POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Techniques for Malware Analysis based on Symbolic Execution

Supervisors

Prof. Cataldo BASILE

Prof. Antonio LIOY

Dr. Guglielmo MORGARI

Candidate

Pietro Francesco TIRENNA

July, 2020

Summary

The landscape of malicious software, more commonly known as malware, grows every year in number, popularity and financial damage. Just like we see happening in the software industry, organizations in the cybercrime world are well coordinated: they hire developers, distributors, maintainers, they advertise their product, offering deployment services to paying customers and channels to signal bugs to fix. Manually examining every potentially malicious executable would be unfeasible to the least, therefore turning towards automated, fast analysis systems is becoming more and more a requirement to be efficient in the industry and offer meaningful results. To interfere with such automated techniques, malware developers will often hide meaningful routines activated only if certain conditions in the execution environment are met. These, in the literature called trigger conditions, become a great obstacle in automated analysis systems: specific dates, directory names or network commands that would expose the malicious nature of a sample will not most likely be triggered in a generic execution context without prior knowledge of their expected values, therefore leading to false negatives and, in general, to a decrease of the analysis coverage. Consequently, designing systems to expose hidden trigger conditions has drawn some interest in the reverse engineering and malware analysis literature. This thesis introduces Symba, a prototype based on symbolic execution that attempts to reveal trigger conditions in executables. Symbolic execution, precisely, is a software analysis technique which has been introduced in the literature a few decades of years ago but only recently – thanks to an increasing attention from the scientific community – is being practically adopted. Its rationale is to transform a binary executable into a set of symbols and equations binding them, which can be at any time mathematically solved to query the executable for interesting properties. In this work, we specifically resort to symbolic execution to handle the problem posed by trigger conditions. By extracting these conditions from both proof-of-concept and real world samples, consequently observing new paths of execution revealed in automated systems, we demonstrated how applying new analysis techniques, such as symbolic execution, on malware analysis can push a step forward towards more intelligent systems.

Acknowledgements

I wish to express my deepest gratitude to the people who helped me during the development of this thesis. To Guglielmo Morgari, who relentlessly sit with me countless times, listening, supporting, kindly introducing new precious ideas in my flow of thought with the clarity and passion that has the shape of genius. Thanks to Emanuele De Lucia, who, in the most friendly and competent way, introduced me to the concepts of malware analysis and cybercrime research. I am truly indebted with Fabrizio Vacca, who gave me the opportunity of working with the wonderful laboratory guys at Telsy, where I had the privilege of growing from both a human and professional perspective. My most sincere appreciation and admiration goes to professor Antonio Lioy, who encouraged me towards finding a research direction that could truly motivate me, something that would give a contribution, for how small, to our community. A special regard goes to Cataldo Basile, who supported me in so many ways that I could not count them. Thanks for giving me your continuous guidance, assistance and advice, in a way that I truly feel grateful for. Thanks to my teammates at PolitHack, where I found the motivation to start tinkering with the amazing angr framework. I wish to acknowledge the constant love and support of my family. To my dad and my mother, Carmelo and Gabriella, and my two sisters, Rossana and Caterina: I would not have been able to reach this point without you, and your tireless help and kind words. To my friends, for being there, helping me with a beer and a chat when the road got tougher. To Chiara, all my love and gratitude. You make every day my best day, even when I spend it analysing disassembled and messy malware.

"Technology is the campfire around which we tell our stories."
Laurie Anderson

Table of Contents

List of Figures

Chapter 1

Introduction

Keywords: cyber security, malware, analysis, automatic malware analysis, threat intelligence, symbolic execution

Nowadays, the field of malware analysis is receiving increasing attention from both the academic and corporate world. terms like *cybersecurity* and *threat intelligence* are trending in journals and technical blogs, and many reverse engineering gurus are less interested in cracking games than what they are in analysing malicious software. Networks of malware developers are growing, recruited by state groups, red teaming operation squads, and, infamously, threat actors. Antivirus evasion, anti-detection, anti-debugging and in general anti-reversing techniques are exchanged in forums and taught in university courses. It is clear how, in this active environment, the number of families tracked by analysts is growing, where not necessarily in complexity, most certainly in numbers.

In this heterogeneous context, analyses of malicious samples can have different scopes, different questions to answer, and each question has to be handled by specific approaches. For instance, the analysis of an Android application to determine whether that piece of code is malicious or benign can help determining whether that specific app should be removed by Google Play Store. This binary classification - malicious or not malicious - often has to be quick. Imagine a scenario where the Blue Team of some corporate organization has detected a suspicious executable uploaded in one of the workstations connected to the network. Every minute lost in analysing the malware leads to a minute more computed on the time of reaction, time spent before appropriate measure of confinements can be taken.

Other kind of analysts do not have the goal of determining whether a sample is malicious or not, but rather understanding in depth what the purpose of the sample is, and which techniques it is employing to obtain it. Analysts assigned to this task have to determine what kind of action the malware will perform in the machine, such as stealing data or executing code sent by a remote server. They will

have to understand how the code is obtaining persistence in the target machine, and which techniques are employed to defeat known analysis approaches. In short time, they have to write a detailed report of the sample, a dissection of its features. This kind of report is extremely valuable.

- Other malware analysts in the network will learn how to handle that family or similar ones.
- AV companies will use those analyses to compute signatures to update their databases and quickly recognize whether a sample belonging to that family has infected the machine they're installed on.
- New techniques equipped on modern samples can be documented in frameworks like the MITRE (`https://attack.mitre.org/`).

It goes without saying that these kind of detailed analysis and reports of malware samples are not performed in a few minutes. Depending on the degree of sophistication of samples, these tasks can require from different hours to even days of work. In certain cases this is completely acceptable. For instance, a malware analyst tasked with analysing complex APT samples developed by threat actors like Fancy Bear [1] will not probably "see" multiple samples per day. However, for companies operating in the corporate and business world, gathering and processing data from samples being delivered daily in email attachments, honeypots, Point Of Sales stations, the requirements are much different. It is enough to research how many samples an engine like Virus Total processes per day to shape an idea of the flow of this market.

How can companies without thousands of analysts cope with this volume of samples? In the past years we have withstood an increasing attention in automation techniques applied to malware analysis. These can be categorized as static or dynamic, each one with their perks and disadvantages. Static analyses have a good coverage and do not need any kind of target environment to execute the malware in. Obfuscation and anti-reversing techniques make this kind of analysis harder, even when scripts and not simple human eyes are used as tools of the trade. On the other hand, dynamic techniques are inherently more capable to observe the behaviour of code, as is the case with malware sandboxes technologies. A sandbox is a controlled environment where malicious code can "explode" and every action it performs on the machine is logged. File system operation, network traffic, API calls, registry keys queried and written, everything is monitored to compute a detailed behavioural report where interesting properties of malware samples can be extracted and processed without human interaction.

However, modern malware will often use various approaches to detect when they are being executed in a controlled environment and not a simple victim machine

as it could be the laptop of a marketing director of a target company, and once detected so they will just terminate without expressing any of their malicious features, thus rendering misleading and not useful the analysis performed.

We present *Symba*, a system to enhance malware analysis capabilities and statically extract meaningful properties from analysed samples through *Symbolic Execution*. Symbolic execution can overcome some key challenges in malware analysis, such as being able to automatically recover properties of interest from samples that mutate on little attributes. For instance, let us take the scenario of a Remote Access Trojan that accepts from a C2 server a suite of commands to execute in the target machine. In order to hinder network detection, commands are obfuscated or encrypted so that the IDS will never see anything like "GET" or "WEBCAM_SCREENSHOT" in the incoming packets. The obfuscation could be simple as XORing every received byte with a fixed reference byte, or a bit more complex as in the case of small permutations on received traffic. Through symbolic execution techniques, an analyst can develop a script able to extract the *clear-text* commands from the samples, applicable to samples belonging to different campaigns where, as it often happens, the key byte could be each time different.

Another key concept is that of *path constraints*, also called *Trigger Conditions* [2] in the relevant literature. The idea is that, in general, the malicious part of a malware is reached through a forest of checks that the execution environment has to pass first. These guard conditions are employed in order to defeat, as previously mentioned, sandboxed mechanisms, but also to ensure that the malware is running in the expected environment, where some registry keys, usernames, or region attributes are needed. It is interesting, for example, the case of malware checking the machine locale before "exploding", to avoid targeting countries belonging to the Russian territory. Dynamic techniques with no proper guidance, therefore, will not help in all these cases at observing any kind of malicious behavior. The way Symba handles this problem is to *taint* interesting sources of data and consequently taint any region of memory "contaminated" by this data. Branching instructions on tainted data can uncover what the code is expecting to retrieve from those API to unlock the different branches hidden inside its code.

1.1 Contribution

This thesis brings the following contributions to the field of Windows Malware Analysis.

1. We introduce Symba, a system to automatically extract trigger conditions from malware targeting the Windows OS.

2. We describe some novel techniques to generate and inject the potential inputs

that are then chained in order to extract properties from malicious sample, to construct the correct environment where malware has to be executed.

3. We evaluate and demonstrate the capability of Symba to extract such properties from real malware samples and benchmarking code typically used to measure the degree of detectability of an environment.

1.2 Outline

The rest of this thesis is organized as follows.

- Chapter 2 states the requirements of this work. Namely, the specific problem that we identified, the ideas designed to solve it, the tools used to implement a system matching the design, and the approach used to verify the validity of such implementation.
- Chapter 3 gives the reader a comprehensive background of the theoretical and technical concepts handled in this thesis. Ranging from the environment where we have worked, the Windows OS, to known malware analysis techniques using both static and dynamic approaches. Finally, an academic overview of symbolic execution will be presented.
- Chapter 4 explores the capabilities of the angr framework, the symbolic execution engine employed for this thesis. Each section will highlight different features of this library, including examples of their use aimed to facilitate reading.
- Chapter 5 describes the design of Symba, first in its entirety, that is, how components interact with each other. Subsequentially, every component is "disassembled", and its role in the workflow is thoroughly characterized.
- Chapter 6 contains some details about the specific implementation of Symba. It incorporates some notable snippets of code extracted from the tool, alongside a few comments on their inner mechanisms.
- Chapter 7 presents the case studies where Symba has been evaluated. It starts with a proof of concept RAT server developed by following documented malware development techniques. Then, Symba is launched against a highly starred project in GitHub that simulates known and less known techniques that a malware would use to detect virtualization, Paranoid Fish. Afterwards, Symba is tested on a real malware for which the source code has been leaked on GitHub, named Wrathrage. The chapter is concluded with a depiction of the system limitations in its design and implementation.

- Chapter 8 describes some other works, consulted from the available literature, that handle the problem of solving trigger conditions, working with Windows malware and using symbolic execution.
- Chapter 9 concludes the thesis with a glance towards the future researches that could be conducted in order to extend this piece of work.

Chapter 2

Requirements

2.1 Problem Statement

The problem tackled in this thesis is also known, in the relevant literature, as *trigger analysis*. In order to define what a trigger is, we observe that malicious software often hide behaviour under precise conditions with different purposes in mind. A few examples of trigger conditions in malware follow.

- Checking the system date in malware that only activate their routines in specific days and months.
- Checking some registry key to detect the presence of a virtualized environment, such as VMWare or VirtualBox.
- Activating a keylogging mechanism only in the presence of a certain window title that matches, for instance, the name of a banking account.
- Listening on the network for packets containing precise commands, returning a fake response otherwise.

There are two main issues with trigger conditions. First of all, the routines hidden behind triggers are often essentially what characterize the executable as malicious. In scenarios where analysts execute the sample to monitor its behaviour and determine the degree of suspiciousness, without the knowledge of those triggers, they would not notice anything malicious in the report of execution, with the risk of falsely classifying a malware as benign.

Moreover, in situations where the examined sample has already been determined to be a malware, revealing triggers is not a simple task. With dynamic analysis techniques, there is a high change that the generic environment where the malware is monitored will not satisfy any of the conditions. Turning to even more dynamic

techniques like fuzzing the malware input to observe new behaviour could be useless. Let us take the example of a server with a particular command suite that listens for packets on the network. The chance that, by fuzzing network input to the binary, one of the commands in the suite is found, has the same statistical features of a complete bruteforcing attack on a password, which as known is generally limited in effectiveness.

Therefore, extracting triggers often turns out to be a manual, time-consuming static approach. The analyst needs to find the correct spots where a condition is checked for, disassembling and figuring out the meaning of all the checks, even when obfuscated between different instructions, and solve them manually. Even in simple scenarios, where the analyst is able to extract the triggers in no more than a couple of hours, such approach becomes impractical when applied to hundreds of samples per day.

With these premises, we believe that the process of finding and solving triggers in potentially malicious executables needs to be pushed towards automation.

2.2 Proposed Solution

The solution proposed in this thesis takes the name of *Symba*, and it stands for *Symbolic Behaviour Analyzer*. The word symbolic introduces the main approach used in this work, *symbolic execution*. Symbolic execution is a technique, described in detail in the background chapter at 3, which transforms executables into a set of symbols and mathematical constraints over them. Therefore, every different state of execution can be described as a precise equation binding all the symbols part of the symbolic analysis, such as registers, memory addresses, opened files, et cetera.

We identify, linked to the concept of trigger, the one of *trigger source*. A trigger source is a function from the Windows API which will produce the value used in the condition checking. For instance, in case of a condition based on the date of execution, we define as trigger source functions in the GetSystemTime family. Once defined trigger sources, we symbolically track their output values, transforming them into controlled symbols in the context of our symbolic execution. Once the condition in the trigger check poses constraints on the symbol, Symba detects it and manages to solve it for all the different branches. This way, concrete values which satisfy all the different branches dependent on the condition are extracted.

The framework chosen to implement symbolic execution in Symba is angr [3]. Angr is a binary analysis framework, with symbolic and dynamic execution capabilities, developed in conjunction between the UCSB and the Arizona State University, integrated in a system designed for auto-exploitation of binary executables for the Cyber Grand Challenge organized by DARPA (`https://www.darpa.mil/program/cyber-grand-challenge`). It comes as a Python 3

library. It exposes a clear, readable and customizable interface, it is well maintained and it has been allegedly proved to perform better than competitors in pure symbolic tasks [4].

2.3 Expected Outcomes

The expected outcomes of the prototype of Symba developed for this thesis follow below.

1. The analyst, or an automatic system in which Symba is integrated, executes a sample in a target environment, observing a certain behaviour exhibited by the binary.
2. The analyst, or an automatic system in which Symba is integrated, feeds the developed tool with the binary executable and a list of API function names configured as trigger sources.
3. Symba symbolically executes the binary, tracking output values of trigger sources, monitoring every constraint dependent on them.
4. Once the symbolic execution process terminates, all the constraints are solved and concrete values to satisfy all branches of the trigger conditions linked to the sources are written to a logfile.
5. The analyst, or an automatic system in which Symba is integrated, takes these logged concrete values, injects them into the context of the target environment, and it executes the binary.
6. In this new execution, the analyst or the automatic system, observes new behaviours unlocked by trigger conditions.

Chapter 3

Background

3.1 Windows System

Windows, developed my Microsoft, is a notorious modern operating system which is heavily used both in the corporate and the customer world. This offers a perfect opportunity for malware developers, since their effort - in terms of money and time - spent in development and distribution can yield a greater return on investment with respect to other niche operating systems. The rest of this section will introduce the Windows API, the interface used by developers to interact with the operating system, to give afterwards an introduction of dynamic and statical analysis approach, and ends with a presentation of symbolic execution, the main approach employed in this work.

3.1.1 Windows API

The Windows API, formerly called Win32 API or Winapi, is a core set of application programming interfaces available in Windows operating systems. The most direct way to use the API is through C and C++ programming languages. In order to use it, users must download and install the Windows Software Development Kit (SDK) which contains all the relevant header files, libraries, documentation and tools needed by the API to develop software. The macro categories which segment the API follow.

- *Base services.*
- *Security.*
- *Graphics.*
- *User Interface.*

- *Multimedia.*
- *Windows shell.*
- *Networking.*

The *Base services* exposes fundamental interface on the OS like file system, processes, threads or registry handling. *Security* API provides programming elements for authentication, authorization and other security related prototypes. The *Graphics* subsystem provides functionality to show graphical content on monitors, printers and generic output devices. The *User Interface* can be used to create windows and controls. *Multimedia* offers the capability to work with video, sound and input devices. *Windows shell* interface allows applications to execute commands in the context of an OS shell. Finally, *Network services* provides access to the network capabilities of the host.
The official reference for the API is the MSDN (Microsoft Developer Network), while the bone and meat implementation of the API specifications is located in Windows DLLs.

3.1.2 Windows API example

In order to show in greater detail how the API are used by programmers and what ends up in a compiled executable, let us describe an example scenario. As developers, our goal is to write a program to query, format and print to a message box the current time of the system. The first step is to query the MSDN documentation to find out specifications of API which we could use. A quick search of "System time" on the search box offered by the MSDN offers results of the relevant API specification, which are reported in the listing below:

```
1 void GetSystemTime(
2   LPSYSTEMTIME lpSystemTime
3 );
```

As specified in the documentation, the parameter lpSystemTime is a pointer to a SYSTEMTIME struct, which has the following specification:

```
1 typedef struct _SYSTEMTIME {
2   WORD wYear;
3   WORD wMonth;
4   WORD wDayOfWeek;
5   WORD wDay;
6   WORD wHour;
```

```
7   WORD wMinute;
8   WORD wSecond;
9   WORD wMilliseconds;
10 } SYSTEMTIME, *PSYSTEMTIME, *LPSYSTEMTIME;
```

The listing is clear and states which words will contain the different fields returned by the operating system. Following the same reasoning, we lookup functions to display message boxes in the Windows OS, and we encounter MessageBox. The documentation of this function follows:

```
1 Displays a modal dialog box that contains a system icon, a set of
    buttons, and a brief application-specific message, such as status
    or error information. The message box returns an integer value
    that indicates which button the user clicked.
```

and the signature:

```
1 int MessageBox(
2   HWND    hWnd,
3   LPCTSTR lpText,
4   LPCTSTR lpCaption,
5   UINT    uType
6 );
```

Every field is well described in the rest of the MSDN page.

As depicted above, interacting with the Windows API is merely a matter of including the right header, consulting the documentation and respecting the interface exposed.

3.2 Windows Malware

Having introduced the basics of how programs interact with the Windows OS, the following section will discuss further on how malwares are classified, based on their behaviour and goals in the system.

3.2.1 Classification

Malware is a generic *umbrella* term used to describe many kinds of malicious, hostile software. It can be used by cybercrime industry in order to gain money, in APT [5] campaigns, both in simulated attacks (Red Teaming Operations) and state actors [6] to damage or control target systems. It can be used by hacktivist groups or as a vector of revenge towards enemies. In this wide context, researchers

developed some criteria to classify [7] [8] [9] the multiple types of malware in several ways. These criteria include:

- *The delivery method and attack methodology.* Victims can get infected simply by visiting an evil or compromised website - *drive-by distribution* - or via attachment in phishing emails that trick victim into executing them. Man-in-the-Middle attacks where the content is injected into traffic packets, or using exploits targeting browsers, in what is called *Exploit kit* [10].
- *The goal or objective.* Different types of malware have different scopes. Some may have purely financial goals, while others are deployed in order to control remotely the target machine, or exfiltrate sensitive data.
- *The platform that the malware targets.* Operating system, architecture, even specific builds.
- *Approach to stealth.* How does the malware attempt to hide itself from user or AV detection?
- *Behavior and peculiar characteristics.* Specific features such as how the malware replicates and spreads, or other distinguishing attributes [11].

Generally, some of the peculiar types of malware tracked by researchers can be listed as follows.

