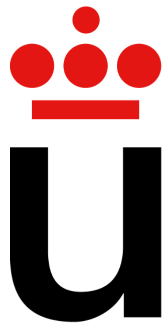

Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DE LA CIBERSEGURIDAD

MMDect: Metamorphic Malware Detection Using Logic Programming

TRABAJO FIN DE GRADO

CURSO 2022-2023

Autor/a: **Luciana Camila Fidilio Allende**

Tutelado por: Joaquín Arias Herrero

Abstract

Malware has become a major concern as the techniques used by the malicious actors improve on an ongoing basis, e.g., by using metamorphic malware, which modifies its own code to a semantic equivalent code. In parallel, anti-malware technologies have advanced, resulting in different techniques for detecting or classifying malicious programs. Even then, each technique has its limitations. For example, classic static analysis is very vulnerable to code changes, dynamic analysis requires the code to be executed in a specific environment and AI behavioral technology reports many false positives, and it is easy to fool once its classification method has been analyzed. In this work, we improve on static analysis approaches by including a metamorphic rules-based technique, which transforms lines of code into semantic equivalents patterns (reproducing what metamorphic malware does). The resulting tool, MDDect can detect variations of malware (following certain metamorphic rules) based on given signatures (patterns of code) that identify malicious behaviors or subroutines. Initially, we implemented MDDect under Python, but to facilitate the extension of the tool with new rules we re-implemented it under Prolog. To validate MDDect we use 4 examples, a (real) use case, and randomly generated programs including from zero to three signatures between harmless code blocks. In the use case, we assume the existence of a signature of a program written in Intel assembly that compromises the confidentiality of the host by printing a file to stdout with potentially elevated privileges.

Resumen

El malware se ha convertido en una de las principales preocupaciones, ya que las técnicas utilizadas por los actores maliciosos mejoran continuamente. Un ejemplo del resultado de estas técnicas es el malware metamórfico, que modifica su propio código para convertirlo en un código semánticamente equivalente. Paralelamente, las tecnologías anti-malware también han avanzado, dando lugar a diferentes técnicas para detectar o clasificar los programas maliciosos. Sin embargo, estas tecnologías tienen sus limitaciones: el análisis estático clásico es muy vulnerable a estas modificaciones de código, el análisis dinámico requiere que el código se ejecute en un entorno virtual, y la tecnología de análisis de comportamiento basado en IA reporta muchos falsos positivos y es fácil de engañar una vez es analizado su método de clasificación, entre otras. En este trabajo, mejoramos el enfoque de análisis estático al incluir una técnica basada en reglas metamórficas, las cuales transforman las líneas de código a analizar según patrones de equivalencias semánticas. La herramienta resultante, MDDect, puede detectar variaciones de malware basándose en firmas almacenadas, es decir, patrones de código que identifican comportamientos o subrutinas maliciosas. Inicialmente, implementamos MDDect en Python, pero, para facilitar la extensión de la herramienta con nuevas reglas, la reimplementamos en Prolog. Para validar MDDect utilizamos 4 ejemplos, un caso de uso (real) y programas generados aleatoriamente que incluyen de cero a tres bloques de código maliciosos entre otros inofensivos. En el caso de uso, suponemos la existencia de una firma de un programa escrito en ensamblador que compromete la confidencialidad del huésped imprimiendo un archivo en salida estándar con privilegios potencialmente elevados.

Acknowledgments

First and foremost, I would like to extend my appreciation to my tutor, Joaquín, for providing me with support and helping me along the development of this work.

Furthermore, I would like to express my sincere gratitude to my dear friends and beloved family members, who have been an unwavering source of assistance and support.

Lastly, but certainly not least, I would like to extend my utmost appreciation to all the forums and blogs that have significantly helped me understand the theory needed to develop this work and solve all the questions that arose along the way.

