eman ta zabal zazu

Universidad del País Vasco | Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

Bagheera: an Advanced Polymorphic and Infection Engine for Linux

Author

Diego Carballeda Martinez

2021

eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering
Computer science

End of degree work

Bagheera: an Advanced Polymorphic and Infection Engine for Linux

Author

Diego Carballeda Martinez

Director(s)
Jose A. Pascual

Summary

Computer viruses have been evolving since the ’80s, adopting new techniques with the intention of avoiding being detected by anti-virus programs. One of these techniques is polymorphism, which is used to change the virus’ structure each time an infection is carried out. This technique was broadly adopted by the virus-writing community and led to the birth of **Polymorphic Engines**, which can grant polymorphism to any virus.

This project focuses on the study of those engines and, in particular, on exploring the techniques used to avoid detection from anti-viruses. In addition, this project also focuses on the analysis and development of techniques to infect ELF binaries on Linux platforms.

The final goal is to design and build a modern polymorphic and infection engine, namely *Bagheera*, and to evaluate its effectiveness against a state of the art anti-virus in a Linux platform.

Disclaimer

This thesis contains software that can be used as a viral threat. The author of this document is not responsible for possible damages that may be caused by using this software. The contents of the document are purely academic and should not be used with any other purpose.

*Mila esker, nire familiari, uneoro nire alboan egotegatik
eta nigan beraien konfiantza sendoena jartzeagatik.*

*Jon, Haizea, Aitor eta fakultateko lagun guztiei,
azken 4 urteetan izandako beste 'familiari',
hain polita den etapa honetan bidelagun onenak izategatik.*

*Eta azkenik nire tutoreari, Jose, informatikako esparru politena
esploratzera animatzeagatik eta emandako babesagatik.*

Contents

List of Figures

List of Tables

List of source codes

1. CHAPTER

Introduction

When it concerns computer security, there is no more well-known term as *computer virus*. Although the term is often misused when referring to worms or malware in general, a self-replicating program is the simplest form of a virus. For most users, these malicious programs infringe unwanted and annoying damage to computer systems, and often anti-virus (AV) software is the easiest and simplest solution to take care of them.

From the virus writers' perspective, there is no bigger enemy than a solid anti-virus. Starting back around the '80s, a fierce fight has been fought through the years between them. Back in the day, virus writers had no anti-viral opposition and could make their work spread as freely as they wanted. Reacting to the dawn of viruses, anti-virus software appeared to appease this threat. There have been several battles over the last few years between the two of them, regarding virus encryption, signature-based scanning or code emulation based detection [1]. Nevertheless, the war still has continued to this day.

In the last part of the 1980s, the idea of using encryption to scramble the presence of a virus was motivated by the fact that antivirus software could identify infections by examining executable files looking for unique viral footprints. As it will be detailed in Section 4.1, this encryption has nothing to do with modern cryptography algorithms.

Motivated by improvements in AVs to detect encryption, viruses adapted to this new environment, adopting a new technique: polymorphism. Polymorphic viruses are ever-changing programs, which challenge signature-based scanning, through morphs[1] of dif-

[1]In biology, polymorphism is the occurrence of two or more different morphs or forms in the population of a species.

ferent sizes and shapes. This ground-breaking technique fascinated the virus-exchange scene, gaining rapid popularity among virus writers.

Every hacker wanted to develop a polymorphic virus, so polymorphic engines (PE) emerged. This tiny module, ready to be attached to any virus, could turn any virus into polymorphic. This new invention increased the popularity of polymorphic viruses. Contributions like *Dark Avenger*'s Mutation Engine (DAME) [2, 3] made these new viruses more accessible and easy to create. As a consequence, their presence in the community increased dramatically.

Even though the first polymorphic virus appeared in 1990, F. Cohen had already introduced the concept of evolution in viruses. In his work, Cohen also stated that the problem of virus detection is undecidable, concluding that: a program that precisely discerns a virus from any other program by examining its appearance is infeasible [4]. With the actual implementation of polymorphic viruses, the problem defined by Cohen is taken to the next level, generating more transformations than those contemplated in his work. Additionally, a recent investigation conducted by D. Spnellis showed that the identification of polymorphic viruses is NP-complete [5].

Throughout history, most viruses have targeted Microsoft Windows family operating systems. Although UNIX-like operating systems are not immune to viruses, their sheer market share has prevented them from being attacked. It is believed that UNIX systems are not susceptible to a virus, but the reality is that the almost non-existent presence of viruses for this platform has led to its users believing that they cannot be attacked by this type of threat.

This project aims to understand the inner workings of a polymorphic virus from a practical point of view, analyzing and developing the basic structure of a polymorphic engine, i.e. Bagheera. Furthermore, various advanced techniques are developed, used to provide additional stealthiness to the engine. The project also ventures into the world of infecting software, implementing a simple yet efficient infection algorithm for Linux machines. Finally, the effectiveness of Bagheera is put to the test against anti-virus software, analyzing if the engine is able of concealing the presence of a virus.

2. CHAPTER

Aims of the project

The project aims to design and build a polymorphic engine that can be linked to any piece of software to evade anti-virus detection. To reach the final goal, the project must be broken down into various steps.

In the first step of the project, a first core implementation of a polymorphic engine must be developed. Future steps need a solid base to evolve correctly, so this basic implementation of the PE must cover all the necessary aspects an engine like this demands. Throughout recent history, various engines have been published in the hacking community. All these implementations share a common structure, which is known to be efficient and proper. Therefore, the PE this project comprehends will follow this common structure.

The second step in the project is the enhancement of the polymorphic engine, adding advanced features to its core functionality. These features make the detection process of anti-virus software more complicated. Countless advanced techniques could be bound to the engine to make it more robust against detection. Just a few of these techniques will be developed since adding more would increase the level of complexity of the project. Additionally, the third step of the project will focus on providing viral properties to the engine. This will be carried out by implementing an infection method for Linux binaries.

Lastly, the engines main purpose must be tested. It must be capable of evading anti-virus detection in a real-world environment. Hence, an open-source antivirus, *ClamAV*, will be protecting a system that the virus has to infect without being noticed by the AV. If the detection software is capable of spotting the virus, the design and implementation of the PE must be revised so that polymorphic weaknesses can be solved. Custom virus signatures can be added to *ClamAV* to mimic a much more harsh environment in which the virus has to spread. By doing this, the PE can be tested to the edge.

One of the main purposes of Bagheera is to be a lightweight addition to existing software to offer polymorphic capabilities. This is why the implementation of the engine must be small and concise. To ensure this goal is met, the engine could be implemented in assembler language. Although the performance of this programming language in this kind of tasks is unbeatable, some kind of abstraction is needed. Therefore, the polymorphic engine will be developed in `C++`, making use of the `AsmJit` library which offers the possibility to write executable code at execution time.

In conclusion, the main tasks that must be developed to reach the projects main purpose are the following:

1. Development of a basic PE implementation
2. Enhancement of the PE via advanced polymorphic techniques
3. Linux binaries infection algorithm implementation
4. Testing the PE against *ClamAV*

3. CHAPTER

Preliminaries

Computer viruses have been around for a long time. Since the first glimpses of modern computers were imagined, self-replicating pieces of software have been present. One of the founders of today's modern computer architecture, John von Neumann, apart from defining the core of advanced computers [6] also contributed to the viral aspect of computation. In his essay back in 1949, he introduced the concept of self-replicating programs and how to design them [7]. That work can be considered as the first formal description of a computer virus, although it was a theoretical work.

Years after von Neummans publication on self-replicating programs, the first computer viruses appeared in the 70's decade. Although these programs followed von Neumann's approach to self-replicating programs, the end goal of these pieces of code was not just mere replication, but to inflict some damage in the system they resided on. These malicious actions are carried out by the virus "payload". This part of the program is in charge of damaging the system. Also, once one system was infected, there was a need to spread and find another susceptible host. The part in charge of searching for a new host is the virus infection algorithm. These two features are the key elements for computer viruses [8]. A clear example of this kind of virus is the primitive `BRAIN` released in 1986. This DOS-based malicious program was one of the first functional evil self-replicating software that emerged in the dawn of the virus era [1].

The computer community considered these pieces of software as a huge threat, and thus a counter-movement occurred; the beginning of anti-virus software. Anti-virus or AV programs tried to detect the presence of previously known computer viruses in computer systems. This detection was performed using pattern recognition, also known as string signatures.

These signatures are just a sequence of bytes stored in anti-virus databases that are extracted from an actual virus. When a program is suspicious of containing a virus, its code is analyzed and the part that is believed to carry out evil actions is the one that forms the series of bytes that unequivocally identify the virus. This sequence is usually not found in normal programs and yells the presence of malicious behaviour in the program that holds these bytes.

Using this pattern, anti-virus software looks for this exact sequence inside binary files. When it finds a match, it tags the binary as a virus, taking the appropriate measures for its neutralization [9]. Similarly, heuristic rules are also used by these programs to try and detect viruses that have yet not been discovered but behave in a similar way to a previously analyzed virus. Although this method is quite accurate for detecting macro viruses[1] [10], the amount of false positives it generates with binary viruses makes this technique not that appropriate for this kind of viral threat. Although they are still used in modern AVs, different virus types often require completely different heuristic rules, making this method quite ineffective [11].

Even though hiding their presence was not a concern in the early stages of computer virus development, the arising of anti-virus software made the virus community look for clever ways to avoid being detected. In this context, various concealment techniques, or self-protection techniques, bloomed and made virus detection way harder for anti-virus software.

As mentioned, the earliest and most primitive technique was encryption [12]. Since basic detection systems focus on analyzing unique patterns in the viruses binaries, encrypting this information onto a nonsense collection of bytes cleared the problem away. The virus is divided into two main parts: a decryption function and the actual body of the virus. The only piece of code that the virus leaves unencrypted is the decryption function. When the virus is executed, the decryption function decodes and gives control to the encrypted code. This way, the actual instructions that define the virus are encrypted until they need to be executed. The only piece of readable and sensible code is the decryption function, which normally is very small and simple to not raise suspicions.

