URL: https://owasp.org/www-community/Fuzzing.

[23] Lars Marius Garshol. *BNF and EBNF: What are they and how do they work?* [Online; accessed 1. Sep. 2024]. June 2021. URL: https://www.garshol.priv.no/download/text/bnf.html.

[24] *Ghidra*. [Online; accessed 14. Jun. 2024]. 2021. URL: https://ghidra-sre.org.

[25] *Hex Rays - State-of-the-art binary code analysis solutions*. [Online; accessed 16. Jun. 2024]. 2024. URL: https://hex-rays.com/ida-pro.

[26] *ImaginaryCTF-2023-Challenges*. [Online; accessed 15. Jun. 2024]. 2024. URL: https://github.com/ImaginaryCTF/ImaginaryCTF-2023-Challenges.

[27] *Intermediate Representation - angr documentation*. [Online; accessed 31. Aug. 2024]. Aug. 2024. URL: https://docs.angr.io/en/latest/advanced-topics/ir.html.

[28] Wei Jiang, Charles Zhang, Zhenqiu Huang, Mingwen Chen, Songlin Hu, and Zhiyong Liu. "QSynth: A Tool for QoS-aware Automatic Service Composition." In: *2010 IEEE International Conference on Web Services*. 2010, pp. 42–49. DOI: 10.1109/ICWS.2010.38.

[29] *Jinja — Jinja Documentation (3.1.x)*. [Online; accessed 13. Jun. 2024]. 2024. URL: https://jinja.palletsprojects.com/en/3.1.x.

[30] Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar Weippl. "SoK: automatic deobfuscation of virtualization-protected applications." In: *Proceedings of the 16th International Conference on Availability, Reliability and Security*. 2021, pp. 1–15.

[31] David Kushner. "The Real Story of Stuxnet." In: *IEEE Spectr* (2024). URL: https://spectrum.ieee.org/the-real-story-of-stuxnet.

[32] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. "Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization." In: *Information and Communications Security: 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings 19*. Springer. 2018, pp. 313–324.

[33] *libvex_ir.h*. [Online; accessed 31. Aug. 2024]. Aug. 2024. URL: https://github.com/angr/vex/blob/a3f558b7f6426d45688dbbf425bee1e4bea9cc5d/pub/libvex_ir.h#L4.

[34] David Liu and Mario Badr. *17.8 Application: Control Flow Graphs*. [Online; accessed 18. Jun. 2024]. 2024. URL: https://www.teach.cs.toronto.edu/~csc110y/fall/notes/17-graphs/08-control-flow-graphs.html.

[35] Arm Ltd. *What is Instruction Set Architecture (ISA)?* [Online; accessed 18. Jun. 2024]. 2024. URL: https://web.archive.org/web/20231111175250/https://www.arm.com/glossary/isa.

[36] *Malware Analysis: Steps & Examples - CrowdStrike.* [Online; accessed 21. May 2024]. 2024. URL: https://www.crowdstrike.com/cybersecurity-101/malware/malware-analysis.

[37] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey." In: *ACM Comput. Surv.* 52.5 (2019). ISSN: 0360-0300. DOI: 10.1145/3329786. URL: https://doi.org/10.1145/3329786.

[38] *miasm.* [Online; accessed 12. Jun. 2024]. 2024. URL: https://github.com/cea-sec/miasm.

[39] *msynth.* [Online; accessed 12. Jun. 2024]. 2023. URL: https://github.com/mrphrazer/msynth.

[40] Lily Hay Newman. "The NSA Makes Ghidra, a Powerful Cybersecurity Tool, Open Source." In: *WIRED* (Mar. 2019). URL: https://www.wired.com/story/nsa-ghidra-open-source-tool.

[41] *objdump(1) - Linux manual page.* [Online; accessed 16. Jun. 2024]. 2024. URL: https://man7.org/linux/man-pages/man1/objdump.1.html.

[42] *Oreans Technologies : Software Security Defined.* [Online; accessed 16. Jun. 2024]. 2024. URL: https://www.oreans.com/themida.php.

[43] *Practical MBA Deobfuscation with msynth.* [Online; accessed 12. Jun. 2024]. 2021. URL: https://synthesis.to/2021/11/11/practical_mba_deobfuscation.html.

[44] *QEMU.* [Online; accessed 15. Jun. 2024]. 2009. URL: https://www.qemu.org.

[45] *RE: Reverse Engineering.* [Online; accessed 27. May 2024]. 2023. URL: https://cs.unibuc.ro/~crusu/re/labs.html.

[46] Rolf Rolles. "Unpacking Virtualization Obfuscators." In: *WOOT* 9 (2009), pp. 1–10.

[47] Ieva Rutkovska and The BlackBerry Research & Intelligence Team. "Threat Spotlight: The Andromeda Botnet." In: *BlackBerry* (2020). URL: https://blogs.blackberry.com/en/2020/05/threat-spotlight-andromeda.