Contents

Abstract i

Resumen ii

Acknowledgments iii

Contents iv

List of Tables vi

List of Figures vii

1 Introduction 1
- 1.1 Objectives 4
- 1.2 Methodology 4

2 Background 7
- 2.1 Different obfuscations techniques used by malware 7
- 2.2 Metamorphic rules as functors 9
- 2.3 Definite Clause Grammars 10

3 MMDect: Metamorphic Malware Detection 13
- 3.1 General diagram 14
- 3.2 Generation of new metamorphic programs 15
- 3.3 Detection of malware using signatures 16

4 Implementation of MMDect 18
- 4.1 Step 1: Translation from Intel syntax 19
  - 4.1.1 Translation for Python 20
  - 4.1.2 Translation for Prolog 20
- 4.2 Step 2: Generate new metamorphic programs 21
  - 4.2.1 Python implementation 22
  - 4.2.2 Prolog implementation 22
- 4.3 Step 3: Detect malware using signatures 24

4.3.1 Python implementation 24
4.3.2 Prolog implementation 25
4.4 Installation and Usage 26
4.4.1 Usage 27
4.4.2 Rule definition 28
4.4.3 Signature definition 28
4.4.4 Grammar definition 29

5 Evaluation **30**
5.1 Examples 30
5.1.1 Basic rule appliance 31
5.1.2 Comments and fake jumps 31
5.1.3 Immediates 32
5.1.4 Additional jumps 32
5.1.5 Real use case 32
5.1.6 Randomly generated programs 33
5.2 Analysis of the Evaluation 34
5.2.1 Achievements 34
5.3 Limitations 35
5.3.1 Static analysis and zero-days 35
5.3.2 Code lines reordering and additional jumps 35

6 Conclusions **36**
6.1 Future work 37

Bibliography **38**

A Appendix **39**

List of Tables

4.1 Code converted to a matrix 19

5.1 Comparison between Python and Prolog 33

List of Figures

2.1 Additional jumps transformation 9
2.2 Behavior of instructions $push(1); pop(12);$. 10
2.3 Behavior of instruction $mov(r12, 1);$. 10
2.4 DCG code for translating a list of animals 12
2.5 Classic Prolog code for translating a list of animals 12

3.1 General structure of inputs and outputs 14

4.1 Function calls 19
4.2 Instruction grammar 21
4.3 Code of rules under Python 23

A.1 DCG grammar corresponding to Intel syntax 39
A.2 Code converted to a matrix 40
A.3 DCG grammar corresponding to Intel syntax 40
A.4 Code converted to a matrix 41

Chapter 1

Introduction

Malware is a piece of code or a program with potentially malicious intentions materialized after its execution in a victim information system.

The race between malware and the development of methods to detect it has been going on for a long time. Since the first antivirus programs in the 1980s, both sides have advanced their methods. Those first antiviruses only scanned files in search of code lines that matched with a set of lines identified as a virus (called signatures). Since then, antivirus programs have been incorporated into a larger suite, called antimalware solutions, which, in addition to scanning potentially infected files, also perform tasks such as scanning processes, connections, etc.

Nowadays, with a new intent for economic gain and new technologies to use, this race has taken on a new importance along with the nefarious impact of new malware. Organizations of all kinds have suffered millions of dollars in losses, meanwhile, it has never been easier and cheaper to launch a cyber attack. In addition, old methods based on comparisons of strings or a list of instructions in a code called heuristic analysis now can be easily evaded, as we can see in several demonstrations of bypassing antivirus solutions modifying comments, the file's name, or function names (Roberts 2022).

However, the growing significance of malware has led to a boost in research investment, resulting in the exploration of novel technologies. New methods such as machine learning or natural language processing have improved malware detection (Singh and

Singh 2021). Nonetheless, even though most research focuses on different applications of artificial intelligence, these methods tend to consume a lot of computational resources and time for training and execution. Furthermore, malware can evade these techniques with a high probability of success (Quertier et al. 2022).

Due to the drawbacks of AI solutions, a traditional approach based on heuristic static analysis should not be discarded, but rather reformulated and adapted to the new era to complement these new techniques and achieve more comprehensive and effective protection. Following this line of thought, new solutions based on signature matching with a slight twist are also being developed. Moreover, these solutions can even achieve better results than the use of artificial intelligence.