Later, anti-virus software also evolved to adapt to this new kind of viruses. Apart from trying to find malicious behaviour patterns in binary files, lots of effort were put into

[1]Computer viruses that use an application's macro programming language to distribute themselves.

finding basic decryption routines. The same detection method as for virus body signature scanning is applied, but in this case, spotting a decryption function is the goal [13]. These functions had a similar structure, which was easily detectable by scanners. In fact, most viruses have a unique decryption routine. This is the biggest flaw of encrypted viruses.

Motivated by this sheer amount of decryption function variations, virus writers developed the technique which this thesis focuses on: polymorphism.

Polymorphic viruses are in some way, an advanced evolution of encrypted viruses. They maintain the encrypted payload, but the decryption function no longer has a unique form. Polymorphic viruses are born aiming to avoid detection schemes based on string signatures. These systems are based on the idea that viruses always preserve several stable bytes in each generation, so the polymorphic virus changes its decryption function in every infection it performs, but still maintaining the same semantics. Every polymorphic virus maintains its semantics intact whilst using different instructions to do so. Generally, these instructions or `opcodes` are chosen at random. A polymorphic engine has to have a large number of possible random equivalences to have a solid structure.[14, 15, 16].

Although this new virus type might look quite undetectable at first, anti-virus software has evolved the same way viruses have done, adapting their detection mechanisms to this kind of techniques. A very common approach to detecting polymorphic viruses is quite simple: let the virus decrypt its payload, and analyze this decrypted data [11].

Letting the virus decrypt its malicious code and then analyzing it is a very effective technique, as well as costly in terms of computation. Although detecting an infected program is the main function of an antivirus, it cannot use the system resources to do so. The detection has to be a light process that allows users of a system to use it without any inconvenience. In contrast, viruses do not have this restriction, as they have no limit on the systems' resource usage. Advanced techniques of stealthiness and polymorphism, entangle even more the decryption process. Thus, the complexity of the decryption function increases, making anti-viruses unable to afford to analyze the decrypted code. The process is so long and tedious that the analysis aborts before the payload is available.

4. CHAPTER

Core Polymorphic Engine

This chapter presents the core structure of the polymorphic engine named Bagheera[1], a module that can be added to any virus to give it polymorphic qualities. First, the basic structure of the engine is defined, as well as the logical relationship between the different parts of the code it generates. Next, the development environment where the engine has been crafted is described, to finish by detailing the execution phases of the engine.

4.1 Overall Structure

A polymorphic engine is a piece of code that can modify an input I into an altered version of it, namely O_i, with $1 \leq i \leq n$, being n a large integer value. Every conversion the engine performs generates a different output, unrelated to the rest of possible $n-1$ outputs. As a result, there is no feasible situation where for any given pair of outputs (O_i, O_j) with $1 \leq i, j \leq n \rightarrow$ that $O_i = O_j$. Figure 4.1 illustrates the differences between the morphs Bagheera creates.

For every O_i generated, the engine must also provide a method to retrieve the original I, since the aim of the morphing is to evade plain sight detection and the original input will be needed at some point in time. This method is often called a **decryption function** (also referred as $decrypt()$). The decryption function generated for a specific output is tightly related to it, making also $decrypt()$ unique the same way O_i is. Alternatively, the engine

[1]The source code for Bagheera is available at `https://github.com/diegocarba99/bagheera`.

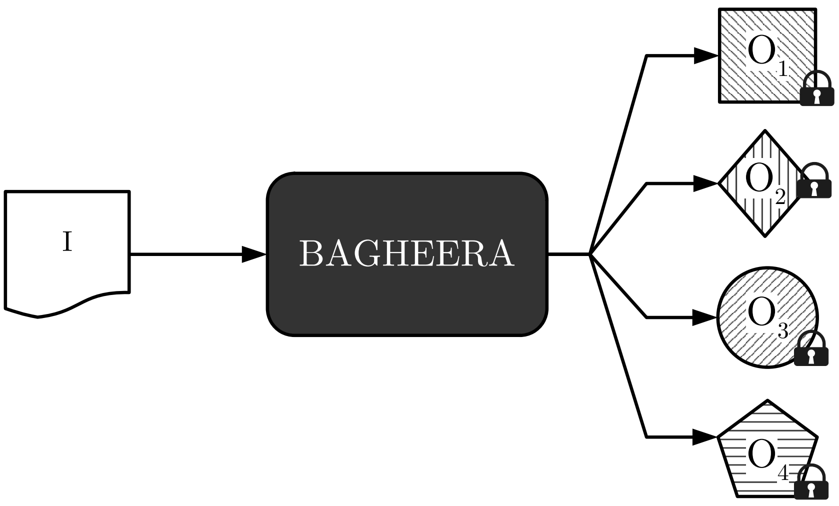


Figure 4.1: Process of morphing an input I into various possible encrypted outputs O_i.

transforms the initial input using an **encryption function** (also refereed as *encrypt*()). This function works inversely to *decrypt*(), as the result generated by the encryption function can be processed by the decryption function to retrieve the original I.

Generally, some basic obfuscation[2] is performed so that the output cannot be related to the input data at a glance. This transformation although effective, does not require any huge amount of computational effort, since it only consists of simple arithmetic or logic operations, described in more detail later on. Obfuscation is carried out by creating a unique encryption **key** (k), which is used as an operand in both the pseudo-encryption and pseudo-decryption process. What gives the engine its polymorphic nature is the fact that, despite the key is different in every morph, the generated functions are also different in every conversion it is performed.

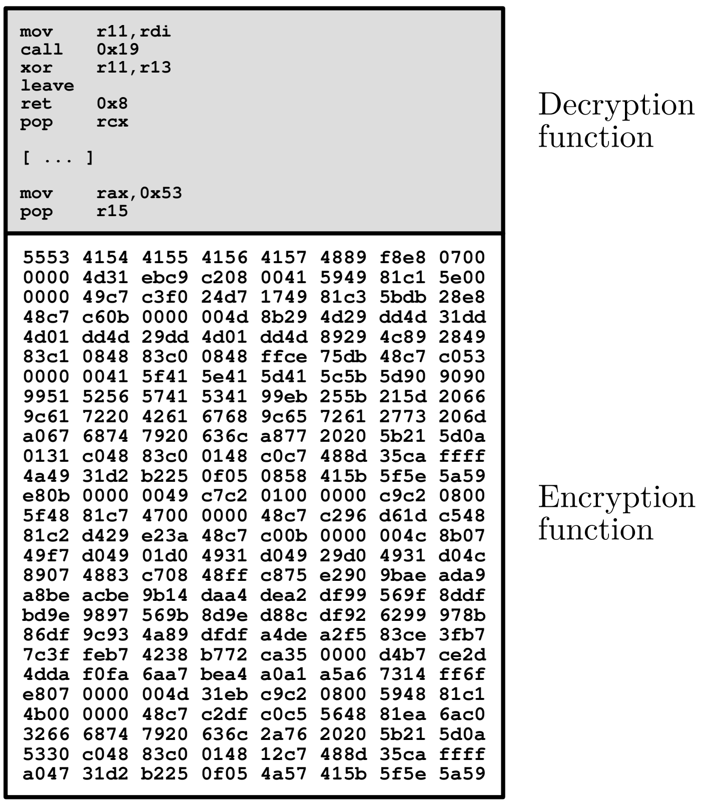
In consequence, it can be stated that:

$$I = decrypt_k(encrypt_k(I)) = decrypt_k(O_i)$$

As mentioned in Chapter 3, the most common input a PE can receive is static binary code (also known as *shellcode*). Since the encryption process aims to obfuscate the meaning of

[2]The deliberate act of creating code or modifying existing one so that is difficult for humans to understand. Code may be obfuscated to conceal its purpose and logic to prevent tampering or deter reverse engineering.

I, the camouflaged data O_i generated by the engine maintains the same semantic meaning as the original input.