- *Dropper/Downloaders.* This kind of malware serves the purpose of being the first stage of infection in the target system. Upon execution, they will typically extract and run a second stage malware which contains the meaningful code to compromise the machine. Depending on the type of dropper, the latter will be retrieved using the internet or it will be hidden inside the file itself.
- *Information stealers/Keyloggers.* Most often, the goal of these programs is to exfiltrate - through email, HTTP, FTP or custom TCP tunnels - sensitive data from the target system. This may include keyboard presses, stored passwords, files, browser cookies, and in general everything that will have a market for the threat actors.
- *Bankers.* This usually sophisticated type of malware will hook specific APIs or DLLs used by Web Browsers in order to inject custom content or exfiltrate information when the unknowing user visits banking web apps. The goal is financial, to extract card information or hijack accounts.
- *Ransomware.* Ransomware encrypt files on the system, requesting money to the user to get them back.

- *Remote Access Trojans*. Once infected by a RAT, the machine will establish a tunnel on the Internet towards a *Command & Control* server, expecting commands to execute on the host, such as exfiltrating data or pivoting further in the network of the victim.
- *Rootkits*. Generally more advanced malware, rootkits establish their persistence at a lower level, which can assume the form of a kernel driver, or even as code hidden in the MBR of the hard disk. Able to hide from plenty of detection techniques, they may survive disk formatting and OS reinstall.
- *Worms*. Typically infamous pieces of code like the Morris Worm and, more recently, WannaCry [12], these malwares require almost zero interaction from the user, since they use wide-spread and powerful vulnerabilities. For this reason, they are able to replicate and spread in a small amount of time and potentially deal great amount of damages to impacted systems..

3.3 Dynamic and static analysis

Before introducing the theory which Symba is built upon, symbolic execution, the following section will describe how analysis of malware is handled in the literature [13], trying to highlight the perks and the disadvantages of both.

3.3.1 Static Analysis

This term embraces all the different kind of analyses that can be performed without actually running the PE file. One relevant example of static analysis is the *disassembling* and *decompilation* of malware samples. Industry-standard tools like IDA Pro, OllyDBG and, most recently, Ghidra [14] are packed up with features, analysis routines, addresses cross-referencing, even scripting interfaces that can be used to easily automate custom routines. Often, malware uses binary packers such as UPX, to avoid being analyzed [15] [16]. Therefore, in order to be disassembled, specific unpacker will have to be used. The typical interface of a modern disassembler is demonstrated in fig. 3.1:

Also, strings can be really good indicators of maliciousness, and in some cases they can yield interesting information and properties of samples, like C2 IP addresses, or files searched in the system.

As previously mentioned, Windows API calls can be used as detection patterns during static analysis; for instance, `CreateRemoteThread`, `LoadLibrary`, `WriteProcessMemory` are suspicious function often used by malicious samples for DLL injection into other processes, and they are rarely used for legitimate code.

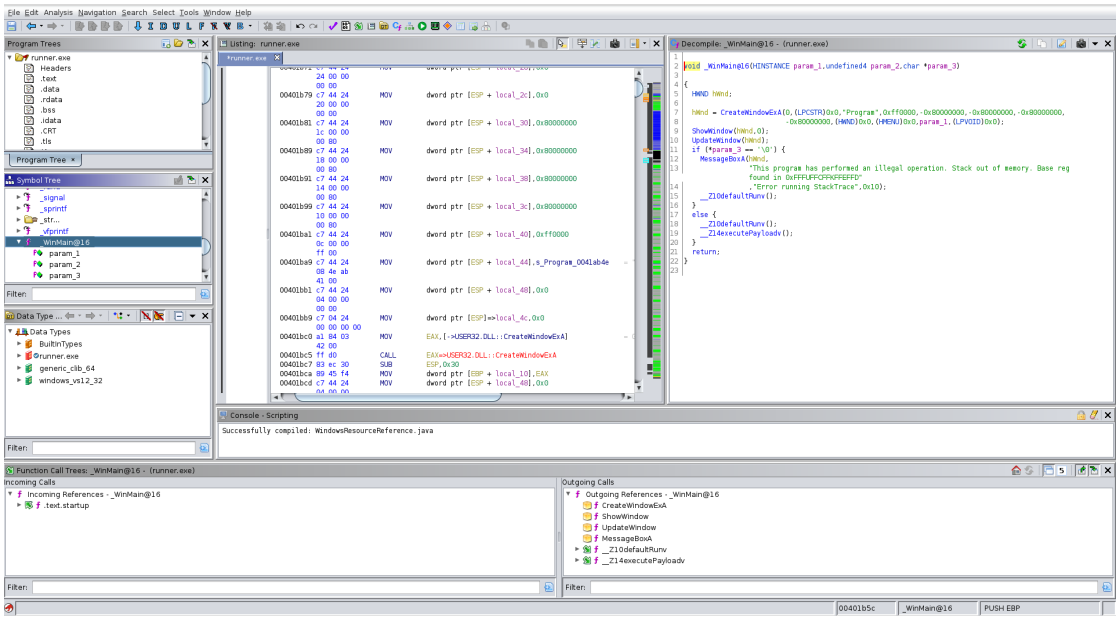

Figure 3.1: Interface of the reverse engineering tool open sourced by the NSA, Ghidra.

A *control Flow Graph* (CFG) [17] is a directed graph where basic blocks of code are represented by graph nodes, and flow paths - like branches - by edges. For instance, in Symba design and implementation, CFG will be used to extract basic blocks preceding and following a certain API function call deemed as particularily interesting. Modern analysis frameworks implement methods and tools to graphically output these kind of graphs, which provides an easier understanding and mapping of the general features of the code, as seen in fig 3.2.

Notably, other static analysis techniques have been employed in the literature, such as the usage of N-grams combined with API calls or opcodes, and file attributes like size or content entropy [18].

3.3.2 Dynamic Analysis

Dynamic analysis [19] is an approach which aims at analyzing code by executing it in a controlled environment where its actions can be observed. The level of depth of this observation can vary [20], from code instrumentation that monitors and operates on executed instructions, to the system as a whole, e.g. by logging registry, file system, network operations.

As with disassemblers for static analysis, one of the most notorious application of dynamic analysis in the software analysis field is *debuggers*. Available for most modern platforms, architectures and operating systems, debuggers provide the

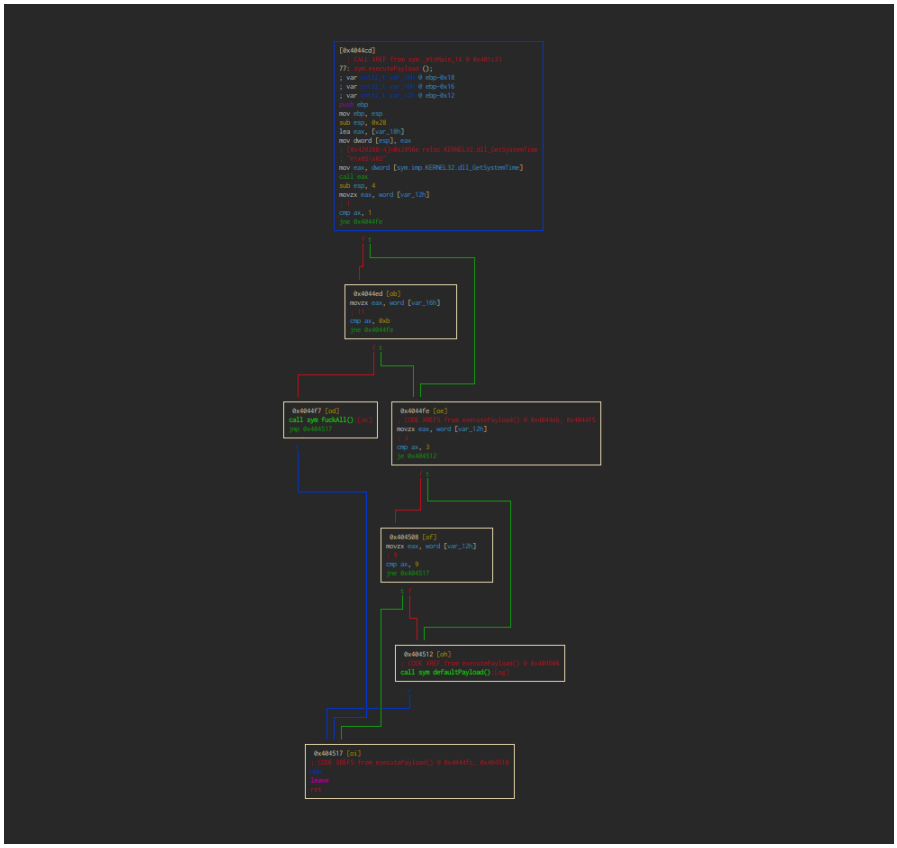

Figure 3.2: Flow Graph generated by radare2 framework.

analyst with the possibility to step through the code, instrution after instruction, to observe in a detailed and controlled way the flow of execution of binaries under scrutiny. While debugging can be used by developers to track down and solve bugs - as the very etymology of the words shows - it constitutes a useful tool in the hands of reverse engineers trying to extract knowledge and properties out of analyzed potentially malicious samples. Incidentally, one of the already mentioned industry standard tools for malware analysis, IDA Pro, also features a capable debugger. Other notable mentions here are WinDBG, ImmunityDBG and x64dbg for Windows, while the de-facto tool in the Linux environment remains GDB. Even though debuggers take the role of manual, precise tools in the field of dynamic analyses, other automatic approaches are used in the wild with good results. For instance, we already mentioned API calls sequencing as a valuable tool to detect and classify in a first quick analysis malicious samples. However, static observation of API calls suffer from the drawback of not being able to monitor which parameters are being passed to API calls. Moreover, some malware developers have been obfuscating their samples by importing API function which have no real use in

the binary, other than confusing analysts trying to extract information from the imports table. For this reason, analysts will often use monitoring tools which will hook API calls performed by binaries alongside their parameters, and output them in a filterable tables. The tool of the trade, in this scenario, is ProcMon of the Sysinternal suite (fig 3.3).

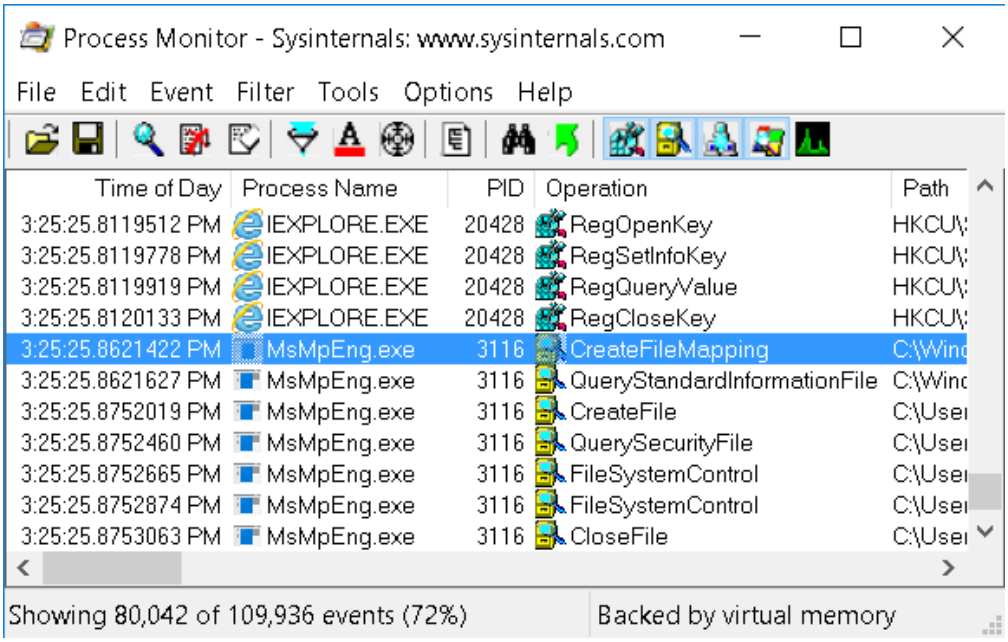

Figure 3.3: Customizable monitoring with Sysinternals procmon.

In dynamic analysis the goal is to observe as many meaningful operation performed by the binary as we can, and one of the most important actions, most often present in malicious samples, is communication with the outside. Therefore, we include to our description of dynamic analysis techniques that of network sniffing, which can be naturally automated based on what the scope and target of the analysis is. A great tool to capture packets going out and in of the network is Wireshark, which, thanks to the library underneath it, allows powerful filtering, inspection and extraction of sniffed traffic. However, since by the time of analysis is conducted the real C2 servers linked to the sample could be offline, or perhaps because traffic outside is not allowed in the analysis environment for security reasons, some analysis environments adopt network services simulators like INetSim, where network interaction is emulated so to observe the functionality of the sample in its totality. All these functionalities, alongside observing changes made to the registry, file system, and in general Operating System, can be automated by using a dedicated *malware analysis sandbox*. Briefly, a sandbox works as follow: for each analysis, a dedicated Virtual Machine is spawned and the sample is executed inside

it. All the operations mentioned up to here, and more, are monitored and recorded, and at the end of the execution, after a given timeout or upon malware termination, they are used to build a detailed report of the execution. Modern sandboxes also automatically extracts "suspicious" signatures - basically, using Yara rules precompiled in the sandbox - out of the monitored facts and use them to compute a score of maliciousness of the analysed samples. These services can be exposed onlines in applications like app.any (`https://app.any.run/`), joe sandbox (`https://www.joesecurity.org/`) or VirusTotal (`https://www.virustotal.com/gui/`), or can be hosted offline by the analyst in projects like the Honeynet Cuckoo Sandbox. An example of suspicious analysis signatures as extracted by Cuckoo is shown in figure 3.4

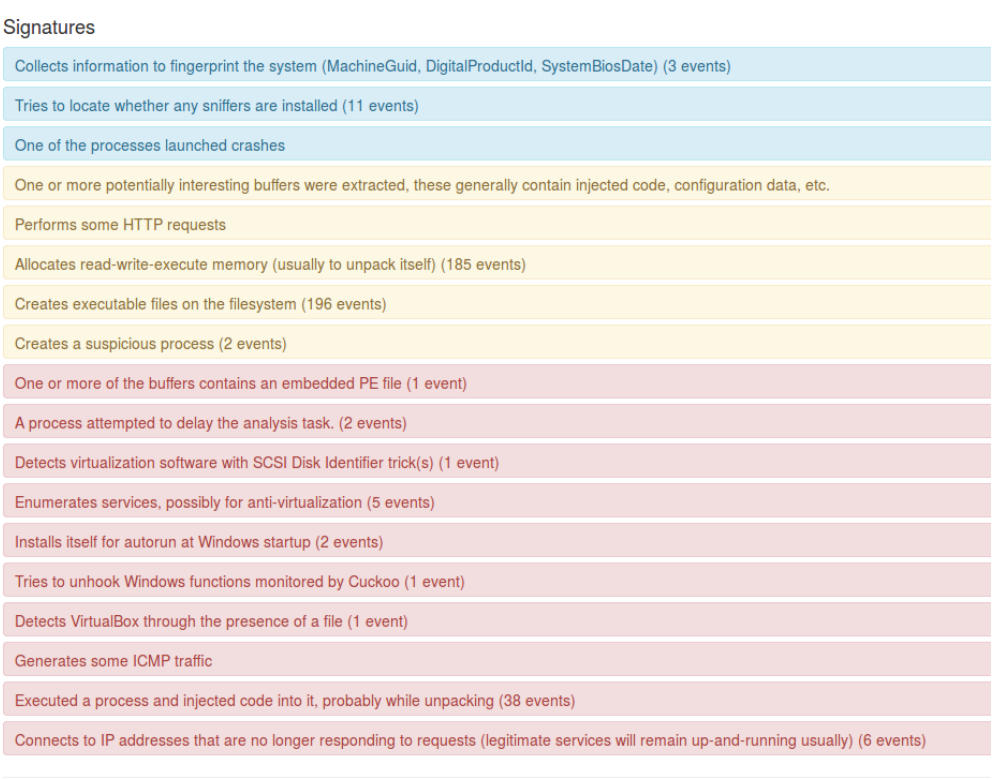

Figure 3.4: List of signatures extracted by cuckoo for a Poweliks sample. Color indicates degree of suspiciousness.

3.3.3 Advantages and Disadvantages

After having defined the boundaries and ideas of the two approaches, let us map, for both, what could be the typical advantages and disadvantages in applying

them during the process of analysis. This section will not take into account the time spent by the analyst using one approach instead of the other, given that this measure heavily depends on the tools employed, which can greatly vary in their capability of automating the analysis process.

Dynamic

With dynamic analysis, suspicious samples are detonated in a controlled environment, thus all its operations are effectively monitored and graded. This allows for a precise report on the malware, rather than a report based on heuristics and static signatures. Any kind of static obfuscation performed on the binary code, fake API imports, and generally counter-measures utilized by malware developers to confuse analysis will have little power in this environment. Moreover, some samples, in order to properly execute their routines, could need some kind of input or information that static analysis simply cannot extract alone. Let us take, as instance, a RAT sample that executes code received on the network as command. Without having executed the malware, thus contacting the C2 server, the analyst cannot fully determine the nature and format of the expected commands, let alone their specific protocol, encryption, and so on and so forth.

This great capability of monitoring the sample in a totally controlled and "real" environment poses a great amount of issues to these kind of solution. The first problem is that malware analysis environments are generally very recognizable. In fact, to evade detection, attackers will code into their malware multiple routines trying to identify whether the sample is being executed in a "legit" victim environment or a monitored sandbox. They can search for various indicators such as the presence of user activity - mouse movement or key presses - unusually sized disk drives, suspicious names of user and processes running – for instance, VirtualBox will typically spawn some processes containing "VBox" in guest machines – registry keys with flagged names or values, presence of other hosts in the network, together with a countless other possible validation that are well documented in the modern literature [21].

Furthermore, detonating potentially malicious samples is most definitely a dangerous activity to perform, especially from machines belonging to corporate organizations connected to the company network, where the unknown activities exhibited by the malware could potentially compromise it. Therefore, dynamic analysis systems have to be secured as much as possible, using one-time, throwaway virtual, isolated environments only valid for one round of execution, without allowing any kind of interaction with the host OS. Considering again the scenario of a RAT trying to establish a network communication, this could mean that the system won't allow the guests to initiate connections to the outside, since this could pose a threat for the entire network. For this reason, network services emulator

which fake plausible responses are sometimes used, while it goes without saying that such security compromise could negatively impact the soundness of the analysis.

Finally, the kind of described environment can be complex to design and build. On one side, different samples require different targets. OS, architecture, even a specific version of a particular piece of software or a particular DLL could be needed by the sample in order to execute as expected, posing a great challenge for systems and tools trying to analyze malware targeting different platforms such as Linux, Windows, Android. On the other hand, it is clear how the entire process of bringing up the virtual machine, transferring the binary file, executing it and monitoring - via instrumentation and hooking - its capabilities, then shutting down or resetting the machine and computing a full report can take several minutes. In environments where thousands of samples are fed into the system on a daily basis, performance is definitely a problem to address.

Static

Unlike dynamic analysis, as has already been mentioned, static analysis simply look at the contents of files as they exists on disk, without executing them whatsoever. It uses patterns, attributes, signatures and artifacts to extract meaningful information from the binary.

Static analysis is definitely more resilient to the same issues that were defined before with regard to dynamic analysis. It is much more efficient, since the process of handling the virtual environment doesn't exist here, and more cost-effective, not requiring an infrastructure of distributed machines and the capability to handle instrumentation, hooking, evade detection and mantaining the system secure at the same time. Static tools can be quickly written specifically for the analyzed binary, independently of the architecture, OS, library or networking capabilities needed. However, this kind of analysis can be easily hindered to the point of rendering it useless by largely available and known techniques:

- *Code obfuscation.* The idea behind code obfuscation [22] [23] is to transform instructions of the original code in some way that maintains the same semantic behavior of the former, but in a way that is much more difficult to analyze. Notable examples are *opaque predicates*, sequence of code ending in a branch that always produce the same result, in a way that would be difficult to detect from static analysis, and *control flow flattening* (fig 3.5), a technique that attepts to make it as difficult as possible to build a useful CFG out of the binary, by bringing all the flow logic down to a list of "leaf" basic blocks and a single (or multiple) dispatcher blocks that use and update a state variable to jump to the relevant blocks. Both techniques have been reportedly used in modern malware and are available in open source tools such as the LLVM obfuscator [24].