This is the case of techniques using abstract interpretation and behavioral heuristic analysis, i.e., analyzing the objective of a piece of software against a defined set of malicious behaviors. One implementation is SAFE-PDF, a solution made to identify malware embedded in PDF files (Jordan et al. 2018), based on the Scalable Analysis Framework for ECMAScript (Lee et al. 2012), or the experimental tool SAFE (Christodorescu and Jha 2003), which method was to elevate the code to a representation made of annotations before analyzing over those annotations. This last technique may represent an application of a behavioral approach to static analysis as early as 2003. SAFE was aimed specifically at the detection of metamorphic malware, one of the major advances in malicious software development. This type of malware can change its code, making it impossible to detect with heuristic analysis. It achieves different versions of the same program undetectable to traditional techniques using different methods. One of them is the transformation of one or more lines of code to others that do the same function. These rules of equivalency between instructions are called metamorphic rules.

A more recent application of abstract interpretation to metamorphic malware, in this case, focused on the infer of their signatures and transformations, is MetaSign (Campion, Dalla Preda, and Giacobazzi 2021). This solution uses the same transformations the malware uses to change its code to learn all the metamorphic rules it has used. However, this project didn't provide a direct implementation of software analysis.

We have created a tool based on this last technique, MMDect, which applies two separate steps, namely generation and comparison, to establish the practicality of utilizing

metamorphic transformations for software analysis since, if the malware has used a known set of rules to mutate, it is possible to generate the original code from one iteration of it.

While MetaSign was aimed at learning the metamorphic rules used by a given malware, MMDect uses the same method of applying transformations but with the purpose of detecting possible infected or malicious programs.

In the generation step, the program will obtain all the possible iterations of a piece of software using all the stored applicable metamorphic rules. Then, in the comparison step, it will compare each iteration against static signatures to effectively detect metamorphic malware. As for the language used, the initial proposals were reduced to two: Python, for its widespread use and ease of programming, and Prolog, for its nature, completely integral with the use of rules and the power that this combination could bring. So we decided to use both: Python would be used to program a common input and output controller, but the internal modules representing each phase would be developed in both languages to test the initial hypothesis regarding Prolog's expressiveness and capability to generate various versions of a program by applying metamorphic rules. The resulting tool, MMDect, is open-source and is available at `https://github.com/Lu-all/MMDect`.

The structure of this paper is the following:

In Chapter 1 it is explained the context and aims of this work, along with research in a similar area, including the work on which this technique is based. Chapter 2 explains the theory on which MMDect is based more in-depth. Chapter 3 presents an extensive description of the design of the tool, while Chapter 4 focuses on its implementation. Chapter 5 details the evaluation process, by testing MDDect with several cases. Finally, Chapter 6 summarizes the results obtained and suggests future lines of research.

1.1 Objectives

The principal objective of this work is to improve the security of IT systems by increasing the ability to detect malware using metamorphic rules. In order to do so, we have developed a static analyzer capable of detecting metamorphic malware.

While developing the analyzer, as we realized the rule-based structure of the code, we became aware that a logic programming language, such as Prolog, would be ideal to simplify the development of the tool. Thus, an additional goal of this work is to test the advantages of applying this programming paradigm and, at the same time, extend its use in the cybersecurity field.

As an additional objective, we wanted to probe the utility of signature-based analysis in the detection of modern malware, as this approach is being neglected in favor of experimental techniques, when, as mentioned in this paper, they are precisely the perfect method to reduce the weaknesses of these new technologies. By using both approaches, a better solution can be achieved than if either of them were discarded, so traditional methods should not be discarded but adapted to the new times and further improved.

1.2 Methodology

While researching metamorphic malware detection, we came across an intriguing tool called MetaSign (Campion, Dalla Preda, and Giacobazzi 2021). This tool has the ability to learn the rules used by metamorphic malware to mutate. The theory behind MetaSign is as simple as it is captivating: it applies the same rules used by malware to reverse transformations, as these changes can occur in both directions. However, MetaSign lacked an implementation that would allow direct analysis of the malware using the acquired rules. This apparent lack of functionality prompted us to take proactive measures and embark on the development of a tool that would fill precisely this gap.