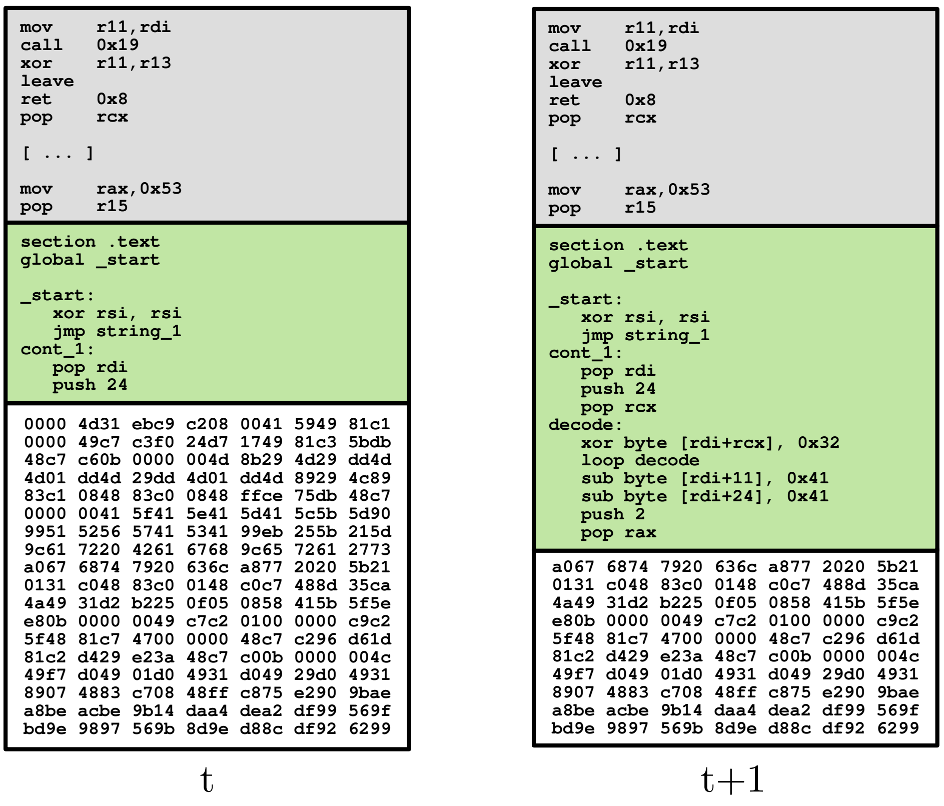
```
mov     r11,rdi
call    0x19
xor     r11,r13
leave
ret     0x8
pop     rcx

[ ... ]

mov     rax,0x53
pop     r15
```

```
5553 4154 4155 4156 4157 4889 f8e8 0700
0000 4d31 ebc9 c208 0041 5949 81c1 5e00
0000 49c7 c3f0 24d7 1749 81c3 5bdb 28e8
48c7 c60b 0000 004d 8b29 4d29 dd4d 31dd
4d01 dd4d 29dd 4d01 dd4d 8929 4c89 2849
83c1 0848 83c0 0848 ffce 75db 48c7 c053
0000 0041 5f41 5e41 5d41 5c5b 5d90 9090
9951 5256 5741 5341 99eb 255b 215d 2066
9c61 7220 4261 6768 9c65 7261 2773 206d
a067 6874 7920 636c a877 2020 5b21 5d0a
0131 c048 83c0 0148 c0c7 488d 35ca ffff
4a49 31d2 b225 0f05 0858 415b 5f5e 5a59
e80b 0000 0049 c7c2 0100 0000 c9c2 0800
5f48 81c7 4700 0000 48c7 c296 d61d c548
81c2 d429 e23a 48c7 c00b 0000 004c 8b07
49f7 d049 01d0 4931 d049 29d0 4931 d04c
8907 4883 c708 48ff c875 e290 9bae ada9
a8be acbe 9b14 daa4 dea2 df99 569f 8ddf
bd9e 9897 569b 8d9e d88c df92 6299 978b
86df 9c93 4a89 dfdf a4de a2f5 83ce 3fb7
7c3f feb7 4238 b772 ca35 0000 d4b7 ce2d
4dda f0fa 6aa7 bea4 a0a1 a5a6 7314 ff6f
e807 0000 004d 31eb c9c2 0800 5948 81c1
4b00 0000 48c7 c2df c0c5 5648 81ea 6ac0
3266 6874 7920 636c 2a76 2020 5b21 5d0a
5330 c048 83c0 0148 12c7 488d 35ca ffff
a047 31d2 b225 0f05 4a57 415b 5f5e 5a59
```

Figure 4.2: Basic structure of the output generated by the polymorphic engine.

As stated before, the generated output data and the decryption function are tightly related, so the engine provides both at the same time. Figure 4.2 illustrates this concept visualizing the outcome created by Bagheera. PEs are used to create polymorphic code, which can be created by supplying the mentioned executable instructions to the engine. Thus, all the generated output can be interpreted as code instructions. The decryption function will consist of benign code that just performs some operations on the data that follows it, i.e. the encrypted payload.

On the other hand, the obfuscated data will presumably look like corrupted or bizarre instructions. As the decryption function is executed, this gibberish will be translated into actual valid instructions, which correspond to the original input data the engine camouflaged. This decryption process is represented in Figure 4.3. The figure shows two snapshots of Bagheera's output in a real decryption process, where the decrypted payload can read the password of the user[3].



```
mov    r11,rdi
call   0x19
xor    r11,r13
leave
ret    0x8
pop    rcx

[ ... ]

mov    rax,0x53
pop    r15
section .text
global _start

_start:
   xor rsi, rsi
   jmp string_1
cont_1:
   pop rdi
   push 24
 0000 4d31 ebc9 c208 0041 5949 81c1
 0000 49c7 c3f0 24d7 1749 81c3 5bdb
 48c7 c60b 0000 004d 8b29 4d29 dd4d
 4d01 dd4d 29dd 4d01 dd4d 8929 4c89
 83c1 0848 83c0 0848 ffce 75db 48c7
 0000 0041 5f41 5e41 5d41 5c5b 5d90
 9951 5256 5741 5341 99eb 255b 215d
 9c61 7220 4261 6768 9c65 7261 2773
 a067 6874 7920 636c a877 2020 5b21
 0131 c048 83c0 0148 c0c7 488d 35ca
 4a49 31d2 b225 0f05 0858 415b 5f5e
 e80b 0000 0049 c7c2 0100 0000 c9c2
 5f48 81c7 4700 0000 48c7 c296 d61d
 81c2 d429 e23a 48c7 c00b 0000 004c
 49f7 d049 01d0 4931 d049 29d0 4931
 8907 4883 c708 48ff c875 e290 9bae
 a8be acbe 9b14 daa4 dea2 df99 569f
 bd9e 9897 569b 8d9e d88c df92 6299
```

```
mov    r11,rdi
call   0x19
xor    r11,r13
leave
ret    0x8
pop    rcx

[ ... ]

mov    rax,0x53
pop    r15
section .text
global _start

_start:
   xor rsi, rsi
   jmp string_1
cont_1:
   pop rdi
   push 24
   pop rcx
decode:
   xor byte [rdi+rcx], 0x32
   loop decode
   sub byte [rdi+11], 0x41
   sub byte [rdi+24], 0x41
   push 2
   pop rax
 a067 6874 7920 636c a877 2020 5b21
 0131 c048 83c0 0148 c0c7 488d 35ca
 4a49 31d2 b225 0f05 0858 415b 5f5e
 e80b 0000 0049 c7c2 0100 0000 c9c2
 5f48 81c7 4700 0000 48c7 c296 d61d
 81c2 d429 e23a 48c7 c00b 0000 004c
 49f7 d049 01d0 4931 d049 29d0 4931
 8907 4883 c708 48ff c875 e290 9bae
 a8be acbe 9b14 daa4 dea2 df99 569f
 bd9e 9897 569b 8d9e d88c df92 6299
```

Figure 4.3: Decryption process performed of a real output of the engine, from time t to time $t+1$. Decryption function is shown in grey, the decrypted payload in green and the encrypted payload in white.

4.2 Development Environment

Bagheera has been developed using the `C++` programming language. Polymorphic engines can be created using programming languages with less abstraction level such as `C` or `Assembly`.

The main reason for choosing `C++` has been dynamic code generation. The engine must generate a unique decryption function every time is executed, thus it must create executable instructions on run-time. This feature is present in Just-In-Time compilation. Just-In-Time or JIT compilation is a technique used to generate code and execute it during the execution of a program rather than before execution [17].

[3]The whole payload is available at http://shell-storm.org/shellcode/files/shellcode-891.php

This project has made use of the `AsmJit` library [18] available for `C++` to generate assembly code at run-time. The engine creates the executable code and the library stores it in one of the `CodeHolder` class' attributes. When the executable code is ready to be assembled, a specialized JIT run-time resolves all the jumps and offsets, adding the generated code to a function pointer. This process is detailed in Section 4.3.4.

4.3 Engine Phases

The engine must complete certain execution phases to generate the output. These phases are executed by the engine in order of appearance. In the following sections, these phases are described.

4.3.1 Create encryption function and key

In this phase, the PE must obfuscate the input data that has received. Since the encrypted data must later be retrieved using the decryption function, this phase is highly related to the one described in Section 4.3.3.

The encryption key used for both functions is also created in this phase. As it will be described in Chapter 5.1, the key will be concealed using advanced methods, but the core functionality remains the same: store a certain value (the key itself) in a register for later usage in the arithmetic and logical operations performed on both encryption and decryption functions.

All the operations that are performed on the input data are cached so that they can be reversely applied by the decryption function. For this purpose, Bagheera creates an array that will contain the opcodes of the operations that will be applied to the data.

All the operations must be reversible so that when they are reverse-applied, no information is lost. From the available set of operations present on most modern CPU's Arithmetic Logic Unit (ALU), only a small set of them can be used for this purpose since not all of them are reversible. Among this operation set, the following ones can be found: Addition (`ADD`), Subtraction (`SUB`), Negation (`NEG`), Bit-wise And (`AND`), Bit-wise Or (`OR`), Bit-wise exclusive Or (`XOR`), Bit-Wise negation (`NOT`) and rotation (`ROT`) [19].

Although many other operations might be supported by specific architectures, only those operations stated in this chapter will be considered for the project. A possible series of

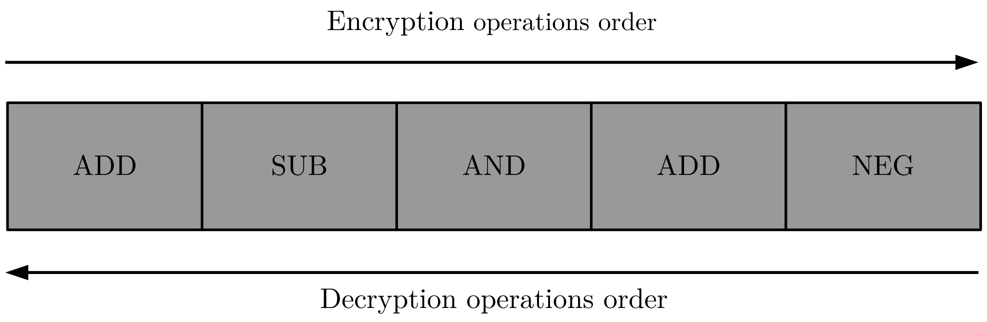


Figure 4.4: Representation of the array of opcodes used for the encryption and decryption functions.

opcodes stored in the array the engine creates is illustrated in Figure 4.4, where the order in which both encryption and decryption functions apply those operations to the data is represented.

Basic PEs, encrypt the data using just one operation, making the obfuscation process quite simple. On the contrary, Bagheera uses multiple operations. This way, the level of camouflage is higher, making the analysis of the generated data even more complex.

Algorithm 1 presents the basic structure of the encryption function the engine generates. Even though the encryption function will consist of different operations and key, the semantics of the initial input will not be altered. When decrypted, all the different morphs that correspond to a certain input will be equal.

