UNIVERSITY OF CALGARY

Towards Automatic Generation of Anti-Virus Emulators

by

Daniel Medeiros Nunes de Castro

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

December, 2010

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Towards Automatic Generation of Anti-Virus Emulators" submitted by Daniel Medeiros Nunes de Castro in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

[illegible]

Supervisor, Dr. John D. Aycock
Department of Computer Science

[illegible]

Dr. Michael E. Locasto
Department of Computer Science

Mark L Bauer

Dr. Mark L. Bauer
Department of Mathematics & Statistics

15 Dec 2010

Date

Abstract

Computer viruses and malicious software (malware) in general are a prevalent threat in today's world. While widely used, static based detection of malware, such as signature detection, is not sufficient to fight this threat. We need to also adopt dynamic approaches.

One of the dynamic approaches is code emulation, specifically using anti-virus emulators. Anti-virus emulators have a strong focus on malware detection and analysis instead of on emulating code to its completion.

However, most anti-virus emulators are unavailable for researchers. Most of them are proprietary and when not embedded in an anti-virus product, they are limited to anti-virus companies' laboratories. In this work, we present AGAVE, a step towards automatic generation of anti-virus emulators. AGAVE is flexible and customizable. Using Python syntax, the researcher can customize library and system calls and even CPU instructions.

To simplify use by the researcher, AGAVE uses previously collected program execution traces as a base of information about system and library calls. This information is accessed using Case-Based Reasoning(CBR), so even when an unknown call is found during the emulation, the emulator can respond to the call properly and the emulation is not interrupted.

In order to emulate low-level CPU instructions, AGAVE uses a third-party CPU emulator. The researcher, the intended user of AGAVE, can choose to change the CPU emulator to the one of their preference. AGAVE provides a common interface for the CPU emulator that can be customized so the researcher can use the chosen emulator.

In this thesis, we discuss the design and implementation of AGAVE, including its CBR modules and how a researcher would use it in a practical example. We also evaluate our implementation, discuss some limitations of AGAVE and propose some future work.

Acknowledgements

I would like to dedicate this thesis to Raquel, Melissa and Iraci.

Most of all, I would like to thank Raquel for her immense patience and understanding, especially during the last couple of months before my defense, when she had most of the work to take care of our "best production of the year", Melissa. Thanks to my mom, Iraci, for all her support, including trying and helping me to see the bright side when I missed the opportunity of traveling to meet her: "At least now you can focus on your work." And, of course, thanks Melissa, definitely my best production of 2010, for her precious smiles when seeing daddy coming home.

Special thanks also to Dr. John Aycock, more than a supervisor, an adviser. His comments and suggestions, either related to studies or to life in general, have always been invaluable.

I also would like to thank my friends from the Double-Secret Security Lab/Programming Languages Lab for their support and interesting discussions.

And last but not least, I would like to thank the force that keeps us going on, in the right direction and doing the right things, independent of religious denomination.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computers have become impressively prevalent, more notably in the last three decades. We do not only find them in universities and companies but also in our very homes, with uses varying from games to home banking, and now even wherever we are, as cellphones and PDA's. However, with all the benefits we have from computers and computing devices, they have also introduced a new concern into our lives: malicious software.

Malicious software (malware) is a generic term that includes all kinds of malicious programs. Some examples are:

- The infamous viruses that, as their biological counterpart does, can "reproduce" themselves by copying their (genetic) code into other programs and that are sometimes considered by media as a synonym for malware; and

- Trojan horses. Inspired by Greek history, these programs seem like a gift (such as a free graphic editor or a fun game), but are hiding a secret danger. Secretly, they may look at our files or detect keystrokes, trying to gather sensitive information, like passwords to our favorite websites or even to our bank accounts.

During this work, the differences between types of malware (virus, Trojan horses, worms and others) are irrelevant and, in order to keep expressions and names as used in the original sources, the words "virus" and "malware" might sometimes be interchanged, sharing the same basic concept of "malicious software". The exception is when "viral code" is mentioned specifically, which refers to the ability of a computer virus to replicate itself.

Detecting, and eventually eliminating, malware is a problem that has been becoming increasingly harder, as the motivation for writing malware has also changed[24]. While the first malware writers did so only for learning how things work, for fun or, in some cases, for fame, nowadays it is used mainly for profit, sometimes with criminal organizations behind them, funding their development[34].

One of the earliest approaches for detection is the use of static analysis[62]. This includes looking at the code being analyzed to understand how it works, using tools like disassemblers (programs that can translate the machine language code in an executable back to code in a higher level language, such as assembly) and also searching for specific sequences of characters (usually called signatures). Signature detection is, by itself, a quite difficult problem, as the number of possible signatures to be found tends to be incredibly high [1].

While still being largely used, static analysis has lost some of its efficacy as malware writers now use different techniques to avoid it. Obfuscation techniques, including the application of cryptography, can be used to hide malicious code until the moment when it is really necessary[21], avoiding earlier detection. It is also possible to create code that is able to adapt itself, essentially changing from one generation to another, making detection, in special signature-based techniques, even harder while still preserving the malicious capabilities. This is achieved by using techniques known as *polymorphism* or *metamorphism*[6].

When static analysis fails, security researchers must adopt what are called dynamic approaches. An example consists in executing any suspicious code in a controlled, monitored, and preferably isolated, environment. By doing this, whenever something considered "bad" happens (as an unexpected file change or removal, or a network connection), no harm to the production system is done and information about the code, as what it

[1]Symantec's website reports a total of over 8.7 million signatures in their signature file in October 2010. Source: `http://www.symantec.com/business/security_response/definitions/certified/index.jsp`

has been doing or what it saved in memory or disk, can be collected to allow further identification and analysis.

It must be noted that, for each time an analysis is performed, the machine would need to be returned to an "initial state", a state when it is considered safe, with no infection (no malware was installed). Depending on the case, a task that might require quite some time to be performed properly, as not only files in the disk can be infected, but there are also cases of malware attacking the BIOS (Basic Input/Output System), the built-in software in all computers that is responsible for the very task of accessing disks and other devices.

Instead of executing the code in a controlled environment, an even safer approach would be emulating its execution.

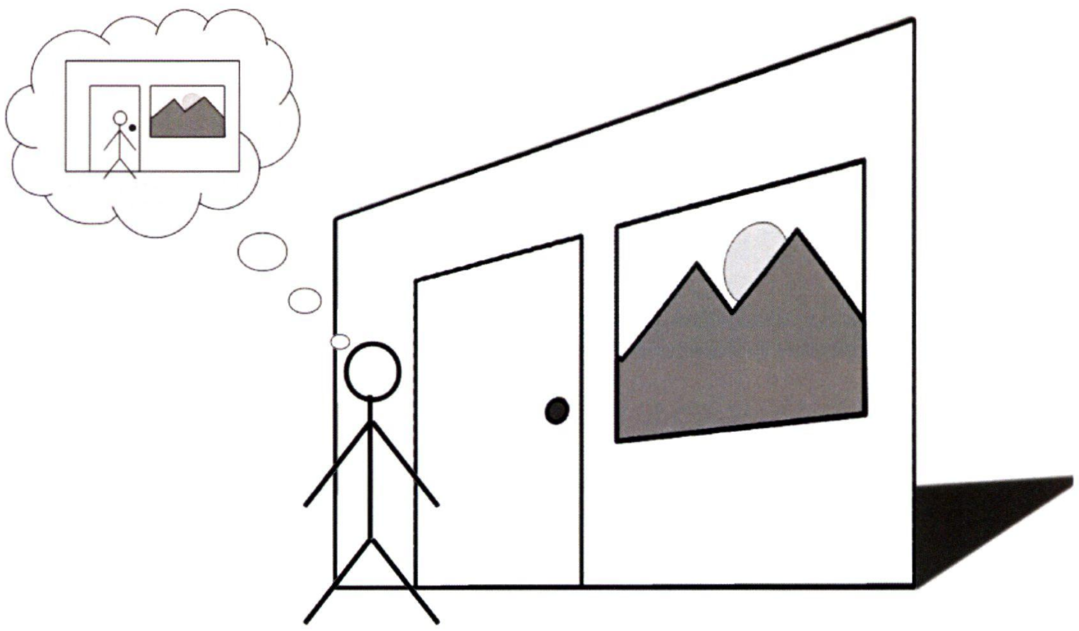


Figure 1.1: Movie set scenery

We can think of emulation as a movie set (Figure 1.1) when you look around, everything seems quite real, but behind a door there is just a wall and a window provides a view to a nice picture. In order to emulate code execution, we need to provide an environment with enough elements so the malware is able to run without noticing that it is only a facade, and the emulated programs become actors in a play, acting as if the facade was real, but with no impact in the real world, other than knowledge, of course.

Still in the realm of movies, we can recall the 1998 movie "The Truman Show", where the main character (Jim Carrey), was part of a TV show for his entire life (30 years) without being aware of it. The entire city was scenery and all the people there, but him, were actors. He lived his life as it was the real world in front of his eyes, and not a TV show with scripts, where even the weather was controlled by people outside that bubble. This is basically what we want in an emulated environment, the program being emulated, our suspicious malware, must not have any hint that it is in an emulation and not in an actual system.

Emulation can be used in several ways and there are several different tools available. A common approach for many security researchers is running suspicious code in virtual machines, which are, in this context, hardware emulators (programs that emulate only the hardware, including processors, memory and video cards) that are able to execute common operating systems. This approach not only removes the need of purchasing new equipment for tests, but it also provides a level of isolation necessary for tests and even allows the researcher to return to the initial state in a short time. However some problems still persist[6]:

- The actual operating system is executed in the virtual machine, so while this assures that the environment reflects a real one, there might be some issues concerning compatibility to the hardware being emulated and also OS copyright and licensing.
- Monitoring capabilities are not usually included or are not designed for the specific task of detecting malware.

- Booting the operating system requires an amount of time that delays the actual task of emulating the code, decreasing productivity. [2]

Another use of emulation may happen internally to some anti-virus products. In order to provide dynamic analysis, they send part of the suspicious code to a secure environment[3] and then emulate the code, so malicious activities can be detected[55, p.164]. In some cases this process can also help to detect malware signatures that were encrypted until then and in order to be used, need to be decrypted in memory. The mentioned "secure environment", known as "anti-virus emulator", sometimes also called a *sandbox*, is not capable of running a full emulation (as graphic operations, among others, are usually nonexistent), but it is still able to emulate some hardware and software operations and this allows it to achieve its main purpose: malware detection.

A drawback for emulating both hardware and software, however, is the fact that, sometimes, an emulation is not able to understand some instructions, in particular undocumented or fairly uncommon ones[56] and, by introducing software emulation (e.g., operating system) to the hardware emulation, we also introduce more chances of having such failures in emulation.

One big challenge in producing emulators for malware detection, especially in the research setting, is that there are a vast number of possible hardware and software characteristics that need to be emulated in order to create an environment able to run suspicious code and identify its malicious features.

Making this challenge even harder is the fact that most of the research on emulation for malware detection happens within anti-virus companies, thus new and innovative methods are usually not publicly (or academically) available, being published just as

[2]A feature that helps on this and that are present in some virtual machines is the capability of using "snapshots". Users can save states of the machine, whatever is in memory and in the processor registers, and eventually return to them, continuing the execution from that point.

[3]An environment where programs can be executed without causing harm to the hosting system, regardless of whether or not the harm is intentional.

patents. Sometimes those companies release information about their results, but with few exceptions, that information is usually embedded into marketing material, which makes the results not too reliable nor repeatable from the point of view of an independent researcher.

This work is motivated by the need for a framework in which researchers, academics or otherwise, can develop new techniques for dynamic malware detection and analysis and also to evaluate existing techniques in an unbiased context. As an additional benefit, we want this framework to allow not only the implementation and testing of novel heuristics in a standalone mode, but also to give the researcher a possibility to "plug" this framework into an existing product, extending its functionality.

In this thesis, we describe AGAVE, a tool designed for automatic generation of anti-virus emulators and that provides an environment for malware analysis. This environment includes hardware and operating system emulation.

AGAVE is meant to be a framework to develop and test new techniques for malware analysis and detection on end-user machines. AGAVE was not designed to be used as a primary tool for malware analysis. Nor was it designed to use all the resources that might be available in an anti-virus laboratory and that are unlikely to be found in computers used by non-experts. The general end-user, the one with little or no expertise with computers, must be kept in mind as the main beneficiary of the techniques and methods that can be developed by using AGAVE.

Like any usual code, malicious software is basically a sequence of instructions, which includes calls to libraries, that provide several resources, and also directly to the operating system, known as system calls. AGAVE is meant to be used by security researchers and, to do so, it also allows implementation of system and library calls, and even low-level instructions, which gives the researcher the ability to customize the environment.

We believe that AGAVE can be used in different scenarios. Security researchers can use AGAVE interactively for malware analysis, so they can learn how malicious software works and how it can be better prevented or detected. Security researchers can also use AGAVE as a testbed for new heuristics for malware detection. AGAVE can be used embedded into anti-malware (anti-virus) products, allowing emulation of suspicious code to detect variations of known malware, using the heuristics already included within those products or ones created in AGAVE.

1.1 AGAVE at a glance

AGAVE is a framework that produces a secure environment for emulating the execution of suspicious code. We believe that one of its most important characteristics is its flexibility, as it not only allows researchers to choose the hardware and operating system to be emulated, but also allows customization of both, as well as development of heuristics and interfaces to third-party products.

Conceptually, the researcher provides information about how the operating system works, what hardware it should emulate and also any customization that should be applied. AGAVE, then, provides an environment where the researcher can emulate suspicious code for analysis and eventual detection of malware. As shown in Figure 1.2, this environment consists of basically three components:

- OS Emulator
- Hardware Emulator
- Controller

In order to provide OS Emulation, we have to understand how an operating system works. In a usual anti-virus emulator, the researcher either implements each and every function that is called by a program or returns a default value for those that are unknown.

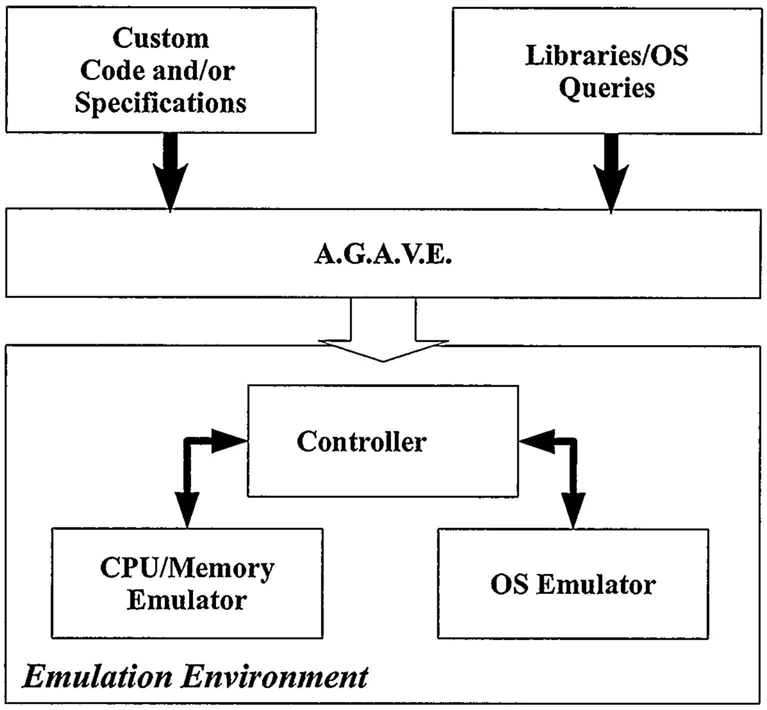


Figure 1.2: Conceptual view of AGAVE

In anti-virus laboratories, sometimes tests are interrupted due to a missing function. So the researcher should implement the function and restart the emulation. To avoid this in AGAVE, prior to the actual emulation, the researcher needs to provide information about the operating system. This information consists of traces of library and system calls, that are parsed and stored in a database. During the emulation, when a call to the operating system is requested, we use Case-Based Reasoning (CBR) to decide how the emulator will respond to it. The researcher can override and customize responses if desired. We provide details on how CBR was implemented in AGAVE in Chapter 3.

Even though it was desirable, AGAVE currently does not produce a CPU emulator due to time constraints; instead we provide an interface from which AGAVE can interact with existing products. While we used a specific emulator for this proof-of-concept (PyEmu), we developed an interface that is general enough to allow researchers to use other hardware emulators, provided some features are available, like the access to memory and CPU and also the execution of a single instruction at a time.

In Chapter 2, we explain how emulation is used within AGAVE. In Chapter 4 we demonstrate the use of AGAVE and also describe some of the problems we faced, demonstrating how to solve these problems.

The Controller is responsible for the flow of execution within the emulation, directly controlling the other two components, and also for the user interface.

The user interface is text-based, where we tried to incorporate elements from both the Python IDE and GDB, a popular debugger used by developers and researchers for debugging and code analysis. Our interface, thus, allows the user to transparently enter not only usual Python commands but also commands and functions that are part of AGAVE itself, allowing access to information about CPU status and to read from and write into the emulated memory.

Researchers can develop code in Python for controlling the code execution, heuristics for detecting malware and interface to third-party products. The AGAVE Controller offers a set of functions (also described in Chapter 2) to help the researcher. These functions, from the programmer's point of view, seem like built-in functions and commands.

Finally, in Chapter 5, we present our conclusions and also suggest some future work.

Chapter 2

Overview

As with many technological developments (artificial satellites, space stations, gene therapy, etc.), the idea of computer viruses was first introduced by science fiction novelists[1]. However, what seemed to be just science fiction now affects our everyday lives.

The idea of computer viruses and how to defend against them was first academically[2] discussed by Fred Cohen[15] and since then, computer viruses, and malware in general, have quickly become a prevalent threat for computers and other devices, such as cellphones or PDA's. The good news is that anti-virus technologies have also advanced at a similar pace, as noticed and discussed by Nachenberg[45].

One approach for detecting malware is the use of static analysis, in particular, signature scanning. This technique consists of looking for specific sequences of characters in files that would indicate the presence of malware.

While rather efficient for a limited number of signatures, both its efficiency and its efficacy are compromised by the increasing number of signatures that are constantly added to the list. In fact, thousands of new signatures are found every day[3]. (Some of these new signatures, however, do not describe new threats, but previously known ones.)

Detection by signatures can be avoided by many ways. Malicious code can be hidden, for example, encrypted, inside an executable. Then, only when the infected executable is running is the malicious code decrypted and executed. Another common practice,

[1]In 1970, Gregory Benford published "The Scarred Man" and, in 1972, David Gerrold published "When Harlie was One", stories that mention not only a computer virus, but also a program to defeat it, thus mentioning "anti-viruses" as well.[6]

[2]Some researchers also refer to John Von Neumann's work "Theory of self-reproducing automata" as the first discussion about computer viruses, e.g., [62].

[3]According to Symantec's website, 259246 new signatures were added to in their "certified" list of signatures during October 2010, an average of over 8,000 new signatures daily. Source: `http://www.symantec.com/business/security_response/definitions/certified/index.jsp`.

inspired by biological mutations, is self-mutating code, by using techniques such as polymorphism and metamorphism. With these techniques, every instance of a program can potentially be different from previous ones, thus having different signatures[46, 59].

Dynamic methods of analysis and detection have, then, been developed not only as an alternative, but as a complement to static approaches. For example, we can observe the behavior of a program (the sequence of actions the program performs) to identify suspicious activities.

In this chapter, we are going to discuss one dynamic approach: the use of emulation for malware detection and analysis. First, we will review what emulation is and some of the ways emulation can be used when working with malware, then we give an overview of AGAVE.