Following this line of work, our main goal was to develop a tool able to detect meta-

morphic malware using transformations between semantically equivalent sets of instructions. As we can see in Chapter 5, the principal objective has been met. These results not only confirm the viability of this technique but also open the door to new paths of development such as code line reordering or integration with rule learning, resulting, as a whole, in a powerful, fast, versatile, and resource-savvy antivirus suite.

To be able to develop this tool, it was necessary to learn logical programming from scratch. Unfortunately, within the scope of my career, there was no direct opportunity to grasp the basics of a language belonging to this paradigm in the career. Nevertheless, by reading blogs and forums aimed at logic programming, it was feasible to develop the planned implementation while achieving a good final result.

The planning and development of this work followed a customized methodology, departing from the traditional approach of agile methodologies that typically involve fixing a minimal viable product (MVP) and working on achieving an executable version of it within the shortest possible time frame. Instead, our approach involved breaking down the minimal viable product into three fundamental steps, which are elaborated upon in the dedicated implementation chapter (Chapter 4). Then, we allocated three-quarters of the overall development time to meticulously develop and test each of the aforementioned steps individually until a sufficient quality, higher than usual in an MVP, was achieved. I.e., instead of a unique product, we defined three MVPs, and then we established an agile process for each of them. Afterward, the final quarter was destined in resolving minor issues, documenting the installation and use of the tool, and improving the overall code quality. This methodology was useful due to the modular but sequential nature of this project, reducing the risk of failure in one step affecting the subsequent steps.

The design process of the tool was focused on achieving our main goal: to develop a tool capable of detecting metamorphic malware using metamorphic rules.

To complete this goal, first we tracked down its requirements. We needed a program that could generate variants and then compare them against signatures. To achieve this, we devised a two-part structure that would effectively address the challenge. In the first part, metamorphic rules should be applied to achieve the greatest number of variants. Then, in the second one, each variant of the original code should be compared with established signatures to effectively detect if the code was malicious. The best course

of action was not doing the process mentioned above in the original Intel syntax, but in a comfortable intermediate language, so as a prior step, we implemented a translator between those.

To test that the conditions presented have been met, we crafted four special cases, one for each major transformation, and a use case involving a real program. However, to reinforce an impartial validation process, later we tested the program against randomly generated programs. All these cases and their results are disclosed in Chapter 5.

By achieving the above objective, it is also demonstrated that static heuristic analysis can be used to detect metamorphic malware. Therefore, a potential integration with AI solutions would not decrease but empower the overall effectiveness of this theoretical suite.

Moreover, while thinking about the structure of the generator module, we noticed the rule-oriented nature of it. Therefore, we formulated the following hypothesis: "Logical programming would simplify the process of writing and incorporating rules and signatures". To demonstrate this theory, we implemented the modules in Prolog and compared its results with the Python implementation, finding that, indeed, the result was simpler and easier to escalate, strengthening the extensibility of the tool.

This work was also presented at the PROLE 2023 conference. The presented article, which summarizes the essence of this research, can be viewed in the tool's repository. The feedback from the judges has been an invaluable source for improving this work.

Chapter 2

Background

In this chapter, it is discussed useful knowledge to understand the goal and development of this work. First, various techniques used by malware developers to disguise their programs and avoid their detection by antivirus software can be found in Section 2.1. In Section 2.2, it is explained the theory and method behind MMDect. Specifically, we will elaborate on the concept of metamorphic rules and how they work. Finally, in Section 2.3, a brief introduction to Definite Clause Grammars (DCG) is given, showing its utility in parsing through a comparison between a classic Prolog implementation and a DCG implementation.

2.1 Different obfuscations techniques used by malware

There is not a single and general technique used by malware developers to evade detection, but rather they have been evolving and branching out according to new measures developed by anti-malware systems. There are newer techniques, such as, for example, mimicking the behavior of a benign program to evade behavioral analysis, while there are other more traditional techniques such as encrypting or compressing the malicious payload of the program, i.e. the instructions that carry out the malicious purpose of the program, and including it with a seemingly innocuous program that would be responsible for reversing that process and executing it. This last technique would be the case of polymorphic programs.

