UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA IN
Ingegneria Informatica

Metamorphic transformations: safeguarding Software IP with Generative Grammars

Relatore:
Prof. **A. Pellegrini**

Laureando:
Matteo Ciccaglione
Matr. **0315944**

ANNO ACCADEMICO 2022/2023

Abstract

This thesis work aims to introduce an innovative obfuscation technique based on the use of generative grammars in the metamorphic field to provide a weapon against reversing attacks on the intellectual property of the software.

The metamorphic engine allows each instruction of an executable to be transformed into a group of one or more semantically equivalent instructions. A generative grammar is used to produce, in a non-deterministic manner, the group of equivalent instructions into which to transform a starting instruction. The created tool can only be applied to binaries in ELF format and for an Intel architecture.

The transformation process is made up of several phases: there is a first initial phase in which we try to cancel the changes made in a previous application of the engine to the executable in order to prevent the size of the binary code from growing exponentially. Therefore, we initially try to reduce the instructions (by performing the initial transformations but in reverse) and to reorder the instructions, trying to cancel the permutation made in the previous iteration. Both of these operations lead to changes in the ELF only if it had previously undergone a transformation process. This is followed by a permutation of the instructions and an expansion phase, in which each instruction is passed as input to a generative grammar to be transformed.

A test-driven development approach was followed, during which the various components of the engine (i.e. the portions of code responsible for different operations) were tested using specific executable files constructed in such a way as to test almost all exceptional cases that may be encountered when trying to modify an executable file in the ELF format; furthermore, they have also

been tested on some executables contained in the /bin directory.

Moreover, tests were carried out to verify the engine's obfuscation capacity, applying the latter to a malware sample and analysing the variation in the detection rate, exploiting the Virus Total API for malware analysis. This also enabled identifying which anti-viruses were most resistant to this obfuscation technique.

Performance tests were also conducted to verify whether the variation in execution times caused by the application of the engine remains almost constant between one transformation iteration and another. The tests were carried out on two benchmarks also to verify that the execution time did not increase too much after the metamorfic transformations.

Chapter 1 introduces the problem of protecting intellectual property, and how obfuscation techniques can be used to prevent reversing activities. An introduction to binary obfuscation is also presented. Chapter 2 discusses the current state of the art and aims to address existing gaps in the literature. Chapter 3 introduces the concept of metamorphism and explains how it can be used in conjunction with generative grammars. Chapter 4 describes the implementation work, including detailed explanations of interesting algorithms used. Chapter 5 discusses the experimental results, and Chapter 6 presents conclusions and ideas for future improvement.

Contents

List of Figures

List of listings

List of Algorithm

1. Introduction

In this chapter there is an introduction to obfuscation techniques in general and how these can be used to solve the problem that this thesis work aims to address, namely the defense of the intellectual property of software from attacks based on reverse code engineering .

1.1 Obfuscation techniques and why they are used

Binary obfuscation refers to different methodologies and techniques that manipulate existing applications to hide away their internal details [43]. They are used extensively in computer engineering, especially for malware applications. In fact, malicious software abuses these techniques to avoid antivirus detection. Obfuscation can also be used for good intentions, for example to defend Intellectual Property (more details in Section 1.2).

Code obfuscation is also used to provide security in Android applications using the code shrinker R8 [36]. In this case, obfuscation is made using both shrinking (roughly speaking, it would be the removal of unused classes) and renaming classes to unreadable short names that will prevent a malicious user from reverse engineering the app.

Some of the most used techniques are:

- binary code permutation;
- binary code encryption and packing;

- metamorphism.

Binary code permutation is a simple reorganisation of the control flow graph of the application [37], while encryption and packing are strong obfuscation techniques largely diffused in malicious application. Encryption is based on the use of cryptography to hide the representation of the program [39], and packing [18] is the art of reducing the size of the code by transforming it into something else that cannot be executed directly but needs the support of another piece of code that is then added, named *unpacker* that is in charge of undoing all the operations performed by the packer. Sometimes malware authors write their own packers, sometimes use a commercial one, like UPX [32].

All of these techniques suffer from a common pitfall: they are inherently static in that, once generated, the program's binary image is not subject to any change. Most of the reverse engineering techniques adopted to violate IP or analyse a malware are based on manual analysis leveraging *debuggers* and *disassemblers* (e.g., [1, 13, 15]).

Metamorphism [5, 4] is a technique that tries to exceed this limitation by changing the image of the executable after each execution. In this way, we will have a different version with the same behaviour but different binary code. This makes it difficult for a reverse engineer to combine both static and dynamic analysis.

Traditional metamorphic approaches found in literature (see, e.g., [5, 41]) rely on a fixed and size-reduced set of transformations that can be randomly applied to binary code instruction. This represents a huge limitation, in fact also if the rule to apply is randomly picked from this set, it will probably repeat from one transformation to another. This limit can be overcome by means of generative grammars that permit us to generate a transformation from one to many instructions in a random way, combining most of the available rules. Also, the use of generative grammar gives the developer the ability to change the set of transformation by simply changing the grammar set of rule, instead of rewriting and recompiling the binary application.

This thesis work focuses mainly on obfuscation by means of metamorphism

used in combination with generative grammars; more details about metamorphism and how it works can be found in Section 3.1.

1.2 Reverse Code Engineering: a threat for Intellectual Property

Reverse Code Engineering is an activity aiming at re-building the high-level representation of a binary executable or to understand the inner workings of an application. It includes many different techniques that can be divided into two main categories: Static;Dynamic.

Static techniques (such as disassembling, file signature analysis, or symbol analysis) are a set of different methodologies with a common aspect: they do not require program execution. Instead, dynamic techniques (e.g. Debugging, Unpacking) are based on the observation of the program when executed. Some of these, such as debugging, execute the program in a controlled manner, while others (such as unpacking) do not.

These techniques are widely employed in *cybersecurity* to discover potential vulnerabilities in a software executable, as well as a sequence of operations that allows us to distinguish between malware and benignware, but they can also be used to reverse engineer a commercial product, in order to hack it (e.g. discover a way to bypass activation key usage) or copy it. In this sense, reverse engineering can be a potential threat for Intellectual Property, since it enables an attacker to violate copyright and exposes internal details of a product. Additionally, vulnerability disclosure can be seen as an attack to intellectual property if done without the consent of the owner.

There are different well-known intellectual property violations, such as the one by George Francis Hotz in 2011, which reversed the code of the PlayStation 3 operating system [14]. Thanks to this analysis, it was able to discover and exploit a firmware vulnerability, which enabled it to execute unsigned software and gain full control of the console.

Another interesting example can be found in a story told by Kevin D.

Mitnick in The Art Of Intrusion [29], about a group of friends that decided to reverse engineer one of the most popular slot machines in casinos. They were able to discover a vulnerability in the *Random Number Generator* used by the machine and used the acquired knowledge to develop a piece of software that was equivalent to the one in the machine's firmware. It was then used to carry out more and more sophisticated attacks, which allowed them to earn millions of dollars.

So now is manifest that a not consensual reverse engineering activity can cause a lot of damage to a Software Company, and then some countermeasure are needed. Binary obfuscation techniques can be useful for this. Not surprisingly, Google has decided to integrate an obfuscation mechanism for Android applications [36], in order to prevent intellectual property violation.

1.3 Thesis' Objective

This work has as objective to design and develop a new obfuscation engine based on metamorphic techniques driven by generative grammars to contrast reverse engineering attempts.

The proposed engine can be integrated into larger tools for the defence of Intellectual Property, such as **MorphVM**. MorphVM is a virtual machine used to run a program that has been written (or compiled) with a custom Instruction Set Architecture (ISA), that can change from one execution to another. Where does metamorphism fit in this context? The use of a custom ISA is an effective reverse engineer repellent, since all of their software used for analysis (static or dynamic) will fail if applied to an unknown and undocumented ISA. But this alone is not enough, since the VM itself must be compiled using a conventional ISA, such as x86 Intel [16]. This makes the VM prone to reverse engineering attacks, and then all efforts made by the software to hide itself using a custom ISA are made useless, since the attacker can obtain information on the ISA structure and how it changes. To contrast this problem, we can include a metamorphic engine in the VM. In this way, we can

obfuscate the VM itself, making it harder for an attacker to break MorphVM's hiding mechanism.

This thesis work is useful in this way, since the entire developed metamorphic engine can be easily included into MorphVM, giving it a hard-to-break obfuscation mechanism.

2. Related Work

Obfuscation techniques are widely discussed in the literature, both from the evasion and detection point of view. The panorama is vast, with techniques such as ROP-based obfuscation [6], opcodes hiding [25], code virtualisation [19, 38], control flow redirection based on exceptions [21] and mimimorphic techniques [44] based on attacking both semantic and statistical analysis. Another type of hiding technique is based on the use of packers, i.e. compressing and encrypting binary code in the executable file, adding an unpacking stub that, when activated, reverses the packing operation before giving control to malware [31, 3, 12, 31].

All of these obfuscation techniques share the same problem: they are static and do not change executable files over time. Therefore, once an analyst has recognised the underlying pattern (e.g. how to unpack or how the code is virtualised), they can easily construct a deobfuscated version of the executable to analyse it without impediments. My solution to this problem will not involve including additional levels of complexity in the binary representations, as many of the above contributions did. Instead, I will switch focus, allowing the obfuscated application to modify itself over time. This will be done through a metamorphic engine that can drastically rewrite, based on some source of randomness, to prevent analysts from observing two very identical versions of the same program.

I plan to rely on generative grammars to simplify the generation, management, and improvement of such a metamorphic approach. Some early work in the literature has dealt with metamorphism and generative grammars, but they always approach the techniques separately. Malware writers primarily use

metamorphic engines to protect their code against detection activities. The most relevant example is Metaphor [41, 40]. Indeed, the author showed how we can build a metamorphic engine to change malware binary code effectively. The techniques reported in [41] were a starting point for this work. Nevertheless, I have already explored the feasibility of applying a grammar-based approach instead of a hard-coded one (as in [41]) to identify a suitable set of transformations to apply.

For what concerns generative grammar works, many works (see, e.g., [45, 46**?** , 8]) describe approaches to use grammars for purposes other than parsing source code [23]. In [45, 46], the authors rely on grammars for automatic code generation in the context of metamorphism. Still, the issue is tackled abstractly, ignoring technical problems (such as a proper definition of grammar tokens), which is highly relevant and challenging in the field. In [**?**], grammars are used for sentence generation based on a user-specified Bison grammar [23], while in [8] grammars are used to generate language sentences for compiler testing. These approaches are highly optimised for their purposes, barely fitting my objectives. Indeed, even though providing a comprehensive grammar for a binary representation were feasible, when matching grammars against existing executables, I would not start from the axiom as these approaches do. Therefore, more carefully-tailored solutions must be explored, which is what i have done in this thesis.

Regarding binary manipulation, several proposals in the literature provide the capability to manipulate executables (see, e.g.,[34, 2, 24]) with different levels of flexibility. The main difference is that all these proposals target the study or the improvement of non-functional properties of the applications; thus, the set of modifications they support is somewhat limited. Conversely, a significantly more extensive collection of changes on binary files (although driver by grammars) shall be supported to provide an effective metamorphic engine. This requires finding careful solutions not to break the application's correctness and extracting information that is not explicitly available in the program, e.g., through ad-hoc heuristics.

3. Grammar-driven Metamorphism

In this chapter we want to explore in detail the methodologies adopted in the thesis work, in particular with regards to metamorphism, the algorithm to follow to create a functioning and efficient metamorphic engine is explained in detail. The chapter also addresses in parallel the topic inherent to the other important component of this work, namely generative grammar, and then concludes with a discussion on how a generative grammar can be used within a metamorphic engine and with what objective.

3.1 Metamorphism

In this section, it is explained in detail how metamorphism techniques work and what are the code blocks to be used to create a metamorphic engine. As said in Section 1.1, metamorphism aims to change the binary code of an executable, building an equivalent copy of it, but with different instructions.

"The Mental Driller", author of Metaphor [41] was the first hacker to use a metamorphic engine inside a malware application for obfuscation purposes in 2002.

The basic idea is to include into the program an engine that:

1. Load the binary code in memory and disassemble it using any disassembling algorithm;

2. Make a reduction of the code, that is to say convert two or more instruction into one that is equivalent (inverse operation of 4);

3. Apply a permutation technique to the entire code, changing the instruction order;

4. Expand the code, that is to say transform one instruction into two or more equivalent ones;

5. Updates any inter-instruction reference that has been broken (for example, a jumped instruction that was moved to another address during permutation and expansion phases)

6. Rewrite the disk image of the file.

All of these operations must be performed every time the program is executed, in order to change the file after each execution.

Let us see in details how each phase works and why it is used.

Phase 1 involves the use of a disassembler to load and correctly read the executable (the type of disassembler used or the algorithm adopted is not relevant). This phase also requires an algorithm to deal with file format specific details; for example, for what concerns the ELF format (the one used for Linux executables) [26], a software that deals with executable header, section header, and other details is needed. A good practise while interpreting the file content is to build an intermediate representation of the file, that permits one to hide low level details, so that for the other phases (excluding the 6) you do not have to deal with this aspects. The use of an intermediate representation makes the development of the rest of the engine easier.

As just said before, phase 2 is the inverse of 4, so for this part of the development process, we need a table of inverse transformation (that are the same used during the expansion). This phase is strictly related to 4, so we will explore both together. To transform the binary code of an executable, in order to build a newer but semantically equivalent version of the software, you need to transform one instruction into two or more that practically does the same thing. There are basically two approaches to do that:

- Use a static set of hard-coded rules for transformation;

- Use a generative grammar to drive the transformation—this approach is the one adopted in this thesis.

In both cases, the selected rule must be chosen randomly, ensuring that the set of transformations applied by the engine will be different in each execution. Once you have selected which rule to apply, you have to perform the transformation effectively, that is, remove the previous instruction and add the new ones.

This is all for what concerns phase 4. To prevent the file size from growing unbounded, phase 2 is needed. During this phase you use the table of inverse transformation to randomly select a rule and use that to reduce binary code, converting two or more instructions into one. In this way, you are (probably because of the randomness) undoing all the operation that was done during the expansion phase of the previous engine execution.

Applying both 2 and 4, you can control the size of the executable. If the file size increases too much, the antivirus may consider it as an indicator of maliciousness.

Phase 3 is used to add more randomness to the transformation process. Basically, it consist in changing the sequence of instructions, making a permutation of them, and adding some *jmp* instruction (see Appendix B) between them, so that the execution flow is not altered. For example, suppose that we have this sequence of instructions:

Listing x86 Asm example pre permutation

```
1 add rax,0x10
2 sub rdx,0x9
3 add rax,rdx
4 ret
```

Listing 3.1: x86 Asm example pre permutation

then, a possible permutation of this can be the following:

Listing x86 Asm example post permutation

```
1     jmp add_a_10
2 add_r_d:
3     add rax,rdx
```

```
4  jmp r
5  add_a_10:
6      add rax,0x10
7  jmp sub_d_10
8  r:
9      ret
10 sub_d_10:
11     sub rdx,0x9
12     jmp add_r_d
```

Listing 3.2: x86 Asm example post permutation

Expansion, reduction, and permutation can break instruction links, such as a conditional jump that refers to another instruction in the code that has been moved at another address. Phase 5 addresses this issue.