2.1 Code emulation

The idea of emulation can be traced back to the 1960's, originally used to make the new and smaller systems (still mainframes at that time) behave like the large and old ones[63]. The objective was keeping existing programs running without adaptation to the new equipment, that was usually not compatible with its predecessors. The basic idea was providing a scenario that programs would recognize as their actual environment and would be able to run normally.

Emulation allows us to run code written for a specific processor in a different type of machine, or code written for a specific operating system can be executed in a different one, sometimes with the entire environment (CPU, memory, devices) also being emulated. This type of emulation is usually called "virtualization", as the main motivation is basically the original motivation for emulation, the creation of a "virtual machine". The virtual machine can be used to execute code that would be incompatible with the available equipment. In this type of application, low-level instructions can be either emulated or translated to the host machine.

Tools as QEMU[10] and Bochs[40] are some of the examples that allow the emulation of a machine within another operating system. Xen[7] and VMWare[66] can go even further, providing the emulation of several machines in one single system, ideal for hosting services.

Virtualization implies emulating an entire environment and in recent years has increased in complexity as the users request new features[35], thus also draining more resources from the host computer. However, virtualization is not suitable for malware detection. Malware detection, especially when performed by anti-virus products, should be efficient and with as little overhead as possible.

However, some researchers have successfully used virtualization tools for malware analysis (e.g., [9], [33]). The environment can be monitored, allowing capture of library calls or detection of changes in registers, memory and stack. Differences from the machine status before and after malware execution can be collected for comparison, so the researcher can understand how the malware works, how it infects the machine and maybe even how to remove it safely.

Virtualization is not the only way of using emulation to fight malware. The importance of emulation in the fight against malware is that emulation provides a way of safely observing what suspicious code is doing. This safety comes from the fact that the environment that runs within the emulation is completely different and isolated from the host.

Sometimes, the idea of emulation being an isolated environment can be confused with sandboxes, but these are different concepts. In a sandbox[51] the code runs in the actual machine, but in a protected and fairly isolated environment, that, ideally, would not affect other programs in the same computer. However, some resources are still shared with other applications (e.g., processor, memory, disk) and, sometimes, they can be used to provoke system interruption. On the other hand, emulation provides an entirely isolated and fake environment, where even access to devices like disks or video has no impact on the real devices, as some of them do not even exist in reality.

We must emphasize, however, that a balance must exist between the use of static and dynamic analysis (emulation). On one hand, emulation techniques are considered more effective and less risky than static analysis, but it is also more complex and, thus, slower. On the other hand, static analysis can help emulation because of its speed and of its ability to discover anti-emulation measures[16]. Also, we must remember that, unless within an anti-virus laboratory, it is not always possible to emulate a whole program for detection. Some programs (e.g., a word processor) simply never finish, unless requested by the user. So, just a limited number of instructions must be emulated[6].

2.2 Malware detection and analysis using emulation

Emulation allows the use of a huge variety of techniques for malware detection[6]. The techniques are commonly based on generic decryption, heuristics or program behavior profiles.

As previously mentioned, static detection, especially signature scanning, has lost some of its efficacy. In part, this is due to the development of polymorphic viruses, capable of mutating from one generation to another. The common way of "mutating" is by encrypting[4] the viral code, so a previously determined signature would not be valid for that particular instance of the virus. In order to be executed, however, it becomes necessary to include into the target program what is called a "decryptor loop". The decryptor loop is responsible for decrypting the viral code (either in memory or on disk)

[4]The concept of encryption here is rather loose. Malware writers call "encryption" techniques such as a simple XOR operation or techniques of code obfuscation. However, actual application of cryptographic algorithms may happen sometimes.

and then executing it. A detailed description of how the code can mutate is beyond the scope of this work, but polymorphism is formally described by Filiol[22] and a more practical approach can be found in [62].

In generic decryption[48], execution of a suspicious program is emulated, looking for signatures (sequence of characters) in the emulated memory to detect when malicious code has been decrypted. In our experiments, we have implemented a simple variation of this approach, as can be seen in Chapter 4.

Heuristics based approaches rely on a set of rules used to define what should be considered a threat[3]. Those rules can help to identify malicious characteristics in unknown programs, for which signature based detection would fail. Heuristics can be used to select events. Sequences of these events can be then compared to previously collected malicious sequences to identify malicious code[31]. We can also emulate a predefined number of instructions of a program, collecting operands, operators and states of registers after each instruction. Then, using a heuristic analyzer to evaluate this data, we could determine a probability of the program containing viral code[69]. Another heuristic approach involves producing histograms based on active instructions (i.e., instructions that modify memory) or on sequences of them. These histograms can be used to identify some obfuscated malicious code[47].

A program's behavior profile can be described as a sequence of actions it performs. Examples of actions considered when building a behavior profile are: a process creation or termination, file opening or deleting, making a network connection. In practice, system calls made by the program are collected to build its behavior profile[23]. One approach is monitoring and storing the behavior of a program when it is first executed. Every time that program is updated, its behavior is again monitored and compared to the original one. Any differences can be then compared to behaviors of known malicious software for identification[64].

Using a different approach, we can emulate code, building a model that characterizes its behavior. The model describes information flow between system calls. The model is compared to a database of previously recorded malicious behaviors (similar to a signature database)[38].

It is also important to keep in mind that emulation does not need to be done entirely by software. Hardware-virtualization extensions, as provided by Intel-VT, may be used to analyze malware, while the analyzer remains transparent and cannot be detected by the malware[17].

A slightly different use of emulation for malware detection is in Network Intrusion Detection Systems (NIDS) or Network Intrusion Prevention System (NIPS)[41]. In this type of application, code is still emulated to identify the presence of malware, but now the suspicious code comes from network devices instead of coming from disk or memory.

For example, a CPU emulator can try to emulate code directly from network streams looking for signs of buffer overflow attacks or attempts of shell-code injection[27]. In order to find attempts of shell-code injection, another approach is the use of "forensics shellcode" to monitor operations[54].

Virtual machine inputs can also be logged and replayed on a separate analysis platform, so input and logs can be used for intrusion detection, bug finding or, in the worst case, forensics[14, 18].

2.3 Anti-virus emulators

"Anti-virus emulator" refers to the use of emulation techniques specifically for malware detection. Note that they are different from traditional emulators, whose main goal is the ability to execute code from one platform on a different one, thus correct and complete code execution is demanded. Platforms, in this case, might refer to different processor architectures or simply to different operating systems.

Code executed by an anti-virus emulator may not have the same results as if executed by another type of emulation or in a real environment, so the problem posed by Martignoni *et al.*[42](which presents a fuzzing-based methodology used to test four different CPU emulators and that found errors in all of them, some even preventing proper execution), is not really a concern when thinking about emulators for malware detection. The anti-virus emulator only requires that the emulation be just accurate enough to keep malware running.

Aycock[6, pp.75-77] describes an anti-virus emulator as having five conceptual parts:

1. CPU emulation: that interprets and emulates the execution of instructions.

2. Memory emulation: used for mapping and controlling emulated memory allocation.

3. Hardware and operating system (OS) emulation: Actual OSs are not actually used in anti-virus emulation. Among reasons, Aycock cites startup time, size, licensing issues and specific monitoring capabilities.

4. Emulation controller: responsible for controlling the flow of the emulation, when the emulation should stop and what to do if something is detected.

5. Modules for extra analysis: allowing the researcher to deal with special cases and also to include new approaches of malware analysis and detection in the emulator.

When talking about anti-virus emulators, we must mention that there are, at least, two types of applications for anti-virus emulators. They can be used in an anti-virus laboratory, as a tool for malware detection, for testing detection and for malware analysis. It also can be embedded in anti-virus products on user machines, providing a more powerful anti-virus approach than static signature detection.

This thesis focuses on anti-virus emulators. We visualize the anti-virus emulator as a way of providing a scenario to the malware, a representation of the environment it would find when being executed in an actual machine. However no access to any actual device neither to any real application nor files should be made available to the suspicious program.

Unfortunately, there is not much academic work related to anti-virus emulators. The research on anti-virus emulators is usually done within anti-virus or anti-malware companies. Thus, most of the available discussion and results in this area can only be found in the form of patents (including patent applications), or in "white-papers" (with the actual data surrounded by or mixed with marketing information).

AGAVE intends to be a tool to help researchers, especially from academia, to become more familiar with techniques and methods used to dynamically detect malware. It can also be used as a testbed for new methods, allowing the researcher to "plug-in" their new methods into existing products.

2.4 Describing AGAVE

AGAVE, our tool for automatic generation of anti-virus emulators, is designed to be used by malware researchers. The ultimate goal of AGAVE is that, for a given pair $(Hardware, OperatingSystem)$, AGAVE should produce an anti-virus emulator that can be used either for malware analysis or for malware detection.

AGAVE is intended to be used both in a "standalone" mode, where a researcher can interactively analyse suspicious code, or embedded in a third-party tool, such as an anti-virus product, so new heuristics can be developed and tested.

Currently, AGAVE is implemented as shown in Figure 2.1. A third-party emulator (1) is responsible for CPU and memory emulation, as described by Aycock[6]. AGAVE

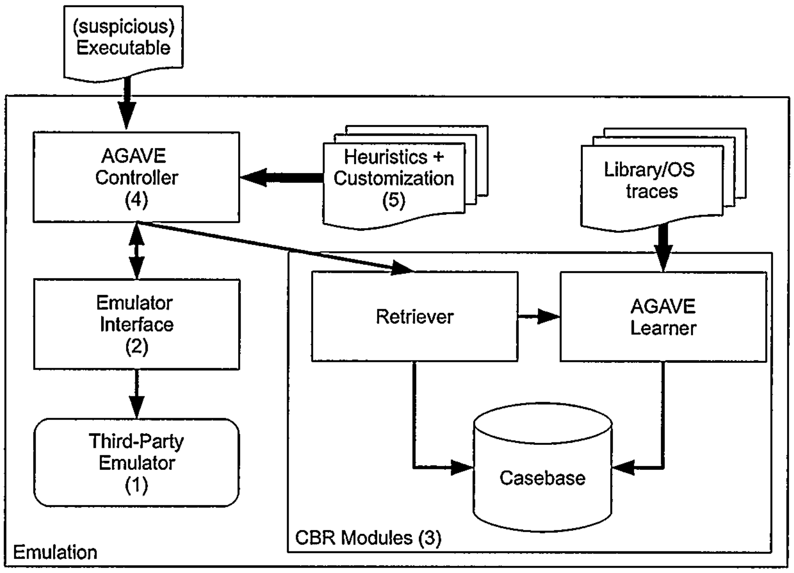


Figure 2.1: Overview of AGAVE

accesses this third-party emulator by an interface (2). A skeleton for writing a new interface is provided and we currently have a working implementation of an interface to PyEmu[5], an x86 CPU emulator written in Python.

Hardware and OS emulation is provided by the CBR modules (3), which are responsible for generating responses to system and library calls. Some functions, however, must be implemented by the researcher, in order to allow correct emulation and, consequently, a correct analysis.

Between the CPU emulator and the CBR modules, we have the AGAVE Controller (4), which is responsible for loading the executable and controlling the flow of execution and the access to OS functions, either by calling implemented functions or by doing requests to the CBR modules. The CBR functions are accessed through the CBR retriever, a module specialized for responding to queries. Heuristics to control the flow of execution

[5]PyEmu Revision r19

and also call extra modules for analysis, mentioned by Aycock, can be implemented by the researcher and called by AGAVE as necessary (5).

The remainder of this chapter discusses CPU and memory emulation and also the AGAVE Controller. A discussion about the CBR Modules is in Chapter 3.

2.5 CPU and memory emulation

Writing a CPU emulator from scratch would obviously take a really long time. Due to time constraints, we decided to use a tool that is already available.

Some characteristics, however, should be present in the product we use:

- It should be preferably open source. Not only would this help to understand how the CPU emulator works, but also it could be necessary to make minor changes, in order to allow an interface to our code[6].
- It should be well documented, both in terms of user manual and also code documentation.
- It should allow easy access to registers and memory, thus allowing the researcher to read and modify them as necessary.

Its fairly good documentation and the fact that it was written in Python, a programming language that was familiar to us, made PyEmu our choice of CPU emulator for this proof-of-concept. Even when the documentation was not complete, the code was clear enough to provide a good understanding.

Our familiarity with Python even allowed us to easily solve some problems that happened because of this specific emulator. We had several interruptions in our experiments,

[6]It turned out that we did have to make small changes, due to some bugs. Bugs that also motivated another feature for AGAVE: Custom implementation for low-level instructions.

especially when trying to emulate the execution of statically compiled code, because of some instructions that were missing. We also found a bug in the PyEmu implementation of the instruction SCASD, which caused us another interruption.

Initially, we added the missing instructions to the emulator code. However, missing instructions were a possibility for any third-party emulator. This motivated us to allow AGAVE's users to implement instructions. This new feature not only helped code to run to its completion, it also gave the user the ability to execute custom code at an instruction level, which can be used, for example, to set up triggers when a specific instruction is called.

A major disadvantage of our choice was definitely speed. An actual processor can execute millions of instructions per second. Our implementation, however, has that speed reduced to approximately 1,000 instructions per second. We believe that it was mainly caused by the fact that both our controller and the emulator are implemented in Python, an interpreted language. Also, the instructions are interpreted sequentially. We have not implemented any sort of optimization or parallelism for the emulation[7].

2.5.1 CPU emulator interface

AGAVE currently uses PyEmu for CPU and memory emulation. A researcher might want to use a different emulator. In order to do this, it is necessary to implement an interface to access the new emulator.

The new emulator should provide:

1. Means to allow interaction with third-party tools (e.g., ability to control the emulator by commands sent by a socket).

[7]There was, however, some optimization in our Python code. We were actually able to increase the processing speed by around 65%. Our original implementation could only interpret around 600 instructions per second.

2. Ideally, a minimum set of functions, that includes means to retrieve information such as registers, stack and memory.

3. Translation instructions from op-codes to mnemonics is also desirable.

To implement an interface to a new emulator, the researcher must create a class derived from class AGAVE and then implement the methods necessary for accessing the emulator. Figure 2.2 shows an example of code necessary to start an implementation of a new interface.

```
from agave import *

class Agave2MyEmulator(AGAVE):

# (...) Defining methods
```

Figure 2.2: Starting implementation of an interface to a new emulator

It is also necessary to implement the methods to access information from that emulator. These methods include loading code into memory and initialization, and reading and writing of memory and registers[8].

Some methods depend on the emulator being used and some methods depend on the type of executable being used (so it can be correctly parsed). Table 2.1 describes the methods that must be implemented and that are emulator specific. Table 2.2 describes the executable specific methods that must be implemented. We have developed a loader for ELF files that can be used. In this case, the executable-specific methods are already implemented. Finally, Table 2.3 describes methods that are not necessary for emulation, but that might provide extra information about the emulator (such as the status of the stack or a list of registers) or extra features (such as snapshots). These extra methods, if implemented, might also be useful for implementing heuristics or monitoring tools.

[8]From this point, any reference to the terms "memory" or "registers" refers to "emulated memory"

Method	Description
get_context()	Returns the CPU context (registers and flags) in a Python dictionary with pairs (register, value). In this case, a register is a key of the Python dictionary.
get_instruction()	Returns the mnemonic for the current instruction.
get_instruction_pointer()	Returns the value of the instruction pointer.
get_memory(*address*, [*n*])	Read *n* bytes from *address* in memory. If *n* is not supplied, reads a word (4 bytes).
get_register(*register*)	Returns the value stored in *register*.
get_stack_pointer()	Returns the value of the stack pointer.
set_instruction_pointer(*address*)	Sets the instruction pointer to *address*.
set_memory(*address*, *value*, [*n*])	Save *n* bytes of *value* in *address* of memory. If *n* is not given, save the length of *value*.
set_register(*register*,*value*)	Stores *value* into *register*.
set_return_code(*value*)	Sets return code for a function that was emulated.
set_stack_pointer(*address*)	Sets the stack pointer to *address*.
skip(*n*)	Skips *n* bytes. Equivalent to adding *n* to the instruction pointer. (Useful for when an instruction is implemented by the researcher.)
step()	Executes a single instruction.

Table 2.1: Mandatory emulator-specific methods to be implemented for interface

Method	Description
load(*filename*)	Loads the executable *filename* into memory. (It might be necessary to parse the executable, for example, to find the entry point.)
check_library_call(*instruction*)	Check if a library call was made and, if necessary, invokes the controller to either emulate or execute it.
check_system_call(*instruction*)	Similar to the previous, but for system calls.
check_custom_instructions (*instruction*)	Similar to the previous, but for custom instructions.

Table 2.2: Mandatory executable-specific methods to be implemented for interface

Method	Description
load_snapshot([filename])	Load snapshot. If filename is not given, reads the last one with auto-generated name.
save_snapshot([filename])	Save snapshot. If filename is not supplied, automatically generates a filename using sequential numbers.
get_memory_pages()	Return a list of allocated memory pages.
registers()	Print list of registers for debugging purposes.
stack()	Print stack for debugging purposes.

Table 2.3: Optional methods to be implemented for interface

It must be noted that we have special functions to deal with the "instruction pointer" and the "stack pointer", as we do not assume which register is used for each case. We do assume, however, that every architecture must have an instruction pointer and a stack pointer, thus the emulator should allow us to read them.

The researcher must also define two variables within the interface.

1. SYSCALL_INSTRUCTION: a mnemonic for the instruction that executes a system call. For Linux on x86, for example, we use "int 0x80".

2. LIBCALL_INSTRUCTION: a mnemonic for the instruction that executes a library call. For an ELF file on x86, we have used "call".

These variables do not need to store the entire mnemonic (including operands), as long as the methods that implement the system and library calls are capable of dealing with this. For example, for dealing with library calls, we look for an instruction "call". If the address does not point to an address in the ELF relocation table, we perform the call, otherwise, we let AGAVE controller decide what to do. (If the function was implemented by the researcher, call the implementation, otherwise, AGAVE controller calls the CBR

or "emulated registers", respectively.

retriever[9] to emulate that function.)

As an example of implementing an emulator interface, defining the aforementioned methods and variables, Appendix C shows our implementation of an interface to PyEmu.

2.6 Controller

As we previously mentioned, the controller is responsible for the flow of execution (emulation). It is implemented as two components: An *interpreter* and the actual *controller*. The *interpreter* is responsible for executing user-implemented code[10]. The *controller* component is responsible for calling the emulator (through the interface) to execute instructions, calling the interpreter to execute the user's code and calling the CBR retriever to emulate library and system calls that are not implemented.

To emulate an executable, we have a loop that reads one instruction from the executable at a time. The pseudocode for the loop is described in Figure 2.3.

The `pre_execution()` and `post_execution()` functions call user defined code, as specified in AGAVE configuration files (described in Chapter 4). This user defined code can be heuristics for malware detection, monitoring tools or any other code the user might want to execute prior to or after an instruction is emulated.

The emulation, by default, starts with batch mode enabled, i.e., the emulator will emulate all the instructions without user intervention. The batch mode can be disabled by a break point, previously defined by the researcher or by code implemented by the researcher, by calling the AGAVE command `pause()`. The batch mode can be re-enabled by the AGAVE command `run()`.

In order to redirect a call or instruction, the user must first define the function, in Python, that is going to implement that call or instruction. This function must deal with

[9]CBR module responsible for searches.

[10]We assume the user is a security researcher, thus with some programming and malware analysis background.

```
enable batch mode

initial_checking()

while not finished:

    if not in batch mode:
       call interpreter for user input

    read instruction from emulator

    pre_execution()

    if instruction is system call:

       if system call is implemented:
          call interpreter for system call
       else:
          call CBR retriever for system call

    else if instruction is library call:

       if library call is implemented:
          call interpreter for library call
       else:
          call CBR retriever for library call

    else if instruction is in list of custom instructions:

         call interpreter for instruction

    else:

         executes instruction using emulator    ## Method step()

    post_execution()

    if break point is reached:
       disables batch mode
```

Figure 2.3: Pseudo-code for the main emulation loop.

issues like reading parameters (either from stack or from registers) and returning a result (usually by calling `set_return_value()`).