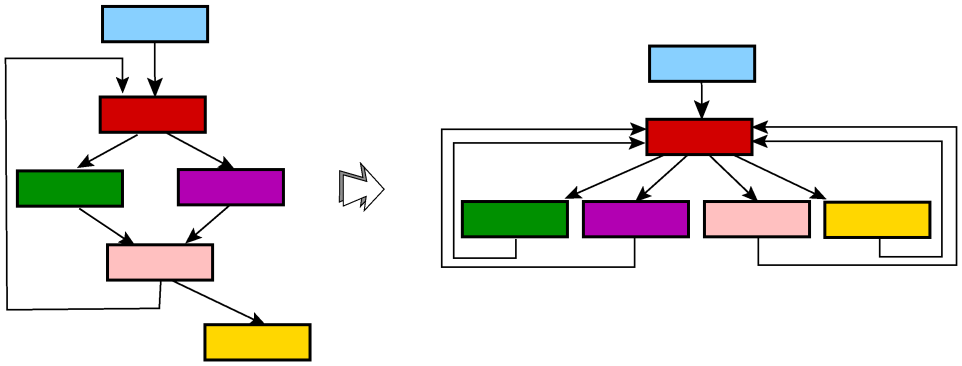

Figure 3.5: Diagram of how the CFG is transformed after the process of flattening (`http://tigress.cs.arizona.edu/transformPage/docs/flatten/index.html`).

- *Packing.* A packed [15] (also referred to as *crypted* in the lingo) malware does not show, when first analyzed, its relevant code anywhere. In fact, the only code available is the *stub*, a small portion of instructions that contain a decompression or decryption routine which, at run-time, will be used to extract the real malware - which is stored in some place, like the resources of PE file - and load it in memory. This way, any kind of static analysis prior to the unpacking will be practically useless.

- *False positives / False negatives ratio.* Since based on heuristics and signatures, static analysis can flag only to a certain extent of soundness a malware as malicious. Without executing it, it is hard - without manual intervention from the analyst - to determine whether those signature are really malicious or just potentially suspicious. In fact, it is entirely possible to craft harmless samples built just in order to trigger static analysis tools for incorrect or malicious attribution to other groups or campaigns or simply to bring confusion inside the analysis environment.

Both analysis approaches have significant drawbacks when dealing with trigger conditions. Statically analysing a trigger essentially mean to disassemble and examine the binary instructions around the condition, trying to reverse engineer the meaning of the conditions, and the values that could solve both arms of branches dependent on it. On the other side, generic dynamic systems, as already depicted, fall short on feeding precise inputs that the malware expects before exhibiting its

malicious demeanor. For this reason, we introduce another technique that is a bridge between static and dynamic analysis, which allows to automatically reason on the code to extract interesting properties. The idea of combining static and dynamic techniques to improve efficiency in malware analysis is not particularily novel [25].

3.4 Symbolic Execution

Symbolic execution is a program analysis techniques introduced fifty years ago in the literature, and which is now withstanding increasing interest in the literature [26]. Its power comes from the ability of studying properties of so-called *aspects of interest* [27]. Aspects of interest are properties of the binary such as that no pointer is equal to NULL at the time of deferencing, or that no overflow in buffer operations occur.

In a real, concrete execution, the program takes inputs from the environment and explores only one control flow path. With symbolic executions, concrete inputs are replaced by symbols. Symbols can simultaneously represent multiple values. Every time an operation is performed on a symbol, a constraint is linked to the symbol. Execution is thus performed by a *symbolic execution engine* which maintains, for each path, the constraints bound to each symbol. This allows symbolic execution to explore multiple paths of a program at the same time, and mathematically reason over each state.

3.4.1 Components

In every moment, a symbolic execution engine manages symbolic states. Each state is characterized by:

- *a memory store.* This maps relationships between symbols and other symbols, or symbols and concrete values. During symbolic execution, assignments and operations on the symbols update the symbolic memory store.

- *a boolean formula.* This component collects, for every *if* statement that handles a symbol, the conditions that have to be bound on symbols to meet each branch. Therefore, for every different path in the execution tree, a different boolean formula is held.

- *a model checker.* An SMT solver which can solve, at any time, boolean formulas to generate concrete values, or to ensure the *unsatisfiability* of the formula.

```
1 void foobar(int a, int b) {
2 int x = 1, y = 0;
3 if (a != 0) {
4 y = 3 + x;
5 if (b == 0)
6 x = 2*(a + b);
7 }
8 assert(x-y != 0);
9 }
```

A detailed overview of the symbolic execution features are described in chapter 4, when discussing of the capabilities of angr. The figure 3.6 taken from [27] describes the symbolic exploration of a simple function, where the interest of the analysis is in finding for which parameters the assert fails. The letters used in the figures represent the following:

- *stmt* is the next statement to be evaluated in the execution path.
- θ is the symbolic memory store, and α_i denotes the symbolic variables.
- π is the path formula.

The branches in the diagram show how a symbolic execution engine updates its path formula and store depending on the values that the symbols simultaneously take. At some point, a state where the assert fails is found, and the symbols are resolved to the value of $(a = 2, b = 0)$.

3.4.2 Weaknesses

Discussing about weak points of symbolic execution is an important part of the analysis of the tools conducted before implementing Symba. Theoretically, symbolic execution is capable of guaranteeing complete soundness on the analysed binary. However, most of the times, a trade-off between quality and feasibility has to be made. Mainly, we can sum up the issues to solve when employing symbolic execution as follows:

- *memory interaction.* While symbolic execution easily handles the modeling of simple variables like integers, or characters as symbols, other more complex data structures bring with them a different spectrum of challenges. How does the engine manages memory addresses that depend on user input or on dynamic allocation, such as strings [28]? If the address becomes a symbol, it is clear how the memory store can grow in complexity.

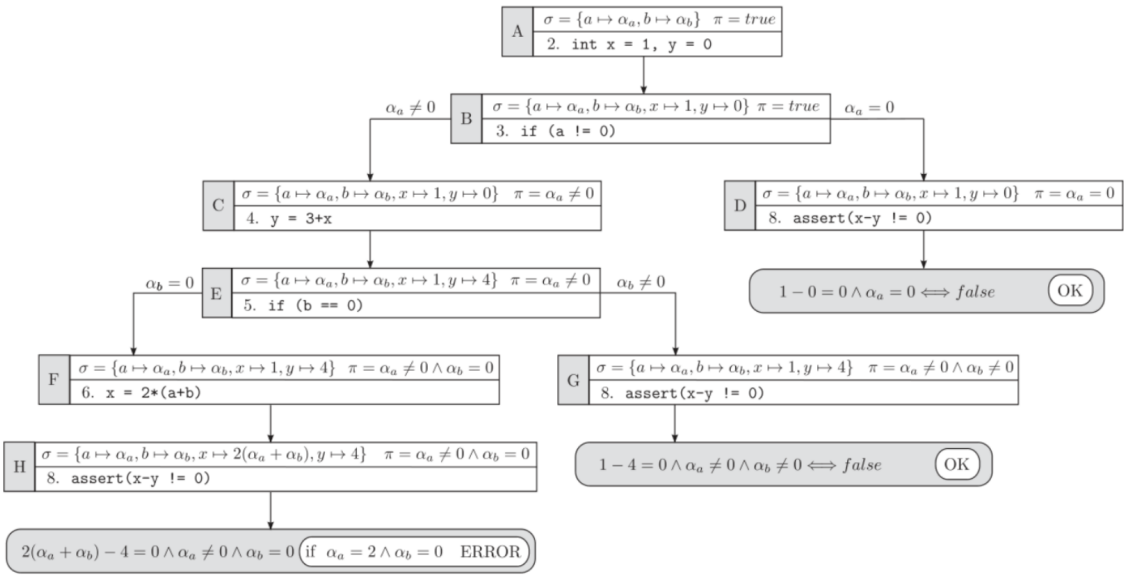

Figure 3.6: Symbolic Execution example flow, from [27]

- *environment.* A binary, and this stands in particular for malware, is not merely a set of instructions independent from everything around. Its flow of execution strictly depends from variables coming from other layers in the software stack: the creation of a file, a registry key, network packets incoming on the wire. A symbolic execution engine has to consider all these factors, to make sure they are consistent within the analysis flow.

- *path explosion.* Every time a branch is encountered, the symbolic execution engine creates a state satisfying, whenever possible, both arms. Clearly, this grows exponentially, therefore it is unlikely that an exhaustive search of all the states is feasible. For this reason, the engine has to take into account optimizations, pruning, and other techniques to keep the exploration feasible.

- *constraint solving.* SMT solvers have the capability of solving hard combinations of constraints over multiple variables. However, finely-grained non-linear arithmetic designed to be mathematically hard to solve can cause serious problems to efficiency.

The framework used in the backend of Symba, angr, provides solutions to manage all the mentioned problems, but it does not, in general, automatically implements any of them. Instead, this power is given to the user of the framework. In the following chapter, we describe some notable features of angr and how they are implemented and used.

Chapter 4

The angr framework

The framework used in this thesis, angr, is packed with interesting features that range from pure symbolic execution to static and dynamic analyses. This chapter will explore some examples of how the interface can be used by the programmer and how to harness most of the functionalities exposed by the library.

4.1 Symbolic Procedures

In angr, Symbolic Procedures are meant to completely replace functions, and they are most prevalently employed to model OS library functions. In fact, by rewriting API calls using the Python interface exposed by the framework, the symbolic execution flow can be freed from all the complexity that would rise from executing library functions symbolically. For the sake of this chapter, let us examine a simple, hello-world example that demonstrates some usage of the framework. The sample code used for it is available in the appendix at A.

The compiled binary, upon execution, returns with the following output:

```
$ ./test
This shouldn't be printed if hooked!
```

To provide a "hello world" of angr usage, consider the python script at A. It simply instruments angr to symbolically execute the binary until the end, thereafter printing the content of stdout – interface modeled and emulated by the framework – to verify that the function func is indeed symbolically called. The output of the script:

```
$ python testsimproc.py
******STDOUT CONTENT******
```

```
3 b"This shouldn't be printed if hooked!\n"
```

The output string matches what we would expect from a 'normal' execution of the binary. Let us now introduce a Symbolic Procedure into the test environment, to observe the change of behavior. The listing at A shows a new version of the previous script, where a SimProcedure is declared and hooked to the binary. The listed script demonstrates the interface used to implement Symbolic Procedures:

1. The class modeling the function must extend angr.SimProcedure, which is the abstract interface exposed by angr.
2. The run method must be implemented. It will take as argument whatever arguments the original function expects, following the calling convention of the binary. Inside the method body, the programmer can choose what to do, either concretely or symbolically, with the function args, including returning values to the callee.
3. Finally, the Symbolic Procedure has to be hooked via call to hook_symbol, replacing the symbol of the function meant to be replaced.

Let us now inspect the output of the script above:

```
1 $ python testsimproc.py
2 The function func was replaced by this function!
3 int_arg: <SAO <BV64 0xa>> / string: <SAO <BV64 0x400742>> /
    other_string <SAO <BV64 0x40073d>>
4 ******STDOUT CONTENT******
5 b''
```

As we expected, the content of stdout is empty, which clearly shows that the original func function has not been executed. On the other hand, we can observe the result of the code inside our run method, where the given arguments are printed. Later in this chapter, we will further expand this example to understand what is the meaning of those seemingly opaque <SAO <BV64>> objects, but the reader is invited to note that the first argument, the integer int_arg, is actually equal to 10 ($0xa$ in hexadecimal notation), which is indeed the value passed to that function by the main.

4.2 Symbolic Exploration

This section will dig inside the meat and bones of the symbolic execution flow, starting from the workings of state exploration in angr. To tackle this task, we will reintroduce a modified version of the example listed in the previous section, to

examine specifically which interfaces of the symbolic framework can be employed and how. The code can be found at A.

The inner bodies of func1, func2, func3 are just composed of a single statement printing the value of the first argument, therefore we won't include that in the listing. This test code introduces the true concept of symbols: in fact, the choice variable cannot have a predetermined value at the time of execution, since it is returned in line 10 as result of an atoi function. This defines the real concept of an unconstrained symbol.

4.2.1 Starting points

The symbolic analysis has to start at some *initial state* of our choice. The angr framework exposes different methods through its factory interface to create this initial state, depending on our needs. In particular, the state can be:

1. *blank_state*. It represents a mostly uninitialized state, which can be filled arbitrarily.
2. *entry_state*. It puts start of symbolic execution at the entry point block of the binary executable.
3. *call_state*. Begins at the first basic block of a function, ordering its passed arguments depending on the calling convention.
4. *full_init_state*. Mostly similar to the entry state, but first it symbolically executes all the constructors and initialization functions that usually are handled by the dynamic loader. Especially useful in C++ binaries with libraries which need to be initialized.

We can see how the mentioned interfaces work with our test binary. An interactive python console will show some information on the computed states.

The first thing will be to create the project:

```
1 In [1]: import angr
2 In [2]: proj = angr.Project("./test")
```

Starting with the blank state, we will see how registers and initial address are populated:

```
1 In [27]: blank = proj.factory.blank_state()
2 In [28]: hex(blank.addr)
3 Out[28]: '0x4006c0'
```

```
In [29]: for reg in ["rax", "rbx", "rsi", "rdi", "rcx", "rbp", "rsp"
    ]:
    ...:     print(getattr(blank.regs, reg).symbolic)
    ...:
True
True
True
True
True
True
False
In [33]: blank.regs.rsp
Out[33]: <BV64 0x7fffffffefff8>
```

From this listing, we can highlight three main points:

- The address, when not specified as argument, is the one of the entry point.
- Most of the registers are symbolic, which means that they are unitialized by the symbolic engine.
- Only the stack pointer has a concrete value.

How would this compare to a state initialized as an entry state? We can analyze this by generating one in the same way we did for the blank state:

```
In [30]: entry = proj.factory.entry_state()
In [31]: hex(entry.addr)
Out[31]: '0x4006c0'
In [32]: for reg in ["rax", "rbx", "rsi", "rdi", "rcx", "rbp", "rsp"
    ]:
    ...:     print(getattr(entry.regs, reg).symbolic)
    ...:
False
False
True
True
True
False
False
```

The scenario is somewhat different: in fact, more registers appear not symbolic. The presence of a concrete RBP suggests that, in this case, some *function prologue* has been executed which prepared the stack frame for execution. It is worth mentioning that, even though we call it *entry* state, the address of the starting basic block can be modified to start somewhere else.

4.2.2 Generating a CFG

As previously mentioned, a Control Flow Graph (CFG) conceptually depicts basic blocks as nodes and jumps, calls, rets - or other instructions modifying the IP - as edges.

Angr currently supports two ways of computing a CFG. One, called *CFGFast*, works statically, so no emulation/execution of the binary is required. For this reason, is faster, with the disadvantage of being possibly less accurate. Another - now deprecated - method is to use *CFGEmulated*, even though the library maintainers do not suggest to use, due to difficulties and missing places in the execution interface, and its very limited performance in terms of speed. For this reason, Symba employs the former to compute CFG of analyzed binaries. Let us explore this functionality in the binary used for tests:

```
In [7]: p = angr.Project("./test", load_options={'auto_load_libs':
    False})
In [8]: cfg = p.analyses.CFGFast()
In [10]: cfg.graph
Out[10]: <networkx.classes.digraph.DiGraph at 0x7f686389c0b8>
In [12]: len(cfg.graph.nodes)
Out[12]: 80
In [13]: len(cfg.graph.edges)
Out[13]: 98
```

The project has been created with a specific option unset, auto_load_libs. This is necessary, otherwise angr would try to compute the CFG of any shared library loaded by our binary, thus increasing enormously the complexity of the graph and its time of computation. For instance, one could imagine the grievousness of the task of analysing the *glibc* implementation. The graph is built using NetworkX [29], a python library used to implement complex networks with a great focus on portability and universality – nodes and edges can be anything.

In the listing below, we start exploring the capabilities of the CFG graph generated by angr, to define the functionalities that will be used by Symba to compute the starting points of the analysis. We will explore how to retrieve basic blocks' addresses, alongside their successors/predecessors nodes, specifying the *jumpkind* in the case of branches.

```
In [23]: entry_node = cfg.model.get_node(p.entry)
In [24]: entry_node
Out[24]: <CFGNode _start [42]>
In [25]: entry_node.block
Out[25]: <Block for 0x4006c0, 42 bytes>
In [26]: hex(entry_node.addr)
```

```
Out[26]: '0x4006c0'
```

The interface is clear and readable. We first get the entry_node by querying for a node matching the address of the entry point. As the output of the block attribute states, the node matches symbol _start at address $0x4006c0$ - the address of the entry point - and its size is 42 bytes. Each block exposes, among the other things, a capstone interface to disassemble instructions found at nodes. Capstone (`http://www.capstone-engine.org/`) is a *lightweight multi-platform, multi-architecture disassembly framework.*, and we can see it in action by playing a bit with the interface:

```
In [43]: p.loader.find_symbol("main")
Out[43]: <Symbol "main" in test at 0x400842>
In [44]: main_node = cfg.model.get_node(0x400842)
In [45]: print('\n'.join([repr(insn.insn) for insn in main_node.block
    .capstone.insns]))
<CsInsn 0x400842 [55]: push rbp>
<CsInsn 0x400843 [4889e5]: mov rbp, rsp>
<CsInsn 0x400846 [4883ec60]: sub rsp, 0x60>
<snip>
<CsInsn 0x400889 [e8e2fdffff]: call 0x400670>
In [46]: p.loader.describe_addr(0x400670)
Out[46]: 'PLT.printf+0x0 in test (0x670)'
```

In the listing above, we:

- Ask the project loader for the address of main symbol.
- Using that address, we retrieve the corresponding node from the CFG.
- Capstone interface is used to disassemble instructions at the basic blocks corresponding to the retrieved node.

We can observe how the basic blocks stops at a call instruction, and the loader confirms that such instruction is a printf@plt, the function which, in the test code, corresponds to printf("Choose number of arguments: ");.

There's no need to disassemble and manually query for addresses: the CFG supports methods to find *successors* and *jumpkinds* from a given node:

```
In [50]: node, jumpkind = cfg.model.get_successors_and_jumpkind(
    main_node)[0]
In [51]: node
Out[51]: <CFGNode 0x400670[6]>
In [52]: jumpkind
Out[52]: 'Ijk_Call'
```

As expected, there is only one successor node, the one calling the printf. In fact, a function call does not "branch" the execution flow in any way. Let us explore how the three subsequent ifs in the code impact the generated CFG.

The CFG allows for querying of symbols of functions listed in the binary. The interface is simple and clear. The listing below extracts the addresses of our functions *func1*, *func2* and *func3*:

```
In [130]: functions = cfg.kb.functions
In [131]: addresses = [(addr, func.name) for addr, func in functions.
    items() if func.name in ["func1", "func2", "func3"]]
In [132]: addresses
Out[132]: [(4196298, 'func1'), (4196334, 'func2'), (4196374, 'func3')
    ]
```

The code above simply builds a list of tuples with the form $(addr, name)$ through a list comprension filtered for those functions named after what we are looking for. For the sake of brevity, and since all of those functions will have a similar behavior in the CFG graph, we will only analyze one of those functions, *func2*. Before proceeding into querying the graph, let us reason on what we would expect, knowing the code.