Finally, once we have transformed our binary executable using the intermediate representation that we built, we have to make these transformations permanent by rewriting the entire file in phase 6.

Why is metamorphism effective as an obfuscation mechanism?

The strength of metamorphism lies in the ability to completely modify the binary code of an executable and, thanks to randomness, to build a new version after each execution of the engine. This means that if an attacker tries to reverse the program, he will not be able to combine static and dynamic techniques. In fact one of the most largely adopted approaches for reverse engineering is to exploit static techniques (like disassembler) to obtain the greatest amount of information and then exploit dynamic techniques (like debugger) to investigate unclear aspects. For example, an attacker could use the disassembler to look for functions and then use a debugger to analyse these functions. Since dynamic techniques trigger the engine activation, he cannot use function addresses obtained with static techniques while using the debugger, since the address is probably not the same.

For comparison with other obfuscation techniques, consider packers. A packer generates an executable with several distinctive traits, which can be seen as indicators by an antivirus. For example, the names of the sections (.text, .data, etc.) change, and furthermore the code section is very small (it

only contains the unpacker) while the data section grows (this is because having to unpack at run time, the code must necessarily be present in a segment that is writable and readable). Indeed, metamorphic executables are indistinguishable from normal ones. This makes metamorphism a more stealthy technique than packer, which is what we want, since a metamorphic engine is intended to provide reverse engineering security for commercial products, and we don't want these software to be seen as malware by antivirus.

As said previously, the set of transformations to be applied can be static or generated by a grammar. The use of a generative grammar allows us to make complex and always different transformation with respect to the static solution. This methodology will be explained more in detail in the following sections.

3.1.1 ELF intermediate representation

Build an intermediate representation of the executable file, according to its format, is useful when dealing with metamorphism, since it allows one to work at a higher level of abstraction. In this thesis work, I mainly focused on the ELF executable file (see Appendix A).

To build an intermediate representation of an ELF, I need to work with:

- Executable Header;
- Section Header Table;
- Program Header Table;
- Section Data;

The program header table contains an entry for each segment header. Segments constitute the memory image of the program and must be properly handled, in order to prevent segmentation fault errors. In fact a segment specifies access permissions; therefore, if the executable code grows to the point of

overflowing into a data segment, it is possible that this does not have execution permissions, thus giving rise to a segmentation fault. To avoid this, it is necessary to update the size of the segments and their position.

The section header table contains an entry for each section header. A section is the smallest unit of an object that can be relocated. The linker uses sections to mix together sections of the same type that come from different modules. A section header contains information like the section name, the section type, and where the start address of the section data.

There are different section types that must be properly handled:

- SHT_PROGBITS, this type of section is assigned to the one containing executable code;
- SHT_SYMTAB, this type of section contains the *symbol table*;
- SHT_RELA, this type of section contains the relocation table which makes use of an addend;
- SHT_REL, this type of section contains the relocation table;
- SHT_DYNAMIC, this type of section holds dynamic linking information.

To build a complete ELF representation, I need to deal with all of this section types.

For what concerns SHT_PROGBITS sections, I need to disassemble their content using the proper algorithm, which depends on the type of ISA used.

Also, I need to deal with symbols that are in the *symbol table* (SHT_-SYMTAB section). In particular, I have to locate each function of the executable since the entire transformation engine is based on the concept of a function.

Each symbol contains:

- A value field, whose meaning depends on the type of symbol, typically it is the address where the content is located;

- A size field, containing the size in bytes occupied by the content of the symbol;
- A type field, which specify the symbol type;
- A bind field;
- A visibility field;
- An index;
- A name.

For each function, there is an entry in the *symbol table* which contains the size of the function and its start address as a value. Using this information together with instruction addresses and their size, it is simple to assign each instruction to its function.

Symbol management is not enough, in fact there are some symbols that contain a zero value. These types of symbols must be handled working on the information in SHT_DYNAMIC section.

The SHT_DYNAMIC section contains a table whose entries contains:

- A tag value which specify the type of the entry;
- A value field, whose meaning depends on the type of the entry, typically is the address where the content is located or a string.

An example of a symbol that must be handled using the SHT_DYNAMIC section are the start address of the _init and _fini functions, as well as the address of the _INIT_ARRAY and _FINI_ARRAY.

Another thing that must be done when building an intermediate representation of an ELF file is reference tracing in the code. There could be two types of reference in the code:

- References between instructions;
- References from instructions to data;

- References from data to instructions.

The first topology is associated with branch operation or in general to instruction which may cause a shift in the execution flow. In a similar scenario, there is a source of the reference (e.g. the branch instruction) and a target. Each change to the binary code could shift these two instructions, making a subsequent adjustment necessary to avoid breaking the original relationship. The same thing is valid for the second topology, in which there is a source instruction, but the target is not another instruction, but rather a memory address. This type of reference is typically related to an operation which moves data from memory to register and vice versa.

In the end, the third topology is typically associated with jump table entries. In fact a jump table entry contains a displacement, which if added to a base address provides the address of the instruction to be executed.

For more details on how the intermediate representation of the ELF was built, see Section 4.1.

3.2 Grammars

In this Section, we introduce the use of grammars, especially generative grammars, in computer science. A grammar is a set of rules, statements, and axioms which describe a formal language [10]. Typically a grammar is used to build a parser that validates sentences of a language, for example, an instruction statement in a program. In fact grammars are used by compilers to validate the syntax of a program.

Grammars can be classified, according to Chomsky Hierarchy [9], in :

1. Type 3, used to generate regular languages, which are formal languages that can be defined by a regular expression;

2. Type 2, also known as Context-Free Grammar (CFG), used to generate context-free languages, which have many application in programming

languages (e.g. most arithmetic expressions are generated by context-free grammars);

3. Type 1, also known as Context-Sensitive Grammar (CSG), used to generate context-sensitive languages;

4. Type 0, used to generate recursively enumerable languages.

Levels of the Chomsky Hierarchy are inclusive; this means that all CFG grammars are regular, but is not true that all regular grammars are CFG.

Context-free grammars are used to specify the syntax of programming languages, enabling the development of parsers and compilers. The Backus-Naur Form (BNF) [27] and the Extended Backus-Naur Form (EBNF) [11] are commonly used notations to define context-free grammars for programming languages. One tool largely used to generate a parser for Context-Free Grammars is Bison [23]. For my thesis work, I used Bison to generate a parser (starting from a metagrammar), in order to validate the generative grammar submitted to the engine.

3.2.1 Generative Grammars

Grammars can also be distinguished in :

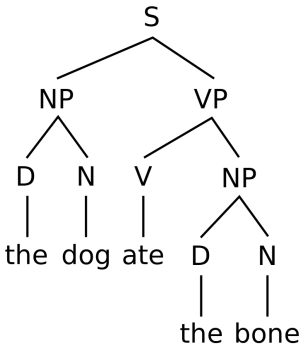
- Prescriptive;
- Descriptive;
- Generative.

The term prescriptive grammar refers to a set of norms or rules governing how a language should or should not be used rather than describing the ways in which a language is actually used, unlike a descriptive grammar, which is an examination of how a language is actually being used in speech and writing.

Instead, generative grammar attempts to get to something deeper—the foundational principles that make language possible across all of humanity.

Generative grammars consider grammar as a set of rules that generate exactly those combinations of words that form grammatical sentences in the given language. They can be used to generate infinite-length sentences of a language.

In reference to Chomsky's hierarchy (3.2), generative grammar can be seen as Context-Free Grammar. In fact, as Chomsky himself said, type 3 grammars are not expressive enough to describe a human language. A derivation of a sentence using a CFG can be seen as a derivation tree, which can be parsed in order to build the generated sentence. Let us see an example of a derivation tree:



CFGs have been widely used in natural language processing (NLP) and parsing algorithms. They are the basis for many syntactic parsers that analyse the grammatical structure of sentences, such as the Earley parser [28] and the CYK parser [30].

The most famous example of generative grammar belongs to Noah Chomsky, and it is the concept of a "finite set of generative rules" for generating an infinite number of sentences in a language. Chomsky introduced this idea in his groundbreaking work "Syntactic Structures," which was published in 1957 [22]. In this book, he argued that human languages are generated by a finite set of recursive rules that can produce an infinite number of sentences.

One of the key illustrations Chomsky used in "Syntactic Structures" is the following sentence:

"Colorless green ideas sleep furiously."

This sentence was intentionally constructed to be grammatically correct

but semantically nonsensical. Chomsky's point was that the grammaticality of a sentence is separate from its meaningfulness. By presenting this sentence, he aimed to demonstrate that generative grammar is concerned primarily with the structure and rules of syntax, not with the meaning of sentences.

Chomsky's work on generative grammar laid the foundation for the transformational grammar framework, which includes transformational rules for generating different sentence structures from a deep structure to a surface structure while preserving meaning. His ideas revolutionised the field of linguistics and continue to be influential in the study of language to this day.

What does a generative grammar look like? Let us see an example of a generative grammar for English language. In English, a basic sentence follows the Subject-Verb-Object (SVO) structure. Here's an example:

- Subject (S): "She"
- Verb (V): "plays"
- Object (O): "the piano"

So, "She plays the piano" follows the SVO structure. A generative grammar will start from this structure to generate a sentence that is syntactically correct, but not necessarily semantically correct.

Generative grammar also includes transformational rules that explain how sentences can be transformed into different forms while preserving their underlying meaning. For example:

Active to Passive Transformation:

- Active: "The cat chased the mouse."
- Passive: "The mouse was chased by the cat."

Transformational grammar laid the groundwork for modern NLP applications, including machine translation. By understanding the transformations between languages, early machine translation systems attempted to convert sentences from one language to another.

Another important example of generative grammars is the Probabilistic Context-Free Grammar (PCFG).

PCFGs are extensions of context-free grammars that assign probabilities to productions. In a PCFG, each production rule in the grammar is associated with a probability. These probabilities represent the likelihood of using a particular rule to generate a specific syntactic structure. For example, in a PCFG for English, you might have a production rule like:

```
1    VP -> V NP [0.6]
```

Listing 3.3: PCFG rule example

For the purpose of this thesis work, the generative grammar must be a transformation grammar, since it has to transform one instruction into two or more equivalent ones, but also a probabilistic grammar because all rules must be selected randomly.

3.3 Using Grammars for Program Metamorphism

Generative grammars can be used to derive sentences for a given language. In this thesis, I decide to use generative grammar to derive a set of instructions that, if executed one after the other, give place to the same effect of a starting instruction. For example, supposed to start from an add operation, such as

```
1    add rax,0x8
```

Then a possible outcome of the grammar can be:

```
1    sub rax,0x4
2    nop
3    mov rdx,rdx
4    add rax,0x10
5    sub rax,0x4
```

This is a set of instruction that brings to the same result of the starting instruction: the content of the register *rax* is incremented by 8.

To build an engine that can do these things, I started from Forson [**?**], which is a tool that receive as input a language description (a .y file containing terms and rules) and generate sentences for this language. Unfortunately Forson doesn't work correctly on x64 machines. So I adapted Forson source code, in order to make it works on x64 machines and I integrate it in my metamorphic engine.

When dealing with grammar engine, the instructions are converted into proper tokens.

A token is a term in the language used, and it reflects the instruction type, for example an instruction like jmp rax has as token JMP.

The set of token used is:

```
1    typedef enum Token_type{
2   ADD,
3   SUB,
4   MOV,
5   LEA,
6   LEAIMM,
7   MUL,
8   JUMP,
9   CMP,
10  JCC,
11  CALL,
12  PUSH,
13  POP,
14  OR,
15  AND,
16  XOR,
17  RET,
18  DEF,
19  NOP
20 }Token_type;
```

Listing 3.4: Token type enumeration

Each instruction type is directly mapped to a token that relies exclusively on the mnemonic without taking into account the rest, except for the lea instruction, which can be mapped to LEAIMM if it is of the type lea reg,[reg+imm], where reg is any register and imm is an immediate value.

These tokens are the terminal values of the grammar. Terminal values are

those generated by the grammar at the end of the derivation process. This means that, in reference to the previous generation example, the grammar has generated the terminal values SUB NOP MOV ADD SUB.

The language description must contain one and exactly one set of rules for each of these tokens, except for the DEF token, which corresponds to an unhandled instruction, which remains unchanged. The grammar engine takes as input a .y file containing the language description and a starting token, corresponding to the instruction that has to be transformed. The language adopted in this thesis work is described by the following .y file:

```
1     %{
2   #include <math.h>
3   #include <stdio.h>
4   int yylex (void);
5   int yyerror (char *);
6 %}
7
8 %token ADD SUB MUL DIV JCC MOV LEA LEAIMM JUMP CMP CALL PUSH POP OR AND XOR
      RET NOP DEF
9
10 %%
11
12
13 single_instruction: add
14     | sub
15     | mul
16     | div
17     | jcc
18     | mov
19     | lea
20     | leaimm
21     | jump
22     | call
23     | cmp
24     | push
25     | pop
26     | or
27     | and
28     | xor
29     | ret
30     | nop
31     | def
32 ;
```

```
33
34
35  add: ADD sub
36      | add sub
37      | add nop
38      | ADD;
39
40
41  sub: SUB nop
42      | add SUB
43      | SUB add
44      | sub add
45      | SUB;
46
47  mul: MUL
48      | mul nop;
49
50  div: DIV
51      | div nop;
52
53  jcc: JCC
54      | jcc nop;
55
56  mov: MOV
57      | push pop
58      | mov nop;
59
60  lea: LEA
61      | lea nop
62      | mov;
63
64  leaimm: LEA
65      | leaimm nop
66      | mov add;
67
68
69  jump: push ret
70      | JUMP
71      | jump nop;
72
73  call: CALL
74      | call nop;
75
76  cmp: CMP
77      | cmp nop;
78
79  push: PUSH
```

```
80      | sub mov
81      | push nop;
82
83  pop: POP
84      | mov add
85      | pop nop;
86
87  or: OR
88      | or nop;
89
90  and: CMP
91      | cmp nop
92      | AND
93      | and nop;
94
95  xor: mov nop
96      | mov
97      | XOR
98      | xor nop;
99
100 ret: RET
101     | nop ret;
102
103 def: DEF
104     | def nop;
105
106 nop: NOP nop
107     | MOV
108     | nop
109     | nop nop;
110
111 %%
```

Listing 3.5: .y file containing the language description

The single_instruction rule is necessary to make the file syntactically correct (each token must appear on the right side of at least one rule), but is never used during the derivation process; furthermore, it is ignored during the construction of the inverse transformation table for the shrinking phase (see point 2 of metamorphism process in 3.1).