A call to `set_system_call()`, `set_library_call()` or `set_custom_instruction()` should then be done to bind the defined function to the call or instruction. `set_system_call()` and `set_library_call()` receive two parameters. The first is the function name; system calls are prepended with "SYS_". The second is a call to the defined function. Additionally to these two parameters, `set_custom_instruction()` receives a third one, which is the number of bytes the instruction pointer will be incremented[11].

For example, a redefinition of the instruction LODSD[12] can be found in Figure 2.4.

```
# Description of LODSD from: http://faydoc.tripod.com/cpu/lodsd.htm
def LODSD():
        interface.log.debug("Executing custom instruction LODSD")
        esi = get_register("ESI")
        df = interface.emu.get_register("DF")
        addr =  esi
        data = get_memory(addr, 4)
        interface.log.debug( "\tMoving \%08x(\%08x) to EAX"\%(addr, data))
        set_register("EAX",data)
        if df == 0:
                set_register("ESI",esi+4)
        else:
                set_register("ESI",esi-4)

set_custom_instruction("lodsd", "LODSD()", 1)
```

Figure 2.4: Redefining x86 instruction LODSD.

A malware detection heuristic is also implemented as a Python function. The main difference here is where the function is going to be evaluated. Our options are calling the heuristic at the beginning of the emulation, before an instruction is emulated or after its

[11]In some cases, it turned out be necessary to increment the instruction pointer inside the implementation, as the number of bytes would depend on the op-code and not on the mnemonic. In these cases, the third parameter should receive 0 (zero).

[12]x86 instruction "LOAD STRING". Its implementation was actually necessary to our tests, as it was not implemented by PyEmu.

emulation (respectively, `initial_checking()`, `pre_execution()` or `post_execution()` steps). This is defined by the AGAVE configuration and we describe it in Chapter 4.

2.7 Related work

Several tools were recently developed that use emulation for malware analysis. Those tools provide services like unpacking, disassembling, tracing and others. One of them can even be accessed via the Internet.

Probably the work closest to AGAVE is PyEmu[52], the very emulator we use in our proof of concept. PyEmu was designed as a tool for malware analysis. AGAVE, on the other hand, can be used for malware analysis but also for malware detection, embedded within an anti-virus product. While PyEmu is designed to support other operating systems, it is only able to work with Windows PE executables[13]. Different from AGAVE, that uses CBR for emulating operating system functions, PyEmu uses actual Windows libraries (DLLs) for emulating code.

Similar to PyEmu, in the sense that the code runs in an emulated operating system environment, with Windows API and native system calls being monitored, in order to provide understanding of the program's behavior, *TTAnalyze*[9] was developed for dynamic analysis of Windows executables.

A similar approach is also found in *Anubis*[8]. Anubis is a platform for dynamic malware analysis, on which binaries are submitted by a web interface and then emulated. Anubis monitors Windows API calls, system services, network traffic and data flow, resulting in a report of the activities.

AGAVE uses traces of execution as input to its learning system. Program execution traces have also been used to improve emulations[33]. A system compares traces from

[13]There are references within the PyEmu source code to emulating a Linux environment, however there is no actual code developed for emulating Linux.

emulated execution of a suspicious program to traces of the same program executed on actual hardware. The emulator then is automatically corrected to a better approximation of the actual platform. A variation of this approach is found in *Renovo*[32], where, instead of using traces, an executable runs in an emulated environment and is monitored by an "Execution monitor". The same code is also monitored outside the emulated environment and any difference between the two executions might indicate some obfuscation or anti-emulation technique is being applied, so the code is extracted for future analysis..

2.8 Summary

In this chapter, we discussed the importance of emulation for malware analysis and detection. We also presented an overview of AGAVE, a tool that we have developed for both malware analysis and also as a testbed for research on malware detection.

AGAVE uses a CPU and memory emulator for interpreting low-level instructions. While our proof-of-concept makes use of the open-source emulator PyEmu, researchers can customize AGAVE to use the emulator of their choice. For emulating operating system functions and library calls, AGAVE relies on a machine learning technique called CBR (Case Based Reasoning), that will be discussed in depth in Chapter 3.

AGAVE's controller is the component responsible for controlling the flow of execution during the emulation. The controller is also capable of interpreting Python code, so researchers can customize OS calls and even CPU instructions as necessary. Examples of custom code are presented in Chapter 4.

Chapter 3

Case-Based Reasoning within AGAVE

In order to emulate a program's execution, it is necessary to understand how the program interacts with the environment, i.e., the operating system (OS) that the program is meant to work in.

The interaction between a program and the operating system can be loosely described by Figure 3.1. A program consists of instructions that can call functions (1) from the operating system's library, a collection of files that contain code and data and that are shared by different programs. Those functions from the operating system's library, after executing some tasks, which include executing other library functions(2) or requesting for a low level operation from the operating system kernel(3), return to the calling function a result that indicates what was done(4). This result can have different meanings: a memory address, the result of an arithmetic operation or even an error code. A program can also directly request the kernel to execute operations(5).

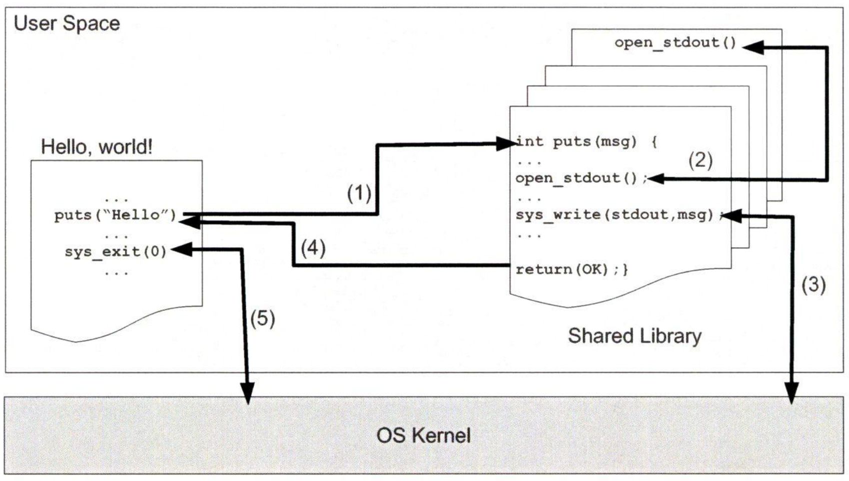


Figure 3.1: Code execution with shared library and system calls

However, according to our observations, those direct requests are not really common for legitimate programs. Legitimate programs usually rely on the shared libraries, for those contain error checking and other operations that might be necessary for correct operation.

In this work, we will refer to the execution of a function within the operating system library as a "library call" and the request for a low-level operation within the kernel as a "system call". System calls do not necessarily need to be executed by library functions, but can also be directly executed by any program.

Our system needs to be capable of inferring how to respond to a call for a library or OS function. This allows a program to run, in our context a suspicious program, as smoothly as possible, without anything unexpected happening.

It must be noted, however, that some calls cannot be learned automatically, due to the obvious complexity of some code. Other calls will require an implementation by the user (that we assume to be a security researcher, with sufficient background for such task), as some basic functionalities must be present to allow actual and effective malware detection, which is one of the main goals of AGAVE. As an example of such required functionalities, we can mention I/O operations, which are essential to allow static-based analysis such as signature detection. The specifics about how a researcher can implement code within AGAVE will be discussed later in this thesis.

In this work, we have used a machine learning technique called "Case-Based Reasoning" or CBR, that applies knowledge from previous experiences to the decision making process. This is analogous to judicial trials, where a judge takes into consideration previous cases in order to decide whether defendants are guilty or innocent.

This chapter is organized as follows: Section 3.1 presents some background on CBR, Section 3.2 describes how cases are defined in AGAVE, followed by a discussion about implementation in Sections 3.3 and 3.4. Finally, in Section 3.5 we evaluate our implementation of CBR.

3.1 Basic concepts

Kolodner[39, p.13] formally defines a *case* as "a contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner". An important keyword in this definition is "contextualized". The context of each case must be clearly defined, otherwise CBR would not be a successful approach.

Some examples of the application of CBR [2] include:

- A physician that is able to diagnose a disease, by observing relevant symptoms. Or that prescribes treatment for a patient, taking into account the patient's history, that includes allergies and reactions to certain medications; or
- A financial analyst that, by observing the economic situation of a certain company and comparing to what has happened to other companies in the same (or quite similar) situation, is able to decide whether or not to recommend if a loan application should be approved.

Note that, in both examples, previously unseen events can happen: Patients can have allergies to a prescribed medication without either patient or physician being aware of it; a new economic crisis can affect the entire world. Even for these situations, each one in its own domain, some previous knowledge is necessary to form a decision. Depending on the situation, the decision process can be rather complicated and retaining information both about the problem and about the decision can be important to help future decisions.

Thus, for a specific context, each one of the situations that we have some knowledge about and also the situation that is being evaluated are considered "cases". The collection of cases we know is usually called a "knowledge base" or a "case base".

But, simply retaining cases, having a large casebase, does not help much if we do not know how to relate those cases to the one we are evaluating.

One of the most important concepts in CBR is the idea of similarity[12]. Defining how to compare two different entities (whatever these two entities are), in order to measure objectively how similar they are, is a key component when designing a CBR system.

Directly related to similarity is the idea of distance. The distance indicates how different two entities are. So, it is said that two entities are close, if they are, up to some level and in a specific context, similar. On the other hand, the distance between two entities is high, i.e., they are far away from each other, if they are really different.

Similarity (and distance) within AGAVE is discussed in Section 3.2.3.

3.1.1 The CBR cycle

Solving a problem using CBR involves four basic steps, or processes, as described by Aamodt[2], that is also referred to as "the four REs":

1. Retrieving: search in the knowledge base for a case or cases that are most similar to the situation to be solved.

2. Reusing: apply the retrieved information to the current problem. This can vary from simply copying the exact same solution as it is, in the knowledge base, to modifying it, adapting the solution to the new problem.

3. Revising: evaluate whether or not the proposed solution is valid for the problem in question.

4. Retaining: add what was learned to the knowledge base, so it can be used for later cases.

We used this cycle as a base for developing our implementation of CBR for AGAVE, as described in Section 3.4.

3.2 Modelling cases

Our first task to implement CBR in AGAVE is defining what we call a "case" in our context.

To allow code emulation, besides emulating low-level instruction execution, which is already provided by the CPU emulator, we need to emulate library and system calls, thus emulating the presence of an OS. Those calls can be described, in their simplest form, as functions that receive parameters as input and return some value as output.

Some functions, in addition to returning a value as output, may also use another mechanism to return information. These functions receive, as parameter, an address pointing to a position in memory. The function can, then, write information there. This mechanism is mainly used for returning structures, objects or strings. Our implementation of CBR for AGAVE does not deal directly with this form of return. However, AGAVE allows researchers to implement functions when necessary or desired. In fact, if a function that uses parameters to return data is important for analysis, it should, ideally, be implemented by the researcher. This implementation would ensure a more precise analysis.

In the remainder of this section, we describe how we collected data and what decisions were made for modelling our data in cases that could be used for our CBR implementation.

3.2.1 Collecting data

As a source of data, we decided to use library and system traces. In a Linux environment, which was chosen to be our proof-of-concept, these data can be collected by using the "`ltrace`" utility[1].

As we wanted as much information as possible, we used some options provided by

[1]There are tools for gathering the same type of information for other operating systems as well.

```
8417 __errno_location()                                  = 0xb76f2898 <0.000036>
8417 malloc(128 <unfinished ...>
8417 SYS_brk(NULL)                                       = 0x08934000 <0.000013>
8417 SYS_brk(0x08955000)                                 = 0x08955000 <0.000013>
8417 <... malloc resumed> )                              = 0x08934008 <0.000208>
8417 gethostname( <unfinished ...>
8417 SYS_uname(0xbf8cbcc6)                               = 0 <0.000011>
8417 <... gethostname resumed> "agave_host", 128)        = 0 <0.000081>
```

Figure 3.2: Sample of ltrace output

ltrace. For collecting information about a program, we used the following command:

```
ltrace -f -T -S -s 1024 -A 64 -o EXEC.ltrace EXEC
```

where:

- *-f*: traces child processes.
- *-T*: shows the time spent in each call.
- *-S*: also lists system calls. System calls will always be indicated by the prefix "SYS_".
- *-s 1024*: maximum number of characters in strings.
- *-A 64*: maximum number of elements in an array to be displayed.
- *-o*: writes ltrace output into a file.
- *EXEC*: this is the name of the executable being traced. The same name, with the extension ".ltrace" appended, is also used for the output.

A sample of an actual output can be seen in Figure 3.2. A close look at this sample shows us a rather interesting situation: a call to `malloc` is shown as "unfinished", two system calls are executed internally to `malloc`, and `malloc` is finally resumed. Only then does the execution time for `malloc` become available.

That situation shows the importance of having the process id (PID)[2] for parsing. When a process forks, both parent and child process share the same trace file, and the PID helps us to discern which instruction belongs to each process, especially in the case of "unfinished" calls.

Unfinished calls can also happen when a signal is sent to the process or when the process pauses to allow the execution of its parent or child.

In our very first implementation in order to evaluate the viability of using CBR for our problem, we decided to work only with system calls (which is a reduced set of functions).

In that implementation we also included a sequence number in our data. This sequence number indicated when in the execution each call was made. The idea was that some functions would return different values depending on where in the code they were called. That is specifically true for functions that deal with I/O operations. For instance, a call to open, when successful, returns a "file descriptor", a number that represents the file opened. This number is usually sequential. So, if there was a call to open, it would probably return a lower number if it happened in the beginning of the code, and a higher number if in more advanced stages of execution.

However, it must be mentioned that this sequence number is no longer part of our data, as its main benefit was increasing accuracy for I/O related calls. As I/O related calls are usually essential for malware detection, they should be preferably implemented by the researcher within AGAVE. The experiments discussed below do not include a sequence number.

The traces, as we have shown in Figure 3.2, have given us an obvious framework for describing a function call. A call consists of:

- a function name, a string that identifies the function and that can be considered a

[2]Process id: a number that is given by the operating system to each process that is executed. In our example (Figure 3.2), the PID is 8417.

short description of that function's purpose;

- a list of parameters (shown between brackets) passed to that function, that can be empty, i.e., some functions might not require any parameters;
- a solution (shown just after the equal sign); and
- an execution time (shown at the end of the line, between angle brackets).

There are, however, other features that are not so obvious, as to define if it is a library or a system call (`ltrace` renames system calls by adding "SYS_" to their names) and a sense of dependency, i.e., what system calls, or library calls, are requested inside another function call. Currently, we do not use this dependency information in our implementation.

3.2.2 From "Calls" to cases

In our earliest experiments, our cases were identified by the function name and for each value passed as a parameter. While that would provide a high level of accuracy[3] (sometimes close to 100%), it was also generating a huge casebase, and search operations tended to be extremely slow. In the worst case, a search for a call would last up to 5 minutes for a knowledge base of around 19,000 cases.

That search time may seem rather surprising, but we must keep in mind that, when using CBR, the searches in our knowledge base are quite different from a search in an usual database (say, a relational database). We are not looking for an exact match, but for similar cases, so our query does not have parameters that can be optimized in some sort of index to speed up the search. This implies a sequential search, thus, quite slow

[3]Accuracy refers to how close the result we found is to the result we would be expecting. Strings were compared by using n-grams and numbers by dividing the lower number by the higher and finding a percent value. Values close to 1.0 would represent similar values.

and computationally expensive. Some measures can be done to avoid a search of the entire casebase, and these measures will be discussed later in the chapter .

One idea to improve performance was grouping cases; fewer cases, less to search. Instead of identifying cases by function name and parameters, we had function names and types of parameters. For instance, $f_1(1, 2)$ and $f_1(2, 3)$, instead of two cases, would now be considered a single case $f_1(\mathit{numeric}, \mathit{numeric})$. Our search operation would give us the most frequent returning value for that case as a result.

We also wanted to consider some special cases. Some values tend to be more frequent than others. Values of 0 (that may also mean "False"), 1 (that usually indicates "True") and -1 (which is a common value for errors), were also designated their own "type". By doing this, we reduced significantly the size of our casebase from around 19,000 to about 300 cases. The significant reduction, however was also due to the fact that by then we were only working with system calls, that were not more than 200[4]. This also brought us an important loss of accuracy, from around 85-95% to around 60-70%.

We had this clear trade-off to consider: granularity (number of cases) versus accuracy. And, ideally, we needed to find a balance. We then decided that, when defining the case, we would keep types for non-numeric parameters, such as strings, structures or arrays, and we would cluster the numeric values in ranges. For our implementation, we have created special clusters for non-numeric types (e.g., StringCluster, ArrayCluster), so we can maintain consistency, working with clusters instead of alternating types and clusters.

For defining the numeric ranges, we looked at the numeric parameters of all the calls in our knowledge base, counting the frequency of each one of them. Then, we used DBScan[20] to generate the clusters. We decided to use this specific algorithm as it does not require us to define a fixed number of clusters in advance, the cluster definition being a matter only of the frequency of each value.

[4]It must be noted that our cases take into account the parameters passed to the function, hence the higher number of cases if compared to the number of system calls.

Using DBScan, however, brought us an unexpected problem. It is a quite expensive algorithm, and, with fairly large databases (sometimes in the order of hundreds of thousands of values to compare), it may take a long time to process. The time for generating clusters was around 10 minutes for a collection of 45,000 calls and resulted in around 1700 clusters.

In order to speed up the process, we organized the frequency of all the values, ordered by the frequency and divided into 5 groups (or five quantiles, as those groups are named in statistics).[5] Figure 3.3 shows an example with the frequency of 25 different values. If the frequency of a value belonged to the last group (marked in red in our example), that value would become a cluster by itself (a cluster of one single element). Numbers that were not in one of those single-element clusters would then be given to DBScan as a range to be grouped into clusters. In our actual data, this process resulted in around 2,400 clusters created in less than one minute.

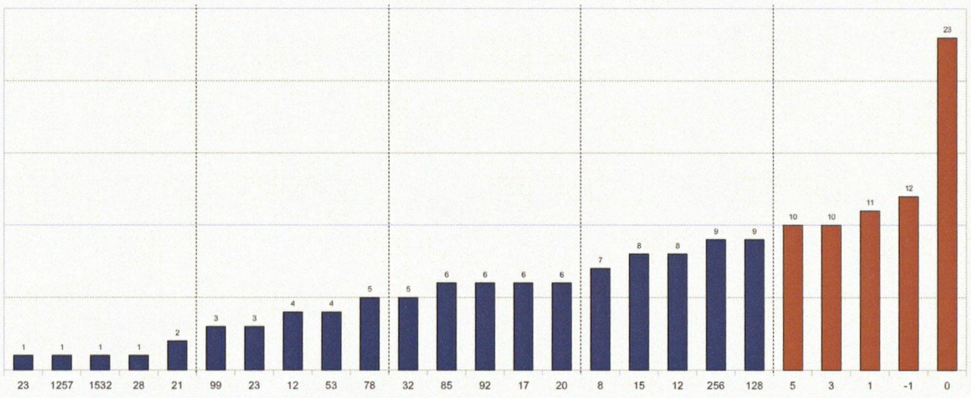


Figure 3.3: Example of clustering by frequency

With the definition of such clusters, that are represented by their limits (highest and lowest value), we can now describe what we consider cases within AGAVE. A case is

[5]We have previously tested different number of quantiles, however if the number was too small, we had clusters for almost every single value, increasing the granularity of our database and, consequently, increasing the number of cases, going back to our original problem. And, if the number of quantiles were too high, we would still have the long process of defining clusters.

defined as having the following features:

- Function name: a string that identifies a function.
- List of clusters: instead of list of parameters, we convert those parameters into clusters.
- List of possible solutions to the case being considered and the frequency of each solution: when generating our casebase, we will store all the solutions for each instance, so we can either return the most frequent solution or use some algorithm based on probability, to return solutions that reflect the behavior of the environment.