Algorithm 1: Encryption function algorithm.

```
Input: Buffer containing the input data and encryption key
Output: Buffer containing the encrypted data
1 foreach Data Block do
2     foreach Encryption Operation do
3         OutputBuffer = InputBuffer opcode EncryptionKey
4 return OutputBuffer;
```

4.3.2 Encrypt the input data.

Once the encryption function has been constructed, the input data provided to the engine can be morphed. This process is performed onto blocks of data taken from the input. The size of these blocks will vary depending on the machine the engine is working on. For 32-bit platforms, the maximum size the engine manages is 4 bytes, and 8 bytes for 64-bit platforms.

The engine will iterate through the input, block by block, applying the operations defined in the opcode array to the data. This modified data will be stored in a temporary buffer until it can be appended to the output.

Since the engine can only process blocks of data, the size of the buffer provided to the encryption function must be aligned with the block size. Bagheera does not impose any restriction on the input size, so some sort of padding is added to the input aligning it to the block size. This padding will fill the remaining space between the input data and the next congruent block size as illustrated in Figure 4.5. As it has been mentioned before, the engine will process static binary code, so the padding consists of `NOP` operations. These instructions are ignored by processors, leaving the initial data's semantics unchanged.

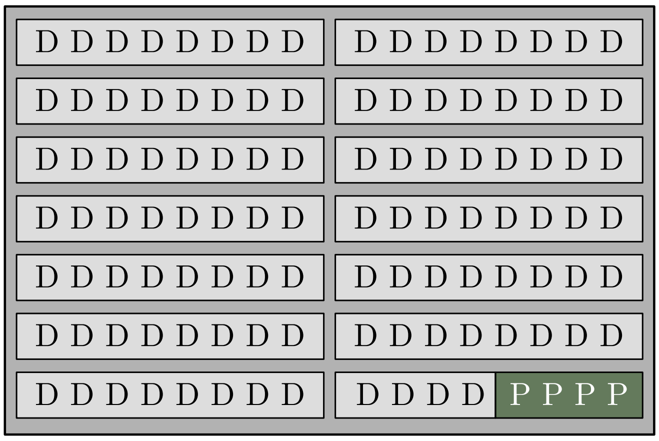


Figure 4.5: Padding applied to an unaligned input data. Data is represented by '*D*' and padding by '*P*'.

4.3.3 Create decryption function.

In this phase, the engine is responsible for creating the actual code that will retrieve the initial input *I* from the obfuscated O_i. To achieve this, the function is divided into different segments. All the code and instructions mentioned in these segments are generated using the `Assembler` class from the `Asmjit` library. This class offers an interface to generate the desired instructions, storing them in the previously mentioned attribute in the `CodeHolder` class. Through both of the classes, the library is in charge of storing all the instructions that are emitted to it, so that later on the actual binary code can be generated.

The different segments that make up the decryption function are described next. These segments appear in the document in the order in which they will be executed in the decryption function.

Function Prologue

The engine can store all the decrypted data into a buffer apart from executing it. This section is responsible for obtaining the buffer's address, which should be large enough to store all the output.

Although the pointer to the buffer is handed to the function as an argument, the way the engine gets access to it differs depending on the platform it's been executed. More precisely, this access depends on the **calling convention** the machine uses. Historically, the way functions and function-callers have interacted has been defined by the platform the software was developed. As multi-purpose machines arose, these diverse specifications merged into various calling conventions, defined mainly by the Operating System the software is run [20, 21].

Calling conventions describe the interface of called code, stating the order of the parameters, how these are passed, which registers' value must be preserved for the caller and how the call is made. Since this area vastly wide, this document will not go into further detail about the peculiarities of each aspect mentioned. However, considering Bagheera has been developed targeting Linux machines, it will follow the calling convention used in this kernel, which is *System V AMD64 ABI*.

Following the System V ABI [22] calling convention, we retrieve the pointer to the output buffer via the `RDI` register. Since the engine just receives a single parameter, this register is the first in the list of 6 used for passing arguments before the stack is used. All the clean-up work is done by the caller of the engine, so Bagheera has no need for preserving the values of the sensitive registers before the execution of the function, as this task is performed by the caller.

Finally, the decryption function must save the values of the registers `RBX`, `RSP`, `RBP`, and

`R12-R15` so that it can restore their original values when the execution ends. This step is performed by pushing onto the stack the values of these registers. Later on, in the function epilogue described in Section 4.3.3, these values will be restored to their original registers.

This last step is crucial for the decryption function to fulfil its purpose. Throughout all the function, various registers serve different purposes. All these different register *types* will change from generation to generation, so all the sensible data described by the *ABI* must be preserved. The different type of registers used throughout the function are listed below:

- `regSrc`: This register will hold a pointer to the memory location where the encrypted data is stored. This location will reside somewhere after the decryption function body.
- `regDst`: This register holds the pointer to the output buffer provided as an argument to the function.
- `regSize`: This register contains the number of blocks that make up the encrypted data and the generated padding attached to it.
- `regKey`: This register holds the value of the decryption key. This value may be calculated onto this register in various ways, but the final effective value used in the main decryption loop is stored in this register.
- `regData`: This register will hold the block of data the function is treating in every iteration of the decryption loop. Data will be temporally modified onto this register and then saved to the corresponding locations.

Any of the available registers in the machine could be used for any of the above-listed purposes. Thus, all sensitive data is saved so no information is lost, and then the registers used by the function are randomly selected. This randomness contributes to the polymorphic nature of the function since most probably two different generations of the engine will not share all the registers laid out the same way.

Listing 1 shows a possible function prologue generated by the function.

Delta offset calculation

The dynamically generated code can be located anywhere in memory and launched from there. In such cases, we cannot refer to parts of the function using absolute memory ad-

```
PUSH RBX   ; \
PUSH RSP   ;  \
PUSH RBP   ;   \
PUSH R12   ;    > Save the sensible registers
PUSH R13   ;   /
PUSH R14   ;  /
PUSH R15   ; /

MOV regDts, RDI  ; Get the function parameter
```

Listing 1: Decryption function prologue.

dresses, because we simply do not know where the function will reside. This address location is stored onto the `regSrc` register, using the following calculation:

$$regSrc = @base\_address + offset\_to\_encrypted\_data$$

Where the `@base_address` is unknown until run-time. The offset to the encrypted data varies depending on the changing size of the decryption function, so its value remains unknown until all the function has been emitted. To overcome this restriction, an age-long technique is used to reference the data using relative addressing, namely the **Delta Offset** technique.

The base address where the decryption routine is located is calculated taking advantage of the behaviour of the `call` instruction. When a function is called using this instruction, the return address is stored onto the stack before the actual jump is done. Precisely, this stored value is the base address the engine is looking for. To obtain it, Bagheera simply emits a `call` instruction that refers to an instruction after it. If the following instruction retrieves the pushed value from the stack, the base address is obtained. Using a simple `pop` instruction is enough to obtain this value.

The delta offset technique is both simple and well known, so it is not stealthy at all. Regular programs do not need to perform this kind of calculations, so a sequence of `call/pop` instructions is suspicious. To avoid triggering all the AV alarms, Bagheera simulates a legit function call. Normally, after a function returns from a call, the caller performs some operations to obtain some information from the called function. Thus, between the `call` and the `pop` function, some non-suspicious code is inserted, emulating a normal program behaviour.

On the other hand, when the instructions to calculate `regSrc`'s value are generated, the final size of the decryption function is unknown and so the offset to the encrypted data is also. Thus, the *offset_to_encrypted_data* value is not emitted yet, but a dummy-value is. Later on, when all the code corresponding to the decryption function has been emitted, this dummy value is updated with the actual offset. In Listing 2 the code generated for the Delta Offset calculation is shown.

```
    CALL delta_offset   ; call to the 'delta_offset' label

    MOV R10, 0x3        ;\  unused instructions
    LEAVE               ; > used to emulate
    RET 0x8             ;/  benign behaviour

delta_offset:
    POP regSrc          ; store base address onto regSrc
    ADD regSrc, 0x000   ; add dummy value, updated later
```

Listing 2: Delta offset calculation.

Decryption key setup

As mentioned in Section 4.3.1, the key is stored the `regKey` register. This process is quite straightforward at its core, as the only action that needs to be carried out is storing a value on a register. However, this is not the way Bagheera manages the key setup, as it consists of an advanced technique that will be described in Section 5.1. The details regarding this segment will be extended in the mentioned section.

Decryption loop

This section is the core of the decryption routine. Here, the function processes the data and restores the original input. The section consists of a main loop that iterates through the encrypted data referred by `regSrc`, and stores it, decrypted, back into `regSrc` and `regDst`.

Before the loop starts, the register `regsSize` is initialized with the number of blocks the loop has to go through. Next, the block pointed by `regSrc` is stored onto `regData`.

Actually, this is where the loop starts. The following instructions operate the value in `regData` with the encryption key located in `regKey`. The number of instructions used for this purpose depends on how the data was encrypted. Precisely, the number of instructions in this part of the program equals the number of entries in the array of opcodes presented in Section 4.3.1. The operations performed onto `regData` use the decryption key as an operand, although not all of them do. Operations like `NOT` or `NEG` do not make use of the key.

After the operations have been performed onto the value stored in the data register, the decrypted data resides in it. The engine saves this value by moving it to the locations pointed by the `regSrc` and `regDst` registers. Finally, these two pointers are updated to point onto the next available block location, and the `regSize` register is decremented as a loop index. The last instruction of this section is a conditional jump, which depending on the value of `regSize` jumps to the beginning of the loop, or continues with the rest of the decryption routine.

An example of the main decryption loop is available on Listing 3. At lines 14 and 15, the data pointers are increment by `0x8`, since this particular example is done in a 64-bit machine where the biggest block size managed by Bagheera in this architecture is 8 bytes.

Function Epilogue

Finally, the engine emits the final code relative to the decryption engine. Basically, the return value is set onto the RAX register, and the sensible registers' values defined by the *System V ABI* are restored.

When this point is reached, the decryption routines final size is known, so the dummy-value described in Section 4.3.3 is updated. The final value is the offset difference between the instruction with dummy value and the beginning of the appended encrypted data.

4.3.4 Generate final output

Right after the main decryption routines body is where the encrypted data resides. This section is constructed embedding the bytes directly using the `DB` and `DQ` instructions, for 32-bit and 64-bit platforms respectively.