Each function is called in a separate arm of an *if-elseif-elseif-else* structure. Therefore, reasonably, basic blocks responsible for calling functions will not have multiple predecessors. In fact, they are reachable only through one path in the code. Also, for the same reason, we expect only one successor, which is, the block executing the return statement. Is this reasoning confirmed by the graph?

```
In [154]: func2_addr = addresses[1][0]
In [155]: func2_node = cfg.get_node(func2_addr)
In [156]: if_body_node = cfg.get_predecessors(func2_node)[0]
In [157]: preds = cfg.get_predecessors(if_body_node)
In [158]: succs = cfg.get_successors(if_body_node)
In [159]: len(preds) == 1
Out[159]: True
In [160]: len(succs) == 1
Out[160]: True
```

As shown clearly in the listing, regarding both the successors and predecessors nodes of the block calling function 2, we only have one direction. As expected, by printing the instructions of the block contained in the successor node, we obtain:

```
In [161]: print('\n'.join([repr(insn.insn) for insn in if_body_node.
    block.capstone.insns]))
```

```
<CsInsn 0x4008cd [488b45c0]: mov rax, qword ptr [rbp - 0x40]>
<CsInsn 0x4008d1 [4889c6]: mov rsi, rax>
<CsInsn 0x4008d4 [bf02000000]: mov edi, 2>
<CsInsn 0x4008d9 [e810ffffff]: call 0x4007ee>
```

Following this idea, we can predict that, by querying the CFG for the predecessor node of the *if arm* body, we will get a node responsible for the cmp instruction that leads to the former. That node will have more than one successor: in fact, depending on the value being compared, the execution flow could or could not pass through that if arm. The listing below confirms this prediction.

```
In [161]: cmp_node = cfg.get_predecessors(if_body_node)[0]
In [162]: print('\n'.join([repr(insn.insn) for insn in cmp_node.block
    .capstone.insns]))
<CsInsn 0x4008c7 [837dbc02]: cmp dword ptr [rbp - 0x44], 2>
<CsInsn 0x4008cb [7513]: jne 0x4008e0>
In [163]: cfg.get_successors(cmp_node)
Out[163]: [<CFGNode main+0x8b [17]>, <CFGNode main+0x9e [6]>]
```

To conclude this section on the features of CFG, we discuss of an important interface that constitutes one of the main components of Symba: the *callgraph*. While the Control Flow Graph describes the relationships between *basic blocks* in the binary, a Callgraph will describe the relationships between functions. In this context, since all three functions are called by the main, should we query the callgraph, we would verify this fact. The CFG exposed by angr contains a suitable callgraph attribute:

```
In [170]: for func in [func1, func2, func3]:
     ...:     pred_addr = list(cfg.functions.callgraph.predecessors(
    func))[0]
     ...:     print(pred_addr == p.loader.find_symbol("main").
    rebased_addr)
     ...:
     ...:
True
True
True
```

4.2.3 Simulation Manager

In the previous chapter, the basics of states and their relationships inside various kinds of control graphs have been discussed. In order to present how Symba implements its analysis flow, one missing bit has to be introduced: the functioning

of the simulation manager, alongside its *exploration techniques*. To implement Symba, a custom exploration technique based on constraint creation has been developed and applied to the manager.

Most definitely, the SimulationManager is the most important control interface exposed by angr. This top level component allows the symbolic engine to explore the program state, progress through symbolic states, merge states together or drop others, detect loops or other structure which could overload the symbolic execution flow, all while maintaining a high degree of control from the point of view of the programmer. One could decide to keep freezed a certain state while stepping forward a certain *stash*. Stashes can be described as buckets of states which share common properties. There are a set of stashes already predefined in angr, though programmers can create custom stashes, if needed.

The default stash for the majority of operations is the active stash, where states are collected at the initialization of a new simulation manager. Other stashes will become useful once symbolic execution starts branching. This fact will become more clear once showing how the "stepping" works in angr. In particular, the stash types available are the following:

- *active*. As already mentioned, this stash contain states that can and will be stepped on the next step().
- *deadended*. This stash collects all states that, for some reason, cannot be stepped anymore. For instance, states reaching termination through an exit call.
- *unsat*. States that are already determined to be unsatisfiable are put in this stash. For example, a state which would need a certain variable to be both equal to 3 and 4 at the same time, would belong to this stash. We will, later in this chapter, show how we can place *constraints* on states using the solver backend. Requires option *save_unsat*.
- *unconstrained*. States where the RIP is controlled by the user or by another source of symbolic input. Requires option *save_unconstrained*.
- *pruned*. With a certain option - LAZY_SOLVES - enabled, angr will not check for state satisfiability at every step. Instead, it will delay this until absolutely required. In this case, if a state is found to be unsatisfiable, all its previous state, until the point it became unsat, is placed inside this stash.

We can observe this behaviour on the test binary, by initializing a simulation manager and inspecting its stashes:

```
In [5]: entry_state = p.factory.entry_state()
In [6]: simgr = p.factory.simulation_manager(entry_state)
In [7]: simgr
Out[7]: <SimulationManager with 1 active>
In [8]: simgr.active
Out[8]: [<SimState @ 0x4006c0>]
```

A simulation manager just initialized on a state - in this case the entry state - has a single stash, the active stash, populated with that state. One of the simplest operation that can be performed on a simulation manager is to run it until all the possible paths have been explored. How would the stashes look in that case?

```
In [9]: simgr.run()
Out[9]: <SimulationManager with 4 deadended>
In [10]: simgr
Out[10]: <SimulationManager with 4 deadended>
In [11]: simgr.deadended
Out[11]:
[<SimState @ 0x1000108>,
 <SimState @ 0x1000160>,
 <SimState @ 0x1000160>,
 <SimState @ 0x1000160>]
```

As expected, the simulation manager after a run call doesn't have active states anymore - if it would, there would still be paths which can be explored. Therefore, all the states are in the deadended stash, which means that they correspond to state termination.

Is it possible to explain the meaning of the four deadended states? Let us reconsider the source code of the test executable at A. Three of the states stopped at the same address, as three are the functions which can be called depending on the value given to atoi. The remaining state, instead, represents the *else* arm which goes through the exit function. This can be interestingly confirmed by accessing the history plugin of the state, which exposes descriptions of the actions performed on the state:

```
In [28]: list(s.history.descriptions)
Out[28]:
 <snip>
 '<SimProcedure atoi from 0x1000100: 1 sat>',
 '<IRSB from 0x4008b2: 2 sat>',
 '<IRSB from 0x4008c7: 2 sat>',
 '<IRSB from 0x4008e0: 2 sat>',
 '<IRSB from 0x4008fd: 1 sat>',
 '<IRSB from 0x4006a0: 1 sat>',
```

```
10 '<SimProcedure exit from 0x1000108: 1 sat>']
```

In fact, the last action performed on the state is verified to be a SimProcedure implementing the exit function.

Programmers can enforce a higher control of degree over the process of execution through an important interface of the simulation manager: the *exploration techniques*. Exploration techniques direct most of the operations that a simulation manager performs on its attributes while stepping through states: which stashes to use, which states go in which stash, when to stop, or which functions should be called on states during execution if certain conditions are met. Some notable examples of built-in exploration techniques exposed by angr:

- *Depth First Search*. Just one state is chosen to be executed and stepped forward. The others are kept in a *deferred* stash until the former terminates.
- *Explorer*. Allows exploration of program states, guided by a set of find and avoid rules. Will dig deeper into this.
- *MemoryWatcher*. Under certain conditions, the quantity of system memory that angr uses can grow uncontrollably. This exploration technique puts a cap into that, ensuring that memory usage will not get past that threshold. This concept is already explored in [30]
- *LoopSeer*. For reasons linked to path explosion, *for loops* can hinder symbolic execution up to the extent of impracticality . LoopSeer detects states which go through too many loops, dumping them into a *spinning* stash until no other active state is available. Other techniques, like the one presented in [31], propose an extension of symbolic execution tailored to handle loops.
- *Veritesting*. An implementation of the Veritesting technique introduced in [32] and mentioned in [33] to find *merge points* automatically and increase symbolic execution performances.

For the sake of understanding, we can apply the *Explorer* technique to the example we are working on. The Explorer instructs angr to continue exploring until some condition has been met. The condition can be represented, simply, by an address - *run until a block containing this address is executed* - or by providing a function returning a boolean, transforming the condition into a *run until function returns true*.

We can observe both behaviors in the test binary. Let us suppose that an analyst is given it, while not provided with the source code, so reverse engineering has to be performed on the executable. The analyst identifies the three func functions, and he identifies as well that an *atoi* function is used to transform the value given

fed to standard input into an integer. The Explorer technique can help the analyst knowing *which value has to return atoi in order to reach a certain function.* Let us start with func1, thereafter using the same method to solve with respect to the other functions.

```
In [8]: func2 = p.loader.find_symbol("func2").rebased_addr
In [9]: hex(func2)
Out[9]: '0x4007ee'
```

Once computed the address of the function, this can be used as it is for the Explorer technique:

```
In [10]: entry_state = p.factory.entry_state()
In [11]: simgr = p.factory.simulation_manager(entry_state)
In [12]: simgr.explore(find=func2)
Out[12]: <SimulationManager with 3 active, 1 found>
```

As shown in the listing above, the simulation manager, armed with the Explorer technique, has created a new stash called *found*. This stash contains states that matched the condition configured into Explorer, which, in this case, is the reach of a precise address. Explorer stops here at the first match, but we could continue running it, gathering all the states matching the rule into that stash.

What we do expect now, therefore, is to have, into the state in found stash, the path needed to reach func2. First of all, let us verify the address:

```
In [14]: found_state = simgr.found[0]
In [15]: hex(found_state.addr)
Out[15]: '0x4007ee'
```

The address of the found state is, indeed, the address of func2. Concerning how the analyst can retrieve the value passed into atoi to answer his starting question, there are various ways. The easiest one is a small workaround based on the state posix plugin. In fact, we know that the value ultimately passed to atoi is fed to the process via stdin. The posix state plugins allows fast solving of opened files:

```
In [19]: found_state.posix.dumps(0)
Out[19]: b'02\xf7\xc7\x00\x10\xbc3\xec\x00\x19PB\x00\x00\x00\x00\x00\
    x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

The returned *bytestring* contains the answer in its first bytes: in fact, the value 02 is exactly the one needed to reach our target function. The other bytes are

just solved by default and, in this case, they're not constrained to anything. We can verify this by wrapping the code into a function and calling it for all target functions:

```
In [22]: def find_atoi_value(find_address):
    ...:     simgr = p.factory.simulation_manager(entry_state)
    ...:     simgr.explore(find=find_address)
    ...:     found_state = simgr.found[0]
    ...:     print(found_state.posix.dumps(0)[:3])
    ...:
In [23]: find_atoi_value(p.loader.find_symbol("func1").rebased_addr)
b'1\xacq'
In [24]: find_atoi_value(p.loader.find_symbol("func3").rebased_addr)
b'3|\xe0'
```

The results show that the number in the solved stdin string matches what we would expect. It is worth considering that we are interested, in this cases, only to *numbers until the first non-digit letter*. This stems from the way atoi works, in particular, without detecting errors in its input, but just converting as long as read letters are digits.

What about the else arm exiting without calling functions? To solve for a value reaching that code, another useful matching rule exposed by Explorer can be used: the avoid parameter. The latter works the opposite with respect to find - it directs exploration avoiding addresses or boolean functions passed to the exploration technique. The solution to our previous question becomes:

```
In [32]: to_avoid = [p.loader.find_symbol(func).rebased_addr for func
    in ["func1", "func2", "func3"]]
In [33]: simgr.explore(avoid=to_avoid)
Out[33]: <SimulationManager with 1 deadended, 3 avoid>
In [34]: to_avoid = [p.loader.find_symbol(func).rebased_addr for func
    in ["func1", "func2", "func3"]]
In [35]: simgr = p.factory.simulation_manager(entry_state)
In [36]: simgr.explore(avoid=to_avoid)
Out[36]: <SimulationManager with 1 deadended, 3 avoid>
In [37]: exited_state = simgr.deadended[0]
In [38]: list(exited_state.history.actions)[-1]
Out[38]: <SimActionData exit() reg/read>
In [39]: exited_state.posix.dumps(0)[:3]
Out[39]: b'\xf5\xd5\xf5'
```

A few notable highlights from the listing above:

- In this case, since no find parameter was passed, there was no active stash.

However, since we were looking for an exited state, the deadended stash became our target.

- A new, avoid stash appeared. As expected, it contains three states, related to the three functions avoided.
- The dumped string doesn't contain, at its first bytes, any digit that can be converted. It is a valid solution - we will discuss later about constraints and the Z3 solver - to reach the exit arm.

The approach used to retrieve the correct value has a few drawbacks. First of all, without having specific knowledge on the entire program flow, finding the correct bytes from the entire stdin symbolic file content would be almost impossible. Moreover, it does not employ at all the power of the angr interface in creating symbols and solving them. Considering this, in the following, we will implement a more robust way to solve for values of interest, a core functionality that Symbolic Execution frameworks expose.

A more precise approach to discover the input needed for a target function starts from the assembly of the executable. The idea is to know, precisely, *where* the fgets function will place the value that will be then passed as input to atoi. In many modern architectures, local variables are referenced through an offset from the register RBP. In this case, by quickly inspecting the disassembly of the main function, we can determine that the offset is $0x30$. Therefore, the buffer which atoi will read our input from will be placed at address rbp – 0x30. Moreover, the address where the atoi function is called is $0x4008ad$. The following listing will show how a symbol of a suitable length is created and injected into memory at that address, for later evaluation at solve time:

```
In [183]: state = p.factory.entry_state(addr=0x4008ad)
In [184]: state.regs.rbp = state.regs.rsp + 0x100
In [185]: state.regs.rdi = state.regs.rbp - 0x30
In [186]: state.memory.store(state.regs.rbp - 0x30, fgets_symbol, 4)
In [187]: simgr = p.factory.simulation_manager(state)
In [188]: simgr.explore(find=p.loader.find_symbol("func2").
    rebased_addr)
Out[188]: <SimulationManager with 3 active, 1 found>
In [189]: sol_state = simgr.found[0]
In [190]: sol_state.se.eval(fgets_symbol, cast_to=bytes)
Out[190]: b'2\x80 \xdf'
```

Let us consider the listing above, to explain how the solution has been found employing the power of symbolic execution. The flow of the code can be explained as follows:

1. A state is created, in line 1, exactly at the address of the call atoi instruction.

2. A stack frame has now to be created. In fact, when creating the state as an entry_state, RSP is initialized with a concrete value, but RBP is not. In order to create a suitable frame, in this case, it is enough to place RBP at 256 bytes above RSP (line 2).

3. As previously mentioned, fgets will use as buffer the address at RBP - 0x30. Therefore, since the same address has to be fed into atoi, we set, in line 3, RDI - the register where the first argument is placed according to the calling convention - equal to that address.

4. Now that the prologue has been set, it is time to actually inject the symbol. In line 4, the previously created fgets_symbol is stored at address RBP - 0x30, with a len of 4 bytes - more than enough in this case.

5. A simulation manager is created from the setup state, and it is executed with the Explorer technique until func2 is called.

6. On the found state, the solver is called to evaluate the concrete solution of the injected symbol. The obtained value is, as expected, 2.

An interesting feature, as indicated formerly, is the capability to use as matching condition a boolean function, to possibly implement constraint more complex than simply touching a given address. Let us see how we can use this, together with the shown solving code, to abstract the discovery of concrete values for the three functions.

For the task, we will employ, once again, the posix state plugin, which wraps many file-system related operations. In fact, every function, in its code, prints a string formatted like: "Function \%d here.\n". This behaviour can be wrapped into a boolean function to find states of interest:

```
1  In [205]: def solve_for_func(n):
2       ...:     def is_func(state):
3       ...:         return True if "Function {} here".format(n).encode
       () in state.posix.dumps(1) else False
4       ...:     state = p.factory.entry_state(addr=0x4008ad)
5       ...:     state.regs.rbp = state.regs.rsp + 0x100
6       ...:     state.regs.rdi = state.regs.rbp - 0x30
7       ...:     fgets_symbol = state.solver.BVS("fgets_input", 4*8)
8       ...:     state.memory.store(state.regs.rbp - 0x30, fgets_symbol,
        4)
9       ...:     simgr = p.factory.simulation_manager(state)
10      ...:     simgr.explore(find=is_func)
```

```
...:     found = simgr.found[0]
...:     return found.solver.eval(fgets_symbol, cast_to=bytes)
...:
```

The only change in the listing above stands in the introduced is_func that checks for the content in the symbolic stdout. If the seeked string is found, the Explorer stops and yields the found state that is later used to solve the symbol. By launching the function with the different integers, we obtain the desired result:

```
In [217]: for n in [1,2,3]:
     ...:     print("A suitable value to feed into fgets to reach
    func{} is: {}".format(n, solve_for_func(n)))
     ...:
A suitable value to feed into fgets to reach func1 is: b'1\x974O'
A suitable value to feed into fgets to reach func2 is: b'2$\xc60'
A suitable value to feed into fgets to reach func3 is: b'3?\xcc\xe7'
```

4.3 Symbolic States

In this section we talk about angr symbolic state plugins, some example of how it is used, and how it is possible to inject and track variable inside symbolic memory.

When performing executions in angr, most of the operations work on specific objects each one "crystallizing" a precise *simulated program state*. We will refer to this object with the name *SimState*.

SimState exposes an interface to interact with the program's memory, its filesystem data, registers – that is, most of the live data that basically constitute a process. In the interest of detailing further this concept, since fundamental to the structure of Symba, let us reconsider the SimProcedure example in chapter 4. When printing to screen values of the arguments that the procedure FuncReplace receives, the output was:

```
int_arg: <SAO <BV64 0xa>> / string: <SAO <BV64 0x400742>> /
    other_string <SAO <BV64 0x40073d>>
```

Arguably, the best way to use angr and explore its capabilities is to use an interactive python interpreter which supports help and tab completion. For this task, I will employ IPython. In this specific case, a breakpoint - represented by the IPython function embed - will be placed right after the print statement in the SimProcedure code. By calling functions in the IPython environment, we are able to interactively explore how angr manages SimStates and memory values.

In a SimProcedure, the SimState can be accessed through the attribute self.state:

```
In [1]: self.state
Out[1]: <SimState @ 0x40063a>
```

The string representation of the state shows the current address of execution. Value of CPU registers can be easily accessed:

```
In [14]: regs = self.state.regs
In [15]: regs.rdi
Out[15]: <SAO <BV64 0xa>>
In [16]: regs.rsi
Out[16]: <SAO <BV64 0x400742>>
In [17]: regs.rdx
Out[17]: <SAO <BV64 0x40073d>>
```

Under x86_64 architecture and *System V* ABI, the first three parameters to functions are passed through the registers RDI, RSI, RDX, in this particular order. In the listing above, we see how the arguments to functions printed in A match those contained in registers. Let us examine furthermore how to interact with concrete values inside those registers. Considering the value of the arguments passed to the function:

```
func(10, "ARG1", "ARG2");
```

A BitVector is essentially the representation of an integer in angr. For this section, it will be enough to state how to transform a BV - symbolic or not - into a real value – in other words, to solve a bitvector:

```
In [54]: rdi
Out[54]: <SAO <BV64 0xa>>
In [55]: state.solver.eval(rdi)
Out[55]: 10
```

In the same way, we can extract the address of the second argument:

```
In [59]: rsi
Out[59]: <SAO <BV64 0x400742>>
In [60]: hex(state.solver.eval(rsi))
Out[60]: '0x400742'
```

Memory can be accessed through the interface state.memory. So, supposing we are to extract the first byte contained at the address in rsi, the code would be:

```
In [63]: addr = state.solver.eval(rsi)
In [64]: state.mem[addr].byte.resolved
Out[64]: <BV8 65>
In [65]: chr(state.solver.eval(state.mem[addr].byte.resolved))
Out[65]: 'A'
```

Abstracting the code above to print strings at given addresses, we obtain the following:

```
In [73]: def print_string(state, addr):
   ...:     result = ""
   ...:     while True:
   ...:         c = chr(state.solver.eval(state.mem[addr].byte.
   resolved))
   ...:         if c is not '\x00':
   ...:             result += c
   ...:         else:
   ...:             break
   ...:         addr += 1
   ...:     return result
   ...:
In [74]: rsi_addr = self.state.solver.eval(rsi)
In [75]: rdx_addr = self.state.solver.eval(rdx)
In [76]: print_string(self.state, rsi_addr)
Out[76]: 'ARG1'
In [77]: print_string(self.state, rdx_addr)
Out[77]: 'ARG2'
```

Indeed, the resulting strings match exactly what is passed to the function in the test code. In line 14, the value 'ARG1', passed as the first argument, is correctly retrieved from the address stored in rsi. Then, the same happens with the second argument, resulting in 'ARG2'. This example is easy to grasp, since all the variables involved are already determined and "hard-coded".

Chapter 5

Architecture design

The design of symba is greatly focused on the aforementioned concept of trigger conditions. The generic flow of the analysis is as follows:

1. The analyst chooses Win32 API function calls that are deemed as interesting. By interesting, here, we mean function calls for which the return value can unlock or block interesting routines - that is, functions that model ‘trigger conditions‘. As we will see, this choice is easily handled by a script which generates the correct configuration.

2. Symba "hooks" these function calls in the binary, basically replacing them with functions automatically generated by the system that return a symbolic, tainted value to the caller. We will see how these tainted value are generated and tracked during the execution.

3. The system then builds a Control Flow Graph of the executable, extracting basic blocks in which the hooked functions will be called. Using the CFG, it builds a list of basic blocks preceding the extracted ones. It will be, substantially, a list of "entry points" basic blocks.

4. Starting from each one of those entry points, Symba symbolically executes the binary, tracking any kind of constraint posed on tracked values. Tracked values could be memory addresses returned by the configured ‘trigger‘ functions or other memory addresses which are directly influenced by those. The mechanism of this tracking will be explained in detail in a following section.

5. Using a custom exploration strategy with some predetermined stopping criteria, symbolic states are collected and stored in a specific stash, until the analysis is stopped - because of termination by the executable, or one of the stopping criteria being met.

6. For each taken path, Symba then collects all constraints on tainted values, solving them using the underlining SMT solver. The computed solutions are then stored into a structured file for further processing - e.g examined by a human analyst or fed into a Cuckoo configuration process for automated analysis.

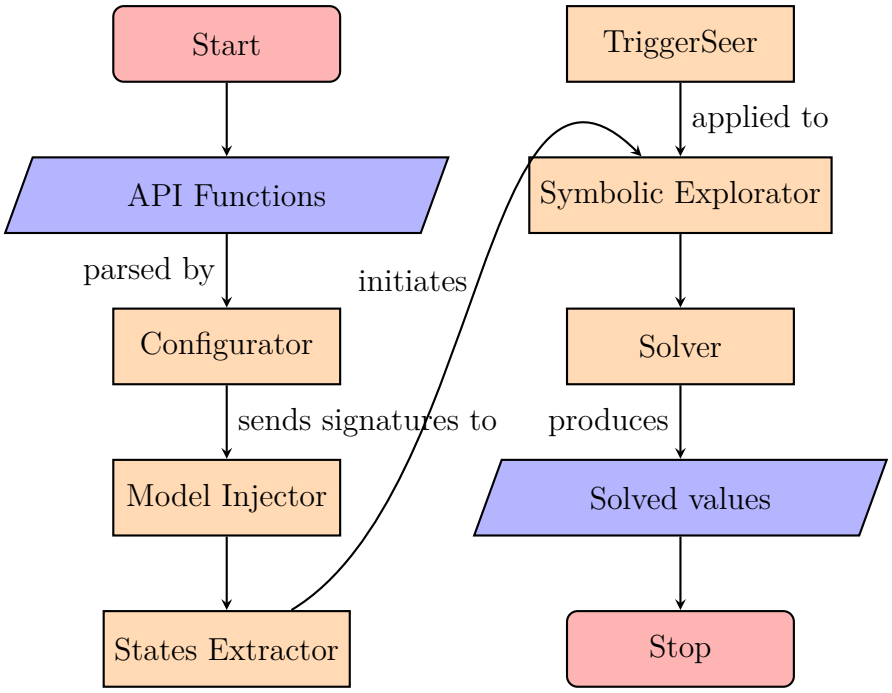

Figure 5.1: Symba flow diagram.

Basically, the main rationale of this system is that malwares will interact with the execution environment through the Win32 API. Results from these API calls will be then probably compared and checked using simple or complex routines, and the results will direct the execution flow towards one branch or the other. By collecting and solving these chain of constraints each time a branch is detected for a tainted value, the system is capable to generate values satisfying one or the other branch arm, thus possibly unlocking new guarded paths in the malware code.

Symba is implemented in a modular fashion, so that most of the features described can be extended and augmented without necessarily touching other parts of the code.

In this chapter, the design of each module will be detailed. The flow diagram of the interaction between components is described in figure 5.1. Specifically, components are:

- *Configurator*.

- *Model Injector.*
- *States Extractor.*
- *TriggerSeer.*
- *Symbolic Explorator.*
- *Solver.*

In the rest of this chapter, each component design will be discussed in detail, highlighting its process inside the analysis flow.

5.1 Configurator

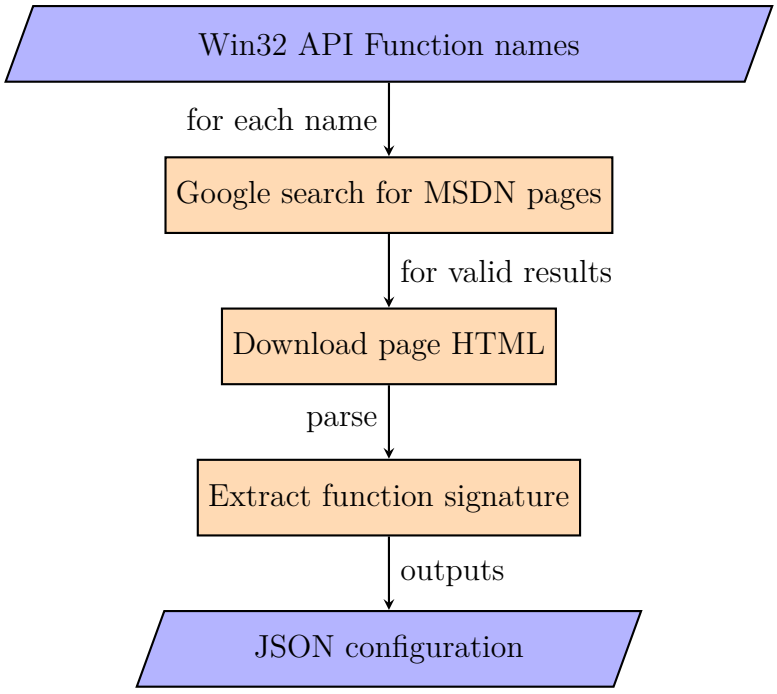

Figure 5.2: Configurator flow diagram.

The configurator is charged with the task of transforming Windows API function names into the format used by Symba. Basically, for each name it is necessary to extract the function signature and build a JSON file with all the relevant fields in a default value, ready to be tweaked by the user. The documentation used to lookup function names is the Microsoft Developer Network, considered to be the official documentation for the Windows API. Since the precise URL depends on

the header file, and since the MSDN does not offer any public API, in order to retrieve documentation from the MSDN a couple of steps are required.

First of all, the name of the function is searched through Google with the `site` filtering keyword pointing to the MSDN site. This way, we will only retrieve pages in the MSDN with the function name contained in the URL. Afterwards, the query results are filtered to make sure the URL looks like an API signature documentation, and if so, the page is downloaded. Since we download the whole page HTML, another parsing step is required to extract exactly the information needed to build the signatures. At last, a JSON file is built with the results. The choice of JSON as the configuration format greatly improves the readability of the file. Moreover, it ensures that the analyst using Symba can easily tweak and customize it, by setting the length of a certain symbol, the name of some parameters, or to decide which one of them should be injectable. Also, there's another reason for having configuration files stating which functions should be traced as trigger sources. The idea is to create multiple JSON files, each one focused on a certain set of trigger sources, that can quickly be switched during analysis. One could have a `lightweigth.json` file to follow only a few functions that are generally deemed as interesting, like API calls that query registry keys. Then, another file which could be named `full.json` will be loaded of more functions, in order to reveal as many conditions as possible from the binaries exhamined. The flow diagram discussed is represented in figure 5.2.

5.2 Model Injector

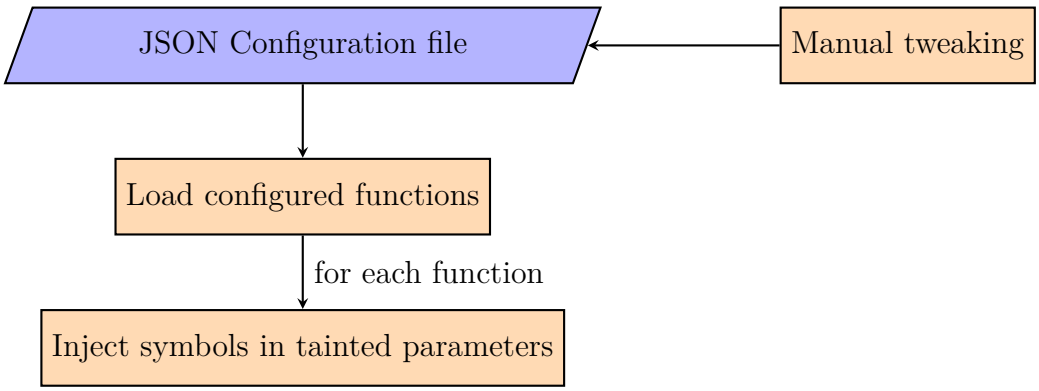

Figure 5.3: Model Injector flow diagram.

The model injector manages the technique that Symba uses to model and represent trigger sources. Basically, functions have to be replaced with a symbolic equivalent counterpart. At the beginning of the flow, the JSON configuration file is loaded and function signatures and options are stored inside some data structure.

Once this data is available, Symba generates a class that matches everything in the signature of the API function: parameters name, type and calling convention. The real difference between the two functions stands in what happens in the function body. Whereas the original API function would have computed the value requested or queried the resource pointed by the executable, the only thing that a Model will do is to select – depending on the configuration file – some specific parameters, lookup their memory address, and inject there a suitable symbol. This will constitute the basics of the tainting process. For each injected symbol, a unique key built using the configuration file name, the function name and the parameter name is stored inside a global repository of keys. From that point on, every time that the executable will try to access in read or write, in a branch condition, the data that has been injected, this will be reflected in the constraints of the symbol. This allows other components that need to monitor the list of constraints associated to a particular symbol, like the TriggerSeer or the Solver, to properly work.

Having a component that handles automatic creation of trigger sources model from the configuration – also automatically generated – is a valuable addition to the project. In fact, without this component, the analyst should manually develop code to represent each modeled function, probably incurring in errors or redundancy. Moreover, it is entirely possible to customize the generation of the model depending on the selected function. The JSON configuration file that is given as input to the Injector, in fact, could be extended to add new options that further personalize different trigger sources. One notable addition to this would be a constraints option. This option would allow analysts to use a simple grammar to define simple constraints on the function parameters. Let us consider, for instance, the GetUserName function. Some analyst could decide that the malware will only look for usernames that start with the letter 'B'. So, a constraint on the lpBuffer parameter that stores the symbol for the generated username could be placed to constrainv the first letter to be equal to 'B'.

5.3 States Extractor

When dealing with symbolic execution in angr, the developer can choose the symbolic state where the emulation has to begin. A naive approach would certainly be to set the initial state at the entrypoint of the binary. This solution, while theoretically feasible, is definitely impractical. In fact, each one of the already mentioned weaknesses of symbolic execution would escalate in such a scenario. For such reasons, it is necessary to design a better way to extract initial states.

The way that Symba handles this starts from the Control Flow Graph of the binary. The CFG contains a few informations on called functions, the data flow graph, the callgraph, from which we can extract dependencies and relationships

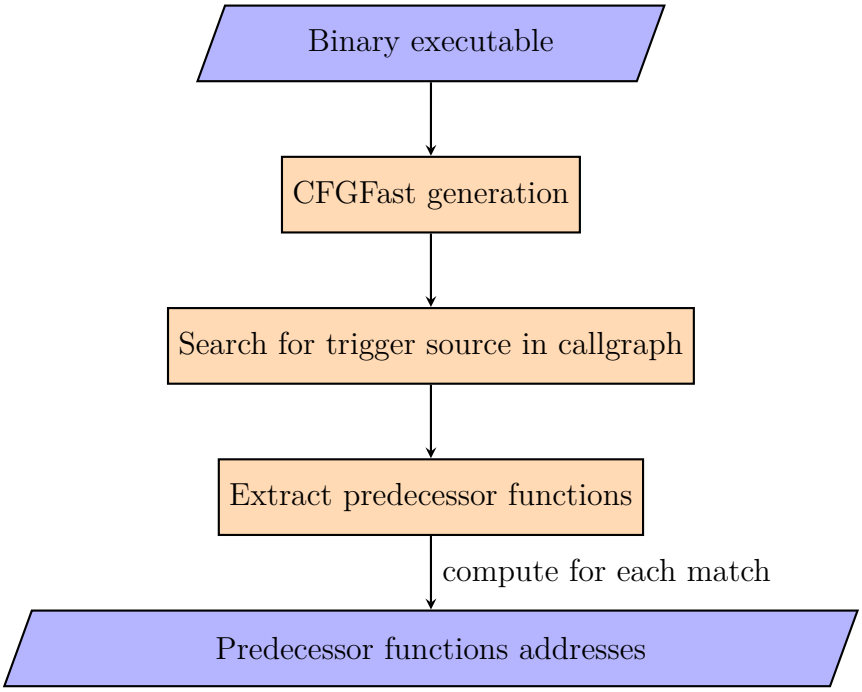

Figure 5.4: States Extractor flow diagram.

between function calls. In particular, from the callgraph, we detect functions in the executable where one or many of the configured trigger sources are called. For each match, we take the predecessor function, that is, the function calling the former function, and we extract the address of this function. This address will be then transformed into a block and that block will be passed as the initializer of a symbolic state.

Note that one advantage of this approach is that the symbolic execution flow will always start from the beginning of a function, in a prologue. This ensures the consistency of variables on stack, with no need for manual intervention and initialization of those fields.

5.4 TriggerSeer

The TriggerSeer is an important component in the execution of Symba. The previous section documented the workflow of the State Extractor, which decides where the symbolic execution will start. The TriggerSeer handles the opposite aspect, precisely the termination condition of the execution. The reason for which Symba needs this hides behind the same reasons already mentioned about the weaknesses of symbolic execution. The main idea of the TriggerSeer is to continue

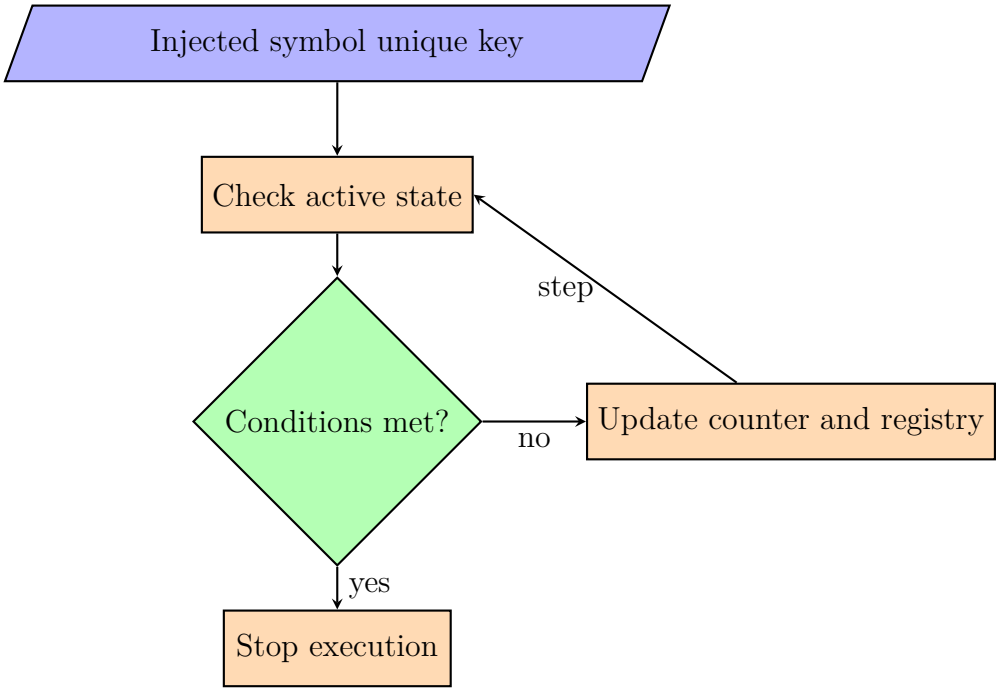

Figure 5.5: TriggerSeer flow diagram.

executing as long as the program is interacting with the tainted symbols. At some point, after a reasonable number of states where the symbol is not fetched in neither read or write, the execution is deemed as complete.

TriggerSeer works with two principal data structures: the *counter* and the *registry*. It is, at its core, an object defined in angr as an Exploration Technique. These objects receive, for each step, the active states, and they decide what to do with those states, and whether the execution is complete. In this case, the operation performed on state is to check, for each active state, the constraints posed on the injected symbol for the trigger source. At the very first step, the registry is initialized with these constraints. In the following step, the new constraints are checked: if anyone new is found, the registry is updated and the counter is set to zero, since the injected symbol has been fetched. Otherwise, the counter is incremented. As soon as the counter passes a certain configured threshold, the execution is completed.

An extension, needed for the correct execution of the exploration technique, has been applied. In particular, since for the reasons previously mentioned, the execution starts from the function which actually calls the trigger source function, for the initial states new constraints on the injected symbol cannot literally be detected, since the symbol is not there yet. Therefore, TriggerSeer actually starts setting the counter only after the unique key bound to the injected symbol has

been inserted in the global state plugin, that is, after the symbolic model of the tracked function has been executed.

5.5 Symbolic Explorator

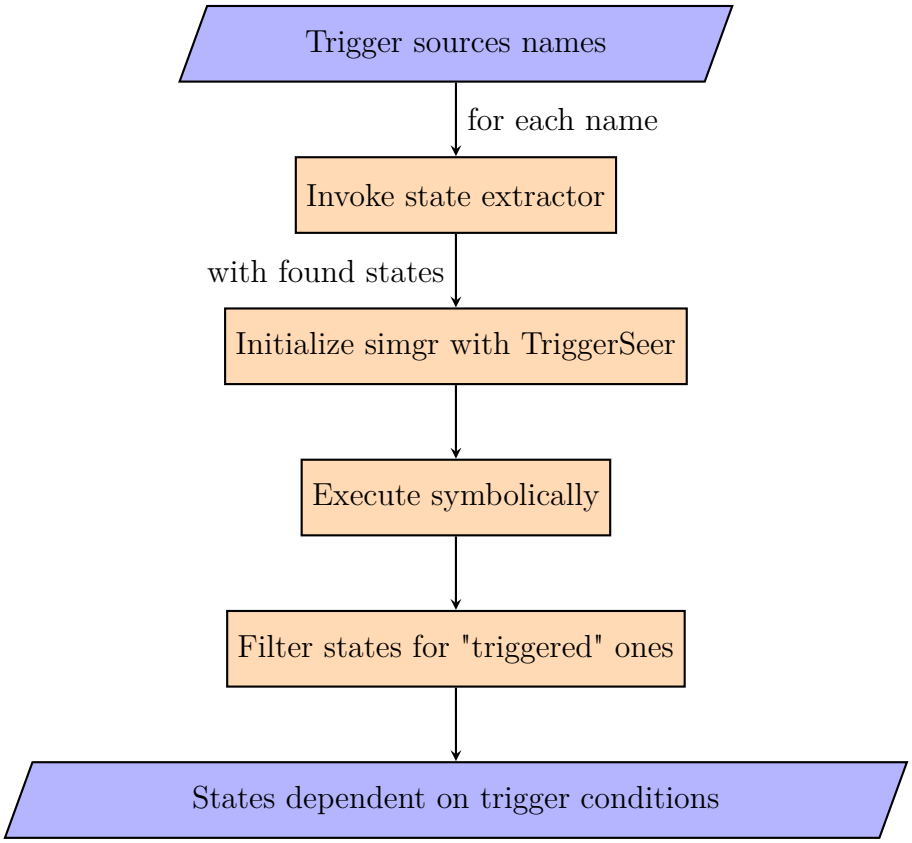

Figure 5.6: Symbolic Explorator flow diagram.

The Symbolic Explorator is essentially the glue code that connects the other components together. It collects the command-line parameters, it builds the configuration from where it will extract the trigger source names. For each trigger source name, it invokes the State Extractor to retrieve starting states for the execution. With those states, it will initialize an execution manager to which it will apply the TriggerSeer exploration technique with the desired threshold. As soon as the execution is complete – with the termination conditions previously described – the explorator retrieves all the states collected during the execution. At this point, it will filter each state, looking only for the "triggered" ones, which is, states that are somehow constrained on the trigger sources. Each one of those

states is returned by the executor, to be afterwards fed to the Solver.

Maintaining this modularity between the different pieces improves code readability, and more important, it makes easier to improve and extend the tool. For instance, if anyone developed a new exploration technique for Symba, it would be enough to replace the line where TriggerSeer is applied with the constructor of the new exploration technique. The same goes with all of the other components.

5.6 Solver

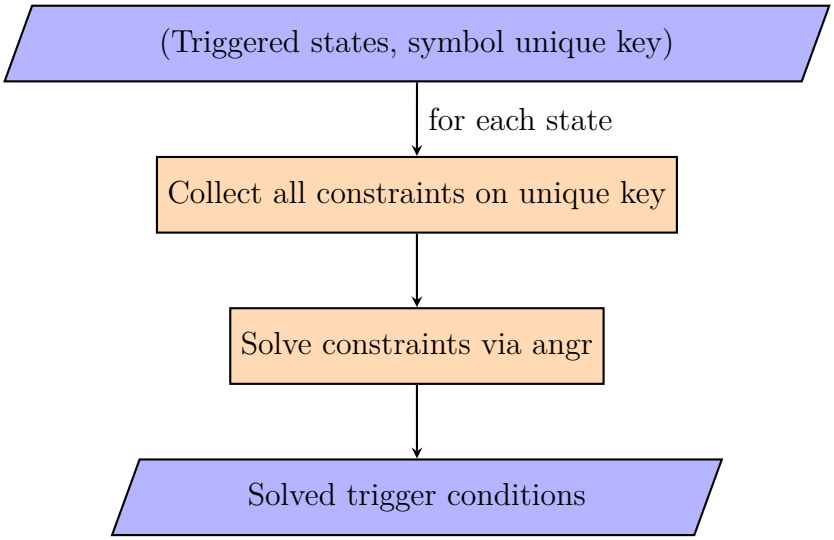

Figure 5.7: Solver flow diagram.

The last component described is also the final component in the workflow of Symba: the Solver. Its task is to take all the triggered states which are generated from the Symbolic Explorator, and solve them, producing concrete values. For this task, it uses an SMT solver, Microsoft Z3, as a backend. The solver is capable of handling both linear and not linear constraints, and it can produce raw bytes or compat numbers as a result. Constraint are identified via the unique key that has been injected by the Model Injector when the trigger source has been replaced. The same key will identify the results in the output log. The key is readable and it contains the name of the configuration file, the name of the trigger source function, and the parameter injected.

In some cases, where the symbol does not have a precise structure, the solver has to produce raw bytes that have to be manually inspected by the analyst. However, for precisely defined *structs*, custom output function can be implemented to produce a more readable result. This will be demonstrated in the evaluation chapter, where

the GetSystemTime results will be formatted to match a Date object.

Chapter 6

Implementation details

6.1 GenericModel

By leveraging the SimProcedure interface, Symba is capable to dynamically, automatically generate new models on the fly - starting from a given configuration - without the need for the analyst to implement them from scratch, a time-consuming and repetitive task.

Specifically, Symba contains a *GenericModel* class that can model the majority of trigger sources one could be interested in tracking. The implementation can be found in the appendix at A.

A GenericModel abstracts the creation of symbolic procedures by exposing an interface (via __init__) where the real difference stands in the function signatures - named fsig in the code. In run method, the unpacking syntax *args is used to represent a variable list of parameters - since different API functions will have a different number of parameters - while the field fsig.params provides the method body with a precise list of parameters, together with its name and type. The rest of the function will be explored further, where the mechanism of symbolic constraints injection will be handled.

Summing up, the GenericModel symbolic procedures constitutes an "abstract" interface for configured trigger sources, lifting the analyst from having to implement the symbolic replacement for API functions on a one-by-one basis.

6.2 SymbaConfig

As previously mentioned, the starting point of the config generation is a list of function names. In this case, it would be:

```
$ cat api_file.txt
GetSystemTime
GetDiskFreeSpaceEx
```

The relevant code used to query Google and the MSDN and parse through BeautifulSoup can be found in the appendix at A.

The workflow of the script follows this path:

1. A file - provided by the user through a command line argument - is opened and its lines are parsed.

2. A non-standard python library - googlesearch is used to automate a Google search. The query employs a Google dork search filter, site, to filter for URLs pointing to the MSDN documentation. Among these, up to two results are selected, and only where the URL contains reference to the Win32 API and the function name, those are selected for parsing.

3. Selected URLs are requested and downloaded using the requests library.

4. The content of the webpages is parsed through BeautifulSoup, a notorious HTML parsing python library, and the relevant parts of the documentation - those pertaining the signature - are saved into a dictionary.

5. The resulting dictionary, shaped into the configuration format, are written to a file whose name is specified through another command line argument.

At the end, the config file shown in the appendix at A is obtained.

To conclude the discussion of how Symba handles configuration of trigger sources, the rest of this section describes how the generate configuration file is parsed. After all, the main target of this configuration process is to "fill" the system with models - generated through the mechanism described in the previous section on Symbolic procedures. The relevant code follows:

```
class SymbaConfig(object):

    def __init__(self, config_file):
        self.config_file = config_file
        self.signatures = []
        self.models = []
        self._parse_json()

        for sig in self.signatures:
            self.models.append(GenericModel(sig, self.config_file))

    def _parse_json(self):
```

```
13          with open(self.config_file, 'r') as f:
14              d = json.load(f)
15          for func in d['functions']:
16              params = [param(**p) for p in func['params']]
17              self.signatures.append(sig(func['name'], params))
```

The logic is clear: the JSON is parsed in method _parse_json - line 20 - and the signatures are extracted. Thereafter, a list of models is filled with various instances of GenericModel, each one representing a different Trigger Source. This data structure will be then available to the main routine of the analysis code, as shown in:

```
1      def _init_configuration(self):
2          for config in self._config_paths:
3              for model in SymbaConfig(config).models:
4                  self.triggers.append(TriggerSource(model))
```

6.3 CFGFast

First of all, we should examine why the CFG is necessary at all in the process of extracting trigger conditions from the associated sources. In fact, a naive solution would consist in symbolically executing the executable starting from main, thus touching all the trigger sources functions. This solution, despite being formally correct, has the big drawback of impracticality: as discussed in the introductory chapter, symbolic execution brings some disadvantages to the table, exposing a few weak points to binaries, especially in the absence of source code - in fact, while angr works at the binary level, other notable symbolic execution engines [34] need the source code to provide more reliability to path explosion or divergence. For all these reasons, an approach *set and forget* where the executable starts from main thru its entire codebase will most likely end up being unsuccessful.

For this reason, Symba employs a different approach to start its symbolic execution. The idea is based upon finding references to the functions defined as trigger sources in the CFG and beginning symbolic execution *just before that*. Specifically, the algorithm works as follows:

1. A CFGFast is computed over the binary.
2. The functions referenced in the CFG are searched for configured trigger sources.
3. For every matching function, a list of predecessor functions is computed from the callgraph.

4. The list such constituted is returned with the name of *call_points*, used later to compute starting addresses.

It should be noted that Symba is not looking for predecessor basic blocks, but functions. In fact, starting symbolic execution from random points inside routine code will probably lead to inconsistencies in local variables or stack, as already demonstrated formerly in this section.

The relevant code is shown below:

```
try:
            if not self.cfg:
                self.cfg: CFG = self.project.analyses.CFG(**
    cfg_options)
            for address, function in self.cfg.functions.items():
                self.l.debug((address, function.name))
                try:
                    if function.name in symbols or not symbols:
                        self.l.info(f"Intercepted call to {function.
    name}")
                        pred = next(
                            iter(
                                self.cfg.functions.callgraph.
    predecessors(
address)))
                        call_points[function.name] = pred
                except StopIteration:
                    pass
        except Exception as e:
            self.l.log(logging.ERROR, f"{e}")
        return call_points
```

Another notable mention in the code above is that CFG is persisted over different executions of Symba, so to avoid having to recompute it each time losing time. In this scenario, it is equivalent to a *Singleton*. Moreover, other than used to find calling points, this function serves another purpose: it actually searches for trigger sources in the binary. In fact, should call_points be empty, the analysis will not start, thus lifting the system from losing computational time over executables where seeked functions are not called anywhere.

6.4 Exploration Techniques

First of all, we have to reason on what the exploration technique must obtain on the execution flow. For each TriggerSource, the simulation manager begins execution at its predecessor function, as shown previously. The reason for having a custom

exploration technique is that, once starting, we could not possibly continue until exhaustion of the program state, for reasons highlighted before. Therefore, using a blank run() is not suggested. Moreover, we do not possess any clear rule to apply to the Explorer technique.

The crucial feature of the needed exploration technique is therefore to know when the simulation manager has to return. A suitable *stopping condition*, or *termination criterion*, must ensure that, once symbolic execution has began tracking a certain trigger source, it collects all relevant constraints placed on its produced value to unravel new paths. So, TriggerSeer models exactly this behavior. The general flow is described below:

1. A *registry* is initialized when the exploration technique is applied first to the simulation manager. It will mantain already registered constraints on the trigger source values.

2. Each time the simulation manager steps forward a state, TriggerSeer checks all the constraints connected to the registered trigger source. If any of them is not in the registry yet, the state is marked as *recently constrained*.

3. If, in a step, no state in the active stash has been recently constrained, a *counter* is incremented. Otherwise, the counter is assigned zero.

4. When the counter passes a certain *threshold* value, TriggerSeer stops execution.

The rationale of TriggerSeer is clear: the idea is that, when producing a value with an API function call, its resulting value will be probably used by its consumer code right after, in a function close in the callgraph. Therefore, we continue executing as long as we see new constraints - that is, operations and branches - applied on our tracked value. Once for a reasonable number of states our value is not "touched" anymore, execution can stop and we can extract meaningful conditions from the collected constraints.

The relevant code follows below:

```
1
2       def step(self, simgr, stash='active', **kwargs):
3           if not any(
4                   self._recently_constrained(state)
5                   for state in simgr.stashes[stash]):
6               self._count += 1
7           else:
8               self._count = 0
9
10          <snip>
```

```
11
12      def complete(self, simgr):
13          return self._count >= self._threshold
```

Notable mentions on the code:

- The step function, in this case, is part of the interface needed to implement a custom exploration technique. It implements what happens when angr executes step on a simulation manager.
- The complete boolean function signals to angr whether a simulation manager should stop its execution.
- The _recently_constrained function will be discussed later when describing how to interact with the solver state plugin.

In this section, most of the core ideas in Symba have been discussed in details, digging down into implementation code and examples. First, fundamentals on how to create a Control Flow Graph, explore its nodes and extract states from it have been depicted. Those states can be used to initialize a simulation manager, the top-level component responsible for every execution-related task, which behavior can be modified and tweaked using built-in and custom exploration techniques, modules which collect states in stash and regulate how states are dropped in each stash, together with defining and respecting stopping criteria. The core of symbolic analysis, that is, creating symbols, has been elaborated in both injecting symbol into memory and later solving them.

6.5 Simulation Manager

The relevant implementation code is listed below:

```
1           call_addr = res[trigger.name]
2           b = self.project.factory.block(addr=call_addr)
3           start_state = self.project.factory.entry_state(addr=b.addr)
4           sm = self.project.factory.simulation_manager(start_state)
5
6           sm.use_technique(TriggerSeer(
7               (trigger.model.config, trigger.model.name)))
8
9           sm.run()
10
11          for state in sm.deadended + sm.active + sm.unconstrained:
12              if trigger.is_triggered(state):
13                  trigger.states.append(state)
```

This part of the code is almost straightforward. For each trigger source, the saved call_addr is retrieved and used to create a new state (line 3). This simulation manager is executed through a call to run. Actually, there are a few special things that should be highlighted in the code:

- The activation of TriggerSeer technique via the interface use_technique. A section later in this chapter will describe in detail how this custom technique works.
- After simulation manager finishes its execution, the three stashes - active, deadended and unconstrained - are scanned to look for interesting states. In particular, we deem as interesting those states passing function is_triggered, a function that checks whether that state contains constraint on the trigger source. The implementation and algorithm of that function will be discussed later.
- Previously in this chapter, we observed that run, when executed on a simulation manager, continues execution until all possible paths have been exhausted; this is not the case: the TriggerSeer technique actually prevents this behavior, stopping the simulation manager when certain conditions are met.

Chapter 7

Evaluation

Symba has been evaluated by executing it, using simple and lightweight configurations, against a proof of concept code and two real-world examples containing trigger conditions of different kinds. For each sample, in this prototype, the results of the tool have been examined manually to extract meaningful information. Then, the sample is executed first in a generic virtual machine with no custom configuration, and afterwards, in a scenario where the extracted trigger conditions are met. Interactions of real malware with the OS are monitored via different methods, like watching debugging output or via the already mentioned Sysinternals ProcessMonitor tool, to observe the shape of new routines in the malware execution flow.

7.1 Proof of Concept

This proof of concept models the general idea behind RAT type malwares. That is, the sample starts a socket listening on a given port, waiting for incoming connections. Upon clients connecting to the server, it will receive a certain number of bytes which constitute the command which should be executed on the victim machine. To discourage simple static analysis techniques or network detection, malware developers will often make use of *Data Obfuscation* approaches, as documented in the MITRE framework. In order to model a known obfuscation technique, this proof of concept RAT employs *XOR obfuscation* with a byte key hardcoded in the binary, which changes for every sample produced in value and location, which in turn makes automating the extraction and fingerprinting of RAT commands suite more complex.

In this scenario, symbolic execution behaves as expected. We mark as trigger source the WinSock recv function, so that every byte received on emulated sockets becomes a symbol and commands based on those bytes can be symbolically solved.

The core control procedure of the malware is described in figure 7.1.

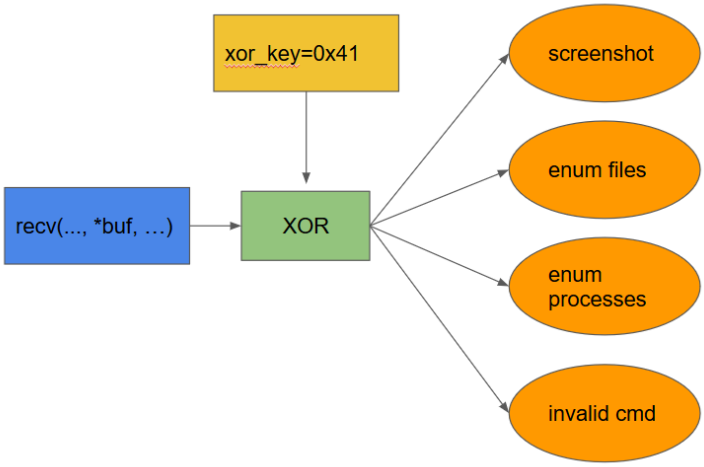

Figure 7.1: Flow of the PoC RATserver.

The three commands hide different routines that model what could happen in a real-world scenario.

1. *Command 1.* Malware reads the value of the "run" registry key, searching for installed startup programs.
2. *Command 2.* Malware enumerates processes currently running.
3. *Command 3.* Malware lists filenames in the *Documents* folder, possibly looking for files to exfiltrate.

The sample has been compiled using the i686−w64−mingw32−g++ tool part of the *MinGW* toolchain. The configuration file has been generated starting from the recv API function, and the buf parameter in the signature has been configured with inject boolean field set to true.