At this point, we are still not using the time of execution in our casebase, however we believe that this might be an important information to avoid some anti-emulation techniques, in particular those based on time checking[13].

3.2.3 Defining similarity

As our cases are rather simple, with not many features, we do not have many elements to compare and define similarities. In fact, during emulation of an actual call, the only information available is the name of the function being called and, sometimes, its parameters.

(For completely unknown functions, we cannot really affirm how many parameters are being passed, if any. We used some heuristics to infer that information, but it is not an accurate process at all, as some obfuscation techniques can be applied in malicious code. These heuristics are discussed later on in this section.)

In order to define the similarity, we assumed that legitimate OS developers tend to name their functions in a somewhat meaningful manner[60]. That good practice allows other developers to infer what a function does based on its name. It also means that functions with similar goals, or similar tasks, should also have similar names. So, we decided to compare function names by using 3-grams[36], which is a rather common way of comparing strings, in order to measure how similar those function names are. For our implementation, we used the package ADVAS[26] that, among other features, provides methods to compare strings using n-grams.

Concerning parameters, we can compare number, types and actual values of them. We have defined two different ways of comparing parameters. For functions that are already known by our system, we know (or at least suppose) how many parameters there are. However, for unknown functions, we need some heuristics to essentially guess them.

So, we defined a similarity measure for *general* use (when we have a function name that exists in our casebase) and a similarity measure for inferring *syntax*, which means that we are not sure about number of parameters or even their types (sometimes, what we see as a number can actually mean where in memory a string is stored, for instance).

The similarity measure is used both when generating the casebase as well as when emulating a program. When generating the casebase, it is used to find neighbors that will be removed during the "Fish and Shrink" algorithm (Figure 3.7). During the emulation, if we are to emulate a function call whose name is not in our casebase, we first use the syntax similarity to look for a function with a similar syntax and assume the number of parameters. Then we use the general similarity measure to find the best match. In both interactions the same "Fish and Shrink" algorithm is used.

Both measures take into consideration the name of functions and also, when comparing parameters, they are compared in pairs by their position in the function, e.g., the first parameter of the query is compared to the first parameter of the element in our casebase. The difference between the two types of similarity is in how the parameters are compared. For *general* similarity, we also consider the number of parameters and their clusters and for the *syntax* similarity we compare only the types of parameters. Currently, we assume that a parameter can either be a string or a number. Data structures (an array, for example), would be represented as a memory address, which we also treat simply as a number. When considering *syntax* similarity, for each parameter of the function in our casebase, we compare to see if the type is the same as the parameter the query has.

For a *general* similarity, the values of parameters are first converted into clusters[6]. If the information cannot be placed within a cluster, we look for the closest ones, i.e., those which one of the limits are numerically closer to the parameter value. The comparison then happens between clusters and not between absolute values. We consider it a match if two parameters are in the same cluster. Otherwise, we calculate the distance from one cluster to another.

The distance is a number between 0 and 1. If the clusters are not the same and at least one of them is not numeric, the distance is maximum, i.e., 1. For calculating a distance between two numeric clusters, we have a list of clusters sorted by the ranges they represent. The distance between two clusters is a value given by the difference between their positions in the list divided by the number of clusters in the list.

Finally, the accumulated parameter similarity (as the inverse of distance[7], in case of numeric clusters) is divided by the number of comparisons[8] to result in an average, that will be used to calculate the total similarity. Pseudo-code for calculating general similarity can be seen in Figure 3.4.

For syntax similarity used for unknown functions, we use a simplified process. When comparing parameters, we simply compare if their types are the same (by checking the

---

[6]As previously mentioned, strings, arrays and structures have special clusters that identify the type of information.

[7]"Inverse" using the mathematical sense of multiplicative inverse or reciprocal for a number, i.e., $similarity = \frac{1}{distance}$.

[8]The number of comparisons depends on the minimum number of parameters between two functions.

```
# Query and elements are AGAVE cases
# Query is the case we are looking for
# Element refers to a case stored in our casebase
function GeneralSimilarity(query, element):
   similarityName ← CompNgrams(query.Name, element.Name)
   numParams ← number of parameters in query
   if numParams > number of parameters in element:
      numParams ← number of parameters in element

   matches ← 0
   # Compare clusters of each parameter
   for i in 0..numParameters-1:
       q ← query.Parameters[i]
       e ← element.Parameters[i]
       if both q and e are numeric:
           if Cluster(q) is equal to Cluster(e):
               matches ← matches + 1
           else:
               distance ← ClusterDistance(Cluster(e), Cluster(q))
               matches ← matches + (1 - distance)
       else  if q and e have the same type:
           matches ← matches + 1
   similarityParameters ← matches / numParams

   # Our formula for similarity
   similarity ← (3 * similarityName + 2 * similarityParameters) / 5
   return similarity
end function
```

Figure 3.4: Pseudo-code for calculating general similarity

```
# Query and elements are AGAVE cases
# Query is the case we are looking for
# Element refers to a case stored in our casebase
function SyntaxSimilarity(query, element) :
   similarityName ← CompNgrams(query.Name, element.Name)
   numParams ← number of parameters in query
   if numParams > number of parameters in element:
      numParams ← number of parameters in element

   matches ← 0
   for i in 0..numParams-1:
       q ← query.Parameters[i]
       e ← element.Parameters[i]
       # We compare the type of each parameter, according to our heuristic
       if type(q) == type(e):
           matches ← matches + 1
   similarityParameters ← matches / numParams
   similarity ← (3 * similarityName + 2 * similarityParameters) / 5
   return similarity
end function
```

Figure 3.5: Pseudo-code for calculating "syntax" similarity

type of the clusters). The idea here is finding a function that resembles the one in our query and not to find a specific case. We want to try and infer how many parameters our query has and what their types are. So, if necessary, we will convert values (memory addresses into strings) for the actual search for the best case in our casebase.

The parameters are either read from the stack or from registers, depending on if it is a library or a system call. As we do not know how many parameters our query has, we configured AGAVE to assume it has 5 parameters. This number can be modified by the researcher, but it was initially chosen after examination of our training set. The number of calls with more than 5 parameters turned out to be quite small[9].

Finally, in order to calculate the similarity measure between two functions, we consider that, as the number of parameters and their types are less reliable information than

[9]Out of 45,000 calls, there were only 96 calls with 6 parameters, all of them to `snprintf` and only one with 7 parameters, calling `getnameinfo`.

the function name, we give more weight to how similar their names are than to their parameters' similarity. The weights as shown in Formula 3.1 were defined after some preliminary experiments and follow this rationale. However, we might note that defining how similar two strings are is still an open problem. As we so strongly rely on the function name, we might revisit our metric in the future in order to provide a more effective comparison.

$$Similarity = \frac{3 * NameSimilarity + 2 * ParameterSimilarity}{5} \tag{3.1}$$

3.3 Generating our casebase

Our process of creation of a case base can be described by the pseudo-code in Figure 3.6. AGAVE deals with system calls and library calls differently, so we decided to store them as two separate case bases.

While it is a simple and straightforward process, depending on the number of calls being evaluated, it can take quite a long time to finish, especially due to the operations that require disk access: looking for the best cluster, checking whether a case is already inserted, and updating or storing a case.

In an earlier implementation of our proof-of-concept, working with a relatively large database (around 35,000 calls), we decided to keep everything in memory, thus avoiding disk access. We then faced a surprising and unexpected problem. In some occasions, we noticed that the computer (a laptop) had restarted, but the problem was intermittent, and usually happened when we left the computer working, but unmonitored. In one of the occasions, however, we witnessed the system restarting during the process. We then noticed that the keyboard was warmer than normal. We immediately installed a temperature monitor and let the process run again. It quickly went over 85 degrees Celsius, and

```
function CasebaseGeneration():
   # Initialize casebase of system and library calls
   syscall_cb ← {}
   libcall_cb ← {}

   list_of_calls ← traces converted into calls
   # Generate clusters based on DBScan
   GenerateClusters()
   for each call in list_of_calls:
        new_case ← new Case(call)
        solution ← new_case.solution

        if new_case.names starts with ''SYS_''
            if new_case is not in syscall_cb:
                insert new_case in syscall_cb
            existing_case ← syscall_cb[new_case]
            existing_solutions ← list of solutions in existing_case
            if solution is in existing_solutions:
                increments solution_counter in existing_solutions[solution]
            else:
                inserts solution into existing_solution
                solution_counter  in existing_solutions[solution] ← 1
            update existing_case with existing_solutions
            update syscall_cb with existing_solutions
        else: # new_case is LibraryCall
            if new_case is not in libcall_cb:
                insert new_case in libcall_cb
            existing_case ← libcall_cb[new_case]
            existing_solutions ← list of solutions in existing_case
            if solution is in existing_solutions:
                increments solution_counter in existing_solutions[solution]
            else:
                inserts solution into existing_solution
                solution_counter  in existing_solutions[solution] ← 1
            update existing_case with existing_solutions
            update libcall_cb with existing_solutions
generate_neighborhoods()
end function
```

Figure 3.6: Pseudo-code for casebase generation

we decided to stop as temperatures over 100 degrees can damage the processor, according to specification[28, p.10] and the maximum component temperature should be under 105 degrees (over 125 degrees Celsius can result in permanent silicon damage[29, p.29]).

We then decided to include a check for temperature during the casebase generation: when the limit of 87 degrees Celsius was reached, a delay of 5 seconds was introduced (calling OS function `sleep()`), which allowed the processor to briefly cool down. This extra delay obviously increased the time to generate a casebase, so additional fans were also attached to the laptop, as an extra protection against high temperatures.

3.4 Applying the "4 REs"

Having a casebase to work on, we will now focus on the "4 REs" from the CBR cycle: Retrieving, Reusing, Revising and Retaining.

3.4.1 Retrieving

Retrieving a case in CBR consists of finding the most similar case to the case in question[10]. The case we eventually find will then be used to produce a solution to the case in question.

A common problem, however, is the performance of search and match operations when the number of cases are in the order of several thousands, as the comparison between cases might slow down the process to the point where it becomes unacceptable[39].

A common approach is reducing the number of comparisons, by selecting just a subset of elements of the casebase. Among the different approaches are ones using genetic-algorithms[30] and fuzzy-logic[4] for self-optimization.

For this project, however, we decided to implement Jörg Schaaf's "Fish and Shrink" algorithm[57]. This algorithm assumes that, if a case being compared to a query has

[10]The "case in question" is often referred to as a "query" in CBR.

	B	C	D
B	100%	75%	55%
C	75%	100%	80%
D	55%	80%	100%

Table 3.1: "Fish and Shrink" example: Similarities in casebase

a similarity measure too low, then other cases similar to that one will also have a low similarity to the query, so we can skip them.

We implemented this algorithm by precomputing the similarity between all the cases in our casebase, creating a graph of "neighbors" (nodes that are more similar to each other). We then perform a sequential search, but if a case of low similarity is tested, we identify all of its neighbors (or "fish" them) and eliminate them from our search space ("shrinking" it).

This conceptual graph[11] of "neighbors" must be created before the code emulation. For testing purposes, we have created several of these graphs, each one with a different pair (Number of nodes, Minimum Similarity). The number of nodes indicate how many cases are eliminated on each step of the "Fish and shrink" algorithm, thus a high number of nodes in the graph (that we sometimes call a "neighborhood") should result in a faster search. The minimum similarity, on the other hand, limits the number of neighbors. To understand why this minimum similarity matters, let us consider the following example.

Consider a new case A, and known cases B, C, D that are in our casebase. Table 3.1 shows the similarity between the cases in our casebase. Table 3.2 shows the similarity between our new case and each one of the elements in the casebase (for demonstration purposes, during emulation this would be calculated as necessary). The cells in these tables indicate similarity between the element in the row and the element in the column.

[11]In our implementation, however, the graph is stored as a list of lists. Each index represents a node and it stores a list of neighbors of that node.

	B	C	D
A	35%	85%	65%

Table 3.2: "Fish and Shrink" example: Comparing new case

If we consider a minimum similarity of 75%, we would have the following list of neighbors for each node:

$$\begin{cases} Neighbors(B) : \{C\} \\ Neighbors(C) : \{B, D\} \\ Neighbors(D) : \{C\} \end{cases} \tag{3.2}$$

When we try to perform a search for A, as the search is sequential, we would first calculate its similarity to B (finding 35% as a result). As the result is too low, it would remove all the its neighbors from the search space, i.e., it would remove C. Then we would compare A to D, finding a similarity of 65% (the best so far). With no more elements, D would be considered the "most similar" node, even though node C is more similar to A.

By increasing the minimum similarity between nodes in a graph to, in this case, 80%, we would have the following list of neighbors:

$$\begin{cases} Neighbors(B) : \{\} \\ Neighbors(C) : \{D\} \\ Neighbors(D) : \{C\} \end{cases} \tag{3.3}$$

Note that the node B does not have any neighbors now. So, the first step would, again, calculate the similarity with B, following by the similarity with C. Now that C is considered during the search and we could find a better result (85% of similarity).

So, we can conclude that the value for minimum similarity avoids that, during the search, we eliminate cases that could be otherwise considered even more similar than the current node being compared to the query. We believed that a higher required similarity, while reducing performance, would tend to increase accuracy.

We understand that this is a contrived example, but it does illustrate the importance of the minimum similarity when generating our graphs of similarity.

In order to increase performance, however, our retrieval algorithm first attempts to find an exact match to the case in question. This search for an exact match is indexed, thus virtually instantaneous. If nothing is found, then we look for similar cases using "Fish and Shrink", as described by the pseudo-code in Figure 3.7.

Currently, during initialization of the system, AGAVE checks if the configured graph is already generated, generating it on-demand if necessary.

3.4.2 Reusing

Reusing a solution can mean one of two different approaches:

1. Using the exact same solution for different instances; or

2. Adapting a previously seen solution to a new instance.

As a general rule, our search for results always returns the most frequent solution for a given case. However, there is, in our implementation, one case that involves some adaptation.

We have noticed that a common occurrence was that some functions would return, quite often, the value of one of the parameters passed. For example, the function `mmap` receives as a first parameter a memory address. If that parameter is not NULL (zero), the kernel should try to map memory around that address. While it does not happen always, in most cases when a memory address was specified, the function would return the same value, or some number close to it. So, we implemented that particular situation by creating an object called *ReturnParameter*, that would contain the index of a parameter

```
# query is the case we are searching for
# type_similarity indicates if  general or syntax should be used
# threshold is the minimum similarity
function FishAndShrink(query, type_similarity):
    if query is Syscall:
     base ← syscall_cb
    else:
     base ← libcall_cb

    list_cases ← list of keys from all cases in base

    if query is in list_cases: #Exact match
     return existing_solutions from list_cases[query]

    best_similarity ← 0
    best_case ← {}
    while list_cases is not empty:
     case ← first case on list_cases
     remove case from list_cases

     if type_similarity is ''General''
     similarity ← GeneralSimilarity(query, case)

     if best_similarity < similarity:
            best_similarity ← similarity
            best_case ← case
     else:
         if similarity <  threshold:
             for each neighbor of neighborhood(case)
                 remove neighbor from list_cases
    return existing_solutions from best_case
end function
```

Figure 3.7: Pseudo-code for casebase search

used to call that function. When reading the traces, for each time the return value matches one of the parameter, we include *ReturnParameter* as a possible solution. During the request, if that object is the most frequent solution, then it would be used as the chosen solution. If an object *ReturnParameter* is returned by the search, the solution that we will give to the caller is the value of the indicated parameter.

3.4.3 Revising

Due to the number of times CBR is called during the emulation, and as our system should be as automatic as possible, revising solutions during the emulation phase would introduce a significant delay for execution. Instead, we decided to revise solutions during the training phase.

After we generate our casebase, we start a validation process, where we read all the calls used for generation and check what solution the system returns. The solution is then compared to what was in the initial trace and, if the similarity between the two solutions is below a predefined threshold (currently 50%), the system will simply store that call as problematic, but no actual change is performed in the casebase.

With this list of "problematic" calls, the researcher can decide whether it is only an exception that requires no correction or if it is necessary to implement the problematic call, in order to achieve the desired result.

It must be noted that, in this case, the researcher can choose to implement the call in a way that it actually performs some actions (which should be seriously considered, especially for I/O operations), or simply implement it to return the correct result.

As an example of these two options, let us consider a call to `puts`, a function that writes a string to the standard output, returning the number of characters written. The researcher could implement `puts` in a function that actually writes a string into a file, so it can be checked later, or on screen, returning the number of characters saved. The researcher could also implement a function that simply returns the number of characters in that string, to assure the correct execution of the code under analysis.

While the first option is, in general, essential for functions that involve some I/O operation, the second allows us to keep the system running even when we have no need to record the operations performed by a function.

3.4.4 Retaining

Our task of retaining cases can be described as having two sub-tasks:

1. Deciding when a new case should be retained

2. Actual retaining process

Not all the cases we encounter need to be stored again, especially considering that our revising step depends strongly on the researcher, so we only want to store new cases when they can bring some positive impact to our system. Correctness cannot be evaluated in run-time (at least not at this point), so we decided to retain cases that can bring speed to our process. When we search for a solution to a problem that is unknown, we will store that problem, and associate the solution with it. We defined that, when the best similarity found was lower than a minimum threshold (currently 50%[12]), this problem would be considered unknown. The definition of this threshold, however, can be configured by the researcher within AGAVE.

The main impact of this retaining operation is on the retrieval step. As, in that step, we first look for an exact match we can speed up that process from several seconds to an almost immediate search by including unknown cases in our casebase.

[12]According to our formula for similarity (Equation 3.1), 50% of similarity would indicate that, in the chosen element, the name, the parameters or both are very different to those from the query.

3.5 Testing CBR

We have tested just the CBR components isolated from the rest of the system in order to answer the following two questions:

1. Is CBR suitable to assist the OS emulation? In other words, would we have enough accuracy so we can test code without having to implement OS emulation?
2. Is it efficient? Or how can we tune our systems to maximize efficiency without losing much accuracy?

Directly related to this tuning process is another factor we wanted to evaluate, even though it was not, by any means, decisive for the success of AGAVE: the size of generated casebases and how the number of neighbors would affect it.

Development and tests of the code, as well as database creation and initial experiments were done on a laptop. However, the experiments presented here required a rather long period of execution (several days). For these experiments we used a desktop computer, equipped with an Intel Core2 2.4GHz processor and 4GB of RAM, running Scientific Linux SL 5.3 and Python interpreter version 2.7.

For our experiments, we collected traces during the execution of 56 Linux commands[13] (listed in Appendix A), storing a total of 45,000 calls. The limit of 45,000 calls was chosen to reduce the time for generating the casebase during our tests. Those calls were converted into 2,008 cases for library calls and 2,355 cases for system calls. All the executables used for generating our casebase consisted of text-based commands. This was done purposely. We speculated that it would be useful to evaluate how our system

[13]We selected text-based commands from the directory /bin. From those, we discarded some that would run indefinitely (e.g., bash, dash). However, as we limited the number of calls, only those 56 commands were used to generate our casebase. In order to allow a more realistic emulation, we suggest that the traces include all commands, when possible. Other programs, such as those with a graphical interface (X-based, Gnome, etc.) should also be included when possible and necessary.

behaves for unknown calls, and a natural source for unknown calls would be executables with a graphical interface.

Our evaluation considered three different situations:

- Only system calls: as their number is quite limited (336 system calls in our version of Linux), we expected a high accuracy.
- Only library calls: we believed that a difference between libraries accessed by our training set and the libraries used by the testing set would reduce accuracy.
- Both system and library calls: A balance between the two previous situations, thus an accuracy intermediate between those two situations.