Metamorphic programs are a type of malware that use transformations in its own code in order to have the same functionality but with different instructions. This technique is aimed specifically at signature-based analyzers. Still, this type can be further categorized depending on the type of transformation that is used.

In order of complexity, the most basic one would be adding comments in random lines of code. A technique derived from this would be the addition of instructions that don't do anything (NOP instructions). These new additions change nothing about the functionality of the code, while adding extra lines that can difficult their automated analysis. The second test (Subsection 5.1.2) was based on this transformation.

Another technique would be renaming the names of variables and changing the value of constants. For example, if one piece of malware is recognized by printing a phrase in a file on a certain date, changing the phrase or the date could drop the number of positives after analyzing the file. The third test (Subsection 5.1.3) was based on this transformation.

A third technique would be replacing sets of instructions with equivalent ones, for example, pushing an immediate to the stack and then extracting the last value of the stack into a register is the same as directly storing the immediate value in the register. This example is explained more thoroughly in Section 2.2. Detecting this type of transformation is the main goal of MMDect, as other transformations would require a different, but not incompatible, approach. The first test (Subsection 5.1.1) was based on this transformation.

Another method of camouflaging from analyzers is the addition of jumps or functions. The code is divided into segments. Then a label is placed in front of each segment. The segments are then randomly reordered and interwoven by unconditional jumps to their assigned labels to preserve their original order. An example is illustrated in Fig. 2.1. The fourth test (Subsection 5.1.4) was based on this transformation.

Each technique is meant to fool one type of analysis, either by encrypting its payload, behavioral pattern mimicking techniques, or changes in its instructions, among others. An analyzer directed to detecting polymorphic programs by the analysis of its entropy can not detect metamorphic malware. Likewise, a program that reverses transforma-

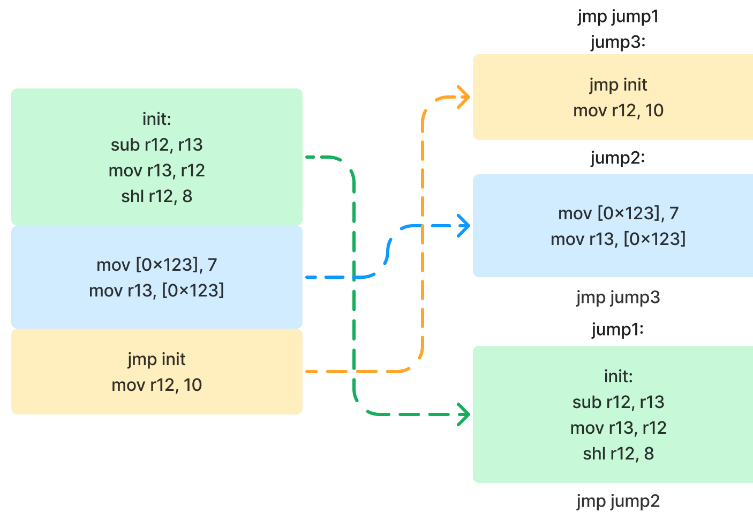

Figure 2.1: Additional jumps transformation

tions in the instructions will not be able to identify polymorphic malware. Because of this, there is not an all-powerful technique able to detect all kinds of malware. The combination of different methods of analysis is the recommended path to follow, as the permutation of different obfuscations is both costly and difficult.

2.2 Metamorphic rules as functors

Metamorphic malware, like any malware, is a program with potentially malicious intentions. Yet, the distinction between metamorphic malware and regular malware is that, with the purpose of fooling analyzers, it goes through a process of finding its own code and changing it into one that does the same job, but with altered instructions.

One possible transformation uses some rules that dictate equivalent sets of instructions, called metamorphic rules. An example in Intel syntax would be:

$$push(1); pop(r12); \Longleftrightarrow mov(r12, 1);$$

Fig. 2.2 shows the behavior of the instructions on the left, $push(1); pop(r12);$, while Fig. 2.3 shows the (equivalent) behavior of the instruction on the right, $mov(r12, 1)$. As we see, these two sets of instructions are semantically equivalent because the initial state and the final state are the same.