During its execution, the engine randomly selects a rule from the set corresponding to the given token, and then if the selected rule contains non-terminal symbols, these are themselves expanded in a recursive process that ends when

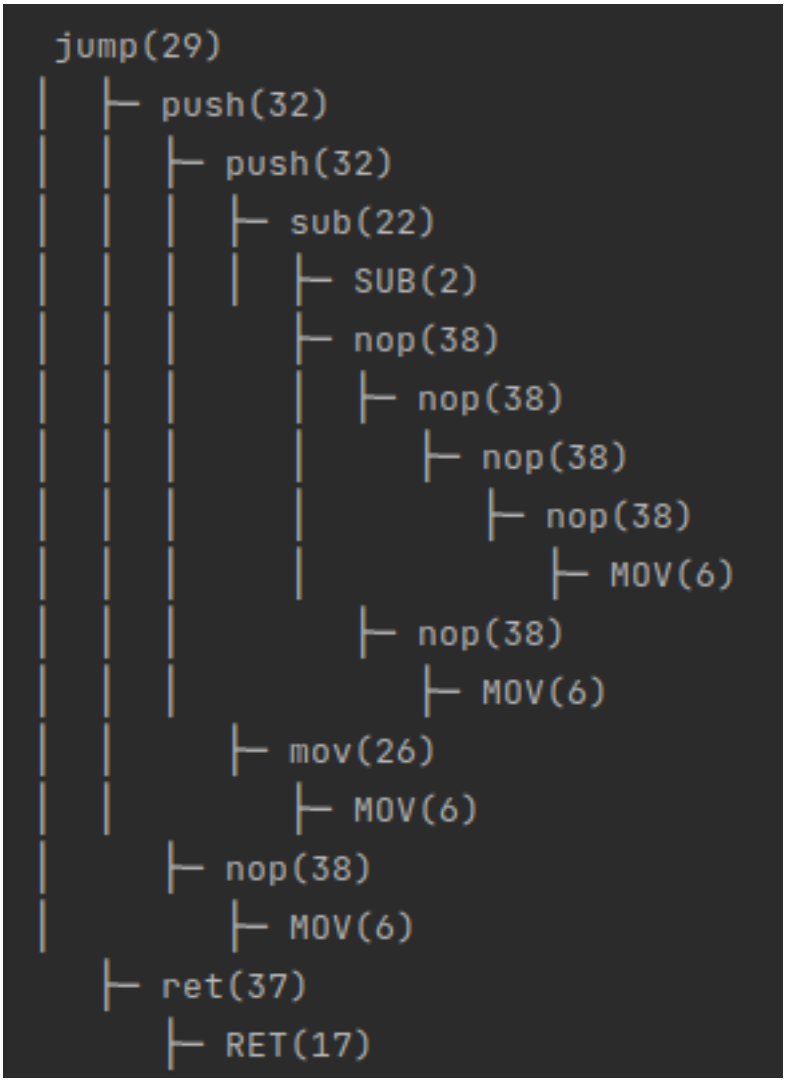


Figure 3.1: Derivation Tree example

all the generated terms are terminal symbols. To prevent the engine from generate an infinite length sentence, I set a recursive cap, that is, a limit to the maximum number of recursive steps, equal to 7 steps. If the engine reaches this cap, it is forced to generate a terminal symbol.

The result of the engine is a derivation tree, which is then parsed by the metamorphic engine to apply the generated transformations and make them permanent. An example of a derivation tree is the following:

In this transformation, the grammar has started from a jump instruction and has generated a transformation into SUB MOV MOV MOV MOV RET. This type of transformation can be applied to an indirect jump (e.g. jmp rax), which can be transformed into a PUSH followed by a RET statements. Numbers in parentheses correspond to internal representation of symbols and rules of the grammar and can be ignored.

Since instruction-to-token mapping depends only on the instruction mnemonic, it is possible that the transformation generated by the grammar cannot be applied to the original instruction. For example, if the derivation tree in Fig. 3.1 is generated for a direct jmp (e.g. jmp 0x20), it cannot be applied, since the instruction is not an indirect jump. In this case, the engine falls back to a transform where it simply adds some garbage statements that are generated by the grammar. Garbage statements are instructions that are no operation (NOP) or equivalent instructions.

This simplistic approach is adequate for the type of grammar provided. In fact, the number of rules in each set is equal to 2 or slightly higher, one of which corresponds to the addition of garbage.

A different approach can be used: to perform an instruction-to-token mapping based on the instruction opcode. This approach does not solve the problem, since two instruction with the same opcode can be different, but it reduces the probability that a generated transformation cannot be applied (we can call this "probability of fail"). However, this comes at a price: an increase in the number of different tokens to manage.

If the supplied grammar becomes more complex, the number of tokens must increase to reduce the probability of fail, otherwise the result of the engine will be a no-operation filled executable.

The grammar used for this thesis work is simplistic, in fact for each group of rules (remember that each group corresponds to a token) there are:

- 1 rule which keeps the instruction unchanged;
- 1 rule which transforms the instruction into two other;

- 1 rule which simply adds garbage instructions.

So if the engine failed selecting 1 rule which cannot be applied, falling back on the "add garbage" rule is not wrong.

I would like to point out that, if the transformation fails, this is only due to the first rule selected during the transformation. In fact, the subsequent rules are applied not to the original instruction, but to "artificial" instructions that have been added by the previous transformation.

4. Reference Implementation

In this chapter the reference implementation and algorithms adopted are presented and discussed in detail.

The structure of the chapter reflects the metamorphic steps presented in 3.1.

4.1 Building the intermediate representation of an ELF

As said in 3.1.1, building an intermediate representation of an ELF gives you greater flexibility during transformation operations on the ELF and also allows you to correctly manage all the metadata relating to the format.

In this way, I can work at an higher level when dealing with instruction manipulation, leaving low-level work to the code part that is responsible for rewriting the ELF.

4.1.1 Data structures

Then I decide to organise the internal details of the file in several data structures that use the ones proposed in the libelf library [20], and also maintain additional data that are useful for ELF rewriting. This is not sufficient, since we had also to read and rewrite all section data, both those containing code (which are the ones with SHT_PROGBITS type) and those containing data, since during the address update phase, it is possible that the sections containing data that have not been directly modified have undergone a change

of address due to the growth of the previous code sections. The data structures used for ELF intermediate representation are the following:

```
1     typedef struct Phdr_table{
2   size_t num_entries;             // number of program headers
3   linked_list *list;              // list of GElf_Phdr structures
4   linked_list *segments;          // list of Segment structures
5 }Phdr_table;
```

Listing 4.1: Data structure for intermediate representation of the Program Header Table

```
1 typedef struct Segment{
2   int index;                      // index of the segment, starting from 0
3   uint64_t start_index;           // segment offset from the beginning of the
      file
4   uint64_t end_index;             // segment end offset, from the beginning of
       the file
5   uint64_t size;                  // size in bytes of the segment
6   linked_list *sections;          // list of sections belonging to the segment
7   unsigned char sensible;     // This bit is 0x1 if the phdr is followed by
      another one with different flags and has to be correctly handled
8 }Segment;
```

Listing 4.2: Data structure for intermediate representation of a Segment

```
1     typedef struct Section_data{
2   Elf_Data *data;                 // content of a section
3   linked_list *instructions;      // list of disassembled instructions, if the
       section is of type TEXT
4   size_t displacement;
5   size_t old_size;
6   size_t padding;
7 }Section_data;
```

Listing 4.3: Data structure for intermediate representation of section's data

```
1     typedef struct Section{
2   Elf_Scn *scn;               // the section handler
3   GElf_Shdr *shdr;                // the section header
4   char *name;                     // the section name
5   linked_list *data;              // list of Elf_Data structures associated
      with the section
6   size_t num_data;                // number of ELf_Data structures
```

```
7   int64_t section_displacement;   // displacement of the section with respect
      to the original offset in the elf, to be reported on elf regeneration
8   size_t old_size;
9 }Section;
```

Listing 4.4: Data structure for intermediate representation of a section header

```
1     typedef struct Sec_table{
2   linked_list *secs;              // List of Section structures
3   size_t num_entries;             // Number of sections
4 }Sec_table;
```

Listing 4.5: Data structure for intermediate representation of Section Header Table

```
1     typedef struct Elf_program{
2   Elf *elf;                       // ELF handle
3   GElf_Ehdr *ehdr;                // Executable header
4   Phdr_table *phdrt;              // Program Header Table
5   Sec_table *sect;                // Section Header Table
6   linked_list *unified_instructions;  // list oif all disassembled
      instructions of the ELF
7   //uint64_t e_entry_addr;    // Address of the entry point
8 }Elf_program;
```

Listing 4.6: Data structure for intermediate representation of the entire elf

```
1     typedef struct Symbol{
2   GElf_Sym sym;                   // the symbol
3   Section_data *sec_data;         // Section data containing the content the
      symbol refers to
4   Section *sec;     // Section containing the content the symbol refers to
5   unsigned long in_data_offset; //Offset in data block of the symbol' content
6 }Symbol;
```

Listing 4.7: Data structure for intermediate representation of an entry of the Symbol table

```
1     typedef struct Asm_function{
2   Symbol *sym;                    // symbol associated with the function
3   linked_list *instructions;      // list of disassembled instructions
      belonging to the function
4   char* name;                     // name of the function
5   uint64_t size;                  // size in bytes of the function
```

```
  uint64_t start_addr;            // initial virtual address of the function
  uint64_t end_addr;              // virtual address of the end of the
    function
  ll_node *fi;                    // Pointer to the first instruction or NULL
    size_t num_instructions;        // number of instructions in the function
    bool uses_jt;     // true if contains a jt
}Asm_function;
```

Listing 4.8: Data structure for function handling

```
    typedef struct Elf_state{
  Elf_program *prog;              // Struct containing all elements from an
    elf file
  linked_list *symbols;           // A list of all symbols
  linked_list *functions;         // A list of all functions in the elf
  linked_list *strings;           // A list of all strings in the elf
  linked_list *j_tables;    // A list of all jump tables in the elf
  linked_list *rela_tables; // A list of all rela tables in the elf
  linked_list *dyn_etries;  // A list of all entries of dynamic table
}Elf_state;
```

Listing 4.9: Data structure for Elf handling

The 4.1.1 structure is used in the engine to have a single access to all the elements that make up an ELF, in order to be able to read and possibly modify them. Single instructions are represented using another data structure in which I take track of all information gather from the disassembler used [34] and additional information that are useful for address adjustment (phase 5 of the metamorphism process). The introduced data structure is the following:

```
typedef struct insn_info_x86 {
    //Previous entry of the struct has been erased since they are copied from
    Hijacker

  // Start of metamorphic engine stuff
  unsigned long long orig_addr;       // original address of the instruction
  unsigned long long new_address;     // new address of instruction
  int64_t displacement;               // displacement of the instruction after
     modifying the elf
  struct insn_info_x86* jump_target;  // target of this instruction
  linked_list *target_of;             // instructions that have this one as
    target
  int jump_op_size;
  unsigned long jump_op_start;
```

```
12    Section_data *sec_data;             // section data where the instruction
        jumps to
13    Section *sec;         // section where the instruction jumps to
14    unsigned long target_addr;      // target addr of original instruction
15    Asm_function *function;       // function to which this instruction belongs
        to
16    bool uses_jt;         // true if this instruction uses a jump table.
17    void *jt;
18    int opd_size;
19    int op[3];
20    unsigned char visited;        // This byte is supposed to be correct only
        during reorder operation. Don't use it outside.
21  } insn_info_x86;
```

Listing 4.10: Data structure for Instruction handling

4.1.2 The jump table problem

A switch statement in a general purpose programming language finds its representation at low-level in the so-called jump table.

A jump table, also known as a branch table or dispatch table, is a data structure used for efficient control flow in software. It is primarily used to optimise decision-making and reduce the need for complex conditional branching in programs. The typical structure of a jump table at the assembly level involves the presence of a table of offset in the read-only memory segment, which is accessed by the code with the use of an indirect jump.

The engine must be able to locate jump tables present in memory and their usage, since this type of instruction references also must be handled. To check if a function is using a jump table, the engine exploit two basic heuristics, which are the following:

1. The function load a memory address in a register;
2. Then uses this memory address plus the content of a register (which is supposed to contain the variable of the switch case) to load the content of a memory location;
3. Then load again the first memory address in a register;

4. The destination of the indirect jump is computed as a sum of the value loaded in point 2 and the one loaded in point 3.

1. The function load a memory address in a register;

2. Then uses this memory address plus the content of a register (which is supposed to contain the variable of the switch case) to load the content of a memory location using a movsxd;

3. The destination of the indirect jump is computed as a sum of the value loaded in point 1 and the one loaded in point 2.

This second type of heuristic is typically associated with an executable which was compiled using some optimisation mechanism (e.g. -O3 option of gcc).

Functions that contain a jump table cannot be touched by the engine, since every modification to the code structure will break this heuristic, making it impossible to recognise the jump table in a subsequent iteration.

4.2 Shrinker

The shrinker is the part of the engine responsible for the ELF reduction phase. As said in 3.1 point 2, the shrinker has to transform two or more instructions into an equivalent one, preventing the ELF size from growing exponentially in subsequent runs of the engine.

4.2.1 Table of inverse transformations

To make instruction reduction possible, I need a table of inverse transformations constructed starting from the .y file provided as a description of the grammar. This table is built by the grammar engine while loading and parsing the .y file. To store the table, the following data structures were created:

```
1     typedef struct INVERSE_RLE_TABLE{
2   inverse_rle **rules;
3   int n_rules;
4   int max_rules;
5 }inverse_rle_table;
```

Listing 4.11: struct of the inverse rules table

```
1     typedef struct INVERSE_RLE{
2   symbol_id start[2];
3   int n_start;
4   symbol_id end[100];
5   int n_end;
6 }inverse_rle;
```

Listing 4.12: struct of the inverse rule

Each rule is actually a set of rules. The array "end" contains each of the possible end symbols that can be obtained starting from the (at most 2) starting symbols present in the "start" array. So the "start" array contains the symbols that identify the set of rules, and end contains each of the possible outcome of the reduction. As an example, since the grammar has a rule that transforms from a jmp instruction to a push followed by a ret (see 3.3), the inverse table will contain an "inverse_rule" entry that has as "start" an array containing both push and ret symbols, and an end array containing at least a jmp symbol

(it can contain more symbols if there are other rules in the grammar that have push and ret as right-hand symbols).