When all the instructions have been emitted to the `Asmjit` library, it is time for the engine

```
    MOV regSize, numBlq    ; get encrypted block number
                           ; numBlq = sizeof_data / blockSize

decryption_loop:
    MOV regData, PTR[regSrc]  ; Get current block

    NOT regData               ;\
    XOR regData, regKey       ; | variable number of
    ADD regData, regKey       ; | decryption operations
    SUB regData, regKey       ;/

    MOV PTR[regSrc], regData  ; Save decryption data
    MOV PTR[regDst], regData  ; to corresponding locations

    ADD regSrc, 0x8           ; update pointer to point
    ADD regDst, 0x8           ; at next block

    DEC regSize               ; decrement encrypted block number
    JNE decryption_loop       ; loop jump
```

Listing 3: Possible generation of decryption loop.

to create the final output. Listing 4 shows the operations the engine performs to fulfill this purpose.

As it has been described throughout this chapter, the decryption function takes a buffer as an input so that the original data can be stored, and it returns the size of the data written to the buffer. Thus, a function pointer is created on line 5, which follows the mentioned definition. First, memory with execution permissions is reserved using the `mmap` function on line 9. Then, the output generated by the `AsmJit` run-time on line 11 is copied onto the executable memory using `memcpy` as shown on line 15. Finally, in line 17 the function pointer is initialized with the executable output.

Finally, the engine can be called using the function pointer defined in Listing 4. When that function is invoked, the decryption function generated by the engine is executed, which decodes the "payload" that it has attached to itself. Finally, you give control to the payload, letting it run. An example of how the decryption function is called is available in the Listing 5.

```
// outputsz - size of the output generated code
// code - CodeHolder class
// output_buffer - buffer to hold the output of the engine

typedef long(*DecryptionFn)(void *);

DecryptionFn non_executable_fn;

void *lib_output = mmap(0, outputsz,
                   PROT_READ | PROT_WRITE ,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

run_time.add(&non_executable_fn, &code);

memcpy(lib_output, (void *)non_executable_func, outputsz);

DecryptionFunc fn = reinterpret_cast<DecryptionFunc>(lib_output);
```

Listing 4: Creation of the decryption function.

```
unsigned long buffer_size = fn(output_buffer);
```

Listing 5: Execution of the decryption function.

5. CHAPTER

Advanced Techniques

To complicate the anti-viruses detection process, this chapter describes some advanced functionalities that have been implemented in *Bagheera*. Thanks to these, the engine can avoid detection with a much higher probability. First, how the encryption key is further obfuscated is described, as well as how the engine generates dead code to increase the load in the system. Finally, the custom pseudo-random number generator used by Bagheera is presented.

5.1 Encryption key concealing

The "**sliding key encryption**" technique used in the metamorphic virus `MetaPHOR` developed by *The Mental Driller* [15], is also implemented in Bagheera. Since the encryption key remains unchanged throughout the encryption process, the data is encrypted uniquely. The encryption key is slightly modified as the decryption progresses, rotating it via a `ROR` instruction. Applying this permutation makes the encrypted data change even more every time is encrypted, making the generated morph even more distinct from the others [23].

Additionally, the actual value of the key is not stored directly into the `regKey` register. Instead, a **key modifier** is used to avoid having the literal value of the key in the code. First, the $k - key\_modifier$ value is written onto the `regKey` register. Next, the modifier is added to the register, leaving the encryption key in the register. All in all, the final value stored in the register is the following:

$$regKey = (k - modifier) + modifier = k + (modifier - modifier) = k + 0 = k$$

After all, the key will be stored in the register. However, the actual value of the key will not appear on the code. This makes reverse-engineering the engine even harder and complicates heuristic analysis from an AV looking for the encryption key.

5.2 Dead code

If the code generated by Bagheera was disassembled, a basic skeleton of a decryption routine would be visible. To avoid having a predictable structure, the engine inserts dead code between each instruction of the generated code. This dead code uses all the registers that the engine does not use so that no relevant information is destroyed.

Additionally, these dead instructions can take up to 45 different forms, that combined with the 9 registers the engine lefts unused, makes up to 405 different instructions. The engine will insert a dead code instruction between useful instructions with a 0.8 probability, making this polymorphic process even more unpredictable.

5.3 Pseudo-Random Number Generator

One of the key components to obtaining solid and diverse morphs out of the engine is randomness. The obvious approach when implementing random events in computing is to use a pseudo-random number generator (`PRNG`) [24]. Most modern operating systems provide an interface to a `PRNG` [25], but the overuse of it can trigger the heuristic alarms of anti-virus software. Bagheera avoids this over-use by implementing its own `PRNG`. This generator is taken from `MethaPHOR`, a Mutation Engine[1] developed by the Mental Driller in 2002 [26, 27]. This `PRNG` uses `seed1` and `seed2`, initialized with the UNIX date and the code's first bytes respectively. On the contrary, Bagheera initializes the `seed2` with the user id that is executing the engine. The `PRNG` follows the next algorithm described in Listing 6.

[1]Mutation Engines are an evolution of Polymorphic Engines, which apart from encrypting the payload, re-write all their code, even the engine itself, changing the semantics of the code.

```c
int prng(){
        seed1 ^= (seed2 + ror_13(seed1 + seed2));
        seed2 = (seed1 + ror_17(seed2)) ^ (seed1 + seed2);
        return seed1 + ror_17(seed1 ^ seed2);
}
```

Listing 6: Pseudo-Random Number Generator C algorithm.

The `ror_XX` operation denotes a circular right rotation by XX bytes. This PRNG is believed to have a period[2] of 40.000, which is quite as much as `glibc`'s generator. The second seed, in both cases, has low randomness, and thus, its period might not be quite large. However, the number of calls to the function is not that high to perceive a cycle in the number generation.

The PRNG is used in various steps of the engine, like the selection of the registers, dead code insertion, key generation, encryption operation number selection and general equivalent instruction selection.

5.4 General Overview

The advanced techniques described in this chapter add new segments to the output, changing the execution flow of the decryption function described in Chapter 4. Moreover, the *Create encryption function and key* engine phase detailed in Section 4.3.1 is also modified using some of these techniques.

Figure 5.1 shows the relation between the segments that make up the basis of the function, the new segments added for the advanced techniques and the phases that exert these very techniques.

Between every segment present in the core structure of the function, a *Dead Code* segment is inserted. This segment generates code unrelated to the rest of the decryption function, or it may not generate any code at all, so the custom PRNG is used to randomize it.

Additionally, both the *Decryption key setup* segment and *Create encryption function and*

[2] A PRNG's period is the number of steps at which the program starts to repeat itself and generates the same numbers it has done before.

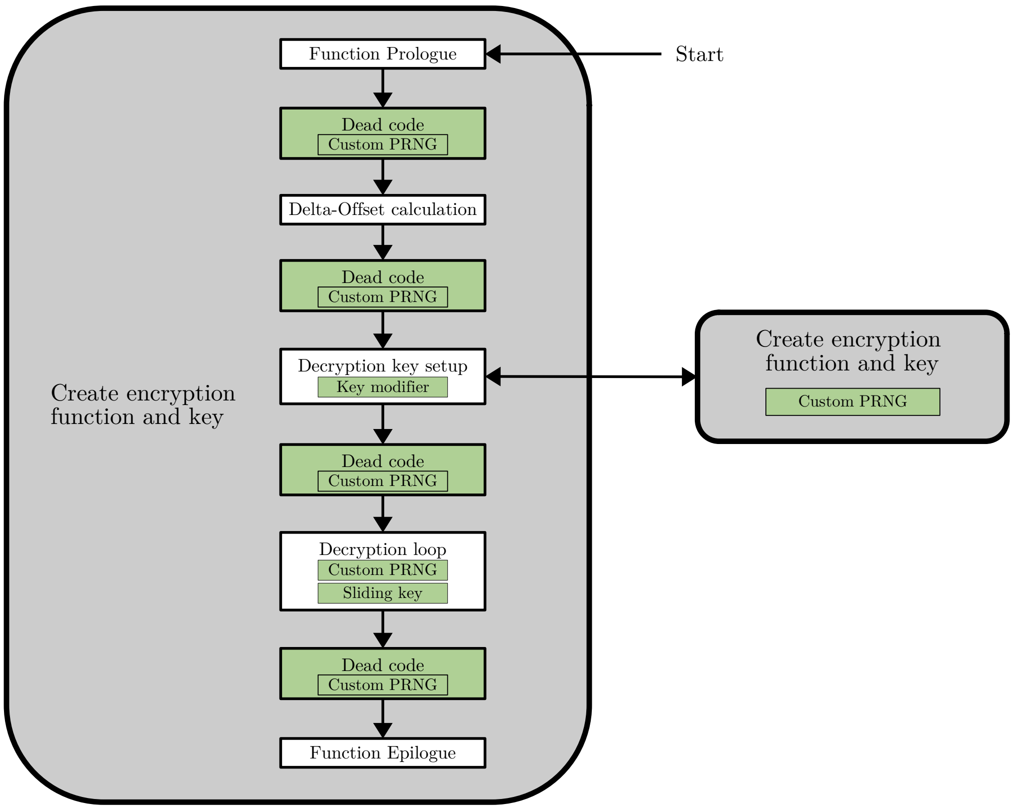


Figure 5.1: Execution flow of the decryption function with the advanced techniques modifications. Green segments correspond to those added by the advanced techniques and, white ones conform the segments that make up the core structure of the decryption function.

key engine phase use the custom PRNG, to generate the key and create the encryption/decryption opcodes respectively.

Furthermore, the *Key modifier* technique is applied in the segment the *Decryption key setup* segment, concealing the actual value of the key from the raw code. Finally, the *Decryption loop* segment is enhanced thanks to the *Sliding key* advanced technique. The remaining segments are left unaltered.

6. CHAPTER

Binary Infection

The main use of a Polymorphic Engine is as an addition to computer viruses, to avoid being detected by anti-virus software. A computer virus is a piece of software that can infect other programs in the same machine or can traverse a network and infect other binaries in different machines. Viruses modify the host's behaviour so that these infected programs contain a copy of themselves [4]. Whenever an infected program executes, also does the virus, possibly infecting more programs in the machine is run at.