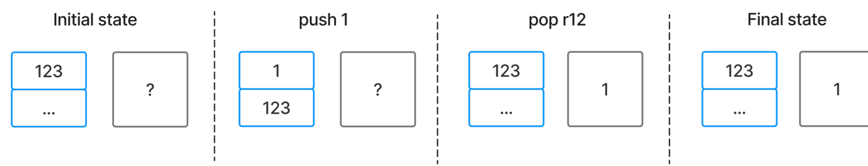

Figure 2.2: Behavior of instructions $push(1); pop(12);$.

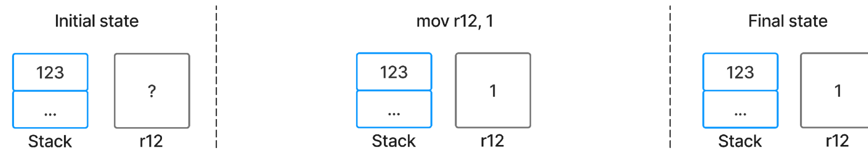

Figure 2.3: Behavior of instruction $mov(r12, 1);$.

The set of metamorphic rules used by a piece of software is called a metamorphic engine. If we know all the rules (or those needed to recreate them) that potential malware has used to rewrite its code, we can modify it in the same way, undoing all the potential mutations it could have made. This is possible because metamorphic rules work both ways, as they are equivalent.

In this work, we have treated metamorphic rules as functions, in the case of Python, and as predicates in the case of Prolog.

2.3 Definite Clause Grammars

DCG, or Definite Clause Grammars, is a group of context-free grammars that can be executed. It is commonly used in the realization of parsers due to its simplicity compared to the equivalent code in classic Prolog.

Just as there are base cases and longer cases in classic Prolog, in Definite Clause Grammars there are terminal symbols and non-terminal symbols since what is defined is a grammar.

A terminal symbol is one that cannot be extended any further. If we take the example of the bidirectional translation of English and Spanish, a terminal symbol would be "Dog = Perro".

```
1    animal([dog]) --> [perro].
```

The equivalent in classic Prolog would be:

```
1     animal(dog, perro).
```

Even though, if we wanted to translate a list of animals, in addition to the terminal cases, it would be necessary to establish a non-terminal symbol, resulting in the code shown in Figure 2.4. This Figure also displays that DCG also offers the possibility to call and execute classic Prolog code in the definition of symbols, if it is specified between brackets.

To call a grammar, it is necessary to use "phrase". Phrase has three arguments:

- The first one is the grammar we want to use and value we wish to achieve by executing it. In the following call, "[dog, cat]", is the value "[Word|Phrase]".
- The second one is the original list we want to parse, in this case, "[perro, gato]".
- The third one would be the remainder left after achieving the transformation.

```
1     phrase(translate([dog, cat]), [perro, gato], Remainder).
```

Starting from a common base, in which 5 animals are translated individually by constant values:

```
1 animal(dog, perro).
2 animal(cat, gato).
3 animal(elephant, elefante).
4 animal(frog, rana).
5 animal(dolphin, delfin).
```

We have programmed a classic Prolog version equivalent to the animal list translation program, in which two different predicates have been defined for each translation direction. If similar results to those achieved with the grammar defined above in Fig. 2.4 are pursued, it is needed to make some adjustments to the code. The result is the code found in Fig 2.5.