```
| symba.analysis | [+] Intercepted call to API function: recv.
| symba.analysis | [+] Injecting a symbol of 32 bytes in parameter buf.
```

Figure 7.2: An excerpt from Symba log during proof-of-concept analysis.

The whole Symba run took 4m2s, and produced four different results which you can observe in figure 7.3. The string covered in the black square represents the recv input where no command is executed, while the red ones represent the three different inputs associated to the malware commands.

Figure 7.3: Results of Symba for proof-of-concept malware analysis.

Let us proof the genuinity of these results by observing what happens by executing the sample, first by providing a random input, then by providing the three commands extracted by Symba. In this case, since the developed proof of concept includes plenty of debugging strings, there is no need to execute the sample in a virtual machine and observe with procmon, but rather it is enough to observe its output, as shown in the four figures from 7.4 to 7.7.

Figure 7.4: Output of the RAT with random command as input.

It is worth noting that, as demonstrated by the byte editor xxd, the command sent to the server are composed exactly of the sequence of bytes extracted by Symba.

7.2 Paranoid fish

Paranoid Fish, or *pafish*, is a demonstration tool that is used in malware analysis to assess the capability of a sandbox to escape evasion techniques used by malware developers. It has 1.4k stars and 304 forks, so it is sufficiently known in the field to ensure the validity of the used techniques. In this case, we will use Symba to focus on a specific evasion measure, extracting trigger conditions bound to it, so that it can be established without manually analysing the code which kind of values the malware is checking to determine whether it is being executed or not in a

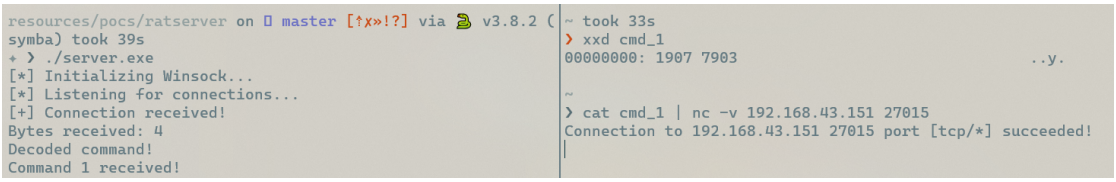

Figure 7.5: Output of the RAT with command 1.

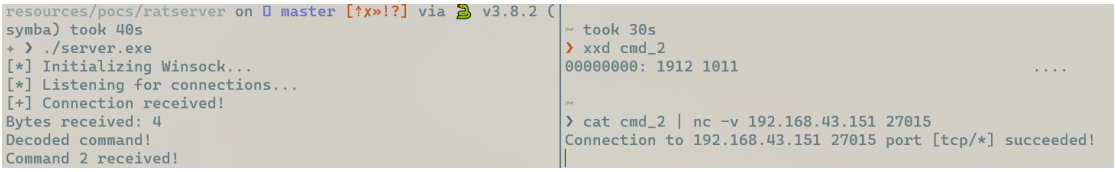

Figure 7.6: Output of the RAT with command 2.

virtualized environment. A snippet of the execution output of the malware in a "generic" Windows 10 virtual machine is shown in figure 7.8.

In this scenario, we focus on the line "[*] Checking username...". It seems that, in this case, the environment is passing the tool check. We will use Symba to determine which values of username would trigger the condition where a virtualized environment is detected. The experiment has been conduced by inserting as input the GetUserName WinAPI function, and configuring the parameter lpBuffer to be injected with a symbol.

As shown in the excerpt at 7.9, Symba finds and intercept as expected a call to function GetUserName. Moreover, the usefulness of the TriggerSeer exploration technique is proved. In fact, by inspecting the last debug logs of it in figure 7.10, we clearly see that the registry which keeps track of all the constraints posed on the trigger condition has been filled with more than 300 constraints. In such cases, posing any kind of "hardcoded" threshold would mean losing some constraints.

Symba analysis took 1m21s and produced 221 different results for the tainted value. While these may look daunting to examine, upon inspection they have the shape depicted in figure 7.11. In fact, most of the solved values in the results file are variations in case of the strings "malware", "sandbox" and "virus". This suggests a call to functions like toupper or tolower to normalize usernames before comparing them to the searched values.

In order to confirm this, we created another username in the virtual machine used for the experiments, named "sandbox". The debugging output of pafish is shown in figure 7.12. As expected, Symba correctly extracted a valid username value which triggered the "traced" check in Paranoid Fish.

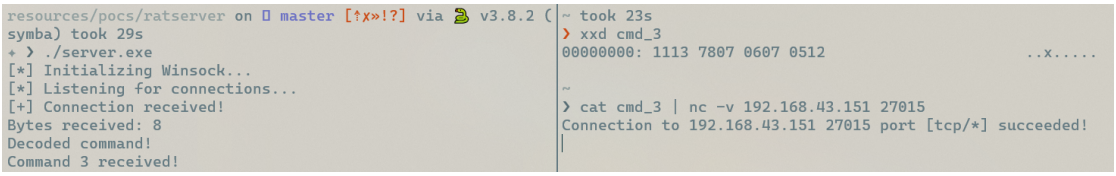

Figure 7.7: Output of the RAT with command 3.

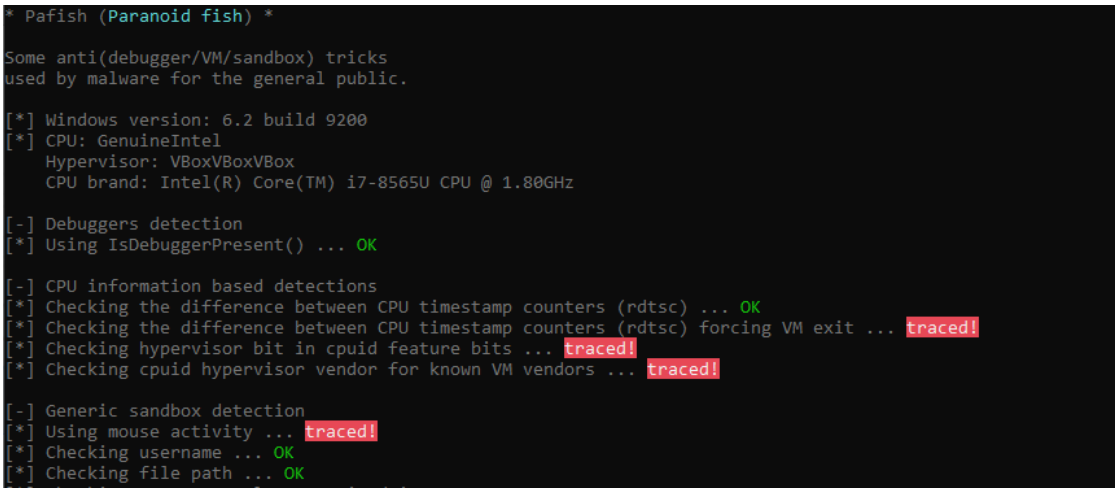

Figure 7.8: Snippet of pafish execution in a generic environment.

7.3 Wrathrage

Wrathrage is a worm which source code has been leaked on github in the kaiserfarrell (`https://github.com/kaiserfarrell/malware`) malware repository. Once executed in the target machine, it will try to send itself towards other nodes via mail or other messaging systems, and it will try to cause some damage on the host depending on the date of execution. In some dates, just a text message will be shown to the user, while in other, unfortunate dates, the malware will attempt to overwrite every file in the filesystem, basically breaking the victim machine up to an unusable state.

To conduct this experiment, we have compiled the source code of wrathrage with the MinGW toolchain, so that we had a binary that Symba could analyze. The latter has been configured using the GetSystemTime function as trigger source – since we are handling dates – where the lpSystemTime has been configured to be injected, and its length has been set to 16 bytes, 8 WORDs of 2 bytes each. This modification is reflected in Symba logs, as shown in figure 7.13. Moreover, in this case, where the symbol has a precise and predetermined structure – it inherits SYSTEMTIME properties – the constraint solving routine has been extended to

```
| symba.analysis | [+] Intercepted call to API function: GetUserNameA.
| symba.analysis | [+] Injecting a symbol of 32 bytes in parameter lpBuffer.
```

Figure 7.9: Function interception in pafish analysis.

```
| symba.analysis.triggerseer | Stepping forward. Count towards threshold: 1. Length of registry: 326
| symba.analysis.triggerseer | Stepping forward. Count towards threshold: 2. Length of registry: 326
```

Figure 7.10: TriggerSeer logs excerpt in pafish analysis.

enhance readability and inspection.

Symba completed the analysis in a few seconds, and produced the output of which an excerpt is demonstrated in figure 7.14. Apparently, three conditions are easily extracted from this file.

1. A datetime containing the 1st of November, 1/11.
2. A datetime containing any date with a day of month equal to 3.
3. A datetime containing any date with a day of month equal to 9.

In order to confirm where these conditions lead to, it would be normally necessary to execute the sample, as we did with the previous experiments, and observe the output. However, in this case, the MinGW compiler apparently produced an invalid binary that, in the virtual machine used for the experiments, is not executed correctly. Nevertheless, we can quickly, for the sake of this experiment, confirm the legitimacy of these results by consulting the source code – which in this case is available – to inspect the function where the GetSystemTime function is called. The relevant snippet is listed below.

```
1  void executePayload() {
2      //payload of the worm will change depending of the day.
3      SYSTEMTIME x;
4      GetSystemTime(&x);
5      if ((x.wDay == 1) && (x.wMonth == 11)) {
6          //this is a very special date for me. it probable represents
       why i make this worm
7          //why i hate some things.. why.. a lot of things. time to be
       a SOB.
8          fuckAll(); //fuck all
9      }
10     else {
11         if ((x.wDay == 3) || (x.wDay == 9)) {
12             //i like number 3, and number 9.
13             defaultPayload();
```

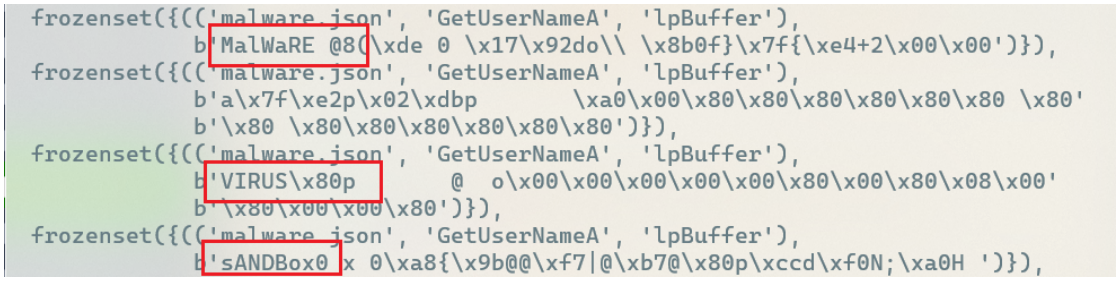

Figure 7.11: Screenshots from results file of pafish analysis.

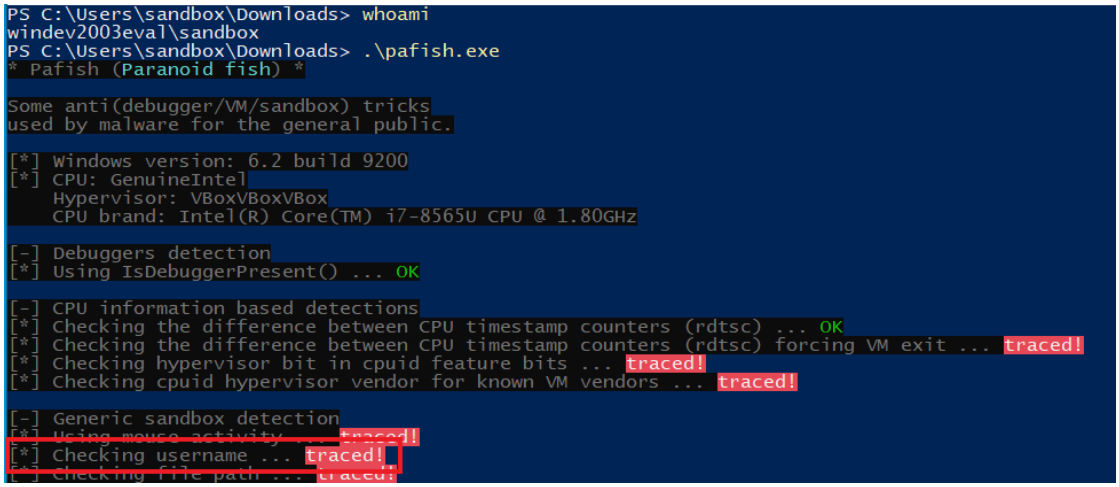

Figure 7.12: Snippet of pafish execution with trigger conditions inserted in target environment.

```
14          }
```

This listing clearly demonstrate that Symba has been able to extract all the relevant trigger conditions from the malware, without needing to consult the source code, which as already mentioned is a rare luxury when dealing with malware analysis. It represented both the condition protecting the if with an OR branch, and the one protecting the if with an AND branch, basically uncovering all the triggers connected to the GetSystemTime function.

7.4 Limitations

There are some limitations that could not be addressed in the implementation and design of Symba. Some of these limitations depend inherently on the techniques employed, while others are part of the test environment that was prepared for

```
| symba.analysis | [+] Intercepted call to API function: GetSystemTime.
| symba.analysis | [+] Injecting a symbol of 16 bytes in parameter lpSystemTime.
```

Figure 7.13: Snippet from Symba logs in wrathrage analysis, showing symbol length override.

```
2: 'wMilliseconds: 0\n'
   'wSecond: 0\n'
   'wMinute: 0\n'
   'wHour: 0\n'
   'wDay: 1\n'
   'wDayOfWeek: 0\n'
   'wMonth: 11\n'
   'wYear: 0',
3: 'wMilliseconds: 0\n'
   'wSecond: 0\n'
   'wMinute: 0\n'
   'wHour: 0\n'
   'wDay: 3\n'
   'wDayOfWeek: 0\n'
   'wMonth: 0\n'
   'wYear: 0',
4: 'wMilliseconds: 0\n'
   'wSecond: 0\n'
   'wMinute: 0\n'
   'wHour: 0\n'
   'wDay: 9\n'
   'wDayOfWeek: 0\n'
   'wMonth: 0\n'
   'wYear: 0'}
```

Figure 7.14: Solved values for GetSystemTime in wrathrage analysis.

Symba.

7.4.1 OS interaction

As already described in the background chapter, one of the greatest challenges posed in symbolic execution comes from the interaction with the Operating System and, more general, the software stack involved. For instances, many interaction with the Windows API are chained as follows:

1. A resource – such as a file – is opened, and a handle is returned.
2. The handle is used to retrieve or modify some property or content of the resource.
3. The handle is closed.

Therefore, those functions interacting with the environment – essentially, the majority of Windows API functions – cannot be simply executed symbolically

without dealing with the operating system. For Linux, angr already models the majority of system calls and interfaces exposed by the *glibc*. However, the same does not stand for the Windows OS. The support for the API that the framework offers is not complete, and often the programmer has to implement the required calls. In a binary-agnostic system like Symba, it is not possible to predict which API calls will be invoked. For this reason, executing the sample from the beginning and waiting to encounter the trigger sources is not feasible, because most likely the execution will not be consistent, errors will be raised, or the memory usage will grow with no control due to path explosion. In order to solve this, one should extend the support that angr provides for Windows, or switch to another symbolic execution engine with a tighter integration with the operating system.

7.4.2 Chained triggers

In the current state, the State Extractor component of Symba takes as the initial state of execution the one of the function that actually calls the trigger source API function. This model helps keeping as light as possible the symbolic execution process, and manages to correctly extract trigger conditions that are local to the function being analyzed. However, this means that Symba does not have knowledge of what happens before the function call that serves as the starting point. This leads to two possible issues.

- The trigger condition extracted may be hidden inside an outer trigger condition that Symba is not tracking. For example, we may be looking to extract the command suite for a RAT server listening on the wire, so we track functions of the recv family. However, before actually starting the server, the malware checks for virtualization in the target environment.
- Values used inside API function calls may not be local to the starting function, and they could have other outer constraints posed on them. For example, a GetUserName call could produce usernames that are checked for a particular user, but this value is saved inside a global variable that is checked in some other part of the code not encountered by the exploration process.

Both the issues above depend from the issues that symbolic execution yields in executing an entire binary file end-to-end. Other approaches, like a mixed dynamic-symbolic execution, could help solve this problem.

7.4.3 Testing samples

The binaries that Symba was evaluated against have been selected because they demonstrate how the system performs against different kinds of input – network

packets, system time, and information on the username. In this sense, they serve well as case studies. However, a better approach to test and improve the system could be more automated, with a flow designed like follows.

1. A great number of malicious samples is extracted using malware repositories, such as VirusTotal.
2. All those samples are executed on a Cuckoo, or similar, sandbox, and analysis signatures are produced.
3. Afterwards, samples are fed to Symba, which extracts for each one new trigger conditions based on a generic configuration file.
4. The validity of conditions is tested by executing again each sample in a Cuckoo instance where they are inserted in the context, monitoring whether or not the triggers unlocked new paths in the analysed samples.

However, most of the samples incoming from these feeds, nowadays, present some issues to this process that made unfeasible to properly setup this testbed during the time spent working on this thesis. Most notably, malware does not come in its ultimate, malicious form so easily. Obfuscation, packing, dynamic loading, loading dlls from the network are just some examples of the techniques used to hinder. Those samples resulted to be too noisy for Symba to properly extract conditions. In order to overcome this, the tool could be best integrated into a pre-existent sandbox framework that would use Symba to extract conditions from the final, dumped, unpacked and unencrypted sample that would eventually go into the hands of the malware analyst. In this direction, the usefulness of a system like Symba stands in its capability to be able to reason over trigger conditions in generally less time than what would take to a human, when they would both start from a somewhat "clean" executable. We mention that some research has been dedicated to the question of symbolic execution with respect to self-modifying code [35] and obfuscation aimed exactly towards hindering symbolic execution [36].

Chapter 8

Related Work

8.1 Symbolic Execution

Symbolic execution has lately drawn quite an interesting amount of interest in the literature world, due to its ability to find errors hidden deeply in the software and greatly increase coverage measures in software testing. In particular, mentioning information security research, an interesting direction of research is the one of automatic exploit generation and vulnerabilities discovery via symbolic execution. This problem is similar enough to what has been investigated in this thesis, in fact, the inputs which would make the program crash can be defined as triggers responsible for states of inconsistency in the program. The real difference is that, where the conditions that Symba is looking for are explicitly coded in the malware, those guarding vulnerabilities and exploits are hidden inside software features. Concerning this research, AEG [37] is an end-to-end system developed for *full exploit generation* based on what the authors call *preconditioned symbolic execution*. The idea is that, among the infinite paths that the symbolic execution could take, the system only focuses on those which are likely exploitable, such as, for instance, those where a maximum length on a buffer is imposed or a network input is processed. Driller [38] focuses on vulnerability discovery rather than full exploitation, and it employs an interesting approach: it merges the ability of symbolic execution to solve for branches condition with the speed and performances of fuzzing engines to explore many paths at once. Specifically, the system is composed of the AFL fuzzer and angr. When the former encounters an interesting branch, it will forward the execution state to angr, which will act as an oracle to determine conditions to explore both branch, feeding them back to AFL. HeapHopper [39] uses symbolic execution and model checking to perform a systematic analysis of different heap allocators implementations. Regarding dynamic allocation, [30] investigates on how to reason over worst-case memory consumption in software. Memoise [40]

implements an approach to re-use results of previous symbolic execution runs to increase efficiency over time.

8.2 Malware Trigger Analysis

The problem of extracting inputs to increase coverage in malware analysis is not recent, and other works have been directed at it, with different approaches. In [41], the authors explore the capabilities of the angr framework to extract trigger conditions from a sample used as case study. The concept is similar to what handled in this work, but Symba is directed towards automatic, binary-agnostic analysis, while the former has been tailored to the sample analyzed. Minesweeper, a tool developed for [42], employ *mixed concrete and symbolic execution* [43] to automatically explore execution paths depending on trigger inputs. Just as in Symba, for Minesweeper the user chooses which trigger types he deems interesting, and values produced by these triggers will be tainted and solved. For the task, the authors developed a system similar to angr, where the binary is transformed to an *Intermediate Representation* to provide easier reasoning for the solver. However, the system is limited to *x86* architecture, and its tight design might prevent easy integration inside other systems. TriggerScope [44] focuses on the Android OS, and it performs *trigger analysis*, a static technique that seeks to automatically identify triggers in applications. Precise detection and extraction of triggers is made possible through a combination of *path predicate reconstruction and minimization* and *interprocedural control-dependency analysis*. Work performed in [45] explores the possibility of using concolic execution frameworks like angr to reverse engineer the malware signatures inside an offline antivirus database. BotMelt [46] symbolically executes botnet malwares, considering network packets as symbol to solve. The framework employed is S2E.

Altough Symba does not provide any dynamic execution component, it could be integrated in any of those systems. This dynamic component would replace the State Extractor: instead of generating a blank state at the beginning of a function, a state filled with memory image, registers values, opened file descriptors and everything connected to the environment could be exported as the initial state. Once conditions are extracted by the Solver, these could be directly inserted in the dynamic execution flow of the binary.

Chapter 9

Conclusions

In normal scenarios, extracting trigger conditions guarding suspicious behaviour requires manual analysis, leading to consumption of time and effort. In this work, we present Symba, a targeted symbolic execution system that tracks and solves appropriate inputs to unlock new and interesting behaviour in binary executables. By combining custom techniques to extract starting and ending points, track interesting symbols, and solve their values, symbolic execution was used in a way that mitigated its typical weaknesses, rendering possible to implement a tool that still completely works in the land of static analyses, with no need for a dynamic environment that would introduce other problems in the design. The system has been capable of extracting and solving trigger conditions for both proof of concept code and real samples, with an average execution time of 2 minutes. Moreover, while the system has been tested on Windows, the high adaptability of angr can be exploited to have a single tool producing results for malwares of different platforms and architectures. The modularity of the system makes it easy to extend and improve its features, and the use of Python, a highly employed programming language in the malware analysis field, as the language of choice, makes it easier to integrate Symba in already operating infrastructures and tools.

9.1 Future work

Symba has been developed as a prototype and its functionalities can be extended furthermore. As described in the architecture design chapter at 5.1, the functions currently tracked as trigger sources are manually configured by the analyst by specifying a list of API functions in a file. While this approach is highly customizable, and it gives analysts the power to tweak their analysis and control time of execution, it also means that Symba could lose some trigger conditions not explicitly tracked in the configuration. For this reason, Symba could be extended with a component

to automatically extract plausible trigger sources. For instance, this component could first identify all the API calls in the binary, and starting to symbolically execute from there. Afterwards, if interesting branches are detected that depend on some of this API call produced values, those function would be marked as possible trigger sources.

As mentioned, Symba works at a statical layer, without actually executing the binary in a real environment. While this brings a few advantages in simplicity and portability, it is also affected by the discussed limitations in chapter 7. Therefore, in a future research direction, Symba could be integrated in a dynamic execution environment [47], which would ensure a more consistent and robust execution flow. Once a trigger source is met, Symba would be consulted as a symbolic oracle that would compute concrete solutions, feeding them back to the dynamic execution environment. Notably, recently, the angr framework integrated a component named Symbion [48], which exposes an API to implement what said and connect angr to a real running target like GDB. The dynamic target could perform more than simply executing once the binary: it could use symbolic execution to enhance an already existent fuzzing process, such as [49] suggests.

Currently, Symba produces a result log with all the solutions of tracked trigger conditions. Those conditions have to be inserted manually, wherever the analyst wishes, to confirm and observe the new behaviour of exhamined samples. In a future iteration, Symba could be directly integrated with a sandbox system like Cuckoo Sandbox, and automatically extract new signatures based on discovered trigger conditions. Overall, the ideas designed for Symba can be integrated and extended in various different analysis scenarios and systems.

Appendix A

Code

This appendix contains some snippets of code that are pointed, mostly, in the implementation details section.

Listing A.1: Test code, version 1.