A possible graphical representation of the table is the following:

```
add_sub:
  add
  sub
add_nop:
  add
sub_nop:
  sub
sub_add:
  sub
mul_nop:
  mul
div_nop:
  div
jcc_nop:
  jcc
mov_nop:
  mov
  xor
lea_nop:
  lea
leaimm_nop:
  leaimm
mov_add:
  leaimm
  pop
push_ret:
  jump
jump_nop:
  jump
call_nop:
  call
cmp_nop:
  cmp
  and
sub_mov:
  push
push_nop:
  push
pop_nop:
  pop
or_nop:
  or
and_nop:
```

```
44    and
45  xor_nop:
46    xor
47  nop_ret:
48    ret
49  nop_nop:
50    nop
```

As can be seen from the given representation, the rules that transform one instruction into an equivalent one are ignored. There are no reasons to even consider those transformations; in fact, the objective of the shrinker engine is to reduce the size of the ELF. obviously it is possible that, by applying one of these transformations, a small quantity of bytes can be recovered, since the x86 assembler is of the CISC type and not all instructions have the same length, but the cost to be paid for doing this further increases the workload of the engine.

The algorithm adopted for the derivation of this table is the following:

Algorithm 1: Algorithm for building the Inverse Rule Table

input : An empty inverse_rle_table
output: A correctly filled inverse_rle_table

$sle \leftarrow symbol\_table$
$start \leftarrow [0, 0]$
while $sle \neq NULL$ **do**
  if *Is_NT(sle) is 1* **then**
    $rle \leftarrow get\_rle()$
    for $j \leftarrow 0$**to**$len(rle)$ **do**
      $sym \leftarrow extract\_sym(rle)$ $start[j] \leftarrow sym$
    end
    $put\_inverse\_rle(table, start, rle)$
  end
  $sle \leftarrow sle \rightarrow next$
end

Where the put_inverse_rle function used in this algorithm is a simple function that inserts the end symbol in the inverse_rle entry corresponding to the given start array.

4.2.2 Reduction algorithm

In this section, the reduction algorithm used is introduced. The algorithm converts each instruction into the corresponding token, then uses two tokens at time and looks at the inverse_rle_table for a rule that can be applied. If there are no such rules, a proper error is returned. The rule is selected randomly, using an rng, from the set of rules available, and if the selected rule is wrong, a new one is selected randomly, until all the available rules have been tested. When testing for the feasibility of a rule, the algorithm must do all the checks on the structure of the starting instructions to see if these can be converted into the instruction specified by the rule, and if this is possible, builds the new instruction using an assembler, which is Keystone [17].

Checks that have to be done differs from transformation to transformation, for example looking at the "from add sub to add" transformation, one checks that have to be done are:

- Are the two starting instruction using an immediate value?
- Are they working with the same destination register?
- Aren't they using a displacement value or a scale value?

If all checks pass successfully, then the transformation can be applied and a new add instruction is built using keystone. Clearly, the add instruction will use the same destination register of the starting ones, and its immediate value will be equal to the difference between the immediate value of the original instructions.

No-operation instructions (or something equivalent) are deleted during the reduction process. This means that the algorithm must be able to identify all instructions that can be transformed into a nop. This is done using the entries of the inverse_rle _table that has nop as the only initial value.

If the reduction is possible, then the first of the two starting instruction is changed (this means that the content of the corresponding struct is updated,

but the struct is not removed from the list of instructions) and the second instruction is removed.

4.2.3 How to remove an instruction

Since the reduction algorithm requires to remove an instruction, let me explain how to do this in a correct manner when dealing with ELF intermediate representation and ELF metadata. As shown in ..., the intermediate ELF representation includes a list of all instructions in the executable. Then, to remove an instruction, it is necessary to remove it from the list (using any algorithm for doubly linked list) and possibly update the medatata of the membership function, like its size and starting address.

What happens if the reduction process removes an instruction that is the target of a jump (or conditional jump) instruction? If this case is not handled properly, the resulting executable will be broken.

There are two possible things that can be done:

- Do not carry out this type of removal;
- Before removing the instruction, update the branch instruction to make it jump to the immediately following instruction.

However, the second solution is not always correct. For example, supposed to have this sequence of instruction:

```
1  foo:
2      push rbp
3      mov rax,[rsp-0xc]
4      jmp label
5      <Don't care about what's here>
6  label:
7      jz rax, label2
8      mov rax,0x0
9      jmp label3
10 label2:
11     mov rax,0x1
12 label3:
13     ret
```

In this case, by removing the instruction on line 7, the function will always return 0, for any value given in input.

The first solution, although simple, is the best to apply in this context. In fact, the reduction algorithm removes only the second instruction, not the first, which is indeed changed to the new ones. This means that, if the second instruction is target of a jump, then I cannot reduce the pair of instruction into a new one since there exists a flow of execution in the program that executes the second instruction but not the first. So, in this case, a reduction is not possible, and then the limit imposed by the first solution is irrelevant. The only scenario in which this limit can hurt is when the engine is trying to remove a no-operation statement. But this is not a problem, since the newer no-operations added by the engine during the expand phase cannot be the target of a jump by construction, and so if the engine fails while removing a no-operation, this means that this nop is an original instruction of the ELF and as such must not be removed.

4.2.4 Shrinking the ELF

The reduction algorithm presented in 4.2.2 must be applied to the entire ELF. The metamorphic engine developed for this thesis work is designed to work on one function at a time. In this way, I avoid problems that mine function structure, such as the reduction of two instruction that belongs to two different functions. If the ELF does not contain any functions, it will be treated as an executable composed of only one function.

That said, we need an algorithm that iteratively performs the reduction on each function. There are some functions that cannot be touched by the engine, in order to preserve the correct functionality of the executable. These functions are the ones added by the gcc compiler (e.g. destructor and constructor) and the functions which contain a jump table. The reasons linked to these issues have already been discussed in detail in 4.1.2.

The shrinking algorithm applies the reduction function to two consecutive tokens at a time and uses the outcome of the reduction function as the first

token for the next iteration, so that it can be further reduced until the entire function has been reduced. This process is applied iteratively to each function. The algorithm described is the following:

Algorithm 2: Shrinking algorithm

```
input  : A set of functions
output: Number of reduction performed
count ← 0
for function : functions do
    tokens ← tokenize(function)
    first ← NULL
    while len(tokens) ≠ NULL do
        if first == NULL then
            first ← pop(tokens)
        end
        second ← pop(tokens)
        first ← reduce(first, second)
        if first ≠ NULL then
            res ← res + 1
        end
    end
end
return count
```

To show a possible outcome of the shrinker engine, let us consider this function that has been modified by the expander:

```
0000000000001976 <foo_3p>:
    1976:       f3 0f 1e fa             endbr64
    197a:       48 83 ec 08             sub    rsp,0x8
    197e:       48 89 2c 24             mov    QWORD PTR [rsp],rbp
    1982:       48 89 db                mov    rbx,rbx
    1985:       48 89 db                mov    rbx,rbx
    1988:       48 89 c9                mov    rcx,rcx
    198b:       48 89 e4                mov    rsp,rsp
    198e:       48 89 e5                mov    rbp,rsp
    1991:       90                      nop
    1992:       90                      nop
    1993:       48 89 db                mov    rbx,rbx
    1996:       48 89 d2                mov    rdx,rdx
    1999:       48 89 c0                mov    rax,rax
    199c:       48 83 ec 10             sub    rsp,0x10
    19a0:       48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
    19a4:       48 8d 05 c5 08 00 00    lea    rax,[rip+0x8c5]        # 2270 <_IO_stdin_used+0x270>
    19ab:       48 89 c7                mov    rdi,rax
    19ae:       90                      nop
    19af:       90                      nop
    19b0:       48 89 c0                mov    rax,rax
    19b3:       48 89 db                mov    rbx,rbx
    19b6:       48 89 e4                mov    rsp,rsp
    19b9:       e8 c2 f6 ff ff          call   1080 <puts@plt>
    19be:       90                      nop
    19bf:       90                      nop
    19c0:       48 89 e4                mov    rsp,rsp
    19c3:       90                      nop
    19c4:       90                      nop
    19c5:       48 89 d2                mov    rdx,rdx
    19c8:       48 89 e4                mov    rsp,rsp
    19cb:       c9                      leave
    19cc:       90                      nop
    19cd:       48 89 db                mov    rbx,rbx
    19d0:       48 89 db                mov    rbx,rbx
    19d3:       48 89 db                mov    rbx,rbx
    19d6:       90                      nop
    19d7:       48 89 d2                mov    rdx,rdx
    19da:       48 89 c9                mov    rcx,rcx
    19dd:       c3                      ret
```

Then one possible result of the application of the shrinker on this function could be the following:

```
00000000000016fe <foo_3p>:
    16fe:       f3 0f 1e fa             endbr64
    1702:       55                      push   rbp
    1703:       48 89 e5                mov    rbp,rsp
    1706:       48 83 ec 10             sub    rsp,0x10
    170a:       48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
    170e:       48 8d 05 5b 0b 00 00    lea    rax,[rip+0xb5b]        # 2270 <_IO_stdin_used+0x270>
    1715:       48 89 c7                mov    rdi,rax
    1718:       e8 63 f9 ff ff          call   1080 <puts@plt>
    171d:       c9                      leave
    171e:       c3                      ret
```


Figure 4.2: Example of a function reduced by the engine

4.3 Resequencer

Resequencer is the algorithm adopted to restore the original order of the instructions, which was changed by the permutator. This operation is done after the application of the shrinker, this is because the expander is executed after the permutation has been carried out, and since shrinking is the inverse operation of expanding, it is correct to apply the shrinker algorithm before reordering.

Hopefully, the input of the resequencer is of the same form as the output of the permutator, that is to say, that each instruction is followed by a direct jump, artificially added to maintain program order. This is not certain, as it depends on the outcome of the shrinker, which, as extensively discussed in section 4.2, randomly selects the shrinking transformations.

For this reason, I cannot assume that the structure of the input function perfectly reflects the output of the permutator. So I need a way to distinguish between "artificial" jump (which are the ones that must be removed) and "natural" jump (where natural means that it was originally present in the executable or it was created by the expander), which cannot rely on the properties of a permutator outcome (e.g. each "artificial" jump is preceded by exactly one "natural" instruction).

The idea to solve this problem is that an artificial jump, by construct, is not the target of any other instruction (roughly speaking, this means that there are no labels associated with an artificial jump).

So, if the algorithm meets a jump instruction that is marked as a target of some other instructions, this jump is an original instruction and must be preserved.

Why does this work? Don't forget that the permutator adds a jump instruction for each other instructions in the function, so if a jump instruction was originally present in the function, then there must exist an artificial jump that points to the first one.

But what if the instruction was added by the expander in the previous

execution of the engine? Well in this case the shrinker will probably remove it, but, if this does not happen, this is not a problem since there are no rules in the grammar used that generates a direct jump. There are also no reasons why the grammar should contain such a rule since a code permutation is executed already by the proper algorithm.

So distinguishing from direct and indirect jump (conditional branch statements are ignored) together with the criterion described can solve the problem of discriminating an artificial statement from a natural one.

The main idea of the algorithm is to look at all the function instructions, marking the touched instruction as visited, but not following the program order; rather, every time an artificial jump is encountered, we proceed to analyse the function starting from the instruction to which the jump jumps. In parallel, the algorithm builds a new double linked list of instructions, putting the instruction in the order in which they are analysed, ignoring the artificial jump instruction.

To prevent the engine from building multiple copies of an instruction (this can happen if the engine analyses the same instruction multiple times), the statements marked as visited are ignored.

If at the end of the process not all the instructions have been visited, the algorithm fails and the function is left as it was.

Broadly speaking, the resequencer algorithm is as follows:

Algorithm 3: Resequencer algorithm

```
input  : A set of functions
output: Number of functions which have been reordered
count ← 0
for function : functions do
    new_ins ← []
    n_ins ← 0
    instruction ← get_first_instruction(function)
    while n_ins ≤ number_of_instructions(function) &
     instruction ≠ NULL do
        if is_visited(instruction) then
            instruction ← next_instruction(instruction)
            go to while
        end
        mark_as_visited(instruction)
        n_ins ← n_ins + 1
        if is_artificial_jmp(instruction) then
            instruction ← target(instruction)
            mark_as_visited(instruction)
            n_ins ← n_ins + 1
        end
        add(new_ins, instruction)
        instruction ← next_instruction(instruction)
    end
    if n_ins = number_of_instructions(function) then
        change_instructions(function, new_ins)
        count ← count + 1
    end
end
return count
```

4.4 Permutator

Another important phase of the metamorphic process is the permutation of the executable code. It is a reorganisation of the code, keeping the execution flow unchanged. For example, the first instruction in the original executable code will not be the first in the new version of the code but will always be executed before all other instructions in the original version.

The idea behind this phase is to add more randomness in the new version of the executable, changing the order in which the instructions are expanded by the grammar engine.

As for the other phases, the permutation can also be applied only to functions that do not use a jump table, and cannot be applied to functions that are added by the gcc compiler. Also, the permutator works only with functions, so if there are no functions in the executable, the entire code will be considered as a unique function.

4.4.1 Permutation algorithm

The algorithm used is very simple and is applied to one function at a time, in this way:

1. Assign to each instruction an index that goes from 0 to number of instructions in the function - 1, organising these index in an ascending sorted array;
2. Then compute a permutation of this array, using a RNG;
3. Build an empty list of instructions;
4. Let us consider the resulting array and look at one element at a time:
    - For each element, build an artificial jmp instruction that points to the instruction that should be in this position according to the initial order, for example if the element taken into consideration

is the second of the array, then the jmp must jump to the second instruction in the initial order;

- Add this instruction to the new list;
- Let us call the value of the element i, then copy the i-th instruction of the initial order in the new list;

This algorithm adds a jmp instruction for each instruction in the original version. This means that after the permutator is applied, the size of the function doubles.