We can observe that, in DCG, it is not necessary to create two different predicates for each direction, which greatly simplifies the task. Moreover, although DCG has a slightly more complex grammar, the resulting code is noticeably simpler to program.

```
1 translate([]) --> []. % Terminal symbol
2
3 translate([Word|Phrase])--> % Non-terminal symbol
4     [Palabra],
5     translate(Phrase),
6         {animal(Word,Palabra)}.
```

Figure 2.4: DCG code for translating a list of animals

```
1 translate([],Frase,Frase).
2 translate(Phrase, Phrase, []).
3
4 translate(Phrase, Remainder, Frase):-
5     nonvar(Phrase),
6     length(Phrase, L),
7     nth1(L, Phrase, Word),
8     animal(Word, Palabra),
9     append([Palabra], Remainder, New_remainder),
10    append(New_phrase, [Word], Phrase),
11    translate(New_phrase, New_remainder, Frase).
12
13 translate(Phrase, Remainder, Frase):-
14    nonvar(Frase),
15    length(Frase, L),
16    nth1(L, Frase, Palabra),
17    animal(Word, Palabra),
18    append([Word], Remainder, New_remainder),
19    append(New_frase, [Palabra], Frase),
20    translate(Phrase, New_remainder, New_frase).
```

Figure 2.5: Classic Prolog code for translating a list of animals

Chapter 3

MMDect: Metamorphic Malware Detection

MMDect is a tool implemented in Python and Prolog. This tool was developed with the goal of increasing the security of an information technology system by targeting the detection of metamorphic malware. To accomplish this, it is employed a static analysis technique consisting in using metamorphic rules to reverse the possible transformations a piece of malware can have been put through.

The basis of MMDect is the set of metamorphic rules used to detect variations of known malware. As a proof of concept, we have defined a set of 19 rules, where the ones with names starting with "G" are inherited from MetaSign (Campion, Dalla Preda, and Giacobazzi 2021). The defined set of rules can be found in the file rules.txt in the tool's repository. Note that MMDect is designed to facilitate the introduction of new grammars, rules, and signatures. New grammars can be used to expand it to new languages and syntaxes, new rules widen the possible transformations we can reverse, and new signatures can broaden the set of malware we can detect. In the Subsections 4.2 and 4.4.2 it is explained how to add a new rule, while the method to build new signatures is detailed in Subsections 4.3 and 4.4.3. The Subsection 4.4.4 elaborates on how to add a new grammar. Additionally, in the Subsection 4.2 it is probed that the Prolog implementation is far superior to the Python implementation in terms of simplicity of escalation. In other words, it is recommended the use of Prolog implementation to further expand the tool by writing new rules.

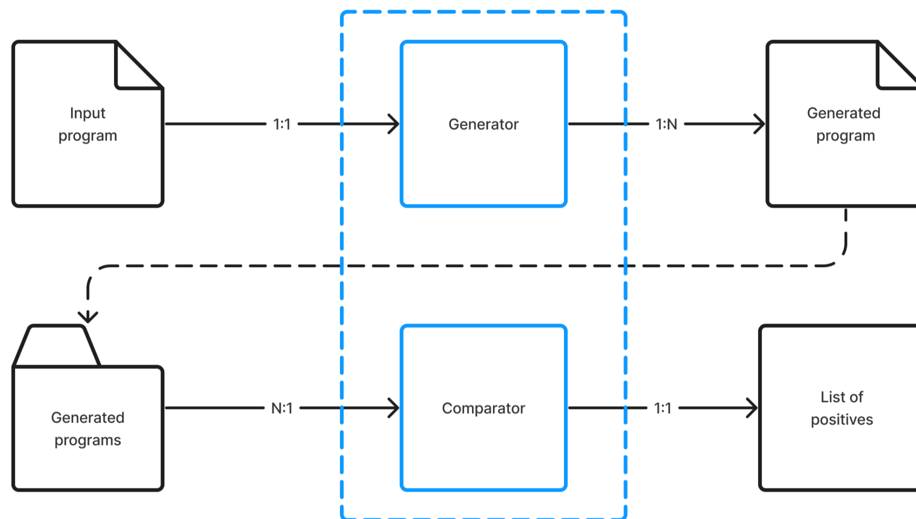

Figure 3.1: General structure of inputs and outputs

This Chapter in particular focuses on the design of MMDect, while the implementation details can be found in Chapter 4. First, a general disposition of the code is reviewed in Section 3.1, providing information about the objective of the two principal modules. Then, the focus is narrowed down to both modules in Sections 3.2 and 3.3.