In a Linux environment, de facto binary programs follow the ELF structure. ELF stands for **Executable and Linkable Format**, used by the Linux kernel to operate objects from a Linking or Executable perspective [28]. The ELF structure is the roadmap used in Linux to understand binary files. It details how the binary should be organized and describes the semantics of the different attributes the structure has. Libraries and external linked programs use the *linking* view of the ELF file, whereas programs that are run use the *execution* view. ELF files support both views, thus following the same specification described in [29]. A representation of both perspectives is shown in Figure 6.1. Although various CPU models have their ELF specifications, this project will consider the generic specification present in [30].

Any ELF file consists of an ELF header, followed by the file's actual data. The data may include a *Section Header Table* detailing zero or more sections, a *Program Header Table* detailing zero or more memory segments, and the actual data referred by both header tables. Sections hold information for linking and reallocation, whereas segments define the regions of the executable that need to be loaded to create the process image associated

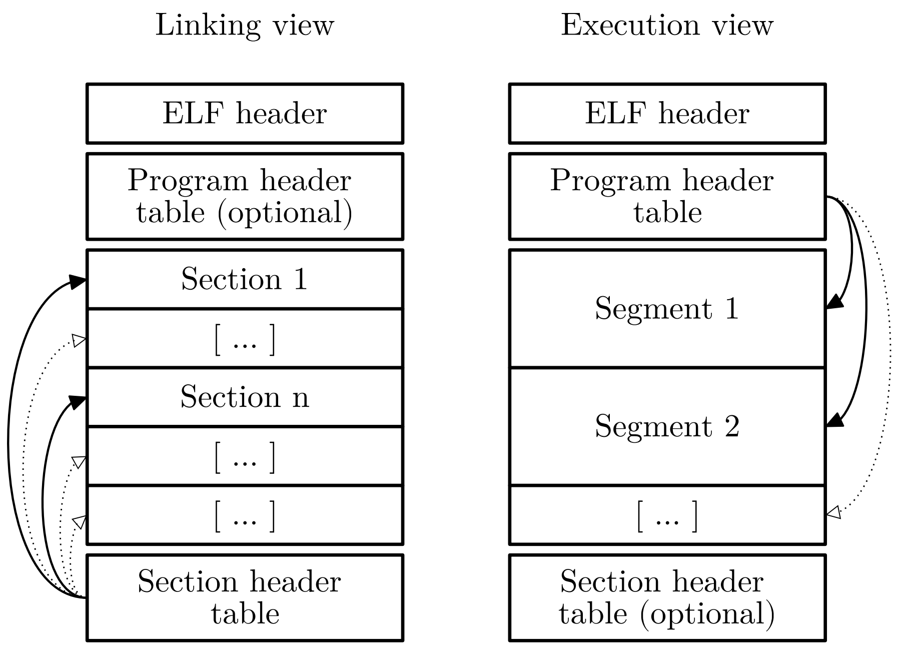


Figure 6.1: ELF file format perspective types.

with the file. A segment may contain multiple Sections.

The ELF header holds information about the binary, i.e. the file type, the required architecture and the entry point of the generated process. This information is stored in the `ElfXX_Ehdr`[1] members, defined in `/usr/include/elf.h`. This header also specifies the location, size and number of entries of the header tables.

From the Linking point of view, the Section Header Table lets one locate all the file's sections. This table is an array of `ElfXX_Shdr` structures, describing the mentioned sections.

When viewed from the Execution view, the Program Header Table describes an array of segment entries described by the `ElfXX_Phdr` structure. The most significant member in each entry, for this project's scope, is the `p_type` member, which tells what kind of segment the array element describes. Some entries describe process segments like `PT_LOAD`, whilst others give supplementary information and do not contribute to the process image as `PT_NOTE` does.

Various ELF infection methods have been developed through the years, most of them shared by virus writers in the **V**irus e**X**change (**VX**) scene. The techniques used in these infection methods are quite diverse, from section corrupting [31], to shared library call redirections [32], and many other advanced techniques [33, 34].

[1]XX can take 32 or 64 values, for 32-bit or 64-bit architectures respectively. It will be referred to this way throughout the document.

Bagheera is capable of infecting an ELF file, hijacking the execution flow by changing the entry point, and inserting its code into some of the supplementary segments using the **PT_NOTE infection** method. This infection method will be explained in the following Section.

6.1 PT_NOTE to PT_LOAD infection

This infection method is based on converting a PT_NOTE segment into a PT_LOAD segment to inject the viral code into a binary. Most ELF Executable files (ElfXX_Ehdr.e_type = ET_EXEC) contain one or more PT_NOTE segments, whose mere function is to store auxiliary information for the binary. This information is expendable, and any binary can execute properly without it. The infection method re-writes the PT_NOTE entry in the Program Header table for a PT_LOAD.

Loadable segments are specified by PT_LOAD elements, whose content will be loaded into memory and thus, available for execution. With the proper changes to the rest of the members of the entry in the Program Header Table, this section can hold viral code without corrupting the binary it resides in.

After the infection, the viral code is appended to the end of the file, and the PT_NOTE segment is pointed to it, hence loading the viral code into memory. To maintain coherence between the Linking and Execution view, so that AV software does not spot the infection with ease, one of the sections inside the PT_NOTE segments is also modified so that it resembles a valid code section.

It must be noted that the developed infection algorithm works only with no-Position Independent Executable (PIE) binaries, such as ET_EXEC type ELF files. PIE binaries are loaded into random memory locations [35]. This random memory addressing makes the calculation of the offsets and jumps that give the ELF back the execution flow an arduous job. However, it is not impossible to infect this kind of binaries. In fact, RIP calculation algorithms offer a workaround to calculating at run-time the random memory where the binary has been loaded [36]. However, this technique has not been implemented in Bagheera, thus it is not capable of infecting this kind of programs.

The custom algorithm Bagheera uses for the PT_NOTE to PT_LOAD infection method is based on the one described in [37]. The algorithm is modified to add the section infection, which was not contemplated in the original algorithm. The algorithm is the following:

1. Open the ELF file to be injected. If the file is not an ELF file, exit the infection process.

2. Check if the file is Executable type. exit infection process if ELF file type does not match.

3. Save the original entry point, e_entry.

4. Parse the program header table, looking for a PT_NOTE segment.

   (a) Convert the PT_NOTE segment to a PT_LOAD segment.

   (b) Change the memory protections for this segment to allow executable instructions.

   (c) Point the offset of this segment to the end of the original binary, where the viral code will be stored.

   (d) Adjust the size on disk and virtual memory size of this section to account for the size of the injected code.

   (e) Modify this segment's alignment to match PT_NOTE alignment.

5. Parse the section header table, looking for a SHT_NOTE section.

   (a) Convert the SHT_NOTE section to a SHT_PROGBITS section.

   (b) Change the memory protections for this section to allow executable and writable instructions.

   (c) Point the offset of the section to the end of the original binary, where the viral code will be stored.

   (d) Adjust the size on disk and virtual memory size of this section to account for the size of the injected code.

6. Change the entry point to the end of the file using the modified segment's virtual memory offset.

7. Write the modified elements back to disk, over the original file.

8. Append viral code to the end of the file.

9. Patch the end of the file with instructions to jump to the previously saved original entry point.

In Figure 6.2 how the topology of the infected ELF file changes after the algorithm is run is shown. The new entry point transfers control to the payload at the beginning of the execution flow. After the viral code finishes, the patch at the end of the file gives the control back to the ELF file, leaving the program's behaviour unaltered. The conversion from PT_NOTE to PT_LOAD means the payload is loaded into memory, the same way legit code segments are.

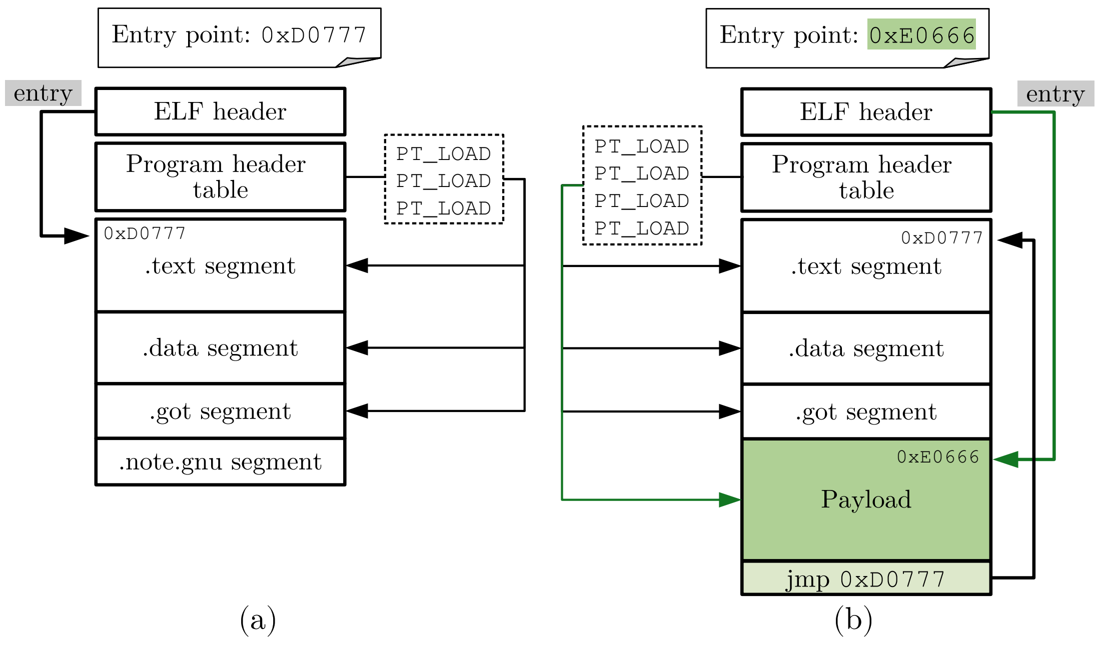


Figure 6.2: ELF file infection using PT_NOTE technique. (*a*) is an un-infected file and (*b*) is an infected file.