Let us see the result of the algorithm applied to this function:

```
0000000000001690 <foo_3p>:
    1690:       55                      push   rbp
    1691:       48 89 e5                mov    rbp,rsp
    1694:       48 83 ec 10             sub    rsp,0x10
    1698:       48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
    169c:       48 8d 05 cd 0b 00 00    lea    rax,[rip+0xbcd]
    16a3:       48 89 c7                mov    rdi,rax
    16a6:       e8 85 f9 ff ff          call   1030 <puts@plt>
    16ab:       90                      nop
    16ac:       c9                      leave
    16ad:       c3                      ret
```

```
000000000000183a <foo_3p>:
    183a:       eb 2d                   jmp    1869 <foo_3p+0x2f>
    183c:       90                      nop
    183d:       eb 0b                   jmp    184a <foo_3p+0x10>
    183f:       48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
    1843:       eb 0f                   jmp    1854 <foo_3p+0x1a>
    1845:       48 89 c7                mov    rdi,rax
    1848:       eb 03                   jmp    184d <foo_3p+0x13>
    184a:       c9                      leave
    184b:       eb 10                   jmp    185d <foo_3p+0x23>
    184d:       e8 de f7 ff ff          call   1030 <puts@plt>
    1852:       eb e8                   jmp    183c <foo_3p+0x2>
    1854:       48 8d 05 15 0a 00 00    lea    rax,[rip+0xa15]
    185b:       eb e8                   jmp    1845 <foo_3p+0xb>
    185d:       c3                      ret
    185e:       48 89 e5                mov    rbp,rsp
    1861:       eb 00                   jmp    1863 <foo_3p+0x29>
    1863:       48 83 ec 10             sub    rsp,0x10
    1867:       eb d6                   jmp    183f <foo_3p+0x5>
    1869:       55                      push   rbp
    186a:       eb f2                   jmp    185e <foo_3p+0x24>
```

Figure 4.3: Outcome of the permutator

4.4.2 How to build an "artificial" jump instruction

The algorithm followed is very simple to understand, but how can we build an artificial jump instruction that has exactly the instruction that we want? We cannot use the original_address field of struct 4.10 because the permutation has moved instructions and then they are not at the same address. There are two possible ways to solve this problem: one might be to recompute the original_address field, but this requires a computation on the entire binary code, which can be very expansive for large size ELF. Another solution, which is the one used in this work, is to build a *dummy* instruction, that is, the engine build a jump instruction which looks like a jump but has a fixed destination address, e.g. jmp 0x2b8, in this way we solve the problem to fill correctly the fields of struct 4.10, in such a way that the instruction is seen by the engine as a jmp. Clearly, this jump instruction is not correct, because it is not linked to the right instruction (as explained in 4.4), but this is not an issue, since we can link it to the correct instruction using the field jump_target of struct 4.10 and then entrust the task of writing the correct jump instruction to the part of the code responsible for fixing the metadata and rewriting the ELF.

4.5 Expander

The expander is the part of the code responsible for the transformation of the instruction. It involves the usage of generative grammars to generate a transformation from one instruction to another. As for the other part of the engine, the expander can only be applied to functions that do not contain a jump table, for the issue explained in 4.1.2.

The basic idea behind the expander is to apply the generative grammar algorithm to one instruction at a time, in order to build a decision tree; then the engine parses this tree and constructs the new instructions which replace the original one.

4.5.1 Grammar generation process

In this section, it is explained in detail how the generative grammar engine works.

The engine takes as input a .y file (as explained in 3.3) and parses it, looking for rules and symbols. A symbol is classified as non-terminal (NT) if there is at least one rule associated to this symbol, while a symbol is classified as terminal if there are no rules associated with it. In reference to the .y file used in this work (see 3.5) the non-terminal symbols are written in lower case, while the terminal ones are written in upper case.

All of these operations are done in an initialisation phase; subsequently the engine can be used for multiple generation using the grow algorithm. This algorithm takes as input a starting symbol and gives as output a decision tree, which is structured in this way: children of a node are the right-side symbols of the applied rule, and the leaves are all terminal symbols. The leaves of the decision tree are the instructions in which the starting one has to be transformed. Clearly, the grammar engine has no knowledge about the fact that it is working on instruction, but is a simple generative grammar which generates sentences that are syntactically correct with respect to the defined language but not necessarily semantically correct. This means that there is a possibility that the final transformation cannot be applied.

The grow algorithm works in this way:

Algorithm 4: Grow algorithm

```
input  : A symbol named starting_symbol
output: A decision tree
stack ← emtpy_stack()
decision_tree ← emtpy_tree()
push_stack(starting_symbol, stack)
while stack_is_not_empty(stack) do
    symbol ← pop(stack)
    if is_NT(symbol) then
        rule ← get_random_rule(symbol)
        push_rule_on_stack(stack,rule)
        print_rule_on_tree(rule, decision_tree)
    end
    else
        print_symbol_on_tree(symbol, decision_tree)
    end
end
return decision_tree
```

The function push_rule_on_stack is structured in such a way that the first symbol on the right side of the rule will be the top of the stack. The function get_random_rule makes use of a random number generator to select a rule. During the initialisation phase, it is possible to specify a fixed seed for the RNG, or decide to use the current timestamp as the seed.

4.5.2 Parsing the decision tree

The decision tree produced by the generative grammar is then parsed by the metamorphic engine to build the new instructions. In fact, the grammar limits itself to deciding which instructions should be generated while remaining at the abstract token level, i.e. determining only the mnemonic of the instructions.

To correctly build the new instructions, the engine parses the decision tree starting from the root node generating all the intermediate instructions up to the final ones. To do that, the engine must be able to process each transformation rule present in the grammar, and this is done by defining a proper function for each rule. For example, since the grammar contains a rule:

push -> sub\mov

then the engine must contain a function that processes this rule.

Notice that the engine does not have to know how to process any transformations which could be generated by the grammar, but only the single rules which can be applied. As an example, supposed that the grammar has generated a transformation from push to sub\add\sub\mov, the engine does not know how to generate these four instructions from a push statement, but it is able to process each single step of the decision tree, and in this way it will generate the final instructions. The algorithm used to parse the decision tree is as follows:

Algorithm 5: Decision tree parsing

```
input  : A decision tree called decision_tree
output: A set of instructions which have to be added in the
        executable
stack ← empty_stack()
instructions ← empty_list()
push_symbol(decision_tree, stack)
while stack_is_not_empty(stack) do
    token ← pop_stack(stack)
    if is_terminal(token) then
        put_instruction(instructions,token.instruction)
    end
    else
        children ← token.children
        res ← process_rule(token, children)
        if is_error(res) then
            return empty_list()
        end
        push_symbols(children)
    end
end
return instructions
```

The functions push_symbol and push_symbols push the symbols given on the stack, in the case of push_symbols, the symbols are pushed "from the right to the left", this means that if the children array is the following:

The function process_rule is explained in more detail in Section 4.5.3. The put_instruction function instead is a simple function that appends the instruction in the token to a list of instructions, this list will be returned by

the algorithm and then will be processed in order to add the instructions to the global list of instructions of the executable. In this way, the instructions are added only if all the decision tree has been correctly parsed, in order to avoid spurious changes.

It is important that the parsing algorithm generates the instructions in the same order as they appear in the right-hand side of the overall transformation. As an example, suppose that the transformation is push -> sub\add\mov, then the algorithm must generate the sub instruction first, then the add and finally the mov statement. To do that, the algorithm goes in depth, always following the child furthest to the left among those not yet visited, until it reaches a leaf, then goes back and continues the descent in depth, always following the child furthest to the left. Since the leftmost statement of the transformation also corresponds to the leftmost leaf in the tree, this ensures that the statements will be generated in the order expected by the grammar. The idea of using a support stack was born precisely to provide the algorithm with the behaviour described above.

4.5.3 Transformation algorithms

The engine must contain a transformation algorithm for each rule in the grammar. This algorithm is responsible for the generation of the resulting instruction(s). This means that there exists a function for each rule in the grammar and that there is a function that is responsible for selecting the correct transformation algorithm.

All of these functions have the same structure, which can be summarised in these four steps:

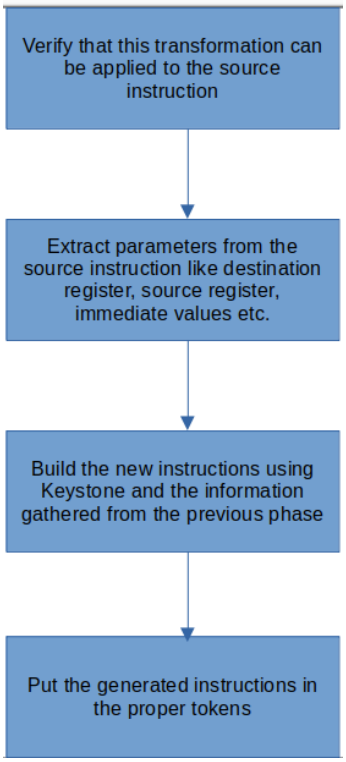


Figure 4.4: Basic structure of a transformation algorithm

The function may fail in any of these phases, so also if an instruction passes all the checks done in the first phase, it may not be transformed because it fails the second or third phase (this may happen in the case of a malformed or corrupted instruction).

The verification phase includes checks on instruction operands and instruction opcodes. The type of controls depends on the transformation, as an example: for the rule add -> add\sub the checks that must be done are: the original instruction:

- is using an immediate value as operand;
- is not using a displacement value (e.g. the instruction is not in the form add [rax+4],0x4).

The function process_rule used in 5 is a simple function that decides which transformation function to perform based on the tokens given as input. If the selected transformation function returns without errors, then the generated instructions are present in the tokens given as the second parameter of the

function.

4.5.4 How to add a new instruction

The last thing that remains to be analysed is how to add a new instruction to the ELF. This type of operation is basically an insert into a global linked list of instructions that will be used in the *rewrite* phase to update the binary code of the ELF. Since we are talking about instructions, we also need to update properly a set of metadata of the function of the instruction, like the size of the function and its final address, as well as the pointer to the first instruction of the function.

Unlike the function that removes an instruction, this cannot fail. In fact, if the function is asked to insert an instruction at a position whose index is greater than the size of the list, then the instruction is inserted at the end of the list.

4.5.5 Expanding the ELF

In summary, to carry out the expansion phase, the engine uses a generative grammar which is applied to one instruction at a time. The function which made the expansion of the entire ELF is invoked for each function of the ELF, then a loop on all the instructions of the function invokes the grow algorithm (see algorithm 4) on each instruction, then the resulting decision tree is parsed, and the global list of instructions is modified by the algorithm 5. The function returns the number of instructions for which the algorithm 5 does not fail. Since the parsing algorithm can fail if the transformation generated by the grammar is not feasible (this means that there was a failure in the validation phase of the selected transformation algorithm), the engine falls on a dummy transformation in which it tries to add some garbage instructions, leaving unchanged the original instruction. If also plan B fails, then the instruction cannot be transformed.

The described algorithm is the following:

Algorithm 6: Expanding the ELF

```
input  : A function
output: Number of instruction which has been changed
changed_ins ← 0
for instruction in function.instructions do
    token ← tokenize(instruction)
    tree ← grow(token)
    res ← parse_tree(tree)
    if is_error(res) then
        token ← nop_token()
        tree ← grow(token)
        res ← parse_tree(tree)
        if is_success(res then
            changed_ins ← changed_ins + 1
        end
    end
    else
        changed_ins ← changed_ins + 1
    end
end
return changed_ins
```

In this algorithm, the parse_tree function includes both the algorithm 5 and an algorithm which adds the generated instructions in the global list.

4.6 Rewrite ELF

In this section, we explain in detail the methodology followed to rewrite the ELF content. When changing the binary code of an ELF, you need to pay attention to how the ELF metadata should be changed. In fact a small change in the .text section could lead to big changes in the entire ELF, because the change made can for example move the entry point, or move the start address of the sections below, which may not even be of type SHT_PROGBITS.

So, the engine needs to update all of the following metadata:

- Program Header Table and its entries;
- Section Header Table and its entries;
- Executable Header;
- Content of the relocation tables;
- Content of the dynamic table;
- Symbol table;

Segments, which are described by the entries of a Program Header Table, must be correctly updated; otherwise, you may experience segmentation fault error when running the modified executable. This happens, for example, if the segment containing the code section is not correctly expanded, and consequently part of the code could be loaded into a segment without execution permissions. Therefore, when the processor tries to load an instruction found in such a segment into the register containing the instruction pointer, it will cause a segmentation fault.

Relocation and dynamic tables must be modified because otherwise the code will not know where the global variables used by the init and fini function are, and then the executable will not be able to start up or terminate in a correct way. This could happen if the .text section grows too much.

4.6.1 How to update all metadata

Let's see how to update all metadata of the executable to build a correct version of it.

Section header refactoring

To adjust the metadata of section header table entries, the engine needs to compute the new offset of the data of all sections and their new size. This is done sequentially starting from the first section and using the results of the previous iteration as input to the next one. The idea is that if the data of a section have been moved to another address, then this is due to some changes in the previous data block. Notice that for the previous data block we mean the set of bytes preceding the one under analysis as established by the virtual addresses present in the original ELF. The size of the data block is computed in this way:

- If the data block is of type SHT_PROGBITS, then the size is computed as the sum of the instructions contained;
- Otherwise the data block size is left unchanged.

Furthermore, because the section header may require an alignment for the content of the section, we need to check if the next data block is correctly aligned, otherwise we have to add some padding bytes in order to move the next data block to the correct address (the one which respects the alignment). If the data block is of type SHT_PROGBITS, then the byte used for padding is 0x90 (opcode of a nop instruction), otherwise the byte used is 0x0 (NULL byte).

Once the data blocks are in place, it is easy to adjust the section header metadata.

Program header refactoring

In order to correctly update the program header table entries, it is necessary to correctly position the segments. To do this, you create a section-segment map (i.e. an association between a segment and the sections it contains) during the initial loading phase of the ELF. The goal is that at the end of the engine execution, the resulting ELF has exactly the same original section segment mapping, so as to avoid any future segmentation fault problems that may occur during execution.

To do this, we use the initial mapping to know which should be the first section contained by the segment and the last one. The first provides us with the address at which to position the segment (which will be the starting address of the data block of that section), while the last allows us to calculate the size of the segment, which is calculated as the difference between the address at which the data block of the last section ends and the starting address of the segment.

The mapping between segment and sections is carried out in the following way: if the data block of a section falls completely or partially into the segment (this means that the segment includes some of the bytes of the data block), then the section is associated with that segment. A section can be associated with multiple segments; this can happen if the segments are of the same type (they have the same permissions).

The segments must be aligned to the size of the page. This is important to avoid the fact that two different segments can occupy the same page. In fact, in this case, it may happen that a segmentation fault error occurs if the two segments have different access permissions.

Adjusting references between instructions

As explained previously, it is necessary to update the references between instructions (see 3.1.1). When doing this, you need to take into account that if the original jump instruction was a jmp short, then it is possible that the

instruction will need to be changed to a jmp long if the new relative offset exceeds the size allowed for a jmp short.

Basically in a jump short type instruction you only have 1 byte available to indicate the rip relative offset, while in a jmp long instruction you have 4 bytes available; consequently, if the new offset cannot be represented in sign using 1 only bytes, then the instruction must be changed. In this case, it will be necessary to update the section header and program header again, as the variation in the size of the changed instruction could lead to changes in the addresses of the data blocks and their size.

4.6.2 Write byte on the file

To rewrite the file it was necessary to write a special algorithm which takes care of converting the intermediate representation of the ELF, appropriately modified, into bytes according to the format expected from an ELF, in order to generate a new version of the perfectly functional executable.

Unfortunately, the support library used for reading the ELF (libelf [20]) does not have adequate functionality for this purpose, so a specific algorithm has been developed which, taking into account the endianess of the machine and the type of processor (32 bit or 64 bit), is able to rewrite the ELF in the expected format. Therefore, a special function was developed to rewrite:

- the executable header;
- the section header table;
- the content of each section;
- the program header table.