We then generated casebases with different minimum similarity for two cases to be considered neighbors and different maximum number of neighbors for each case. We used graphs (that we sometimes called "neighborhoods") of 50, 100, 250, 500 and 750 nodes, with minimum similarity of 70%, 75%, 80%, 85%, 90% and 95% .

For testing sets, we collected traces of the execution of 4 different programs:

- `ls`: Unix command for listing directories. While an execution of `ls` is part of the training set, for testing we are using a different trace of the same command. The sequence of calls, however, should be the same and, even though the results are different, we expect a high level of accuracy in this test.
- `ping`: TCP/IP standard command for sending ICMP messages through the network.
- `who`: Unix command for listing on-line users.
- `xcalc`: a X11 based calculator. This program uses libraries that are not likely to be used by the previous ones.

Program	Syscalls	Libcalls	Total
ls	181	291	472
ping	163	86	249
who	217	144	361
xcalc	870	88	958

Table 3.3: Types of call in testing traces

Table 3.3 shows the number, on each trace, of system and library calls, so we can understand the impact of each one in the results.

There is one other factor that was considered for increasing performance and it is a minimum *threshold* of similarity, for comparing a query case and our case base. During a search in our casebase, when this threshold of similarity is reached, we have found an acceptable similarity level, i.e., the element we compared is "similar enough" to the one we are searching for.

This threshold should not be confused with the minimum similarity for generating neighborhoods. In that case, a high similarity increases performance, as that similarity is used for reducing the search space. Here the opposite happens. A low threshold would actually result in a quicker, but more inaccurate response, as we would interrupt the search earlier. The tested thresholds were 70%, 80%, 85%, 90%, 95% and 100%. The last one (100%) meaning that we either would find an exact match or all the graphs would be checked.

3.5.1 Evaluating accuracy

Our tests consisted in reading the traces from the testing set, converting each trace into a case. We then made our CBR module to search for that case in the casebase and return what solution that call would be resulting on. That result was compared to the trace. Accuracy is, thus, given by how similar the result is to the trace it originated from.

For discussing accuracy in our implementation of CBR, we will be using one single neighborhood. The implementation of different neighborhood sizes had the main objective of improving performance. While we believe that tuning performance has some impact on accuracy, we first only wanted to evaluate what degree of accuracy we could achieve. Only then we would deal with performance issues.

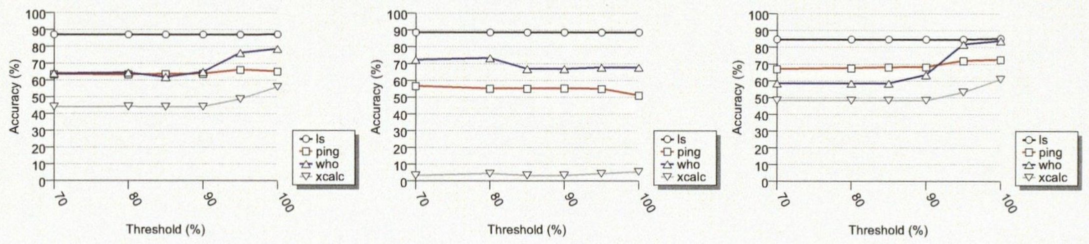
By Figure 3.8a, we can see that `ls` achieved a high accuracy, around 90%. This was expected as we had an example of `ls` in our training set, so all the calls were in our casebase. In the case of `ping` and `who`, while they were not in our training set, they are text-based, and we still could keep an accuracy that varied from 60% to 80%.

For `xcalc`, however, due to its graphical interface, we were expecting a lower accuracy. In this particular case, the number of library calls in this case was only about 10% of the number of system calls. Because of this difference, the impact of the bad accuracy when searching for library calls (Figure 3.8b) was compensated by a better accuracy for system calls (Figure 3.8c).

Library calls, in general, will be executing system calls internally. Thus, we tend to find a higher number of system calls than library calls. We believe that this will allow us to keep an acceptable accuracy, if considering traces of executions.

A problem we would face, however, is the fact that for actual execution, we do not have all the information, i.e., most programs have calls to the operating system library, not to system calls. Even though we are currently storing what calls (either library or system calls) are executed inside a library call, we are not yet using this information. Thus, if we were exclusively using our current implementation of CBR for emulating `xcalc` within AGAVE, our accuracy would be actually reflected by Figure 3.8b.

For our current implementation of AGAVE, there would be two ways of improving this accuracy. The researcher could implement the functions that are missing (or at least those that are relevant), or the researcher could collect traces of trustworthy programs



(a) All calls (b) Library Calls (c) System calls

Figure 3.8: Evaluating accuracy

that use the same library and process these traces within AGAVE. (The program `xcalc`, for example, could have its accuracy improved by the use of traces from other X11 based programs).

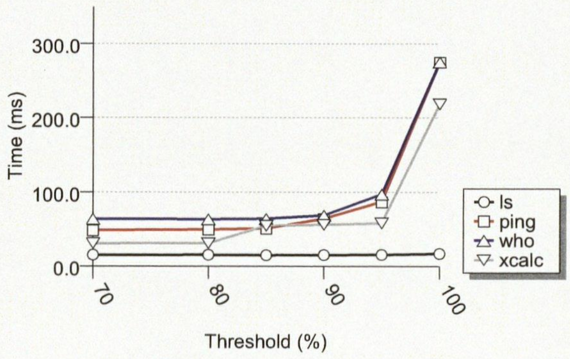
3.5.2 Evaluating performance

Thus, our experiments confirmed our expectations. There are two factors that impact performance:

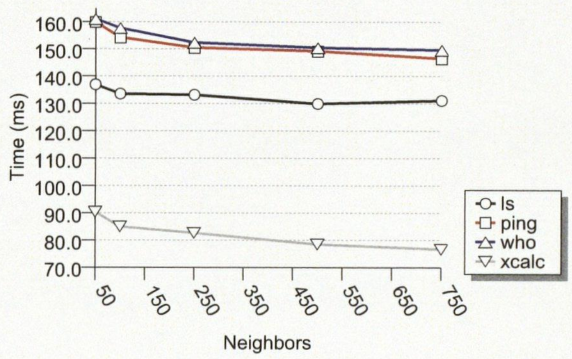
1. The *threshold* used to indicate that a "similar enough" element was found, which could interrupt the operation without necessarily looking at the entire search space.

2. The *neighborhood* definition, defined as a pair (*Number of Neighbors*, *Minimum Similarity*). As previously discussed, a higher number of neighbors would provide a better performance for search operations. More elements would be removed from the search space, during the "Fish and Shrink" algorithm, limiting the number of comparisons, but negatively impacting accuracy. On the other hand, the minimum similarity for elements to be considered neighbors keeps us from removing potentially good results, but reduces performance and improves accuracy.

First, we look at the impact on the threshold. In Figure 3.9a, the y-axis represents the average time to look for a call, in milliseconds. All the programs in this test used the



(a) Impact of the threshold on performance



(b) Impact of the number of neighbors on performance

Figure 3.9: Evaluating performance

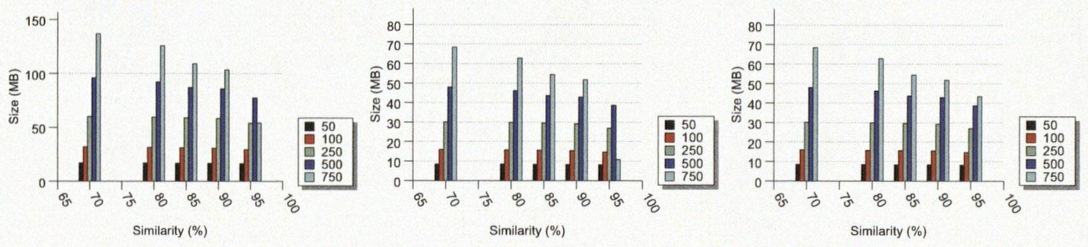
same parameters for the neighborhood[14].

As for evaluating the impact of changing the neighborhood parameters, we defined the threshold at 100%, thus forcing the search in the entire database unless a perfect match was found. In Figure 3.9b, we have the emulation of the commands with different configurations for neighborhoods when keeping the minimum similarity at 85%.

The good performance presented by `xcalc` might seem surprising. Xcalc uses calls from libraries that were not in the training set, so most of the library calls were unknown. The system, then, retained the results for those calls, and when they were later needed, it would use the results just retained. We should remember, however, that this rather good performance was accompanied by a low accuracy (Figure 3.8).

As we previously discussed, the minimum similarity does have some impact on performance, but not directly. The actual impact on the minimum similarity is by reducing the actual neighborhood size for some of the function calls. The minimum similarity is, therefore, more related to the casebase size (which obviously impacts on performance). A discussion about the casebase size is in the following section.

[14]500 Neighbors with 85% of minimum similarity.



(a) All calls (b) Library Calls (c) System calls

Figure 3.10: Evaluating casebase sizes

3.5.3 Casebase sizes

Figure 3.10 shows us the size of the files that store our casebases (neighborhoods). As expected, the higher number of neighbors, the bigger our casebase will be. An alternative to reduce the casebase size would be by increasing the minimum similarity for entities to be considered neighbors. The expected trend, however, only can be observed for system calls (Figure 3.10c).

For library calls (Figure 3.10b), and consequently for the total size of our casebases (Figure 3.8a), we observe a significant reduction in the casebase size for 750 neighbors with a minimum similarity of 95% between them. The number of different names is higher for library calls than for system calls and the names have a strong impact on the similarity measure. So, the number of 750 neighbors for library calls is rarely achieved and the actual neighborhood has less neighbors than expected.

3.6 Summary

In this chapter, we discussed how CBR was implemented within AGAVE. We first presented the definition of a case for AGAVE, then discussed details of decision and implementation and also some of the problems that we faced during the development.

Finally, we discussed tests that were performed to evaluate accuracy and performance under different conditions. Our tests showed promising results for a proof-of-concept; however, there is still a lot of improvement necessary. Not only are there some features that should be explored, but there is information that could be collected and was not considered during this project.

An important source of information, for instance, is the relation between calls. We are able to collect which calls are performed during the execution of a function. This information could be used for inferring more details of execution, which would lead the researcher to a better understanding of the code under analysis.

Another clear limitation of CBR in AGAVE is that, by using only traces as a source of information, we are not able to deal with functions that return information in one of their parameters. We believe that this information returned in parameters can be possibly inferred by other means, but we have not yet explored this alternative. However, when necessary for analysis, the researcher can implement the function within AGAVE. This would assure a correct code execution and a more precise analysis.

Chapter 4

AGAVE Usage and Testing

During this thesis, we have presented how AGAVE is organized and how CBR is implemented for allowing emulation. In this chapter, we will discuss how the researcher can use AGAVE to detect malware.

First, we present how AGAVE can be configured, using a configuration file and, for some options, the command line. The configuration file is similar to what is frequently found for various programs. The command line can allow the user to easily modify the behavior of AGAVE for a specific session of emulation.

We then discuss how to program AGAVE, either for implementing a system or library call to achieve a more correct emulation or for implementing means for detection. This discussion is presented in the form of a tutorial, so the reader can have an insight of the AGAVE experience. As part of this, we describe AGAVE testing.

4.1 Configuration

The AGAVE configuration file is a standard text file, divided in sections, similar to ".INI" files found on Windows. Each section has a header, identified by the name of the section between square-brackets (e.g., *[SectionName]*) and the parameters in each section are in the form "parameter=value".

By default, AGAVE looks for a configuration file named "agave.conf". In order to provide a different file, the user must call AGAVE using the option "-c":

```
user@host $ ./run-agave -c MY_CONFIG SUSPICIOUS_EXE
```

The remainder of this section presents the options that can be used to configure AGAVE.

4.1.1 Section "emulator"

This section is responsible for configuring which hardware emulator we will use, by choosing the emulation interface. Details on how to implement this interface were previously discussed in Chapter 2.

```
[emulator]
module=agave_pyemu
name=Agave2PyEmu
log_level=DEBUG
```

Figure 4.1: Emulator section in agave.conf

In Figure 4.1, we have an example with the available options, which are:

- **module**: name of the module file where the interface was implemented. It must be a valid Python module (so we assume ".py" or ".pyc", as extension). A module file must contain one or more interfaces.
- **name**: the name of the interface that must be used within the module file.
- **log_level**: level of logging for the emulator. Valid levels are: "DEBUG", "INFO", "WARN" and "ERROR", in this order. For example, if a level of "INFO" is defined, no debugging information is logged. If "ERROR" is chosen as the log level, only error messages will be logged.

4.1.2 Section "cbr"

The parameters for the CBR module are specified in this section, defining three things. First, the parameters for the fish-and-shrink algorithm. Second, a minimum threshold for similarity value, that, when achieved, interrupts the search and returns the result that was found. Third, the logging level specific for the CBR module. This is similar to the configuration for the emulator that was previously presented.

An example of this section can be seen in Figure 4.2.

```
[cbr]
casebase=Linux-x86
neighbors=500
neighbor_similarity=75
threshold=75
log_level=DEBUG
```

Figure 4.2: CBR section in agave.conf

The available options are:

- **casebase**: indicates the name of the casebase to be used. We assume that a researcher might work with different operating systems and different versions of them. So, for each operating system we generate a casebase. If the researcher wants to experiment with code in a different operating system, changing this parameter would select a different casebase.
- **neighbors**: the number of neighbors used when generating the neighborhoods for the Fish-and-Shrink algorithm.
- **neighbor_similarity**: the minimum similarity for neighbors, also used for neighborhood generation.
- **threshold**: the minimum similarity for searches. When a case with similarity greater or equal to this parameter is found, its result is returned.
- **log_level**: level of logging for the CBR module.

The values for *neighbors* and *neighbor_similarity* are used to select the casebase the fish-and-shrink algorithm will search in. If the casebase does not exist, it will be generated. It is important to be aware that this generation might take a long time, up to several hours, depending on the number of cases stored and the parameters used, but this is precomputation that is not repeated for each emulator run.

4.1.3 Section "scripts"

```
[scripts]
1=libc.agave
2=linux_syscalls.agave
3=instructions.agave
```

Figure 4.3: Scripts section in agave.conf

This section contains scripts to be executed during the initialization of AGAVE. These scripts contain definitions of functions that are used for customizing the emulation (OS or CPU) and also for other purposes, such as initializing memory for specific operations. The scripts must be stored in the `scripts` directory. Even though the scripts can be considered valid Python code, we suggest the extension `.agave` for such files, as they contain AGAVE-specific code.

The parameter name (which we suggest to be a sequential number) indicates in which order the scripts will be executed, as some scripts might depend on others. If two scripts are referenced by the same value, only the last one will be considered valid (the first one will be ignored).

Scripts can also be loaded by using "--load-script" or "-ls" in command line.

```
user@host $ ./run-agave --load-script SCRIPT1.agave --load-script
SCRIPT2.agave EXECUTABLE
```

or

```
user@host $ ./run-agave -ls SCRIPT1.agave -ls SCRIPT2.agave EXECUTABLE
```

Scripts that are supplied on the command line will have higher precedence than those specified by configuration file. AGAVE first loads scripts specified in configuration files and only then the ones given on the command line. This means that if a function is declared twice (in a configured script and in a script passed by the command line), the last declaration will override the first, i.e., the definition in the file given on the command line will prevail. This approach gives the researcher the flexibility of changing the execution for a single session of emulation without modifying the configuration file.

4.1.4 Section "heuristics"

Heuristics are sets of rules that guide how an activity is performed. In malware analysis, the term heuristics sometimes refers to methods used for detection.

Configuring heuristics within AGAVE requires the researcher to perform two steps:

1. Implement heuristics within AGAVE, preferably using a script file, and declare them using the command `set_heuristics`.

2. Modify the configuration file to inform AGAVE when the heuristics will be evaluated.

An implementation of a simple heuristic is presented later in this chapter.

For the configuration file, a "heuristics" section is used. Each entry in this section is in the form "NAME=EVENT[, ...]". NAME is the name of the heuristic as defined by the command `set_heuristics`(its first parameter). EVENT informs AGAVE when the heuristic will be evaluated and can be one or more of the following:

- **begin_execution**: the heuristic is evaluated before the emulation starts.

- **before_step**: the heuristic is evaluated before an instruction is executed.
- **after_step**: the heuristic is evaluated after the execution of an instruction.

For example, in Figure 4.4, `findEICAR` will be performed at the beginning of the emulation and after each instruction is executed. `CheckStack`, on the other hand, will be performed before the execution of each instruction.

```
[heuristics]
findEICAR=begin_execution,after_step
CheckStack=before_step
```

Figure 4.4: Configuring heuristics

4.2 Testing AGAVE emulation

As we previously mentioned, programming AGAVE is quite simple, in the sense that it uses the Python interpreter, on which AGAVE is interpreted, to interpret user code as well. AGAVE, however, provides some functions to be used by the researcher to access information about the CPU and memory that are necessary to achieve their objective: detecting malware.

A comprehensive list of the functions provided to allow the researcher to program AGAVE is in Appendix B but, instead of going over each one of them, we believe that a demonstration by example is a more effective approach.

In the remainder of this chapter, we will demonstrate a case of malware detection. In our demonstration, we will encounter some of the problems a researcher would have to face, and demonstrate how AGAVE customization can overcome those problems. For

```
#include <stdio.h>
int main() {
  puts("This is the EICAR File content:\n
    X5O!P%@AP[4i\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*");
}
```

Figure 4.5: Source code for hello_eicar

this demonstration, we assume Linux as the operating system, running on an x86 (32 bit) platform.

4.2.1 Hello EICAR

Our final goal is using AGAVE for malware detection. One of the standard tests for malware detection, specially for signature detection, is detecting the EICAR test file.

The "EICAR Standard Anti-Virus Test File"[19] is a valid DOS executable, consisting only of ASCII human readable characters that, when executed, prints "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!". The European organization EICAR proposed the use of this file as good practice to test anti-virus and anti-malware products without risk of real infection. The EICAR file is successfully detected by a great majority of anti-virus products on the market.

However, this file is meant to be executed in a DOS/Windows environment and we have developed a proof-of-concept having ELF files in mind, i.e., it would interpret and emulate only ELF files, so our system was not able to deal with the EICAR file as it is distributed. Instead, we created a small program in C that prints the EICAR file content (Figure 4.5) and compiled it using gcc[1]. By doing this, we have an ELF executable that contains the EICAR string.

Note that traditional anti-virus will not detect our executable as having the EICAR signature. According to the EICAR specification, the EICAR test file must appear as

[1]gcc (Ubuntu 4.4.3-4ubuntu5) 4.4.3

the first 68 bytes of a file, while ours is in the ELF file. However, as our objective is finding the EICAR file in memory, we believe that our implementation actually follows the original motivation: a safe alternative for testing anti-virus products.

Detecting the EICAR file string in our executable is trivial. A simple string search would find it. A malware writer would likely use some sort of obfuscation to avoid detection.

4.2.2 Packing Hello EICAR

Packing is essentially the process of compacting, combining and/or obfuscating an executable, while still keeping it executable[61]. Packing can be used for saving storage space, to create an installer, or to avoid reverse engineering of proprietary code. The actual code must be unpacked to be executed, so, in general, what happens is that the code is compressed and an executable stub is added. This stub decompresses the file (or parts of it) in memory and then executes it.

While the technique of packing has its legitimate uses, it is also often used by malware writers to make analysis by researchers more difficult.

We want to know if AGAVE was capable of allowing us to detect packed malware. We are assuming that we could run the emulation of the stub, it would unpack the actual malicious code in memory and, just before it was executed, we would be able to easily detect (by looking for signatures, for example) the malware in memory.

One of the most common[25] tools for packing is called UPX (standing for Ultimate Packer for eXecutables)[50], and we would use UPX[2] to pack our *hello_eicar* and test if we could use AGAVE to detect the EICAR file.