[48] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. "Symbolic deobfuscation: From virtualized code back to the original." In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer. 2018, pp. 372–392.

[49] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)." In: *2010 IEEE Symposium on Security and Privacy.* 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[50] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. "Automatic reverse engineering of malware emulators." In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009, pp. 94–109.

[51] "SMT Solvers in Software Security." In: *6th USENIX Workshop on Offensive Technologies (WOOT 12)*. Bellevue, WA: USENIX Association, 2012. URL: https://www.usenix.org/conference/woot12/workshop-program/presentation/Vanegue.

[52] Zhanyong Tang, Kaiyuan Kuang, Lei Wang, Chao Xue, Xiaoqing Gong, Xiaojiang Chen, Dingyi Fang, Jie Liu, and Zheng Wang. "Seead: A semantic-based approach for automatic binary code de-obfuscation." In: *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE. 2017, pp. 261–268.

[53] *The LLVM Compiler Infrastructure Project*. [Online; accessed 14. Jun. 2024]. 2024. URL: https://llvm.org.

[54] *TOML: Tom's Obvious Minimal Language*. [Online; accessed 13. Jun. 2024]. 2024. URL: https://toml.io/en.

[55] *UIUCTF-2023-Public*. [Online; accessed 15. Jun. 2024]. 2023. URL: https://github.com/sigpwny/UIUCTF-2023-Public.

[56] Unicorn. *Unicorn – The Ultimate CPU emulator*. [Online; accessed 16. Jun. 2024]. 2023. URL: https://www.unicorn-engine.org.

[57] *Valgrind Home*. [Online; accessed 13. Jun. 2024]. 2024. URL: https://valgrind.org.

[58] *VMProtect Software*. [Online; accessed 16. Jun. 2024]. 2024. URL: https://vmpsoft.com.

[59] David Walker. *CS320: Compilers: Intermediate Representation*. [Online; accessed 18. Jun. 2024]. 2016. URL: http://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf.

[60] Fish Wang and Yan Shoshitaishvili. "Angr - The Next Generation of Binary Analysis." In: *2017 IEEE Cybersecurity Development (SecDev)*. 2017, pp. 8–9. DOI: 10.1109/SecDev.2017.14.

[61] *What are the different types of malware?* [Online; accessed 11. Jun. 2024]. 2023. URL: https://www.kaspersky.com/resource-center/threats/types-of-malware.

[62] *Writing Disassemblers for VM-based Obfuscators*. [Online; accessed 12. Jun. 2024]. 2021. URL: https://synthesis.to/2021/10/21/vm_based_obfuscation.html.

[63] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. "A generic approach to automatic deobfuscation of executable code." In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 674–691.

[64] *Z3*. [Online; accessed 16. Jun. 2024]. 2024. URL: https://github.com/Z3Prover/z3.

[65] *Zeus_Malware_Analysis_Case_Study*. [Online; accessed 7. Jun. 2024]. 2021. URL: https://github.com/Dulanaka/Zeus_Malware_Analysis_Case_Study.

[66] *ZeusVM analysis - Miasm's blog*. [Online; accessed 12. Jun. 2024]. 2016. URL: https://miasm.re/blog/2016/09/03/zeusvm_analysis.html#results.

[67] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. "Information hiding in software with mixed boolean-arithmetic transforms." In: *International Workshop on Information Security Applications*. Springer. 2007, pp. 61–75.

Appendix A

The following Listing is a possible solution for the `vmcastle` challenge, using a plugin generated with the `arch-genesis` tool.

```python
from vmcastle import *
import angr
import claripy

START = 0x17e8      # start symbolic execution from this address
LOSE = 0xa1dc       # avoid this address
SYSCALL = 0x100000
WIN = 0xa1a4        # find this address

p = angr.Project("program")
entry_state = p.factory.entry_state(addr=START)
sm = p.factory.simgr(entry_state)

# create an array of symbolic values, representing the desired flag
flag_chars = [claripy.BVS(f'flag_{i}', 8) for i in range(85)]
flag = claripy.Concat(*flag_chars)

# add the symbolic bytes of the flag,
# at their correct position on the stack
for c in flag_chars:
    sp = sm.one_active.regs.sp
    sm.one_active.regs.sp += 4
    sm.one_active.mem[sp].byte.store(c)
sm.one_active.regs.sp -= 4

# add constrains on the symbolic values,
# corresponding with the printable ASCII range
for c in flag_chars:
    sm.one_active.solver.add(c >= 0x20)
    sm.one_active.solver.add(c <= 0x80)

sm.one_active.options.add(angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS)
# start symbolic execution
sm.explore(avoid = lambda s: LOSE <= s.addr < SYSCALL, find=WIN)
```

```python

# retrieve the solution
sol = sm.one_found.solver.eval(flag, cast_to=bytes)
print(sol)
```