5. Experimental Evaluation

For the experimental evaluation of the metamorphic engine created, three different studies were carried out:

- A study on the variation of the detection rate for a malware sample as the transformation steps vary;
- A study on the variation of the performance for two benchmark as the transformation steps vary;
- A study on the persistence of byte blocks following the application of the engine as the block size varies.

For all of these studies, Python scripts were developed as tools to support data collection. The data were then graphed using a specific Python script that uses the matplotlib library [35].

5.0.1 Study on the variation of the detection rate of malware

In this study a sample of 70 malware has been used. All the malware used have been studied and classified as malicious software by external actors, and they are ELF format files. Virus Total [42] was used as a tool to analyse malware and collect data related to the detection rate.

The study comprises different phases. At first, the malware detection rate and performance for each individual antivirus were collected for the original malware sample. Then the engine has been used to obtain three new different

versions of the original malware, applying the engine on each ELF file in the sample. The different versions have been constructed in this way: the engine has been applied on the original version of the malware, then the result has been used as input to the next iteration of the engine, obtaining in this way a second version of the malware, and then in the same way the third version has been obtained.

After the three new versions of the malware sample were obtained, we moved on to collecting data related to the detection rate and antivirus performance on each new version of the sample.

This is a boxplot that graphically reports the results obtained:

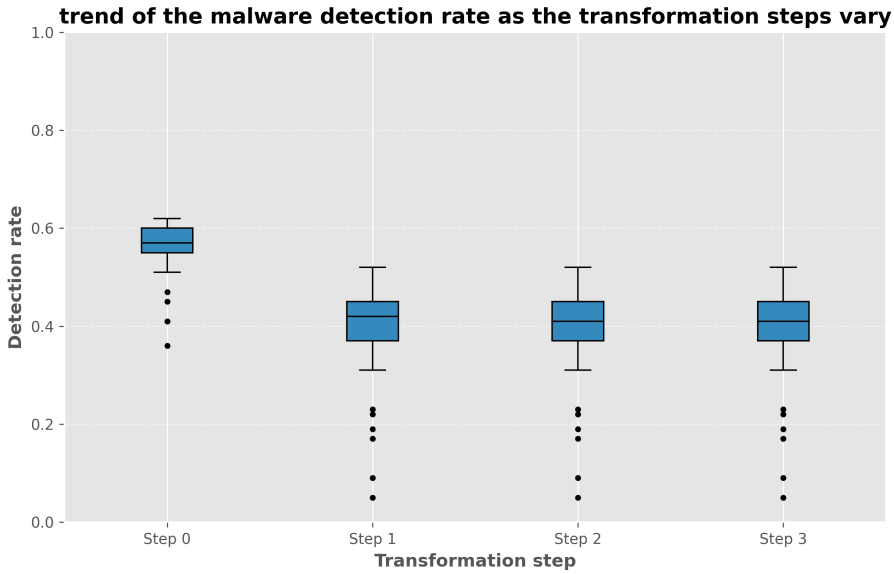


Figure 5.1: Trend of the malware detection rate

As we can see, there is a consistent decrease in the malware detection rate from step 0 to step 1, this is because by applying the engine we are adding an additional layer of obfuscation, while the detection rate remains almost constant between step 1 and step 2, as well as between step 2 and step 3, this is due to the initial shrinking and reordering phases carried out by the engine. In fact, these two phases cancel all the changes made in the previous iteration, and so in steps 2 and 3 we are not really adding an additional layer of obfuscation compared to step 1.

This behaviour is also due to the fact that the same generative grammar was used in each transformation step, even if the random generator was initialised with different seeds. In the case in which the generative grammar is changed (for example, by using a metagrammar), we can expect a variation in the detection rate also in steps 2 and 3, since by varying the grammar the basic rules vary, and therefore also the set of instructions that can be modified and the transformations that can be applied.

Together with the study on the variation in the malware detection rate, a study was carried out on the variation in the performance of antiviruses, taking into consideration the 5 best (i.e. the antiviruses with the least loss of performance) and the 5 worst (i.e. those with the greatest loss of performance).

There are two bar plots showing the results obtained:

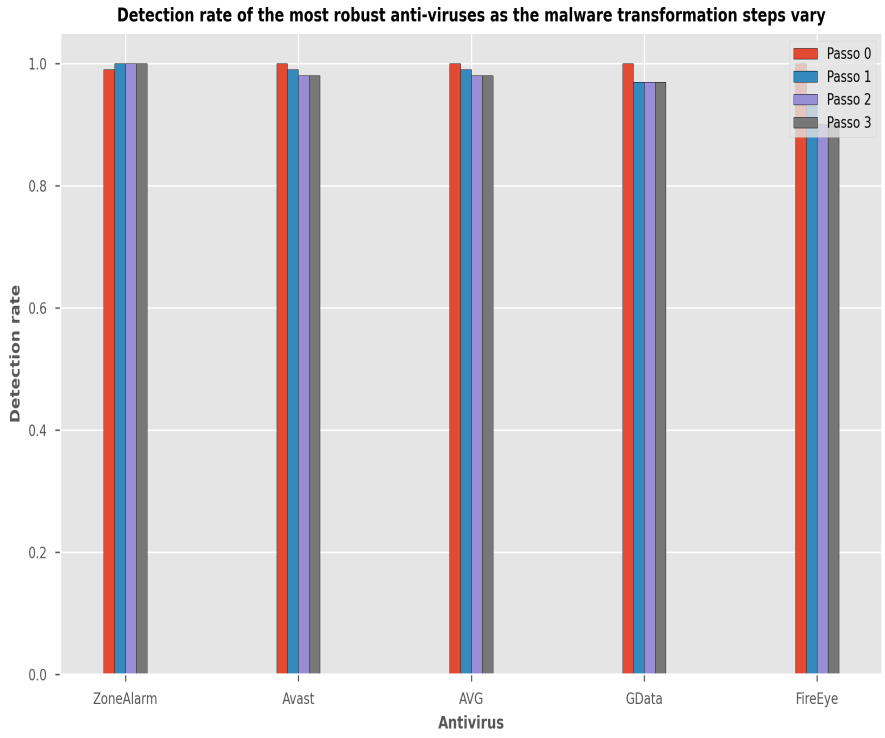


Figure 5.2: Detection rate of the best 5 antivirus

As shown in the 5.2 graph, there is an antivirus named ZoneAlarm that has a better detection rate on malware modified by the engine. This could be due to some particular indicator used by this single antivirus, probably linked

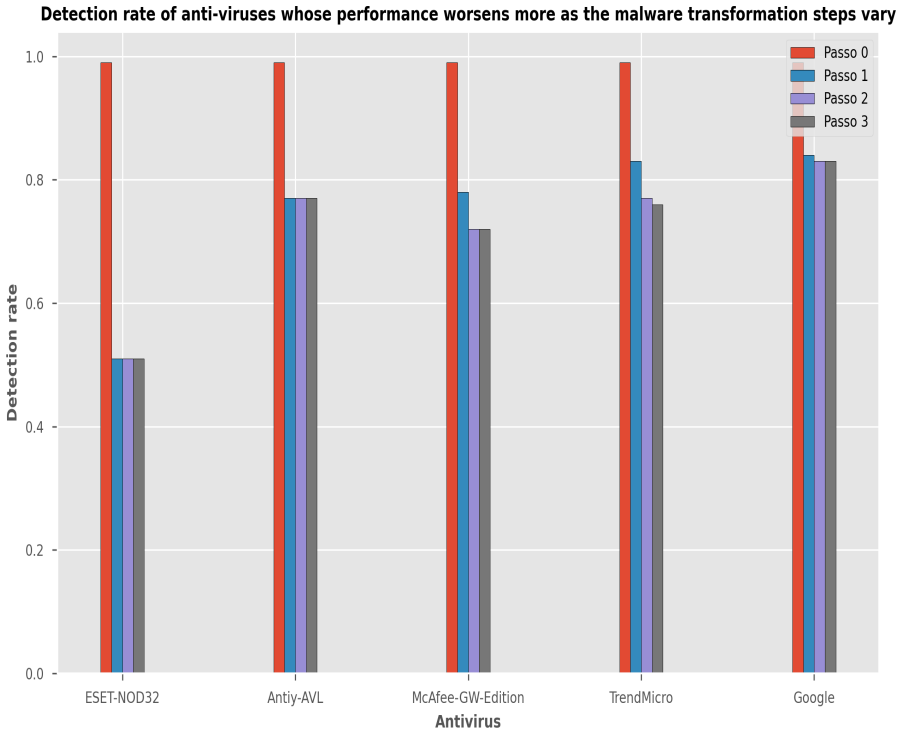


Figure 5.3: Detection rate of the worst 5 antivirus

to the fact that the malware has been metamorphically transformed, while for the other four antiviruses a decrease, albeit slight, in the detection rate is observed.

In the graph 5.3, we can observe a significant drop in the performance of the five antiviruses. For all of these 5 antiviruses the initial detection rate is almost one, while the detection rate at step 1 and the following is less than 0.8; furthermore, we can see how the performances between step 1 and 2 and between steps 2 and 3 change slightly, consistently with what is seen in the graph 5.1.

These results show that the engine performs very well as an obfuscation technique, as it manages to confuse most antiviruses on the outcome of the analysis carried out on a set of software known to be malware, in some cases reducing the detection rate even below 0.4%

5.0.2 Study on the variation of the performance for two benchmark

The transformations applied by the engine can worsen, even drastically, the performance of CPU-intensive software (for example, benchmarks), since they transform an instruction into a set of equivalent instructions which could, however, slow down the execution pipeline. Furthermore, if some particularly efficient instructions are modified by the engine, then the original advantage of using that type of instruction is lost.

For this reason, we decided to carry out an experiment aimed at verifying that the execution times of two benchmarks do not worsen drastically following the application of the engine.

The two benchmarks used are named pagerank [7] and mooncalc [33].

The experiment was carried out according to this modality: for each benchmark, starting from the original version, 10 different versions were created, using a different seed for the random number generator. Each of these 10 versions was subjected to a further 9 transformation steps. In general, the experiment was carried out on a total of 10 transformation steps of the original file, but each step consists of ten different versions of the software. This was done to reduce the random effect on execution time variation due to the nature of the engine. The times used in the study were then taken as an average over the times of the individual versions (the ten in each step), and the times of the individual versions were calculated as an average over five consecutive runs, to have a more accurate measurement, these are execution times that can be influenced by the CPU workload. Execution times for the original versions of the software were taken as the average over 10 program runs.

The physical machine used for the measurements was reserved exclusively for running the benchmarks during the experiment. Each run of the software was carried out sequentially and not in parallel, this means that if a version of the software was running, then this was the only (or almost) only user-level software running.

Python scripts have been developed as support tools for the automated generation of the different versions of the benchmark using the engine and always assigning a different seed, and for the collection of average execution times as explained previously.

Let us see the graphs that show the results obtained from this analysis:

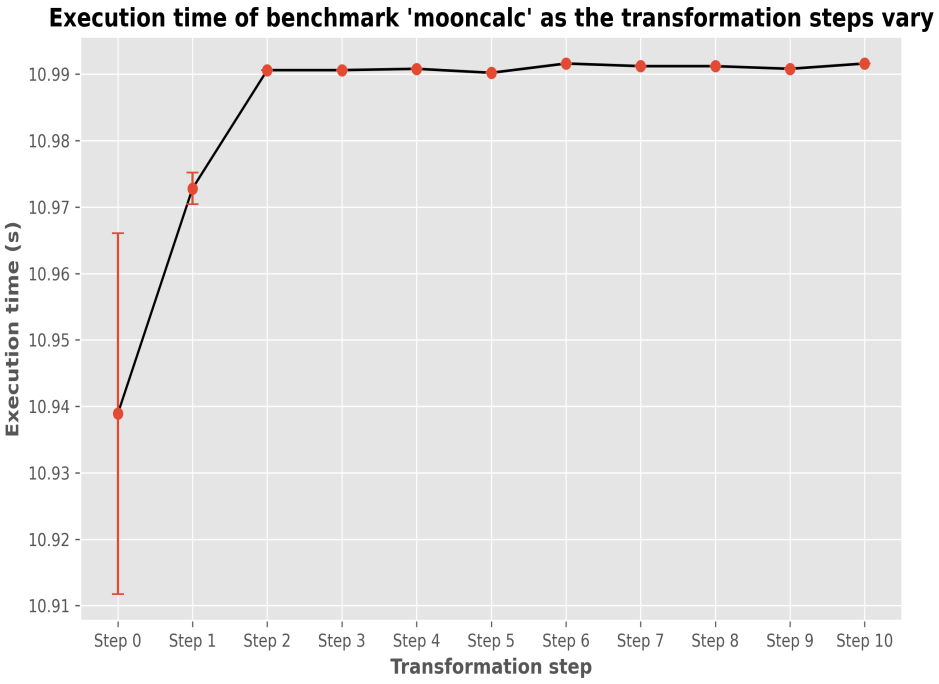


Figure 5.4: Increase in the performance of mooncalc

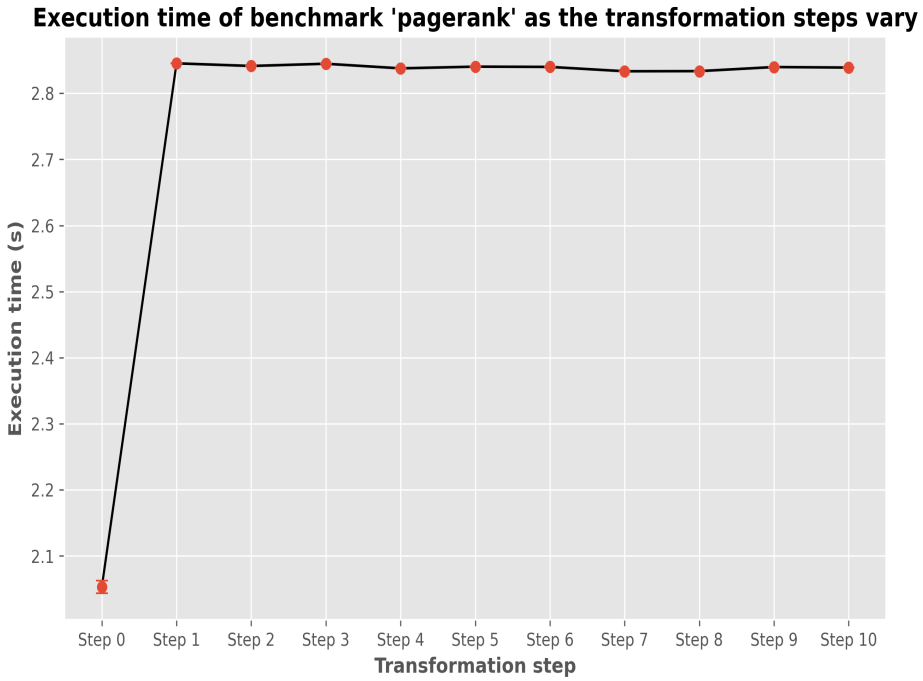


Figure 5.5: Increase in the performance of pagerank

The red line in the plots indicates the variance.