However, simply packing our *hello_eicar* executable did not work. We received an intriguing error message from UPX, shown in Figure 4.6. That message was not explained

[2]UPX version 3.04

```
daniel@dssl$ upx test
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2009
UPX 3.04        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 27th 2009

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
upx: test: NotCompressibleException

Packed 1 file: 0 ok, 1 error.
```

Figure 4.6: Packing hello_eicar

```
#include <stdio.h>
int main() {
  puts("This is the EICAR File content:\n
    X5O!P%@AP[4i\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*");
  __asm__("nop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\n"); // 10 NOPs
  __asm__("nop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\n"); // +10 NOPs
  // ... line above was repeated 180 times
  __asm__("nop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\nnop\n"); // +10 NOPs
  }
```

Figure 4.7: Source code for hello_eicar with NOPs

anywhere in the UPX documentation. We downloaded UPX source code and analysed it, so we found that UPX tests how the `.data` section, with the stub, was compressed, returning the error where the compressed version has the same size of the original. This may be indicative that the main focus of UPX was for compacting not for obfuscation.

To work around this problem, we decided to simply increase the size of our data section. We edited our C code to include NOP instructions[3] (as we can see in Figure 4.7). We decided to use this approach because we wanted to guarantee that we had an executable with exactly the same functionality as the original, although just a bit larger, almost 2 kilobytes larger.

[3] "No Operation" instructions

4.2.3 First attempt at emulation

Having an executable to use in our tests, the next step is trying to emulate its execution. We run AGAVE, passing our packed "hello_eicar" as a parameter.

```
user@host $ ./run-agave hello_eicar
```

Our "hello_eicar" is loaded by AGAVE, the user receives a series of messages and a prompt ("AGAVE ⟩") is shown. In interactive mode, AGAVE presents a text-based interface, where a prompt is displayed and the user can type commands. Python commands as well as the functions listed in Appendix B can be used in this interface. The user can, then, start the emulation by using the AGAVE command `run()`.

```
AGAVE > run()
```

Here a great weakness of AGAVE becomes evident. While the non-emulated code is normally unpacked and executed in a few milliseconds, AGAVE runs for several minutes. As previously mentioned, having AGAVE on top of PyEmu turned out to be a bad combination. In tests, we rarely could emulate more than 1,000 instructions per second. When CBR was necessary it would be even worse. For comparison, a 8088 processor in the early 1980's could execute more than 500 times faster than our current version of AGAVE.

The emulation then is interrupted by a memory access error. A quick analysis shows that the program tried to push a value onto the stack. However the stack pointer was pointing to an address in memory that was not initialized yet.

Looking further in the logs, we find that there were early attempts of performing a system call to SYS_mmap. SYS_mmap's main purpose is mapping a file into memory. However, when SYS_mmap is executed in "anonymous" mode, a virtual memory page (from a swap file) will be mapped into memory, essentially allocating memory. As we do not have SYS_mmap actually implemented, AGAVE simply calls its CBR retriever that returned the requested address to the program. In fact, the result returned to the program is exactly the one the program was expecting, as shown in the excerpt from the log in Figure 4.8. However, AGAVE did not perform any actual operation in memory, which finally resulted in an attempt of accessing memory that was not allocated.

```
[INFO] SYSCALL TO SYS_mmap at 00c01c00
[INFO] Executing code for SYS_mmap from casebase
[INFO] Executing SYS_mmap('9824212', '4096', '12587008', '12590116', '2572')
[INFO] Results from retriever: 9824212
```

Figure 4.8: CBR result for SYS_mmap

4.2.4 Implementing a system call

Our next step to have the code running is to implement SYS_mmap. A researcher may not always have complete information to perfectly implement a third-party function. So, for this particular example, we will not implement SYS_mmap to its full functionality, only what is necessary to allow us to execute our "hello_eicar".

Implementing a system call within AGAVE means implementing a simple Python function and then declaring it as a system call.

It must be noticed that implementing system calls depends on the operating system being emulated. Each operating system deals with system calls differently: assembly instructions are different (for example, int 0x2e on Windows and int 0x80 on Linux), CPU registers and flags are configured differently, memory is managed differently and, something that will directly affect our example of implementation, parameters are passed to system calls by different mechanisms.

For example, in both Windows and Linux, the CPU register EAX is used to indicate what system call is being performed. However, parameters are passed using different registers. On Windows NT, for example, only the CPU register EDX is used to pass parameters to a system call. The parameters are pushed onto the user stack and EDX receives the memory address that points to the parameters' location in the user stack[53].

On Linux, on the other hand, several registers are used to pass parameters to a system call. The first parameter is referenced by EBX, the second by ECX, the third by EDX. If more parameters are necessary the registers ESX and EDI are also used (in this order). In some cases, either because the registers are not enough for all the necessary parameters or for organization purposes, instead of storing actual values, the registers are used to store memory addresses. These addresses point to structures that contain all the information and those structures are stored on the user stack.

SYS_mmap, the system call we are going to implement, receives only one parameter (thus we will only need to read EBX), that points to an mmap_arg_structure(Figure 4.9), as defined in Linux source code[4].

```
struct mmap_arg_struct {
     unsigned long addr;
     unsigned long len;
     unsigned long prot;
     unsigned long flags;
     unsigned long fd;
     unsigned long offset;
};
```

Figure 4.9: mmap_arg_structure, used as parameter for SYS_mmap

For this example, our implementation of SYS_mmap will only able to deal with anonymous mapping (when a flag MAP_ANONYMOUS is used). This is different from the mapping of files into memory. An anonymous mapping has the same basic functionality

[4]This type definition can be found at /usr/src/linux/arch/x86/kernel/sys_i386_32.c.

of memory allocation and initialization, with the contents initialized to zero[5].

Our implementation of SYS_mmap can be seen in Figure 4.10. Some details about this implementation must be noticed:

- When an address is not explicitly given, we need to dynamically allocate a page. For this implementation we directly accessed the PyEmu memory object to find an available memory page. Memory management, allowing AGAVE direct emulated memory access without relying on the emulator being used and having the ability of pointer tainting[68], is a feature that we want to add to AGAVE in the future.
- We used the AGAVE command `set_memory()` to initialize the memory, setting the content to zeroes. This command also allocates memory pages if necessary.

To define the function "sys_mmap" to be executed as the system call "SYS_mmap", we use the AGAVE command `set_system_call`. The first parameter is the system call name and the second parameter is a call to our function.

This code is saved in a file called "demo.agave", and we try to emulate our "hello_eicar" again. But now, we will inform AGAVE to load the file we just created with our SYS_mmap[6].

```
user@host.$ ./run-agave --load-script demo.agave hello_eicar
...
AGAVE> run()
```

Different from the first attempt, the code now crashes almost immediately. The reason now is different: an "unsupported instruction". Most CPU emulators do not offer support to all the possible instructions defined by the architecture they emulate. In fact,

[5]According to the mmap manual page.
[6]A different option would be include the script in the AGAVE configuration file, as explained earlier in this chapter.

```
def sys_mmap():
        ebx = get_register("EBX") # Reads first parameter of sys_mmap
        parameters = []
        address = ebx
        for i in range(6): # Reads structure from memory
                data = get_memory(address, 4)
                parameters.append(data)
                address = address + 4
        (addr, length, protection, flags, fd, offset) =  parameters

        # If address not supplied, get the first available page
        if addr == 0:
                addr = interface.emu.memory.get_available_page(0x08000000)

        if flags & 0x20: # Checks if it is an anonymous mapping
                data = '\x00' * length
                set_memory(addr, data, length) # Initialize memory with zeroes.
        else:
                # Not an anonymous mapping, do nothing for now
                pass

        set_return_code( addr) # Returns the address

set_system_call("SYS_mmap", "sys_mmap()")
```

Figure 4.10: Implementation of SYS_mmap for AGAVE

a common technique used by malware writers to detect if emulation is taking place, and then avoid analysis, is attempting to execute obscure or unusual instructions[13]. This is no different for PyEmu, the emulator we are using for our proof-of-concept.

The instruction responsible for the crash is LODSD. This instruction is not implemented in PyEmu and, as we will see later, this is not the only one. To continue our emulation, then, we need to implement this instruction.

4.2.5 Implementing x86 instructions

LODSD is a mnemonic for "Load String". This instruction is used to load a string from ESI into EAX. The instruction also increments or decrements ESI, depending on the bit DF being enabled in the CPU register EFLAGS. In practical terms, we simply need to store the address from ESI to EAX and then modify ESI. The implementation is in Figure 4.11. We increment (or decrement) ESI by 4 because LODSD assumes addresses are double words (4 bytes).

```
def LODSD():
        esi = get_register("ESI")
        df = get_register("DF")

        data = get_memory(esi, 4)

        set_register("EAX",data)

        if df == 0:
                set_register("ESI",esi+4)
        else:
                set_register("ESI",esi-4)

set_custom_instruction("lodsd", "LODSD()", 1)
```

Figure 4.11: Implementation of the instruction LODSD for AGAVE

To declare our function "LODSD" as a custom instruction for AGAVE, we use the

AGAVE command `set_custom_instruction`. This command is similar to the previously seen `set_system_call`, but, as mentioned in Chapter 2, an additional parameter is necessary and it is the length of the instruction in bytes, so the instruction pointer can be incremented accordingly.

As we briefly mentioned, this is not the only instruction that is unimplemented by PyEmu. And to successfully emulate our "hello_eicar", it is also necessary to implement STOSD and BTR.

STOSD (Store string) performs the reverse operation to LODSD, i.e., it stores a string from EAX into EDI. BTR (Bit Test and Reset) is an instruction that selects a bit from a bit string and stores it in the bit CF from the register EFLAGS. Implementing these instructions is similar to the implementation of LODSD and, for reference, can be found in Appendix D.

Additionally to these instructions that were not implemented by PyEmu, we have a case of an instruction that was implemented but had to be overwritten. This is the instruction REP, which repeats the following instruction according to the number stored in the register ECX. During the emulation of our "hello_eicar", there are some calls for "REP LODSD" or "REP STOSD" (part of the unpacking code for UPX). While PyEmu has an implementation for REP, it does not recognize our LODSD and STOSD, implemented within AGAVE. The solution is, thus, implementing the repetition instruction for those.

We decided to mention this implementation (Figure 4.12) because it also has some interesting details of implementation:

- We have one single function for two different instructions. AGAVE allows us to use parameters when defining functions that are used as instructions. The same can also be used for defining system calls, library calls or heuristics (example of these last two will be seen shortly).

```
def REP(instruction):
        ecx = get_register("ecx")
        while ecx > 0:
                (command, size) = get_custom_instruction(instruction)
                eval(command)
                ecx = ecx -1
        set_register("ECX",0)

set_custom_instruction("rep lodsd","REP('lodsd')", 2)
set_custom_instruction("rep stosd","REP('stosd')",2)
```

Figure 4.12: Implementation of the instruction REP for AGAVE

- We use the AGAVE function `get_custom_instruction()`. This function returns a tuple (command, size of instruction) and this information can also be used for our implementation.

With all these instructions implemented and saved in the file that we previously created ("demo.agave")[7]. Then, we try to emulate our code again:

```
user@host $ ./run-agave --load-script demo.agave  hello_eicar

...

AGAVE> run()
```

Again, the emulation takes a while to run (around 2 long minutes) and finally ends gracefully. The long time is due to the number of instructions being executed for the unpacking process (allied to our slow speed). During the execution we observe requests to the CBR retriever for system calls that we did not implement (such as SYS_mprotect or SYS_munmap).

[7]Ideally, we would recommend the researcher create different files for instructions, system calls, library calls and heuristics, as it would allow a better customization. For example, Linux system calls are independent of processor-specific instructions and keeping them separate from each other would allow to easily perform tests of a 32 bit processor running a version of Linux and then the same tests of a 64 bit processor using the same version of Linux.

But, then, something different happens. AGAVE finds a call to "__libc_start_main", a function defined in the shared library "libc". As it is not implemented within our "hello_eicar" and AGAVE does not know what to do, it calls the CBR retriever for that library call.

That function is part of the initialization process for an ELF file and should, among other actions, call the `main` function in our "hello_eicar". In fact, reading this call shows us that our original code for "hello_eicar" is running. Until now, all the code emulated was for the unpacking process inserted by UPX.

And, again, our CBR retrieves and returns a value that is consistent to what the code was expecting, but as no action was really performed, the program ends when it finds the next instruction, a hlt (for Halt). We need, thus, to implement "__libc_start_main".

4.2.6 Implementing a library call

The function __libc_start_main is responsible for initializing the environment to execute code in Linux. It should perform tasks such as thread initialization and registering handlers for a clean exit. It also must appropriately call the "main" function of an executable and, when "main" is finished, the "exit" function[1].

For simplicity, our implementation of __libc_start_main simply calls the main function, as shown in Figure 4.13. This implementation is also saved in our "demo.agave" file. The implementation is straightforward: __libc_start_main receives as parameter the memory address for the "main" function. We simply move that address to the instruction pointer.

Now our "hello_eicar" can run to its completion. Looking at the emulation output, we see a call to "puts" (emulated by our CBR module) and then a sequence of NOPs.

However, completely emulating this program is not our goal. Our goal is detecting malware, which, in this experiment, is represented by the EICAR file signature. In order to do this, we will implement a simple and basic heuristic.

```
def start_main():
        stack = get_stack_pointer()
        main_address = get_memory(stack+4, 4)
        set_instruction_pointer(main_address)

set_library_call("__libc_start_main", "start_main()")
```

Figure 4.13: Implementation of __libc_start_main

4.2.7 A simple heuristic

For this proof of concept, we demonstrate by performing a string search in memory. Our executable is packed with UPX, so we only will find the string when the decryption is complete. It does not make sense to try to perform the search after every emulated instruction, so we need to detect the end of the decryption phase.

A good indicator of decryption being complete is detecting when an instruction is being executed from a position in memory that was recently written. And, for our heuristic, we use this indicator as a trigger to perform the string search.

The AGAVE code used for our heuristic can be seen in Figure 4.14. The function `jumping_to_data` keeps a list of the pages of memory that were used during the execution (when the code emulation starts, a list is created with the pages currently allocated for the process). For this implementation, `jumping_to_data` is specific to PyEmu. If our code tries to execute from a page that is not in that list, it returns True (and adds all the currently allocated pages to the list). Again for simplicity, `elf_in_memory` is also specific for the problem we are tackling, in this case, UPX. UPX decompresses the entire ELF file in memory. So, we search for an ELF header in the new pages. Those two conditions were sufficient to detect the end of decryption.

Finally, the `search_EICAR` function is a simple string search, returning "True" when the EICAR file is found. Our simple heuristic prints a message informing that EICAR

```
def find_eicar():
    if jumping_to_data() and elf_in_memory():
        print "Jumping into data section"
        if search_EICAR():
            print "EICAR Standard Test File was found"
            pause()

set_heuristics("FindEICAR", "find_eicar()")
```

Figure 4.14: Simple heuristics

was detected and executes the AGAVE command `pause()`, which stops the emulation. This command is particularly useful for testing purposes.

As we have done with system calls, library calls and CPU instructions, we need to declare the function as a heuristic. For this, we use the AGAVE command `set_heuristics`. The complete implementation of "findEICAR" can be found in Appendix E.

In order for this heuristic to be executed, we also need to configure AGAVE. This configuration is necessary to define when the detection will be performed. As we mentioned earlier in this chapter, this is done by modifying the "heuristics" section in the AGAVE configuration file. There is no command line option for configuring heuristics.

For our detection, we decided to perform the detection of EICAR after the execution of an instruction. So, in the configuration file, under the section "Heuristics", we include the following line:

```
FindEICAR=after_step
```

Running our emulation for the last time, it takes around 4 minutes, ending with the message "EICAR Standard Test File was found" and the AGAVE prompt is displayed.

4.2.8 Discussion

While detection using AGAVE is possible, it is still a really slow operation. We believe that another emulator might help to improve speed. Some optimizations for emulation within AGAVE might also be an alternative.

During our experiments, we attempted to emulate a statically compiled program. It was an interesting test case for the emulator speed and it was when we could achieve the best performance. During that experiment, there were no CBR requests for library calls. System calls, on the other hand, were frequent and, to allow a correct emulation, many of them needed to be actually implemented. We also encountered a higher number of unsupported instructions. Due to time constraints, we decided to abandon that experiment for now. Further tests, using statically compiled executables, will be necessary.

The total time for emulation was rather long. And if a researcher tries to follow this chapter, the use of snapshots can prove to be quite efficient. For example, after an unsupported instruction is found, the researcher can save a snapshot of the emulation, exit AGAVE, implement that instruction, restart AGAVE and restore the snapshot, returning to the point where the emulation failed. This simple strategy might save a lot of time for the researcher.

Implementation of system calls and library calls are not exclusive to AGAVE. When using emulation for research purposes, including within anti-virus companies, the implementation of those calls is usually necessary. The process is basically, to try to run the emulation until it fails, implement what is missing and then resume the emulation. AGAVE, however, by using CBR to emulate some of the functions that were not implemented, allows the researcher to perform emulation without necessarily implementing every function that is called. This characteristic of AGAVE potentially reduces the time required for performing analysis.

4.3 Summary

In this chapter we presented how to configure and use AGAVE. We used a “tutorial” approach to demonstrate, step by step, some of the tasks a researcher would have to do while using AGAVE.

These tasks include implementing library calls, system calls and even CPU-specific instructions. We also implemented a simple heuristic to search for the EICAR file in memory as an example of dynamic detection.

During this experiment, the emulation using AGAVE with PyEmu achieved poor performance. Even without the extra overhead for operations using the CBR retriever, we rarely had more than 1,000 instructions processed per second. Some optimization is definitely necessary and experiments with other emulators are recommended to evaluate performance.

However, even with the problem of bad performance and frequent interruptions due to instructions that were not supported by the emulator, we were able to successfully detect the EICAR signature in memory, before it was executed. This demonstrates that AGAVE can be used for dynamic detection.

Chapter 5

Conclusion and Future Work

In this thesis, we described AGAVE, a tool designed as a step towards automatic generation of anti-virus emulators.

This particular type of emulator has a strong focus on malware detection or analysis, depending on how it is used. Anti-virus emulators can be embedded in an anti-virus product or they can be a tool for malware analysis, in research labs (likely inside an anti-malware company).

AGAVE currently relies on a third-party CPU emulator for CPU and memory emulation. We implemented a general interface that allows the integration of AGAVE to these third-party CPU emulators. For our proof-of-concept, we implemented an interface for the open source CPU emulator PyEmu.

The operating system and library calls are emulated by an implementation that relies by default on case-based reasoning (CBR). Usually, researchers must implement all system and library calls for emulating an operating system. Even though precision is not always a requirement, it is still necessary in some implementations. By using CBR, on the other hand, AGAVE takes this burden off the researchers, restricting implementation to a minimum. Our CBR modules reply to system and library calls based on previously collected program execution traces in the operating system it intends to emulate. We have achieved an average accuracy rate over 60%, with cases of over 90% accuracy. This accuracy depends on the training set of collected traces (our tests used traces from text-based Linux commands; a more varied training set might achieve even better average accuracy).

In this thesis, we also demonstrated the use of AGAVE. The demonstration was based on a case study: an attempt to dynamically identify a malware signature in memory before the malware is executed. For demonstration purposes, the malware signature was the EICAR file signature and the executable was packed with UPX. Our demonstration showed some of the problems that a researcher may face, as well as some mechanisms of AGAVE to deal with those problems.

We understand that AGAVE still suffers from a number of limitations. Performance issues must be addressed to turn AGAVE into a more useful tool. Our implementation of CBR also needs some improvements for accuracy and even for completeness (e.g., returning information in one of the parameters). However, our experiments show that, even with those limitations, AGAVE is a viable approach.