This infection method is quite straightforward since it does not require any advanced parsing of the ELF file, like for example, the .text section infection does [34]. On the other hand, since the implementation is so elementary, a sharp eye could detect the infection with ease in case the binary file is examined thoroughly. This trade-off between ease of infection and detection must be taken into account. However, the infection is so simple and effective that it outnumbers the possible drawbacks it can have. If a virus is capable of infecting a huge amount of programs, detection may not be effective at all.

If a regular virus spreads rapidly inside a system, but the anti-virus can detect it after various infections, it will presumably take the appropriate actions to neutralize it, like disinfecting the corrupted files. This neutralization of the viral threat is possible since the AV can look for the virus code in the file system.

However, if a polymorphic virus also spreads at a swift speed in a certain system, a single detection of one of the multiple infected files does not endanger the rest of the infections.

As the code of each of the infected files is different, the anti-virus cannot find the code of the detected file. This way, detection is evaded thanks to polymorphic techniques.

7. CHAPTER

Detection Avoidance

One of the main motivations for developing a Polymorphic Engine is the capacity to evade being detected by anti-virus software. To test if Bagheera fulfils its purpose, some of its outputs have been put to the test against an AV, the open-source ClamAV to be exact.

Since the PE can generate standalone encrypted executable binary code and infect existing binary programs, both methods are tested against the AV. Using signature-based scanning, whether ClamAV can detect a virus payload is tested. Also, in the hypothetical case where the engine's output is marked as a virus, the various polymorphic generations are compared, to test if the function changes enough to not be detected.

To detect the presence of viral code inside certain files, first, a signature of it must be obtained. Two different signatures will be provided to the AV. The first will consist of a raw byte-code sequence corresponding to the engine's input I, referred from now on as "*sig-evil-payload*". In fact, this input I is a simple assembly program named "*evil-payload*" that prints a message onto `stdout` using the `write` system call. The second signature corresponds to the byte-code sequence of the decryption function of a O_i output, referred as "*sig-decrypt-fn*".

When AV software conducts a static scan, a virus database is compared against the contents of the scanned files. If there is a match between a database entry and a portion of a file, this file is tagged as a threat and pertinent risk contention actions are performed.

Using the signatures mentioned before, the following test cases have been conducted:

- **Input obscuring**: Detect an input I using it's signature *sig-evil-payload* against a regular file containing the output O_i.

- **Infection concealing**: Detect an input I using it's signature *sig-evil-payload* against an ELF file infected with the output O_i.
- **Morph detection**: Detect the presence of a Polymorphic decryption function O_i with signature *sig-decrypt-fn* against m files containing each of them a different output O_j, with $0 \leq j \leq m$.

In all of the conducted test cases, the anti-virus failed to detect the input I. None of the scans were able to detect I or its morphs. This evasion is possible since the byte-code sequence used to construct the signature does not appear in the scanned files.

In a real-world scenario, the signature of a detected virus would be added to a virus database. If an input I or one of its morphs O_i would be added to this database, it is highly unlikely that an AV would detect another morph of I contained in O_j. Due to the high polymorphism degree Bagheera produces, the morphs are so distinct and the encryption is so efficient that there would not be any match.

8. CHAPTER

Project management

Planning is a key component on large scale projects of this sort. Organizing all the different stages of the project and taking into account possible drawbacks is vital. Only when this is done, potential risks as well as different development phases can be detected and handled ahead of time, in order to manage the most valuable resource in this kind of projects, that is, time. The aim of this chapter is to accurately detect all the different aspects of the project that must be finished within a deadline so that tasks can be efficiently distributed along the project lifetime. With that, potential setbacks and delays can be prevented and the project can be successfully completed in time.

The management work will be divided into three main phases, namely: project management, project development and documentation of the project. These phases cover the major aspects and the crucial decisions, as well as how difficulties have been confronted through the entire process. To help outreach the main goal of each one of them, phases have been divided into tasks. These tasks constitute the focus points of each phase that must be accomplished for a correct achievement of the projects main objectives and landmarks.

8.1 Description of the phases and their features

The following sections show how the project has developed over time, detailing how each of the phases has evolved during the lifetime of the thesis. Each of the sections details all the work that has been done for the attainment of the main goals, as well as a sharp-eyed observation of the entire process as a whole.

8.1.1 Management phase

The main goal of the management phase is to correctly estimate all the tasks that must be accomplished to reach the objectives of the project, as well as estimate the cost of each of the tasks in order to efficiently foresee its evolution. Major issues and delays have also been taken into consideration. These deviations can transform the projects main course, so special attention must be placed on these. This estimation phase is vital for the correct development of the project since the main end goals must be accomplished on time. Also, being able to track all the major or minor deviation throughout the projects lifetime is useful for estimating its productivity. During the project and after it is completed, these prior estimations can be used for a comprehensive view of the project's efficiency.

After choosing the specific tasks that have to be carried out, the most volatile part consists on placing them in the correct moment and for the right amount of time. Additionally, the project has to be tracked periodically to ensure that the milestones set for the project are being met or if there is need for some readjustments that allow to better handle tasks for the best possible outcome.

When each of the sub-tasks has been defined, the most essential part of this phase is to accurately arrange the tasks in order and to estimate their possible duration. Moreover, the project must be tracked periodically to ensure that its final goals are reached. Besides, this phase is useful for obtaining the best possible outcome from the tasks. This is done by revising the tasks if some readjustments are needed. The elasticity of the planning is vital for a project of this sort so that unexpected deviations do not ward it off from its main goal

There are three main modules that define the management of the project to take into account:

- **Planning**: The main points of the project have been covered estimating their possible duration and the resources needed for their realization, as well as searching for when these objectives can be fulfilled and the order in which they are completed. The result is a set of activities ordered and placed across the time with their respective milestones and with a risk management plan to be ready for risks with prior knowledge on how to avoid them. Finally, the scope of the project is determined.

- **Tracking**: In order to check whether the objectives are being completed under the given time and to counter the present risks that can cause unexpected delays, the project is analyzed during its progression. Finding new risks, modifying the milestones and creating new objectives or replacing older ones is the main purpose.

- **Communication**: This part joins the previous two modules and an assessment is conducted to evaluate how is the project progressing and to identify which tasks are going according to plan and which others are not. All of this is explained in detail to the directors of the project so that they can be up to date and informed of any kind of alterations in the plans. These issues are handled in periodical meetings when certain milestones have been finished or when a new risk has emerged. In the end, the tasks until the next meeting are decided, which are usually intentions for the short term.

8.1.2 Development phase

The project is heavily oriented towards learning the insights of polymorphic engines and how can a viral program be developed in a Linux environment. Most of the development phase is centred on learning and understanding how these engines work and how they fulfil their purpose, the additions that can be done to make them more robust against detection and how a develop a viral behaving program. Additionally, advanced knowledge in assembly language must be obtained, so that the result obtained in the project is of quality and precise.

8.1.3 Documentation phase

The last part involves the report of the project, where the most relevant knowledge and information is gathered about the research conducted, both for the theoretical part and for the practical application. The theoretical aspect is very relevant, but there is not much work published around this topic, and most of the information comes from particular's publications in blogs and e-zines[1].

[1]Virtual magazines about viruses and viral techniques written by underground virus-writing groups, popular in the VX scene.

8.2 Estimations

After covering the three phases that compose the project, the initial estimation of time and the finally needed amount of time to complete the tasks will be displayed. In bold, the estimation for a phase is registered, while below, for each phase, each specific task of the project is broken down belonging to that phase. The estimated time and the final time of each task are summed for finding out how much time each phase has required in Table 8.1.

	Estimated time	**Final time**
Management phase	**65**	**55**
Planning	30	20
Tracking	20	18
Communication	15	17
Development phase	**105**	**101**
Creation/use of a polymorphic engine	40	50
Addition of advanced techniques to the PE	20	13
Binary infection development in Linux platforms	40	36
Testing of the PE against an anti-virus	5	2
Documentation phase	**152**	**159**
Documentation of the project memory	100	109
Learn about how Polymorphic engines work	10	10
Analyze anti-AV techniques used in common computer viruses	5	6
Research on advanced polymorphic techniques	10	7
Learn about basic assembler language and advanced notation	5	9
Analyze common AV techniques to detect malware	10	4
Find out about the development of PEs in history	7	6
Learn about ELF file infection techniques and algorithms	5	8
Total amount of time	**322**	**315**

Table 8.1: Estimation of tasks and their final required time.

8.3 Risk management plan

In all projects, deviations and unforeseen events occur and therefore a risk plan must be developed to deal with them. However, many times these unforeseen events are due to factors that were not taken into consideration in the initial risk plan, so it is necessary to have a plan that has both a preventive and a proactive basis.

With a view to avoiding most of the problems arising from the calculus of the costs of tasks, the most important or extensive tasks have been allocated a higher estimated time than could be estimated initially. This is because a task with a significant time cost will always be more exposed to deviations.

In order to avoid problems derived from the documentation and report phase, the dates available for the presentation of final degree projects have always been taken into account. However, in the development section, the first of these dates has been taken into account as the final date, in order to avoid time dilation and to prevent the problems overcome and the knowledge acquired from falling into oblivion.

8.4 Deviations

There have not been any major deviations in the project development. In fact, the overall time spent on the project has been less than what was initially estimated.

Regarding management, the communication with the tutor has been very fluid, resulting in a reduction in planning and tracking time. Also, there have been unexpected meetings due to minor problems, so the final communication time has suffered a little deviation.

Additionally, the required time for tasks in the development phase where not calculated very accurately, since some of the main tasks were underestimated, and many other simpler ones overestimated. In general, the actual development of the core engine has required more time than expected. On the contrary, the advanced techniques additions and ant-virus testing have demanded less workload than expected.

Finally, the lack of experience writing technical documents and the need to revise concepts related to assembly language resulted in a minor deviation in the documentation phase. Some areas, like ELF infection, were wider than expected, which in the end consumed more time than expected at the beginning of the project.