As can be seen from the graphs, the increase in execution times occurs between the original version and the first two transformation steps, whereas in the subsequent steps the times remain roughly constant. This is due to the shrinker and reordering phases, which remove, in part, the changes made to the executable in the previous iteration. In the absence of these phases, in fact, one could expect a linear increase in execution times as the size of the software, and therefore the number of instructions necessary to carry out a single operation, grows with each iteration.

The maximum increase that can be observed is, however, acceptable; in fact, it is equal to 0.05 in the case of mooncalc and 0.7 in the case of pagerank.

Thanks to these results, it can be concluded that experimentally the engine causes a visible increase in the execution times of a software, but this is still minimal. It can be concluded that after appropriate preliminary verification, it is possible to use the engine to transform hard or soft real-time software, provided that the execution times after a transformation step do not exceed the assigned deadlines.

5.0.3 Analysis of metamorphism-resilient byte blocks

The metamorphic engine behaves as well as the fewer the number of byte blocks that remain present following a transformation process. In fact, if a new version of the software produced by the engine retains blocks of bytes of non-trivial size (e.g. larger than 60 bytes) present in the original version, then this block of bytes can be used to build a signature of the executable, and consequently be able to reconnect the engine's output to its original version. For example, let us suppose we have three files in ELF format, called A, B and C for simplicity. All three files are subjected to a transformation step, now if the engine leaves some blocks of bytes unchanged, these could be used to build a distinctive signature for A, B, and C and consequently be able to understand which of the modified files was originally A, which B, and which C.

Clearly we don't want this to happen, so it was decided to test the engine from this aspect. The test was carried out on the two benchmarks and on

some of the malware. For computational simplicity, it was decided to consider exclusively 2 transformation steps. The idea of the experiment is to verify, for a variable block size, how many byte blocks of that size survive a transformation step.

The following graphs represent the results for the two benchmarks:

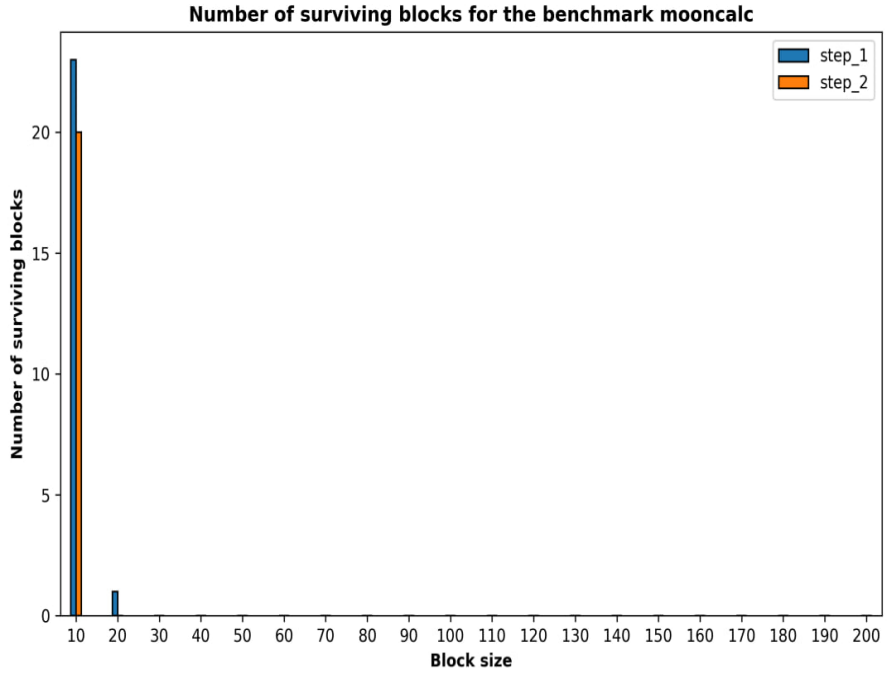


Figure 5.6: Metamorphism-resilient byte blocks for mooncalc

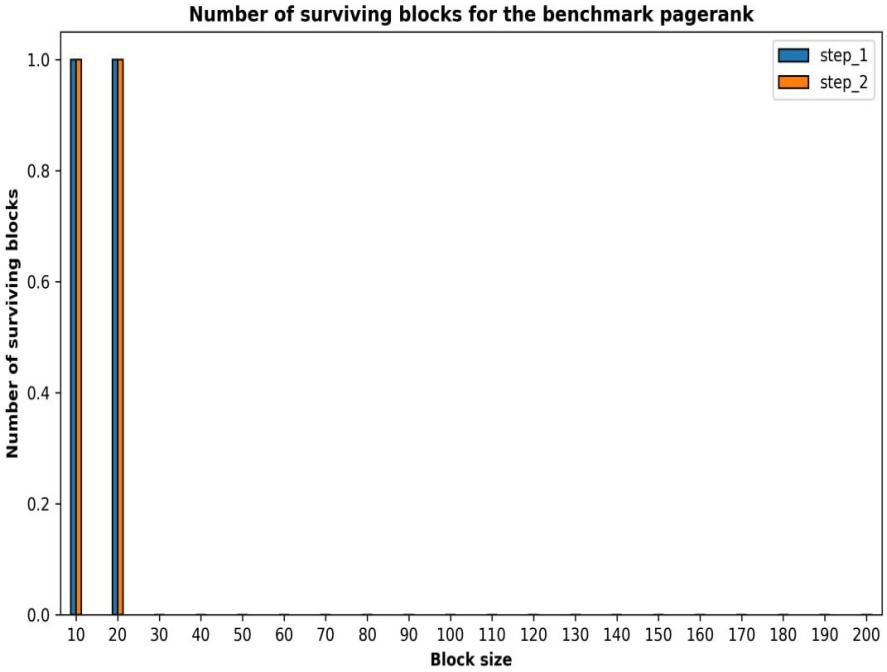


Figure 5.7: Metamorphism-resilient byte blocks for pagerank

The results are very interesting; in fact, as can be seen, there are blocks

of small dimensions that survive, but for dimensions useful for calculating a signature there are no blocks that remain unchanged between one transformation step and another. This result is mainly due to the permutation activity, which, having been carried out by moving every single instruction of the function, allows recurring blocks of bytes in the original executable to be broken with a high probability.

The survivent blocks are small and cannot be used to construct a digital signature.

The results for the test performed on malware are even more satisfactory:

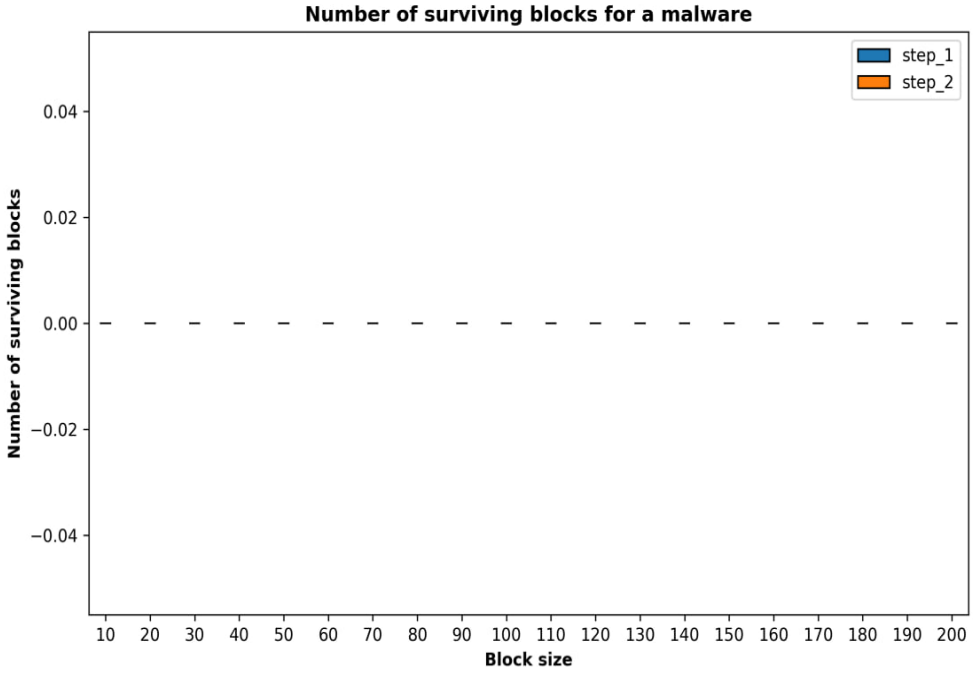


Figure 5.8: Metamorphism-resilient byte blocks for malware

In fact, in this case there are no blocks that survive the application of the engine! These same results appeared for 4 different malware subjected to the same type of testing.

We can therefore conclude that the engine is well designed to avoid leaving possible spots untouched, which can be used as a signature, but rather the new versions produced of the software are completely different from the point of view of byte code compared to the version original, and therefore there is

nothing that directly connects them.

In particular, the tests were conducted considering exclusively the content of the SHT_PROGBITS type sections. If you also wanted to consider the content of the other sections, then it might make sense to integrate the metamorphic engine with a dedicated encrypter for encryption of the data sections.

6. Conclusions and future works

The objective of this thesis was to present an innovative way to protect Software Intellectual Property from attacks based on reverse engineering of the code. A metamorphic engine has been introduced that allows the code to self-mutate, transforming the instructions that make up the binary code of an executable file into other semantically equivalent instructions.

To advance the state-of-the-art, it was decided to integrate the engine with a generative grammar capable of determining in a non-deterministic manner the transformations to be applied to the code, starting from a set of rules defined for each possible managed instruction. The interesting aspect of using a generative grammar is that it can be modified between one iteration and another without the need to recompile the engine, thus allowing it to have a different set of basic rules each time, making this way the versions of the executable generated by the engine are very different from each other.

Although the grammar used in this work is very simple, the results obtained from the experimental evaluation are very satisfactory. The engine is able to significantly reduce the detection rate of the antiviruses used by Virus Total; this is a sign that the obfuscation technique works well. Furthermore, thanks to experiments on the execution times of some benchmarks, it was possible to verify that the engine does not significantly worsen the performance, in terms of execution time, of the executable on which it is applied.

To improve the work done, we can think of integrating the engine with a metagrammar. A metagrammar is a generative grammar responsible for producing new generative grammars, which in turn are responsible for generating transformations for binary code instructions.

The use of a metagrammar allows you to implement what was described above, that is, to provide a new generative grammar to the engine after each iteration, automatically, as this generative grammar will be produced in a non-deterministic manner by means of the application of some of the rules of the metagrammar.

This is an image showing what the entire engine will look like, with some metagrammar added:

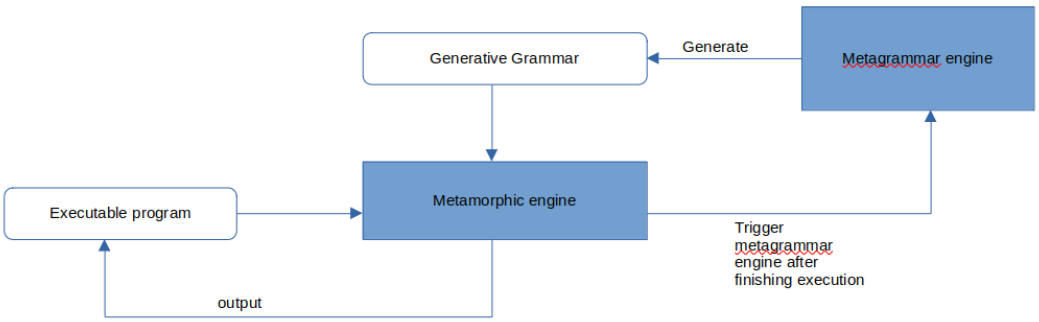


Figure 6.1: Engine scheme with some metagrammar added

In this way, the metagrammatics is called into question at the end of each iteration, to thus generate a new generative grammar, which will then be the input for the next iteration of the engine.

However, the fact that the input generative grammar is different at each iteration opens up a significant implementation problem: the engine uses the generative grammar to derive the table of inverse transformations, as widely discussed in 4.2, but since the generative grammar changes at each iteration, the engine is not able to build the table of inverse transformations related to the previous iteration. In fact, at each run, the engine must shrink according to the rules applied in the previous iteration.

To overcome this problem, one could think of storing the table of inverse transformations somewhere in memory, ensuring that it is not accessible to a possible attacker, for example, by exploiting encryption mechanisms.

This thesis work can be seen as a starting point for the development of new detection algorithms to be integrated into antivirus, in order to strengthen such software against malware that uses metamorphic engines. We could start from

the proposed methodology to try to identify traits common to all metamorphic engines, and in this way the antivirus would be able, through an appropriate dynamic analysis (e.g. using a sandbox), to identify the possible presence of a metamorphic engine in a malware.

A. Executable and linkable format

Introduction to ELF

The ELF (Executable and Linkable Format) is a common file format used for executables, shared libraries, and object code in Unix-like operating systems. It is designed to be flexible, extensible, and platform-independent, making it a fundamental component of the Unix software ecosystem. This Appendix provides an overview of the ELF file format and its structure.

ELF File Types

ELF files can serve different purposes and are categorised into three main types:

- Executable Files (ET_EXEC): These files contain machine code and data that can be directly executed by the operating system or a program loader.
- Shared Object Files (ET_DYN or ET_SO): These files contain code and data that can be linked to other programs at runtime, allowing multiple programs to share the same code.
- Object Files (ET_REL or ET_OBJ): These files contain relocatable code and data that can be combined with other object files to create an executable or shared object.

ELF Header

The ELF file format begins with an ELF header, which provides essential information about the file. The header is located at the beginning of the file and has the following structure:

```
    struct Elf32_Ehdr {     // For 32-bit ELF files
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
    Elf32_Half    e_shstrndx;
};
```

Key fields in the ELF header include:

- e_type: Specifies the file type (executable, shared object, or object file);
- e_machine: Indicates the target architecture (e.g., x86, ARM);
- e_entry: The virtual address where program execution begins;
- e_phoff and e_phnum: Describe the program header table's offset and number of entries;
- e_shoff and e_shnum: Describe the section header table's offset and number of entries;
- e_flags: Contains processor-specific flags;
- e_ehsize, e_phentsize, e_shentsize: Provide the header's size, program header's size, and section header's size, respectively.

Program Header Table

The program header table is present in executable and shared object files (not in object files). It describes how the segments of the file should be loaded into memory. Each entry in the table has the following structure:

```
1     struct Elf32_Phdr {     // For 32-bit ELF files
2     Elf32_Word p_type;
3     Elf32_Off  p_offset;
4     Elf32_Addr p_vaddr;
5     Elf32_Addr p_paddr;
6     Elf32_Word p_filesz;
7     Elf32_Word p_memsz;
8     Elf32_Word p_flags;
9     Elf32_Word p_align;
10 };
```

Key fields in the program header include:

- p_type: Describes the type of segment (e.g., code, data, dynamic linking);
- p_offset: The offset of the segment in the file;
- p_vaddr and p_paddr: Virtual and physical addresses of the segment in memory;
- p_filesz and p_memsz: Size in the file and in memory;
- p_flags: Permissions (read, write, execute) for the segment;
- p_align: The alignment of the segment in memory.