5.1 Outline of contributions

Anti-virus emulators are a powerful and important tool for both malware analysis and for malware detection. Their continued development is essential for improving anti-virus and anti-malware efficacy.

However, most of the work in this area is restricted to anti-virus or anti-malware companies. There are few results published in academic papers. Publications such as white-papers and press releases do give us an insight of what is happening in the industry, but those are not always a reliable source of information, as results can be masked for publicity purposes.

In fact, during the time we were involved in this research, we realized that there seems to be a gap between industry and academia. In informal discussions with other researchers from academia, we noticed that for many of them malware analysis can be resumed as static analysis of code or running malware on a virtual environment to monitor its activities. It actually seemed that some of them even considered the use of anti-virus emulators an outdated approach, taking a strong stand for virtualization methods instead. On the other hand, researchers from anti-virus companies were not only more familiar with this subject but they seemed more receptive to research in this field.

We believe that our work can help to make the still obscure world of anti-virus emulators more accessible for researchers from outside the anti-virus and anti-malware industry. By giving researchers a tool for experimenting and developing heuristics, we expect that novel methods of detection can be created and implemented.

We consider that the main specific contributions of this work are:

1. The design and implementation of a framework that allows both malware detection and malware analysis. Our framework includes over 6,000 lines of Python code[1]. Code was implemented to make AGAVE highly configurable and flexible, and give it mechanisms for both malware detection and analysis.

2. A novel use of Case-Based Reasoning. CBR has often been used to predict the behavior of customers[65], the behavior of computer users[58] and even pure human behavior[43][11]. However, to the best of our knowledge, this is the first time CBR is used to predict the behavior of software, more particularly, of an operating system. Using CBR within AGAVE takes from the researcher the need for implementing code for each and every unknown function that is encountered. We believe that this can bring a significant increase of productivity for the researcher working with malware analysis and detection[2].

[1]This line count does not include PyEmu and other code that we did not develop ourselves.

[2]One could argue that implementing library and system calls is an one-time cost that would provide a better result, as the emulation would be more accurate. However, we would still be subject to the problem of unknown functions, for example, in case of a new version of the operating system or any of its libraries.

5.2 Future work

AGAVE is not a mature tool yet. During our experiments, we identified a number of limitations. We want to make AGAVE a more robust tool and several improvements are definitely necessary, especially in regards to performance. Some of these improvements are already under development; others will come later.

5.2.1 CBR

In our CBR modules , several improvements are necessary or desired.

For this project, our casebase is stored using the “Flat memory” scheme[39], where cases are stored in the form of a list and the search is performed sequentially. In our case, by using the “Fish and Shrink” algorithm, we are able to reduce the number of cases that are compared, but it is still a sequential search. We want to be able to perform parallel searches, in order to improve performance on the CBR modules.

Integration of our casebase with a relational database, such as MySQL or SQLite, is currently under development. Integrating a casebase with a database has been done before (e.g., [5]) and we expect to use the resources provided by a mature database system to achieve better performance, simplify development and provide a more intuitive platform for case retrieval.

While our CBR modules are already capturing the hierarchical relation between calls (library calls that call other library functions or that perform system calls), we have not used this information in our implementation. We believe that accuracy could be improved using this extra context, as this information can provide us a better understanding of how the operating system works.

Also, the ability of returning values within parameters that were passed to a function call is a feature that we want to implement in the future. The use of traces as input for our learning system makes this a non-trivial problem. We believe that this can be inferred by analysing the relation between calls, but further investigation is still necessary.

5.2.2 Emulation

On the emulation portion of AGAVE, there are also a number of features that we want to implement.

One of the methods used by malware writers to avoid analysis, or to make it harder, is to check for specific circumstances to execute the malicious activity; this method is known as "multiple execution paths". For example, if the malware detects that is running on an emulation, the code will not perform any malicious activity. Another approach is having malicious code that is activated only occasionally. For example, a malware may only perform its malicious tasks one tenth of the times it is executed[6]. We want AGAVE to be able to explore those multiple execution paths, checking for when a condition was not met and forcing the execution of the alternative path[44].

To deal with the poor performance of emulating CPU instructions, we want to implement our interface for other CPU emulators. JPC, for instance, a CPU emulator that is written in Java, claims to have a huge gain in performance over its original implementation[49]. We consider JPC an interesting candidate for the next emulator to have an interface in AGAVE.

A module for memory management within AGAVE, as we mentioned previously, is another feature that we want to implement. Currently, the memory is managed directly by the CPU emulator. This is not desirable because we lose portability, i.e., code implemented within AGAVE that deals with memory might be tied to the CPU emulator being used. This module ideally will also offer support to pointer tainting.

5.2.3 AGAVE in general

There is also some work to be done that is not specific to any module of AGAVE.

Currently, AGAVE can be used for malware analysis and for developing new methods of detection. However, we also want to implement a mechanism which will allow the researcher to embed AGAVE into third-party tools, such as an anti-virus product like ClamAV[37].

Another approach for addressing the performance problem would be a partial refactoring of AGAVE. We believe that the use of Python does give important flexibility for the researcher, that they can easily write their own code and customize AGAVE to their own needs. So, in order to keep this flexibility, we want to rewrite some of the AGAVE modules in C, especially the controller and the CBR modules, but keeping them as Python modules.

All of our tests with AGAVE were based on Linux. However, most of the malware found in the wild is for Windows platforms. A natural evolution for AGAVE is its adaptation to work with Windows. However, a number of challenges will have to be faced, such as lack of documentation and the number of available system and library calls. For example, while Linux has around 300 system calls defined, Windows has at least three times this number of system calls[67], many of them not well documented or not documented at all.

Finally, more immediate work will be the implementation of more complex heuristics for malware detection. One of the goals of AGAVE is to be a testbed for heuristics, providing an unbiased environment for evaluation and experimentation. From this implementation, we also want to evaluate how much effort from the researcher the use of AGAVE can save, when compared to implementing such heuristics using other tools for malware analysis.

References

[1] Linux standard base core specification 3.1. available online at: `http://refspecs.freestandards.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/book1.html`.

[2] Agnar Aamodt and Enric Plaza. Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[3] Randy Abrams. Understanding and teaching heuristics. In *Proceedings of 10th Annual AVAR International Conference*, 2007.

[4] Kareem Aggour, Marc Pavese, Piero Bonissone, and William Cheetham. Soft-CBR: A self-optimizing fuzzy tool for case-based reasoning. In Kevin Ashley and Derek Bridge, editors, *Case-Based Reasoning Research and Development*, volume 2689 of *Lecture Notes in Computer Science*, pages 1065–1065. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-45006-8_4.

[5] Jonathan R. C. Allen, David W. Patterson, Maurice D. Mulvenna, and John G. Hughes. Integration of case based retrieval with a relational database system in aircraft technical support. In *Proceedings of the First International Conference on Case-Based Reasoning Research and Development*, ICCBR '95, pages 1–10, London, UK, 1995. Springer-Verlag.

[6] John Aycock. *Computer Viruses and Malware (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[8] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

[9] Ulrich Bayer, Andreas Moser, Chirstopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1), August 2006.

[10] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[11] W. Boehmer. Analyzing human behavior using case-based reasoning with the help of forensic questions. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 1189 –1194, April 2010.

[12] Hans-Dieter Burkhard and Michael M. Richter. *Soft Computing in Case Based Reasoning*, chapter 2: "On the notion of similarity in case based reasoning and fuzzy theory", pages 29–45. Springer-Verlag, London, UK, 2001.

[13] Xu Chen, J. Andersen, Z.M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177 –186, June 2008.

[14] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, pages 1–14, 2008.

[15] F. Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

[16] Drew Copley. Computer behavioural management using heuristic analysis. Patent Application, April 2007. Patent Application US 2007/0,079,375.

[17] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

[18] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36:211–224, December 2002.

[19] EICAR. The Anti-Virus or Anti-Malware test file (version of 7 September 2006). Published on-line at : http://www.eicar.org/anti-virus-test-file.htm, 2006.

[20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings 2nd International Conference on Knowledge Discovery and Data Mining Portland OR AAAI Press*, pages 226–231. AAAI Press, 1996.

[21] Eric Filiol. Strong cryptography armoured computer viruses forbidding code analysis: the BRADLEY VIRUS. In *V. Broucek ed., Proceedings of the 14th EICAR Conference*, 2005.

[22] Eric Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(1):pp. 70–75, April 2007.

[23] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Conference on Workshop on Intrusion Detection and Network Monitoring*, pages 51–62. USENIX Association, 1999.

[24] Sarah Gordon. Technologically enabled crime: Shifting paradigms for the year 2000. *Computers & Security*, 14(5):391–402, 1995.

[25] Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh. A study of the packer problem and its solutions. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 98–115. Springer Berlin / Heidelberg, 2008.

[26] Frank Hofmann. AdvaS Advanced Search - version 0.2.3. Available on-line at http://advas.sf.net, January 2005.

[27] Hsiang-Lun Huang, Tzong-Jye Liu, Kuong-Ho Chen, Chyi-Ren Dow, and Lih-Chyau Wuu. A polymorphic shellcode detection mechanism in the network. In *InfoScale '07: Proceedings of the 2nd international conference on Scalable information systems*, pages 1–7, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[28] Intel. *Intel® CoreTM 2 Duo Mobile Processors on 45-nm process for Embedded Applications –Thermal Design Guide*, June 2008. Order Number: 320028-001.

[29] Intel. *Intel® CoreTM2 Duo Mobile Processor, Intel® CoreTM2 Solo Mobile Processor and Intel® CoreTM2 Extreme Mobile Processor on 45-nm Process – Datasheet*, March 2009. Document Number: 320120-004.

[30] Jacek Jarmulak, Susan Craw, and Ray Rowe. Genetic algorithms to optimise CBR retrieval. In Enrico Blanzieri and Luigi Portinale, editors, *Advances in Case-Based Reasoning*, volume 1898 of *Lecture Notes in Computer Science*, pages 159–194. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-44527-7_13.

[31] Myles Jordan. System and method for computer virus detection utilizing heuristic analysis. Patent, June 2007. US Patent 7,231,667.

[32] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53, New York, NY, USA, 2007. ACM.

[33] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *VMSec '09: Proceedings of the 1st ACM workshop on Virtual machine security*, pages 11–22, New York, NY, USA, 2009. ACM.

[34] Sandeep Karanth, Srivatsan Laxman, Prasad Naldurg, Ramarathnam Venkatesan, J Lambert, and Jinwook Shin. Pattern mining for future attacks. Technical Report MSR-TR-2010-100, Microsoft, July 2010. available online at: `http://research.microsoft.com/apps/pubs/default.aspx?id=135599`.

[35] P.A. Karger and D.R. Safford. I/O for virtual machine monitors: security and performance issues. *Security & Privacy, IEEE*, 6(5):16 –23, September/October 2008.

[36] Jong Yong Kim and John Shawe-Taylor. Fast string matching using an n-gram algorithm. *Software - Practice and Experience*, 24:79–83, 1994.

[37] Tomasz Kojm. `http://www.clamav.org`.

[38] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and Efficient Malware Detection at the End Host. In *18th USENIX Security Symposium*, 2009.

[39] Janet Kolodner. *Case-Based Reasoning.* Morgan Kaufmann Publishers, Inc., 1993.

[40] Kevin P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux Journal*, 29:7, 1996.

[41] Michael Locasto, Ke Wang, Angelos Keromytis, and Salvatore Stolfo. Flips: Hybrid adaptive intrusion prevention. In Alfonso Valdes and Diego Zamboni, editors, *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 82–101. Springer Berlin / Heidelberg, 2006. 10.1007/11663812_5.

[42] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *ISSTA '09: Proceedings of the eighteenth international symposium on software testing and analysis*, pages 261–272, New York, NY, USA, 2009. ACM.

[43] Y. Miyanokoshi, E. Sato, and T. Yamaguchi. Suspicious behavior detection based on case-based reasoning using face direction. In *SICE-ICASE, 2006. International Joint Conference*, pages 5429 –5432, October 2006.

[44] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, 1997.

[46] Carey Nachenberg. Understanding and managing polymorphic viruses. Available online at: `http://www.symantec.com/avcenter/reference/striker.pdf`, July 1999. Symantec Whitepaper.

[47] Carey Nachenberg. Histogram-based virus detection. Patent, November 2005. US Patent 6,971,019.

[48] Carey Nachenberg and Alex Haddox. Generic decryption scanners: The problems. *Virus Bulletin Magazine*, 8:6–8, August 1996.

[49] Rhys Newman and Chris Dennis. *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*, chapter 9: "JPC: An x86 Emulator in Pure Java". O'Reilly, 2009.

[50] Markus Franz Xaver Johannes Oberhumer, László Molnár, and John F. Reiser. UPX - The Ultimate Packer for eXecutables. Published on-line at http://upx.sourceforge.net, 2010.

[51] David S. Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, Berkeley, CA, USA, 2002. USENIX Association.

[52] Cody Pierce. PyEmu: A multi-purpose scriptable IA-32 emulator. In *Blackhat Conference*, 2007. Available online at: `https://www.blackhat.com/presentations/bh-usa-07/Pierce/Whitepaper/bh-usa-07-pierce-WP.pdf`,.

[53] Matt Pietrek. Poking Around Under the Hood: A Programmer's View of Windows NT 4.0. *Microsoft Systems Journal*, August 1996. Available online at: `http://www.microsoft.com/msj/archive/S413.aspx`.

[54] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 15–27, New York, NY, USA, 2006. ACM.

[55] Nicolas Puech and Eric Filiol. Computer viruses and applications. In *Computer viruses: from theory to applications*, Collection IRIS. Springer Paris, 2005. 10.1007/2-287-28099-5_11.

[56] Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. Available online at `http://invisiblethings.org/papers/redpill.html`, November 2004.

[57] Jörg Schaaf. Fish and shrink. a next step towards efficient case retrieval in large scaled case bases. In Ian Smith and Boi Faltings, editors, *Advances in Case-Based Reasoning*, volume 1168 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0020623.

[58] Silvia Schiaffino and Analia Amandi. User profiling with Case-Based Reasoning and Bayesian Networks. In *Open Discussion Track Proceedings of the International Joint Conference IBERAMIA-SBIA 2000*, pages 12–21, 2000.

[59] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 541–551, New York, NY, USA, 2007. ACM.

[60] Richard Stallman et al. *GNU Coding Standards*. Free Software Foundation, September 2010.

[61] Gabor Szappanos. Exepacker blacklisting. *Virus Bulletin*, October 2007.

[62] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[63] S. G. Tucker. Emulation of large systems. *Communications of the ACM*, 8(12):753–761, 1965.

[64] Peter A. J. van der Made. Computer immune system and method for detecting unwanted code in a computer system. Patent, August 2006. US Patent 7,093,239.

[65] Mark van Setten, Mettina Veenstra, Anton Nijholt, and Betsy van Dijk. Case-Based Reasoning as a Prediction Strategy for Hybrid Recommender Systems. In Jesus Favela, Ernestina Menasalvas, and Edgar Chvez, editors, *Advances in Web Intelligence*, volume 3034 of *Lecture Notes in Computer Science*, pages 13–22. Springer Berlin / Heidelberg, 2004.

[66] Vmware, 1998. http://www.vmware.com/.

[67] Miao Wang, Cheng Zhang, and Jingjing Yu. Native API based Windows anomaly intrusion detection method using SVM. In *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, volume 1, June 2006.

[68] Jun Xu and Nithin Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[69] Trevor Yann and Oleg Petrovsky. Detection of polymorphic virus code using dataflow analysis. Patent, 6 2006. US Patent 7,069,583.

Appendix A

List of Linux commands used for casebase generation

```
bzcat                      lsmod
bzip2                      mknod
bzip2recover               mount
cat                        mv
chgrp                      nano
chown                      nc.traditional
chvt                       netstat
cpio                       ntfs-3g
dbus-cleanup-sockets       ntfs-3g.probe
dbus-daemon                openvt
dbus-uuidgen               ping
dd                         ping6
df                         ps
dnsdomainname              pwd
egrep                      readlink
false                      rm
fgconsole                  rmdir
fuser                      sed
fusermount                 setfont
ip                         sleep
kbd_mode                   stty
kill                       su
less                       sync
lessecho                   tar
lesskey                    tempfile
ln                         touch
login                      true
ls                         ulockmgr_server
```

Appendix B

List of functions

B.1 Functions implemented in AGAVE

- execute(*command*)
- exit()
- get_breakpoint_list()
- get_custom_instruction(*instruction*)
- get_heuristic_list()
- get_library_call(*name*)
- get_library_call_list()
- get_system_call(*name*)
- get_system_call_list()
- load(script_name)
- memory_dump(address, size)
- memory_save(filename, address, size=4096, append=False)
- next()
- pause()
- remove_breakpoint(*breakpoint*)
- run()
- set_breakpoint(breakpoint)
- set_custom_instruction(instruction, action, size)
- set_heuristics(name, action, before=True)
- set_opcode(name, action, parameters=None)
- set_register(register, value)

- set_library_call(name, action, parameters=None)
- set_system_call(name, action, parameters=None)
- stop()
- get_arguments()
- set_arguments(args)

B.2 Functions implemented in CPU emulator interface

- get_context()
- get_instruction_pointer()
- get_memory(address, size=4)
- get_register(register)
- get_instruction_pointer()
- set_instruction_pointer(addr)
- get_stack_pointer()
- load_snapshot(filename=None)
- set_stack_pointer(addr)
- restart()
- set_return_code(return_value)
- save_snapshot(filename=None)
- set_memory(address, value, size=4)
- stack(size=64)
- startenv()
- step()