```c
#include <stdio.h>

int func(int n, char* string, char* other_string) {
    printf("This shouldn't be printed if hooked!\n");
}
int main(int argc, char* argv[]) {
    func(10, "ARG1", "ARG2");
}
```

Listing A.2: angr hello world code.

```python
import angr

# Init project
proj = angr.Project("test")

# Init symbolic execution manager
simgr = proj.factory.simulation_manager()

# Step until all states terminate
simgr.run()

# Prints output collected to stdout
final_state = simgr.deadended[0]
print("******STDOUT CONTENT******")
final_state.posix.dumps(1)
```

Listing A.3: Test code, version 2.

```c
int main(int argc, char* argv[]) {
    char buf[32];
    char* arg1 = "ARG1";
    char* arg2 = "ARG2";
    int choice = 0;

    // Get number of arguments
    printf("Choose number of arguments: ");
    fgets(buf, 32, stdin);
    choice = atoi(buf);

    // Choose function
    if (choice == 1) {
        func1(1);
    }

    else if (choice == 2) {
        func2(2, arg1);
    }

    else if (choice == 3) {
        func3(3, arg1, arg2);
    }
    else {
        exit(1);
    }
}
```

Listing A.4: Example code of a SimProcedure

```python
import angr

# Init project
proj = angr.Project("test")


# Define Symbolic Procedure
class FuncReplace(angr.SimProcedure):
    def run(self, int_arg, string, other_string):
        print("The function func was replaced by this function!")
        print(f"int_arg: {int_arg} / string: {string} / other_string
    {other_string}")
        return 0

# Hook func, replacing it with the SimProcedure
proj.hook_symbol("func", FuncReplace())


# Init symbolic execution manager
```

```
simgr = proj.factory.simulation_manager()

# Step until all states terminate
simgr.run()

# Prints output collected to stdout
final_state = simgr.deadended[0]
print("******STDOUT CONTENT******")
print(final_state.posix.dumps(1))
```

Listing A.5: Implementation of the generic model injected.

```
class GenericModel(SimProcedure):
    """
    It will work as a charm.
    A GenericModel receives in input
    A Symba function signature,
    and it models the run() function
    just by injecting into memory
    symbols as specified into config,
    which needs to be inherited by angr
    standard, accordingly.
    """

    def __init__(self, fsig, config_name, default_len=32):
        self.fsig = fsig
        self.name = fsig.name
        self.config = config_name
        self.params = [inspect.Parameter(
            p.name, inspect.Parameter.POSITIONAL_OR_KEYWORD) for p in
    fsig.params]
        self._default_len = default_len

        super().__init__(num_args=len(self.params))
        pass

    # angr asks, we please him.
    def run(self, *args):  # pylint: disable=method-hidden
        # inspect.signature(self.run).bind(*args)
        params = self.fsig.params
        # Symbol Injection process
        for i, param in enumerate(params):
            [...]
```

Listing A.6: MSDN retriever script.

```
<snip>

with open(args.api_file, 'r') as f:
```

```
    funcs = [line.strip() for line in f]

msdns = []
for funcname in funcs:
    query = "site:docs.microsoft.com {}".format(funcname)

    for s in search (
        query,
        tld='it',
        lang='en',
        tbs='0',
        safe='off',
        num=2,
        start=0,
        stop=2,
        domains=None,
        pause=2.0,
        tpe='',
        country='',
        extra_params=None,
        user_agent=None):
        if 'win32/api/' in s and funcname.lower() in s:
            msdns.append(s)

out = {}
out['functions'] = []
# Matter of time and readability man
functions = out['functions']

for msdn in msdns:
    soup = BeautifulSoup(requests.get(msdn).text, 'html.parser')
    code = soup.find('code').get_text().split('\n')[:-2]

    # Shall we begin? Let's parse this code!

    function_name = code[0].split()[1][:-1]
    params = []

    for i, param in enumerate(code[1:]):
        t = param.split()[0]
        # Last argument is the only one without ','
        n = param.split()[1] if i == (len(code[1:]) - 1) else param.split()[1][:-1]
        params.append({'type': t, 'name': n, 'inject': False, 'length': "<DEFAULT>"})
    functions.append({'name': function_name, 'params': params})

with open(f"{args.outconfig}.json", 'w') as f:
    json.dump(out, f, indent=2)
```

Listing A.7: JSON Configuration file for Symba.

```
{
  "functions": [
    {
      "name": "GetSystemTime",
      "params": [
        {
          "type": "LPSYSTEMTIME",
          "name": "lpSystemTime",
          "inject": false,
          "length": "<DEFAULT>"
        }
      ]
    },
    {
      "name": "GetUserNameA",
      "params": [
        {
          "type": "LPSTR",
          "name": "lpBuffer",
          "inject": false,
          "length": "<DEFAULT>"
        },
        {
          "type": "LPDWORD",
          "name": "pcbBuffer",
          "inject": false,
          "length": "<DEFAULT>"
        }
      ]
    }
  ]
}
```

Bibliography

[1] Benjamin Jensen, Brandon Valeriano, and Ryan Maness. «Fancy bears and digital trolls: Cyber strategy with a Russian twist». In: *Journal of Strategic Studies* (Jan. 2019), pp. 1–23. DOI: 10.1080/01402390.2018.1559152 (cit. on p. 2).

[2] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. «Automatically Identifying Trigger-based Behavior in Malware». In: vol. 36. Jan. 1970, pp. 65–88. DOI: 10.1007/978-0-387-68768-1_4 (cit. on p. 3).

[3] Yan Shoshitaishvili et al. «SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis». In: May 2016, pp. 138–157. DOI: 10.1109/SP.2016.17 (cit. on p. 7).

[4] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael Lyu. «On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs». In: *IEEE Transactions on Dependable and Secure Computing* PP (Dec. 2017). DOI: 10.1109/TDSC.2018.2866469 (cit. on p. 8).

[5] Zainab Saud and M. Hasan Islam. «Towards Proactive Detection of Advanced Persistent Threat (APT) Attacks Using Honeypots». In: *Proceedings of the 8th International Conference on Security of Information and Networks*. SIN '15. Sochi, Russia: Association for Computing Machinery, 2015, pp. 154–157. ISBN: 9781450334532. DOI: 10.1145/2799979.2800042. URL: https://doi.org/10.1145/2799979.2800042 (cit. on p. 11).

[6] Samuel Greengard. «The New Face of War». In: *Commun. ACM* 53.12 (Dec. 2010), pp. 20–22. ISSN: 0001-0782. DOI: 10.1145/1859204.1859212. URL: https://doi.org/10.1145/1859204.1859212 (cit. on p. 11).

[7] Giuseppe Primiero, Frida Solheim, and Jonathan Spring. «On Malfunction, Mechanisms and Malware Classification». In: *Philosophy & Technology* (Nov. 2018). DOI: 10.1007/s13347-018-0334-2 (cit. on p. 12).

[8] Alan Lee, Vijay Varadharajan, and Udaya Tupakula. «On Malware Characterization and Attack Classification». In: *Proceedings of the First Australasian Web Conference - Volume 144*. AWC '13. Adelaide, Australia: Australian Computer Society, Inc., 2013, pp. 43–47. ISBN: 9781921770296 (cit. on p. 12).

[9] Yunting Guo and Wenqing Fan. «Feature Collection and Selection in Malware Classification». In: *Proceedings of the 2019 International Conference on Artificial Intelligence and Advanced Manufacturing*. AIAM 2019. Dublin, Ireland: Association for Computing Machinery, 2019. ISBN: 9781450372022. DOI: 10.1145/3358331.3358342. URL: https://doi.org/10.1145/3358331.3358342 (cit. on p. 12).

[10] Ivan Nikolaev, Martin Grill, and Veronica Valeros. «Exploit Kit Website Detection Using HTTP Proxy Logs». In: *Proceedings of the Fifth International Conference on Network, Communication and Computing*. ICNCC '16. Kyoto, Japan: Association for Computing Machinery, 2016, pp. 120–125. ISBN: 9781450347938. DOI: 10.1145/3033288.3033354. URL: https://doi.org/10.1145/3033288.3033354 (cit. on p. 12).

[11] Ondřej Pluskal. «Behavioural Malware Detection Using Efficient SVM Implementation». In: *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*. RACS. Prague, Czech Republic: Association for Computing Machinery, 2015, pp. 296–301. ISBN: 9781450337380. DOI: 10.1145/2811411.2811516. URL: https://doi.org/10.1145/2811411.2811516 (cit. on p. 12).

[12] Guy Martin, Saira Ghafur, James Kinross, Chris Hankin, and Ara Darzi. «WannaCry—a year on». In: *BMJ* 361 (June 2018), k2381. DOI: 10.1136/bmj.k2381 (cit. on p. 13).

[13] Rami Sihwail, Khairuddin Omar, and Khairul Akram Zainol Ariffin. «A survey on malware analysis techniques: static, dynamic, hybrid and memory analysis». In: 8 (Jan. 2018), p. 1662. DOI: 10.18517/ijaseit.8.4-2.6827 (cit. on p. 13).

[14] Roman Rohleder. «Hands-On Ghidra - A Tutorial about the Software Reverse Engineering Framework». In: Nov. 2019, pp. 77–78. ISBN: 978-1-4503-6835-3. DOI: 10.1145/3338503.3357725 (cit. on p. 13).

[15] Dhruwajita Devi and Sukumar Nandi. «Detection of Packed Malware». In: *Proceedings of the First International Conference on Security of Internet of Things*. SecurIT '12. Kollam, India: Association for Computing Machinery, 2012, pp. 22–26. ISBN: 9781450318228. DOI: 10.1145/2490428.2490431. URL: https://doi.org/10.1145/2490428.2490431 (cit. on pp. 13, 20).

[16] Marco Gaudesi, Andrea Marcelli, Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. «Malware Obfuscation through Evolutionary Packers». In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. GECCO Companion '15. Madrid, Spain: Association for Computing Machinery, 2015, pp. 757–758. ISBN: 9781450334884. DOI: 10.1145/2739482.2764940. URL: https://doi.org/10.1145/2739482.2764940 (cit. on p. 13).

[17] Andrey Mikhailov, Aleksey Hmelnov, Evgeny Cherkashin, and Igor Bychkov. «Control flow graph visualization in compiled software engineering». In: May 2016, pp. 1313–1317. DOI: 10.1109/MIPRO.2016.7522343 (cit. on p. 14).

[18] Zhuojun Ren, Guang Chen, and Wenke Lu. «Space Filling Curve Mapping for Malware Detection and Classification». In: *Proceedings of the 2020 3rd International Conference on Computer Science and Software Engineering*. CSSE 2020. Beijing, China: Association for Computing Machinery, 2020, pp. 176–180. ISBN: 9781450375528. DOI: 10.1145/3403746.3403924. URL: https://doi.org/10.1145/3403746.3403924 (cit. on p. 14).

[19] Waqas Aman. «A Framework for Analysis and Comparison of Dynamic Malware Analysis Tools». In: *International Journal of Network Security & Its Applications* 6 (Oct. 2014). DOI: 10.5121/ijnsa.2014.6505 (cit. on p. 14).

[20] 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 (Sept. 2019). ISSN: 0360-0300. DOI: 10.1145/3329786. URL: https://doi.org/10.1145/3329786 (cit. on p. 14).

[21] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. «Malware Dynamic Analysis Evasion Techniques: A Survey». In: *ACM Comput. Surv.* 52.6 (Nov. 2019). ISSN: 0360-0300. DOI: 10.1145/3365001. URL: https://doi.org/10.1145/3365001 (cit. on p. 18).

[22] Melissa Chua and Vivek Balachandran. «Effectiveness of Android Obfuscation on Evading Anti-Malware». In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. CODASPY '18. Tempe, AZ, USA: Association for Computing Machinery, 2018, pp. 143–145. ISBN: 9781450356329. DOI: 10.1145/3176258.3176942. URL: https://doi.org/10.1145/3176258.3176942 (cit. on p. 19).

[23] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. «Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?» In: *ACM Comput. Surv.* 49.1 (Apr. 2016). ISSN: 0360-0300. DOI: 10.1145/2886012. URL: https://doi.org/10.1145/2886012 (cit. on p. 19).

[24] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. «Obfuscator-LLVM: Software Protection for the Masses». In: *Proceedings of the 1st International Workshop on Software Protection*. SPRO '15. Florence, Italy: IEEE Press, 2015, pp. 3–9 (cit. on p. 19).

[25] Blake Anderson, Curtis Storlie, and Terran Lane. «Improving Malware Classification: Bridging the Static/Dynamic Gap». In: *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*. AISec '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 3–14. ISBN: 9781450316644. DOI: 10.1145/2381896.2381900. URL: https://doi.org/10.1145/2381896.2381900 (cit. on p. 21).

[26] Cristian Cadar and Koushik Sen. «Symbolic Execution for Software Testing: Three Decades Later». In: *Commun. ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. URL: https://doi.org/10.1145/2408776.2408795 (cit. on p. 21).

[27] 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 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657 (cit. on pp. 21–23).

[28] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. «Symbolic Execution of Programs with Strings». In: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. SAICSIT '12. Pretoria, South Africa: Association for Computing Machinery, 2012, pp. 139–148. ISBN: 9781450313087. DOI: 10.1145/2389836.2389853. URL: https://doi.org/10.1145/2389836.2389853 (cit. on p. 22).

[29] Aric Hagberg, Pieter Swart, and Daniel Chult. «Exploring Network Structure, Dynamics, and Function Using NetworkX». In: Jan. 2008 (cit. on p. 28).

[30] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. «Symbolic Execution for Memory Consumption Analysis». In: *SIGPLAN Not.* 51.5 (June 2016), pp. 62–71. ISSN: 0362-1340. DOI: 10.1145/2980930.2907955. URL: https://doi.org/10.1145/2980930.2907955 (cit. on pp. 34, 69).

[31] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. «Loop-Extended Symbolic Execution on Binary Programs». In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. Chicago, IL, USA: Association for Computing Machinery, 2009, pp. 225–236. ISBN: 9781605583389. DOI: 10.1145/1572272.1572299. URL: https://doi.org/10.1145/1572272.1572299 (cit. on p. 34).

[32] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. «Enhancing Symbolic Execution with Veritesting». In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 1083–1094. ISBN: 9781450327565. DOI: 10.1145/2568225.2568293. URL: https://doi.org/10.1145/2568225.2568293 (cit. on p. 34).

[33] Vaibhav Sharma, Michael W. Whalen, Stephen McCamant, and Willem Visser. «Veritesting Challenges in Symbolic Execution of Java». In: *SIGSOFT Softw. Eng. Notes* 42.4 (Jan. 2018), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/3149485.3149491. URL: https://doi.org/10.1145/3149485.3149491 (cit. on p. 34).

[34] Cristian Cadar, Daniel Dunbar, and Dawson Engler. «KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs». In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224 (cit. on p. 54).

[35] Lian Li, Yi Lu, and Jingling Xue. «Dynamic Symbolic Execution for Polymorphism». In: *Proceedings of the 26th International Conference on Compiler Construction*. CC 2017. Austin, TX, USA: Association for Computing Machinery, 2017, pp. 120–130. ISBN: 9781450352338. DOI: 10.1145/3033019.3033029. URL: https://doi.org/10.1145/3033019.3033029 (cit. on p. 68).

[36] 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*. ACSAC '16. Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 189–200. ISBN: 9781450347716. DOI: 10.1145/2991079.2991114. URL: https://doi.org/10.1145/2991079.2991114 (cit. on p. 68).

[37] Thanassis Avgerinos, Sang Cha, Brent Hao, and David Brumley. «AEG: Automatic Exploit Generation.» In: vol. 57. Jan. 2011. DOI: 10.1145/2560217.2560219 (cit. on p. 69).

[38] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. «Driller: Augmenting Fuzzing Through Selective Symbolic Execution». In: Jan. 2016. DOI: 10.14722/ndss.2016.23368 (cit. on p. 69).

[39] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. «Heaphopper: Bringing Bounded Model Checking to Heap Implementation Security». In: *Proceedings of the 27th*

USENIX Conference on Security Symposium. SEC'18. Baltimore, MD, USA: USENIX Association, 2018, pp. 99–116. ISBN: 9781931971461 (cit. on p. 69).

[40] Guowei Yang, Sarfraz Khurshid, and Corina S. Păsăreanu. «Memoise: A Tool for Memoized Symbolic Execution». In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 1343–1346. ISBN: 9781467330763 (cit. on p. 69).

[41] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. «Assisting Malware Analysis with Symbolic Execution: A Case Study». In: June 2017, pp. 171–188. ISBN: 978-3-319-60079-6. DOI: 10.1007/978-3-319-60080-2_12 (cit. on p. 70).

[42] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. «Automatically Identifying Trigger-based Behavior in Malware». In: *Botnet Detection: Countering the Largest Security Threat*. Ed. by Wenke Lee, Cliff Wang, and David Dagon. Boston, MA: Springer US, 2008, pp. 65–88. ISBN: 978-0-387-68768-1. DOI: 10.1007/978-0-387-68768-1_4. URL: https://doi.org/10.1007/978-0-387-68768-1_4 (cit. on p. 70).

[43] Dorottya Papp, Levente Buttyán, and Zhendong Ma. «Towards Semi-Automated Detection of Trigger-Based Behavior for Software Security Assurance». In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ARES '17. Reggio Calabria, Italy: Association for Computing Machinery, 2017. ISBN: 9781450352574. DOI: 10.1145/3098954.3105821. URL: https://doi.org/10.1145/3098954.3105821 (cit. on p. 70).

[44] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. «TriggerScope: Towards Detecting Logic Bombs in Android Applications». In: May 2016, pp. 377–396. DOI: 10.1109/SP.2016.30 (cit. on p. 70).

[45] Aubrey Alston. «Concolic Execution as a General Method of Determining Local Malware Signatures». In: *ArXiv* abs/1705.05514 (2017) (cit. on p. 70).

[46] Byeongho Kang, Jisu Yang, Jaehyun So, and Czang Yeob Kim. «Detecting Trigger-Based Behaviors in Botnet Malware». In: *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*. RACS. Prague, Czech Republic: Association for Computing Machinery, 2015, pp. 274–279. ISBN: 9781450337380. DOI: 10.1145/2811411.2811485. URL: https://doi.org/10.1145/2811411.2811485 (cit. on p. 70).

[47] Konrad Jamrozik, Gordon Fraser, Nikolai Tillmann, and Jonathan De Halleux. «Augmented Dynamic Symbolic Execution». In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: Association for Computing Machinery, 2012,

pp. 254–257. ISBN: 9781450312042. DOI: 10.1145/2351676.2351716. URL: https://doi.org/10.1145/2351676.2351716 (cit. on p. 72).

[48] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. «SYMBION: Interleaving Symbolic with Concrete Execution». In: *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*. June 2020 (cit. on p. 72).

[49] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. «Deferred Concretization in Symbolic Execution via Fuzzing». In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 228–238. ISBN: 9781450362245. DOI: 10.1145/3293882.3330554. URL: https://doi.org/10.1145/3293882.3330554 (cit. on p. 72).