Section Header Table

The Section Header Table (SHT) is a critical component of the ELF (Executable and Linkable Format) file structure. It provides detailed information about the various sections present in the ELF file, such as executable code, data, symbol tables, and more. Understanding the Section Header Table is essential for analysing and working with ELF files.

Each entry in the Section Header Table has the following structure, which is defined in the ELF specification:

```
1     struct Elf32_Shdr {     // For 32-bit ELF files
2     Elf32_Word sh_name;       // Section name (index into section header
      string table)
3     Elf32_Word sh_type;       // Section type (e.g., SHT_PROGBITS, SHT_SYMTAB)
4     Elf32_Word sh_flags;      // Section attributes and permissions
5     Elf32_Addr sh_addr;       // Virtual address when loaded into memory
6     Elf32_Off  sh_offset;     // Offset in the file where the section starts
7     Elf32_Word sh_size;       // Size of the section in bytes
8     Elf32_Word sh_link;       // Section header index that holds related
      information
9     Elf32_Word sh_info;       // Extra information about the section
10    Elf32_Word sh_addralign;  // Required alignment when loaded into memory
11    Elf32_Word sh_entsize;    // Size of each entry (if applicable)
12 };
```

Let us break down the key fields in a Section Header Entry:

- sh_name: An index into the section header string table (.shstrtab) that holds the name of the section. This allows for efficient storage of section names and reduces redundancy;
- sh_type: Specifies the type of the section, which determines its content and purpose. Common section types include:
  - SHT_PROGBITS: Contains program-specific data (e.g., code, data);
  - SHT_SYMTAB: Symbol table containing information about symbols used for linking;
  - SHT_STRTAB: String table for storing various strings (e.g., symbol names, section names);
  - SHT_DYNAMIC: Contains dynamic linking information
  - SHT_RELA and SHT_REL: Relocation information for linking;
- sh_flags: Defines section-specific attributes and permissions, often encoded as bit flags. Common flags include read (SHF_ALLOC), write (SHF_WRITE), and execute (SHF_EXECINSTR);

- sh_addr: Indicates the virtual memory address at which the section should be loaded when the ELF file is executed. This field is typically used for executable sections;
- sh_offset: Specifies the offset in the ELF file where the section's data begins. It points to the section's data within the file;

  sh_size: Represents the size of the section in bytes. It specifies how much space the section occupies in the ELF file;

  sh_link and sh_info: These fields are used for various purposes depending on the section type. For instance, in a symbol table (SHT_SYMTAB), sh_link may point to a string table, and sh_info may hold the index of the associated section;

  sh_addralign: Defines the required alignment of the section's data when loaded into memory. It specifies the boundary on which the section should be placed in memory;

  sh_entsize: This field is used for sections that contain fixed-size entries (e.g., relocation sections). It specifies the size of each entry in the section;

ELF files can contain a variety of sections, each serving a specific purpose. Some common sections include:

- .text: Contains executable code;
- .data and .bss: Hold initialized and uninitialized data, respectively;
- .rodata: Read-only data (constants);
- .symtab and .strtab: Symbol table and string table for linking and debugging;
- .rel and .rela: Relocation sections for dynamic linking;
- .dynamic: Contains dynamic linking information;
- .plt and .got: Used for Procedure Linkage Table (PLT) and Global Offset Table (GOT) in shared libraries;

B. Intel x86 Instruction Set Architecture (ISA)

The Intel x86 Instruction Set Architecture (ISA) is a widely used and well-documented architecture for microprocessors. It has a rich history dating back to the Intel 8086 processor, and it continues to evolve with modern x86_64 processors. This appendix provides an overview of some essential elements of the x86 ISA.

Register Names

The x86 ISA features a variety of registers, each with a specific purpose. Here are some commonly used registers in the x86 architecture:

1. **EAX**: Accumulator
2. **EBX**: Base register
3. **ECX**: Counter register
4. **EDX**: Data register
5. **ESI**: Source index
6. **EDI**: Destination index
7. **EBP**: Base pointer
8. **ESP**: Stack pointer

64-bit Extensions (x86_64 mode)

In 64-bit mode, the general-purpose registers are extended to 64 bits:

1. **RAX**: Accumulator
2. **RBX**: Base register
3. **RCX**: Counter register
4. **RDX**: Data register
5. **RSI**: Source index
6. **RDI**: Destination index
7. **RBP**: Base pointer
8. **RSP**: Stack pointer

Instruction Mnemonics

x86 instructions are represented using mnemonics, which are human-readable representations of the operations that a processor should perform. Some common mnemonics include:

- **MOV**: Move data from one location to another.
- **ADD**: Add two values.
- **SUB**: Subtract one value from another.
- **CMP**: Compare two values.
- **JMP**: Unconditional jump.
- **JE**: Jump if equal (used for conditional branching).
- **CALL**: Call a subroutine.
- **RET**: Return from a subroutine.
- **NOP**: No operation (useful for padding or alignment).

REX Byte

The REX byte, introduced in the x86_64 (64-bit) extension of the architecture, is used to extend the functionality of registers and operands. It is optional and appears before an instruction. The REX byte consists of four fields:

- **R**: Bit 2 is the R field, which extends the base register (e.g., EAX to RAX).
- **X**: Bit 1 is the X field, which extends the index register (e.g., ESI to RSI).
- **B**: Bit 0 is the B field, which extends the base register (e.g., EBX to RBX).
- **W**: Bit 3 is the W field, which specifies the 64-bit operand size when set (e.g., MOVQ for 64-bit data).

Example of a REX byte in assembly code:

```
48 89 E5  ; MOV RBP, RSP (64-bit mode)
```

Here, 48 is the REX byte, indicating 64-bit mode (W=1), and the instruction moves the value of the stack pointer (RSP) into the base pointer (RBP).

Bibliography

[1] James H Anderson and Yong-Jik Kim. 2001. GDB: The GNU Project Debugger. *Distributed Computing* 14 (2001), 17–29. https://doi.org/10.1007/PL00008923

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. *SIGPLAN Notices* 35 (2000), 1–12. https://doi.org/10.1145/358438.349303

[3] Giorgio Bernardinetti, Dimitri Di Cristofaro, and Giuseppe Bianchi. 2022. PEzoNG: Advanced packer for automated evasion on windows. *Journal of computer virology and hacking techniques* 18, 4 (Feb. 2022), 315–331. https://doi.org/10.1007/s11416-022-00417-2

[4] Jean-Marie Borello. 2011. *Étude du métamorphisme viral : modélisation, conception et détection.* Ph. D. Dissertation. Université Rennes 1, Rennes, France.

[5] Jean-Marie Borello and Ludovic Mé. 2008. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology* 4, 3 (Aug. 2008), 211–220. https://doi.org/10.1007/s11416-008-0084-2

[6] Pietro Borrello, Emilio Coppa, and Daniele Cono D'Elia. 2021. Hiding in the particles: When return-oriented programming meets program obfuscation. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '21).* IEEE, Piscataway, NJ, USA, 555–568. https://doi.org/10.1109/dsn48987.2021.00064

[7] Andrea Carrara. [n. d.]. pagerank: C implementation of Google's PageRank algorithm.

[8] A Celentano, S Crespi Reghizzi, P Della Vigna, C Ghezzi, G Granata, and F Savoretti. 1980. Compiler testing using a sentence generator. *Software: practice & experience* 10, 11 (Nov. 1980), 897–918. https://doi.org/10.1002/spe.4380101104

[9] Noam Chomsky. [n. d.]. *Three Models for the Description of Language*. Professional Group on Information Theory.

[10] N Chomsky. 1988. Generative grammar. *Studies in English linguistics and literature* (1988).

[11] Richard Feynman and Chapter Objectives. 2016. Ebnf: A notation to describe syntax. *Cited on* (2016), 10.

[12] Marco Gaudesi, Andrea Marcelli, Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. 2015. Malware Obfuscation through Evolutionary Packers. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*. Association for Computing Machinery, New York, NY, USA, 757–758. https://doi.org/10.1145/2739482.2764940

[13] Ghidra [n. d.]. https://github.com/NationalSecurityAgency/ghidra.

[14] James Gilliland. 2011. STIPULATION TO FINAL JUDGMENT UPON CONSENT AND PERMANENT INJUNCTION by Sony Computer Entertainment America LLC. , 34–34 pages.

[15] IDA pro [n. d.]. https://hex-rays.com/ida-pro/.

[16] Intel technologies. [n. d.]. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed: 2023-9-NA.

[17] Keystone. [n. d.]. Keystone. https://www.keystone-engine.org/. Accessed: 2023-9-NA.

[18] Min-Jae Kim, Jin-Young Lee, Hye-Young Chang, Seongje Cho, Yongsu Park, Minkyu Park, and Philip A Wilsey. 2010. Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 80–86.

[19] Johannes Kinder. 2012. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE '12)*. IEEE, Piscataway, NJ, USA, 61–70. https://doi.org/10.1109/wcre.2012.16

[20] Joseph Koshy. 2010. libelf by Example. *Web site: http://people. freebsd. org/jkoshy/download/libelf/article. html* (2010).

[21] Byoungyoung Lee, Yuna Kim, and Jong Kim. 2010. binOb+: a framework for potent and stealthy binary obfuscation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*. Association for Computing Machinery, New York, NY, USA, 271–281. https://doi.org/10.1145/1755688.1755722

[22] Robert B Lees and Noam Chomsky. 1957. Syntactic Structures. *Language* 33, 3 (July 1957), 375.

[23] John Levine. 2009. *flex & bison*. O'Reilly Media.

[24] Chi-Keung Luk, B C Ed, F C G Hi, E D Q Rs, A Tu, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05* 40 (2005), 190. https://doi.org/10.1145/1065010.1065034

[25] William Mahoney, Joseph Franco, Greg Hoff, and J Todd McDonald. 2018. Leave it to weaver. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop (SSPREW '18)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/3289239.3291459

[26] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2006. *System V Application Binary Interface AMD64 Architecture Processor Supplement.*

[27] Daniel D McCracken and Edwin D Reilly. 2003. Backus-Naur form (BNF). In *Encyclopedia of Computer Science.* John Wiley and Sons Ltd., GBR, 129–131.

[28] Philippe McLean and R Nigel Horspool. 1996. A faster Earley parser. In *Compiler Construction.* Springer Berlin Heidelberg, 281–293.

[29] Kevin D Mitnick and William L Simon. 2009. *The Art of Intrusion: The Real Stories Behind the Exploits of Hackers, Intruders and Deceivers.* John Wiley & Sons.

[30] Antinus Nijholt. 1991. The CYK approach to serial and parallel parsing. *Lang. Res.* 27, 2 (1991), 229–254.

[31] Jon Oberheide, Michael Bailey, and Farnam Jahanian. 2009. PolyPack: an automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX conference on Offensive technologies (WOOT'09).* USENIX Association, USA, 9.

[32] Oberhumer M F X. [n. d.]. UPX the ultimate packer for eXecutables. https://github.com/upx. Accessed: 2023-9-NA.

[33] Paul Salanitri Paulrho. [n. d.]. mthtest2: maths benchmark test (numerical position of moon).

[34] Alessandro Pellegrini. 2013. Hijacker: Efficient static software instrumentation with applications in high performance computing. In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation (HPCS).* IEEE, Piscataway, NJ, USA, 650–655. https://doi.org/10.1109/hpcsim.2013.6641486

[35] Valentina Porcu. 2018. Matplotlib. In *Python for Data Mining Quick Syntax Reference*, Valentina Porcu (Ed.). Apress, Berkeley, CA, 201–234.

[36] PreEmptive Solutions, LLC. [n. d.]. The unofficial R8 documentation. https://r8-docs.preemptive.com/. Accessed: 2023-9-4.

[37] Michael Sikorski. 2012. *Practical Malware Analysis: The hands-on guide to dissecting malicious software.* No Starch Press, San Francisco, CA.

[38] Jae Hyuk Suk and Dong Hoon Lee. 2020. VCF: Virtual Code Folding to Enhance Virtualization Obfuscation. *IEEE Access* 8 (2020), 139161–139175. https://doi.org/10.1109/ACCESS.2020.3012684

[39] Péter Ször. 2005. *The art of computer virus research and defense.* Addison-Wesley Professional, Boston, MA, USA.

[40] Péter Ször and Peter Ferrie. 2001. *Hunting For Metamorphic.* Technical Report. Symantec.

[41] The Mental Driller. 2002. *Metamorphism in practice or How I made MetaPHOR and what I've learnt.* Technical Report 29A 6. 29A.

[42] Virus Total. [n. d.]. VirusTotal. http://www.virustotal.com. Accessed: 2023-9-NA.

[43] Gregory Wróblewski. 2002. *General Method of Program Code Obfuscation.* Ph. D. Dissertation. Wrocław University of Science and Technology.

[44] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. 2010. Mimimorphism: a new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10).* Association for Computing Machinery, New York, NY, USA, 536–546. https://doi.org/10.1145/1866307.1866368

[45] Pavel V. Zbitsky. 2009. Code mutation techniques by means of formal grammars and automatons. *Journal in Computer Virology* (2009).

[46] Éric Filiol. 2007. Metamorphism, Formal Grammars and Undecidable Code Mutation. *International Journal of Computer and Information Engineering* (2007).

Ringraziamenti

Ci tengo a dedicare uno spazio nell'elaborato alle persone che mi hanno sostenuto durante questo percorso di studi. In primis, desidero esprimere la mia sincera gratitudine al Prof. Alessandro Pellegrini per il suo straordinario contributo a questo lavoro di tesi. Il Prof. Alessandro Pellegrini è stato molto più di un semplice supervisore; è stato un mentore, un modello da seguire e una fonte inesauribile di ispirazione.

Un ringraziamento speciale va alla mia famiglia, che mi ha sempre sostenuto in questo percorso, gioendo con me nei momenti felici ed aiutandomi a vedere la luce nei momenti più bui. Ci tengo in particolare a ringraziare Siria, che con il suo amore mi ha aiutato a tirare fuori il meglio di me.

Grazie a tutti i colleghi e le colleghe con i quali ho avuto il piacere di collaborare e condividere idee ed opinioni. Un grazie particolare va ad Andrea, un collega ma soprattutto un vero amico.

Grazie ai miei nonni, in particolare a nonno Domenico, che ha atteso con ansia il risultato di ciascun mio esame, benché fosse sempre sicuro del mio successo.

Infine, grazie a tutti coloro che mi hanno sostenuto anche solo con un pensiero, vi voglio bene