Appendix C

Implementation of interface to CPU emulator (PyEmu)

```
#!/usr/bin/env python
import os, sys

sys.path.append("third-party")
sys.path.append("third-party/pyemu")
sys.path.append("third-party/pyemu")
sys.path.append("learner")
import cPickle
import re
from agave import *

class Agave2PyEmu(AGAVE):

    def save_snapshot(self, filename=None):
        if filename == None:
            filename = self.get_snapshot_name()

        exe = self.executable
        crc = self.header_hash

        memory = {}
        for page_addr in self.emu.memory.pages:
            page = self.emu.memory.pages[page_addr]
            memory[page_addr] = {"Address":page.address, "
               Permissions":page.permissions, "Data":page.data
               [::-1]}

        context = self.emu.cpu.get_context()

        registers = {}
        for register in dir(context):
            value = getattr(context, register)
            if type(value) in [type(0), type(0L)]:
                registers[register] = value

        filepath = "%s/%s%s"%(self.SNAPSHOT_DIR,filename, self.
           SNAPSHOT_EXTENSION)
        cPickle.dump([exe, crc, registers,memory, self.
           __CODE_PAGES__, self.emu.os.TLS], open(filepath ,"w"))
        self.log.info("Snapshot saved as %s"%filename)
```

```
def load_snapshot(self, filename=None):
    if filename == None:
        filename = self.get_last_snapshot()
        if filename == None:
            self.log.error("No snapshot found for this
               executable")
            return False

    self.startenv(False)

    filepath = "%s/%s%s"%(self.SNAPSHOT_DIR, filename, self.
       SNAPSHOT_EXTENSION)
    [exe, crc, new_context, memory, self.__CODE_PAGES__, self
       .emu.os.TLS] = cPickle.load(open(filepath ,"r"))

    if exe != self.executable or crc != self.header_hash:
        self.log.error("Invalid snapshot for this executable"
           )
        return False

    self.emu.memory.pages = {}

    for page_addr in memory:
        page = memory[page_addr]

        self.set_memory(page_addr, page["Data"], len(page["
           Data"]))
        self.emu.memory.pages[page_addr].permissions = page["
           Permissions"]

    context = self.emu.cpu.get_context()
    for reg in new_context.keys():
        value = new_context[reg]
        setattr(context, reg, value)
    self.emu.cpu.set_context(context)

    self.log.info("Snapshot loaded from %s"%filename)

def get_context(self):
    cpu_context = self.emu.cpu.get_context()
    context = {}
    for register in dir(cpu_context):
        if "_" not in register:
            context[register]=getattr(cpu_context,register)
    return context
```

```
def __init__(self, exename, address=0x80000000):
    AGAVE.__init__(self) # Starts AGAVE variables

    self.executable = exename

    self.load(exename, address)
    self.startenv()
    self.count_pushes = 0
    self.SYSCALL_INSTRUCTION = "int 0x80"
    self.RELOCATION_INSTRUCTION = "call"
    self.header_hash = self.get_header_hash() # Doing this to
        improve


def set_return_code(self, return_value):
    self.log.debug( "Returning %d (setting EAX)"%return_value
        )
    self.emu.set_register("EAX", return_value)

def get_memory(self, address, size=4):
    data=None
    try:
        data = self.emu.get_memory(address, size)
    except:
        self.log.error("Error reading memory")
        data=0
    return data

def get_memory_pages(self):
    return self.emu.memory.pages.keys()

def set_memory(self, address, value, size=4):
    return self.emu.set_memory(address, value, size)

def step(self):
    if self.check_jump_to_data():
        self.check_for_elf()
    try:
        result = self.emu.execute()
    except Exception as exc:
        (type_exc, value, traceback) = sys.exc_info()
        self.log.error( "Unexpected error(%s): %s"%(type_exc,
            value))
        sys.exc_clear()
        self.set_batch(False) # Forces a pause
```

```
            if self.LOG_ALL:
                self.stack(16)
                self.registers()
            result = False
        return result

    def stack(self, size=64):
        self.emu.dump_stack(size)

    def registers(self):
        self.emu.dump_regs()

    def get_instruction(self):
        if self.emu == None:
            return False
        instruction = self.emu.cpu.get_disasm()
        return instruction

    def get_stack_pointer(self):
        if not self.emu.frame_pointer:
            return self.get_register("EBP")
        else:
            return self.get_register("ESP")

    def set_stack_pointer(self,addr):
        if not self.emu.frame_pointer:
            self.set_register("EBP",addr)
        else:
            self.set_register("ESP",addr)

    def load_relocation_table(self, elf_addr):
        # get one page of data. Sufficient for header?
        data = self.get_memory(elf_addr, 4096)
        elf = elffile.ELF(data = data, parse_all = False)
        elf.parse_program_headers(elf.ELF_HEADER.e_phoff, elf.
           ELF_HEADER.e_phnum, elf.ELF_HEADER.e_phentsize)
        addr = 0
        size = 0
        for x in xrange(len(elf.program_headers)):
            ph = elf.program_headers[x]
            if ph.p_type == elffile.SEGMENT_TYPES["PT_DYNAMIC"]:
                (addr, size) = (ph.p_vaddr, ph.p_memsz)
                segment = self.get_memory(addr, size)
                elf.parse_dynamic_segment(segment)
                break
        if elf.dynamic_tags == []:
```

```
        return False

    tags={}
    for tag in elf.dynamic_tags:
        if tag.d_tag in elffile.DYNAMIC_TAGS:
            tags[elffile.DYNAMIC_TAGS[tag.d_tag]] = tag.d_val

    relocation = {}

    try:

        if "DT_REL" in tags:
            addr = tags["DT_REL"]
            size = tags["DT_RELENT"]
            numb = tags["DT_RELSZ"]
        else:
            addr = tags["DT_RELA"]
            size = tags["DT_RELAENT"]
            numb = tags["DT_RELASZ"]
        addr = tags["DT_JMPREL"]

        previous_offset = None
        for reloc in xrange(numb):
            offset = self.get_memory(addr,4)
            if previous_offset != None:
                if offset != previous_offset + 0x04:
                    break
            previous_offset = offset
            info = self.get_memory(addr+4,4)
            symbol = info >> 8

            symbol_addr = tags["DT_SYMTAB"] + tags["DT_SYMENT
               "]* symbol
            symbol_data = self.get_memory(symbol_addr, 4)
            symbol_name = self.get_string(tags["DT_STRTAB"]+
               symbol_data,256)
            relocation[offset] = symbol_name

            addr += size
    except:
        return False
    for addr in relocation:
        self.library_calls[addr] = relocation[addr]

    return True
```

```
def check_library_call(self, instruction):
    if self.RELOCATION_INSTRUCTION not in instruction:
        return False
    (call, addr) = instruction.split()
    try:
        address = int(addr[2:],16)
    except:
        return False # If I cannot do that, there's no chance
            of relocation
    self.log.info("Checking relocation instruction for %s"%(
       instruction))

    data = self.get_memory(address, 16)
    # Checking if the next instructions match a stub
       signature from ELF
    # ff 25 XX XX XX XX    jmp [XXXXXXXX] (GOT -> Next
       instruction)
    # 68 XX XX XX XX       push dword XXXXXXXX
    # e9 XX XX XX XX       jmp [XXXXXXXX] (PLT)
    if not (data[0] == chr(0xff) and data[1]==chr(0x25) and
       data[6] == chr(0x68) and data[11] == chr(0xe9)):
        self.log.debug("Stub signature doesn't match")
        return False

    GOT_address = self.get_memory(address+2,4)
    GOT_instr = self.get_memory(GOT_address,4)
    if GOT_instr  != (address + 6):
        self.log.debug("Address doesn't match (%08x != %08x)"
           %(GOT_instr, address+6))
        return False

    offset = self.get_memory(address+7,4)
    PLT0_offset = self.get_memory(address+12,4)
    PLT0_address = (address + 16 + PLT0_offset )&0xffffffff
       # 16 is total size of the stub

    self.log.debug("GOT Address: %08x"%GOT_address)
    self.log.debug("Offset     : %08x"%offset)
    self.log.debug("PLT0 Addr  : %08x"%PLT0_address)

    if GOT_address in self.library_calls.keys():
        self.emu.execute()
        return self.emulate_library_call(GOT_address)
    else:
        return False
```

```
def check_system_call(self, instruction):
    if self.SYSCALL_INSTRUCTION in instruction:
        call = self.get_register("EAX")
        syscall = self.retriever.get_system_call(call)
        if len(syscall) != 0:
            name = syscall[0].name
            self.log.info( "SYSCALL TO %s at %08x"%(name,
               self.get_register("EIP")))
            if name in self.system_calls.keys():
                self.set_register("EIP", self.get_register("
                   EIP")+2) # TODO: improve this adding 2 to
                   EIP (int 0x80 = 2 bytes)
                self.log.info( "Executing user code for %s as
                    %s at EIP=%08x"%(name, self.system_calls[
                   name][0], self.get_register("EIP")) )
                self.process_command(self.system_calls[name
                   ][0])
                return True
            else:
                parameters = []
                for register in ("EBX","ECX","EDX","ESI","EDI
                   "):
                    parameters.append(self.get_register(
                       register))
                self.set_register("EIP", self.get_register("
                   EIP")+2) # TODO: improve this adding 2 to
                   EIP (int 0x80 = 2 bytes)
                self.log.info( "Executing code for %s from
                   casebase"%(name) )
                return self.learned_exec(name, parameters,
                   SYSCALLS=True)
        else:
            self.log.error("System call %d was not identified
               "%call)
    return False

def check_custom_instructions(self, instruction):
    if instruction in self.custom_instructions:
        self.process_command(self.custom_instructions[
           instruction][0])
        self.set_register("EIP", self.get_register("EIP")+
           self.custom_instructions[instruction][1])
        return True
    else:
        cmd = instruction.split()[0]
        if cmd in self.custom_instructions:
            self.process_command(self.custom_instructions[cmd
               ][0])
            self.set_register("EIP", self.get_register("EIP")
               +self.custom_instructions[cmd][1])
            return True

    return False

def skip(self, n):
    self.set_register("EIP", self.get_register("EIP")+n )


def get_parameters_from_stack(self, max_parameters=5):
    address = self.get_register("ESP") ## Return address
    self.log.fine("ESP = %08x"%address)

    parameters = []
    for i in xrange(max_parameters):
        address += 4
        data = self.get_memory(address, 4)
        self.log.debug( "Parameter %d from stack at %08x: %08
           x"%(len(parameters), address, data) )
        parameters.append(data)

    return parameters


def guess_syntax(self, max_parameters = 5):
    syntax = []

    entrypoint = self.elf.ELF_HEADER.e_entry
    intervals = []

    parameters = self.get_parameters_from_stack(
       max_parameters)

    for table in (".strtab",".dynstr"):
        if table in self.elf.sections:
            start_strtable = entrypoint + self.elf.sections[
               table].sh_offset
            end_strtable = start_strtable + self.elf.sections
               [table].sh_size
            intervals.append((start_strtable, end_strtable))

    for i in xrange(len(parameters)):
            data = parameters[i]

            if data < entrypoint: # This should be a number
                syntax.append(agave_types.AgaveType("NUMERIC"))
            else:
                is_string = False
                for (start,end) in intervals:
                    if start <= data <= end:
                        syntax.append(agave_types.AgaveType("
                           STRING"))
                        is_string = True

                if not is_string: # Large number or address
                    syntax.append(agave_types.AgaveType("NUMERIC"
                       ))
        self.log.debug("Guessed syntax: %s"%syntax)
        return syntax


    def get_syntax(self, name, parameters=[], SYSCALLS=False):
        self.log.debug("Getting syntax for %s"%name)
        if name == "puts" and 1 == 2:
            self.log.debug( "Special case for testing: PUTS" )
            syntax = [agave_types.AgaveType("STRING")]
            syntaxes = [syntax]
        else:
            syntaxes = self.retriever.get_syntax(name, SYSCALLS=
               SYSCALLS)

        if len(syntaxes) > 0:
            for syntax in syntaxes:
                self.log.debug( "Syntax found: %s"%syntax )
        else:
            self.log.debug( "Syntax for this call was not found"
               )
            syntax = self.guess_syntax()

        return syntaxes


    def finish_call(self, n_parameters):
        return_address = self.pop() # Removing return address
           from stack
        self.log.debug("Return address: %08x"%return_address)
        for i in xrange(n_parameters):
            self.pop()          # Removing each parameter from
            stack

    self.emu.set_register("EIP",return_address)


def get_string(self, address, max_string_size=2048):
    _len = len
    _chr = chr
    string = ""
    char = self.get_memory(address,1)
    while char != 0 and _len(string) < max_string_size:
        string = string + _chr(char)
        address += 1
        char = self.get_memory(address,1)
    return string

def load(self, exename, address=0x800000):
    if exename:
        self.executable = exename
        try:
            elf = elffile.ELF(exename)
        except:
            self.log.error( "Invalid executable")
            sys.exit(1)
    else:
            self.log.error( "Blank filename specified" )
            sys.exit(2)

    self.elf = elf

def get_instruction_pointer(self):
    return self.get_register("EIP")

def set_instruction_pointer(self,addr):
    self.set_register("EIP",addr)

def startenv(self, initial=True):
    elf = self.elf
    entrypoint = elf.ELF_HEADER.e_entry
    self.log.debug( "Entry Point Addr: 0x%08x\n" % (
       entrypoint))

    self.emu = elfpyemu.ELFPyEmu()

    for x in xrange(len(elf.program_headers)):
            ph = elf.program_headers[x]
            segment = elf.segments[x]

            self.log.debug( "Loading segment %d (size=%x/%x) at %
               x"%(x,len(segment),ph.p_memsz, ph.p_vaddr) )
            if ph.p_type not in elffile.SEGMENT_TYPES:
                segment_type = "UNKNOWN(%s)"%ph.p_type
            else:
                segment_type = elffile.SEGMENT_TYPES[ph.p_type]

            self.log.debug("%-20s\t%08x\t%08x\t%08x\t%08x\t%08x"
               %(segment_type, ph.p_offset, ph.p_vaddr, ph.
               p_paddr, ph.p_filesz, ph.p_memsz))

            if initial:
                for i in xrange(len(segment)):
                    c = segment[i]
                    self.emu.set_memory(ph.p_vaddr+i, int(ord(c))
                        , size=1)

        for section_name in elf.sections:
            section = elf.sections[section_name]
            self.log.debug( "Loading section %s (size=%x/%x) at %
               x"%(section_name,len(section.data),section.sh_size
               , section.sh_addr) )
            data = section.data
            address = section.sh_addr
            if initial:
                for i in xrange(max(len(data), section.sh_size)):
                    if i >= len(data):
                        c = chr(0)
                    else:
                        c = data[i]

                    self.emu.set_memory(address + i, int(ord(c)),
                        size=1)

        self.emu.set_register("EIP", entrypoint)

        if initial:
            self.start_stack()

        self.__CODE_PAGES__ = self.emu.memory.pages.keys()

        self.finished = False
```

```
def start_stack(self):
    self.log.debug("Initializing stack with args and
       environment variables...")
    addr = self.emu.memory.get_available_page(0x00ff0000)

    self.push(0)    # NULL (ending list of environment
       variables)
    env_path = "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin
       :/usr/bin:/sbin:/bin:/usr/games"
    self.set_memory(addr, "%s\x00"%env_path, len(env_path)+1)
    self.push(addr) # argv

    self.push(0)    # NULL (ending list of arguments)
    addr = addr + len(env_path)
    self.set_memory(addr, "%s\x00"%self.executable[::-1], len
       (self.executable)+1)
    self.push(addr) # argv[0]
    addr = addr + len(self.executable)+1
    self.push(1)    # argc


def check_jump_to_data(self):
    EIP=self.get_register("EIP")
    if EIP & 0xfffff000 not in self.__CODE_PAGES__:
        self.__CODE_PAGES__ = self.emu.memory.pages.keys()
        self.save_snapshot()
        return True
    return False

# Check for ELF necessary for reload relocation table
def check_for_elf(self):
    result = False
    if "__ELF_PAGES__" not in dir(self):
        self.__ELF_PAGES__=[]
    for page in self.emu.memory.pages.keys():
        data = self.get_memory(page, len(elffile.
           ELF_SIGNATURE))
        if data == elffile.ELF_SIGNATURE:
            if page not in self.__ELF_PAGES__:
                self.__ELF_PAGES__.append(page)
                self.load_relocation_table(page)
                result = True
    return result

def pop(self, size=4):
        self.log.debug("Frame pointer? 0x%08x"%self.emu.
           frame_pointer)
        if not self.emu.frame_pointer:
            address = self.emu.get_register("EBP")
        else:
            address = self.emu.get_register("ESP")

        self.log.debug("Address = %08x"%address)
        value = self.get_memory( address, size )
        self.emu.set_memory(address, 0x00, size)

        address += size

        if not self.emu.frame_pointer:
            self.emu.set_register("EBP", address)
        else:
            self.emu.set_register("ESP", address)
        return value

    def push(self, value, size=4):
        if not self.emu.frame_pointer:
            address = self.emu.get_register("EBP")
        else:
            address = self.emu.get_register("ESP")

        address -= size

        self.emu.set_memory( address, value, size )

        if not self.emu.frame_pointer:
            self.emu.set_register("EBP", address)
        else:
            self.emu.set_register("ESP", address)
        return value

    def show_instruction(self):
        address = self.emu.get_register("EIP")
        if address == 0:
            self.log.info( "%08x\tProgram finished\n"%(address))
            self.finished = True
            self.set_batch(False)
        else:
            raw_instruction = self.get_memory(address, 32)

            if type(raw_instruction) != type(""):
                self.log.error( "Error reading instruction at 0x
                %08x (%s)"%(address,raw_instruction))
            return

        instruction = pydasm.get_instruction(raw_instruction,
            pydasm.MODE_32)
        bytes = ""
        for i in xrange(instruction.length):
                bytes += ("%02x "%ord(raw_instruction[i]))

        disasm = self.emu.get_disasm()
        self.log.info("[ASM] %08x\t%-30s\t%s"%(address, bytes
            , disasm))

def check_finished(self, instruction):
    if "hlt" in instruction:
        self.log.debug( "Program execution completed." )
        return True
    elif self.emu.get_register("EIP") == 0:
        self.log.debug( "Instruction pointer is empty. " )
        return True
    return False

def check_stack(self):
    esp = self.get_register("ESP")
    stack_page = (esp & 0xfffff000)
    if stack_page not in self.emu.memory.pages:
        self.log.warn("Memory page for stack not available...
            Initializing %08x for %08x...\n"%(stack_page, esp
           ))
        empty_page = "\x00"*4096
        self.set_memory(stack_page, empty_page, 4096)

def set_register(self, register, value):
    if type(register) == type(False) and reg == False:
        self.log.error( "Invalid register %s"%register)
        self.set_batch(False)
        return
    else:
        if type(value) == type(""):
            if value[0:2] == "0x":
                if not self.emu.set_register(register, int(
                    value[2:],16)):
                     self.log.error( "Error setting register %
                        s=%d(%08x)"%(register, value, value) )

            elif not self.get_register(value) and type(self.
                    get_register(value)) == type(False):
                     source = value.upper()
                     value = self.get_register(source)
                     if not value:
                         self.log.debug( "Invalid register: %s"%
                            source )
                         return
                     else:
                         if not self.emu.set_register(register,
                            value):
                             self.log.debug( "Error setting
                                register %s=%s(%08x)"%(register,
                                source, value) )

            else:
                if not self.emu.set_register(register,value):
                    self.log.debug( "Error setting register %s=%d
                       (%08x)"%(register, value, value) )

    def get_register(self, register):
        reg = self.emu.get_register(register)
        if type(reg) == type(False) and reg == False:
            self.log.error( "Error getting register %s"%register
               )
            return 0
        else:
            return reg
```

Appendix D

Implementation of instructions within AGAVE

D.1 Implementing STOSD

```
def STOSD():
    edi = get_register("EDI")
    df = get_register("DF")
    eax = get_register("EAX")

    set_memory( edi , eax,4 )

    if df == 0:
        set_register("EDI",edi+4)
    else:
        set_register("EDI",edi-4)

set_custom_instruction("stosd", "STOSD()", 1)
```

D.2 Implementing BTR

```
def BTR():

    # Return instruction as a mnemonic
    instruction = get_instruction()

    (cmd, operand_list) = instruction.split()
    operands = operand_list.split(",")

    values = []
    debug_msg = []
    for op in operands:
        try:
            value = get_register(op)
        except:
            interface.log.error("ERROR geting op=%s"%op)
            value = 0
        if value == None:
            value = 0
        values.append(value)
    base = values[0]
    bit_offset = 2 ** values[1]

    if (base & bit_offset) > 0:
        base = base - bit_offset
        cf =  1
    else:
        cf = 0

    set_register("CF",cf)
    set_register(operands[0], base)

# 3 bytes, instruction + 2 operands
set_custom_instruction("btr","BTR()",3)
```

Appendix E

Implementation of heuristics within AGAVE

```
from elffile import ELF_SIGNATURE

def jumping_to_data():
    if "CODE_PAGES" not in dir():
        CODE_PAGES = []
        EIP=get_instruction_pointer()
        if EIP & 0xfffff000 not in CODE_PAGES:
                CODE_PAGES = get_memory_pages()
                return True
        return False


def elf_in_memory():
    if "ELF_PAGES" not in dir():
        ELF_PAGES=[]
        for page in get_memory_pages():
                data = get_memory(page, len(ELF_SIGNATURE))
                if data == ELF_SIGNATURE:
                        if page not in ELF_PAGES:
                                ELF_PAGES.append(page)
                                result = True
        return result

def search_EICAR():
        EICAR_SIGNATURE = ""
        for page in get_memory_pages():
                if EICAR_SIGNATURE in get_memory(page, 4096):
                        print "Found"
                        return True
        return False

def find_eicar():
    print "Find EICAR",
    if jumping_to_data() and elf_in_memory():
        print "Jumping into data section"
        if search_EICAR():
            print "EICAR Standard Test File was found"
            pause()
    else:
        print "Nothing yet"

set_heuristics("FindEICAR", "find_eicar()")
```