9. CHAPTER

Conclusions and future work

The motivation for the first polymorphic viruses was to evade the powerful and effective techniques anti-viruses used to detect their presence. Improvements in viral technology have led to more complex detection systems. In the same way, this development in anti-viruses has led to stealthier and more sophisticated evasion techniques. This cat and mouse game push the area into much greater levels of finesse.

Even though polymorphic viruses are capable of remaining undetected, the anti-virus community reacted and developed techniques to try and spot them. These techniques, such as code emulation, or recent usage of machine learning, are capable of unmasking polymorphic viruses. The obvious reaction from the virus-writing community came with the development of metamorphic viruses. These viruses use a mutation engine to mutate their whole body, even the engine itself.

Metamorphic viruses are way more powerful than their polymorphic counterpart, but they also require a higher level of complexity. Anti-viruses also evolved with metamorphic viruses, and are capable of detecting them, at the expense of a great computational cost. Viruses can use as many resources as they wish, but anti-viruses must remain user friendly, and cannot afford a high CPU rate. This limitation is vital, since a delay in detection of a few hours or a day, can lead to a full-system infection.

Virus writers know this and do not put too much effort into the development of their work. Virus development is often done with not too much care, and the advanced techniques mentioned in this document as well as many others are left out of the equation. Laziness is the biggest drawback in this community since anti-viruses are so constrained by resource usage that simple viruses can fulfil their goal with very little effort.

Therefore, by analyzing the most used techniques in virus writing, anti-viruses can take a step forward onto tackling these new threats, and thus enhancing the area making virus writers come up with new ideas to avoid being detected.

On the other hand, most of the computer viruses developed in history have targeted mass consumer operating systems, like Windows. There has been little research on Linux viruses since the outcome virus writers can take out from it is quite sheer. Virus writers seek some profit, so the focus on virus development is focused on the platform with more exploitable users. This is why Linux is believed to be a "virus-free platform", something this project has demonstrated it is not. Further research in Linux viruses, like it has happened in the past, will bring more robust techniques in virus detection, thus securing the operating system even more.

Looking ahead in time, there are various aspects of the project which could be enhanced in future work. First of all, the robustness of the engine could be increased including more advanced obfuscation techniques. The management of the keys is an aspect to improve, being able to add techniques that increase the initial entropy of the keys. In this way, the randomness of the engine could be higher, proportionally increasing the degree of polymorphism of the result.

Subsequently, the data access profile is linear, something that could set off the alarms in an anti-virus. To avoid this, techniques like `PRIDE` implemented in `MetaPHOR` [23] could be added to Bagheera. This technique protects the virus from detection using heuristics, accessing the encrypted data randomly, emulating the data access profile of a regular program.

Finally, as mentioned in Chapter 6, the infection method is very simple but easily detectable at the same time. Other infection algorithms and techniques can be implemented to achieve a more stealthy binary hijacking. Although their stealthiness is bigger than the one of the PT_NOTE infection, this method does not restrict the size of the payload, something to take into account with, for example, `.text` segment infection methods. As it has been shown, the PT_NOTE method, combined with Polymorphic techniques, is capable of providing a sturdy infection. On the other hand, expanding the type of files that can be infected to PIE binaries is a very valuable addition, since it enlarges the number of binaries susceptible to infection greatly.

Bibliography

[1] T. Chen and J.-M. Robert, "The Evolution of Viruses and Worms," 2004.

[2] G. Smith, "The Virus Creation Labs: A Journey into the Underground." Available at: https://vx-underground.org/zines/29a/29a3/29A-3.2_A.txt, 1994. Last accessed 14 June 2021.

[3] Dark Avenger, "The Mutation Engine." Available at: https://web.archive.org/web/20120204065804/http://vx.netlux.org/vx.php?id=em11, 1991. Last accessed 14 June 2021.

[4] F. Cohen, "Computer viruses: Theory and experiments," *Comput. Secur.*, vol. 6, pp. 22–35, 1987.

[5] D. Spinellis, "Reliable identification of bounded-length viruses is NP-complete," *IEEE Transactions on Information Theory*, vol. 49, no. 1, pp. 280–284, 2003.

[6] J. von Neumann, "First Draft of a Report on the EDVAC," tech. rep., June 1945.

[7] J. von Neumann, *Theory of Self-Reproducing Automata*. Champaign, Illinois: University of Illinois Press, 1967. Edited and completed by Arthur W. Burks.

[8] W. Schneider, "Computer viruses: What they are, how they work, how they might get you, and how to control them in academic institutions," *Behavior Research Methods, Instruments, & Computers*, vol. 21, no. 2, pp. 334–340, 1989.

[9] B. B. Rad, M. Masrom, and S. Ibrahim, "Evolution of Computer Virus Concealment and Anti-Virus Techniques: A Short Survey," 2011.

[10] Microsoft Support, "WD: Frequently Asked Questions About Word Macro Viruses." Available at: https://web.archive.org/web/20110604162558/http://support.microsoft.com/kb/187243/en, 2006. Last accessed 07 June 2021.

[11] P. Szor, “The Art of Computer Virus Research and Defense,” 01 2005.

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

[13] C. Nachenberg, “Computer Virus-Antivirus Coevolution,” *Commun. ACM*, vol. 40, p. 46–51, Jan. 1997.

[14] Mister Sandman, Jacky Qwerty, and GriYo, “PE infection under Win32.” Available at: `https://vx-underground.org/zines/29a/29a2/29A-2.3_1.txt`, 1998. Last accessed 14 March 2021.

[15] The Mentall Driller, “Advanced Polymorphic Engine construction.” Available at: `https://vx-underground.org/zines/29a/29a5/29A-5.204.txt`, 2000. Last accessed 14 March 2021.

[16] Rajaat, “Polymorphism.” Available at: `https://vx-underground.org/zines/29a/29a3/29A-3.2_A.txt`, 1998. Last accessed 14 March 2021.

[17] J. Aycock, “A brief history of Just-In-Time,” *ACM Computing Surveys*, vol. 35, pp. 97–113, 2003.

[18] P. Kobalicek, “AsmJit Project official documentation.” Available at: `https://asmjit.com/`, 2021. Last accessed 02 June 2021.

[19] F. Cesaroni, S. Di Marco, E. Gennari, and S. Gentile, “A general purpose arithmetic logic unit,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 260, no. 2, pp. 425–429, 1987.

[20] C. Lindig, “Random testing of C calling conventions,” in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pp. 3–12, 2005.

[21] M. C. Bolingbroke and S. L. Peyton Jones, “Types are calling conventions,” in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pp. 1–12, 2009.

[22] H. Lu, M. Matz, M. Girkar, J. Hubiçka, A. Jaeger, and M. Mitchell, *System V Application Binary Interface, AMD54 Architecture Processor Suplpement*, 2021.

[23] P. Beaucamps, “Advanced Metamorphic Techniques in Computer Viruses,” in *International Conference on Computer, Electrical, and Systems Science, and Engineering - CESSE’07*, (Venice, Italy), Nov. 2007.

[24] J. Eichenauer and J. Lehn, “A non-linear congruential pseudo random number generator,” *Statistische Hefte*, vol. 27, no. 1, pp. 315–326, 1986.

[25] P. Lacharme, A. Rock, V. Strubel, and M. Videau, “The linux pseudorandom number generator revisited,” 2012.

[26] The Mentall Driller, “Metamorphism in practice or "How I made MetaPHOR and what I’ve learnt".” Available at: https://vx-underground.org/zines/29a/29a6/29A-6.205.txt, 2002. Last accessed 14 June 2021.

[27] The Mentall Driller, “MetaPHOR v1B.” Available at: https://vx-underground.org/zines/29a/29a6/29A-6.602.txt, 2002. Last accessed 14 June 2021.

[28] M. Van Oers, “Linux Viruses–ELF File Format,” *Virus Bulletin Conference*, vol. 123, 2000.

[29] Tool Interface Standard, *Executable and Linking Format (ELF) Specification*. 1995.

[30] The Santa Cruz Operation Inc., *System V application Binary Interface*, ch. 4-5. 1997.

[31] Mayhem, “The Cerberus ELF Interface.” Available at: http://phrack.org/issues/61/8.html#article, 2003. Last accessed 05 June 2021.

[32] S. Cesare, “Shared Library Redirection via ELF PLT Infection.” Available at: http://phrack.org/issues/56/7.html#article, 2000. Last accessed 05 June 2021.

[33] S. Cesare, “UNIX Viruses.” Available at: https://ivanlef0u.fr/repo/madchat/vxdevl/vdat/tuunix01.htm, 1998. Last accessed 05 June 2021.

[34] S. Cesare, “UNIX ELF Parasites and virus.” Available at: https://ivanlef0u.fr/repo/madchat/vxdevl/vdat/tuunix02.htm, 1998. Last accessed 05 June 2021.

[35] Red Hat Inc., “Position Independent Executables (PIE).” Available at: https://access.redhat.com/blogs/766093/posts/1975793, 2012. Last accessed 05 June 2021.

[36] S01den, “Return To Original Entry Point Despite PIE,” *tmp.out*, vol. 1, no. 11, 2021. Available at: https://tmpout.sh/1/11.html. Last accessed 05 June 2021.

[37] Sblip, “Implementing the PT_NOTE Infection Method in x64 Assembly,” *tmp.out*, vol. 1, no. 2, 2021. Available at: https://tmpout.sh/1/2.html. Last accessed 05 June 2021.

[38] S. Pearce, “Viral Polymorphism.” Available at: https://vx-underground.org/archive/VxHeaven/lib/asp00.html, 2003. Last accessed 14 March 2021.

[39] Rogue Warrior, “Guide to improving Polymorphic Engines.” Available at: https://vx-underground.org/archive/VxHeaven/lib/vrw02.html, 1996. Last accessed 14 March 2021.

[40] The Black Baron, "A general description of the methods behind a polymorph engine." Available at: https://vx-underground.org/archive/VxHeaven/lib/vbb01.html, 2007. Last accessed 14 March 2021.