3.1 General diagram

In a normal execution, we first translate the original Intel syntax into an intermediate syntax to handle the code more comfortably.

Once the original syntax is translated, the code is divided into two modules that work independently. We can internally connect the output of the generator module to the input of the comparator module, or use only one of them. A diagram of the inputs and outputs can be seen in Fig. 3.1.

- The generator module will apply rules to obtain different versions of a program. It takes the original file as input and outputs one or more files.
- The comparator module has one or more programs as input and compares them with different (stored) signatures to get matches.

An analysis of a file would first call the generator module, which outputs the metamorphic versions. Then the comparator module would compare the original file and the generated versions against the stored signatures.

Each module is implemented both in Python and in Prolog, with different results.

3.2 Generation of new metamorphic programs

The second step of the execution, the generation of new metamorphic programs, can be used to reduce the number of lines in a program, improve its readability, and create different equivalent versions of a program.

The program goes through the file and looks for matches with cases where certain rules can be applied. The theory and an example of this rules can be found in Section 2.2. In Prolog, it can choose to apply or ignore them to generate all possible versions, while in Python it will always apply them. That is, in Python mode, only the version that applies all possible rules is written to a file. In the default mode (Prolog), however, every possibility is given.

For example, considering the following program (translated into the intermediate language detailed in Section 4.1):

```
1 ["mov", "[123]", "0x6477737361702FFF"],
2 ["push", "[123]"],
3 ["push", "12"],
4 ["pop", "r12"],
5 ["push", "r12"],
6 ["mov", "r13", "13"],
7 ["mov", "r12", "0xFFFFFFFF6374652F"],
8 ["xor", "rax", "rax"]
```

The implementation of MMDect under Python generates only one version:[1]

```
1 ["push", "7239381865414537215"],
2 ["mov", "r12", "12"],
3 ["push", "r12"],
```

[1] We may extend this implementation to generate all the possible combinations. However, since Python is deterministic, that extension requires more effort than the equivalent implementation using Prolog, as we already discussed.

```
4 ["mov", "r13", "13"],
5 ["mov", "r12", "18446744071083156783"],
6 ["xor", "rax", "rax"]
```

In the meantime, using the implementation under Prolog, MMDect gives the following 6 possible variations. Focusing only on the instructions in lines 3–5, i.e., $push(12);pop(r12);push(r12)$:

- Considering the application of rule g1, $push(12);pop(r12) \iff mov(r12,12)$, MMDect applies the following transformation:

$$push(12);pop(r12);push(r12) \iff mov(r12,12);push(r12)$$

- While considering the application of rule f19, $pop(r12);push(r12) \iff NOP$, MMDect provides the following (semantically equivalent) transformation:

$$push(12);pop(r12);push(r12) \iff push(12)$$

Note that one of the possible variations is the one generated by the implementation under Python.

3.3 Detection of malware using signatures

In this step, we compare the generated programs with our base data of signatures to detect if the input program is infected or malicious. Depending on the version of our tool we define the signature differently:

- In Python, the signatures are defined in regular expressions (regex):

```
.*('mov',\s*'((r\w\w?)|(e\w\w))',\s*'0x6477737361702FFF'
.*
'push',\s*'((r\w\w?)|(e\w\w))'
.*
'mov',\s*'((r\w\w?)|(e\w\w))',\s*'0xFFFFFFFF6374652F'
.*
'push',\s*'((r\w\w?)|(e\w\w))').*
```

- In Prolog, the signatures are a list of functors:

```
mov(reg(_Reg), imm('0x6477737361702FFF')),
_,
push(reg(_Reg)),
_,
mov(reg(_Reg),imm('0xFFFFFFFF6374652F')),
_,
push(reg(_Reg))
```

Regular expressions were chosen as the format to write rules in the Python implementation because it allows a certain flexibility that the intermediate language has by default. Both formats allow wildcard arguments and command lines, a specific type of argument (register, immediate, memory, or tag), or the indication of a particular one.

Still, while a Prolog signature is easier and faster to write, it may have some difficulty in wildcarding a command and fixing its arguments, whereas in regex it is as simple as defining other factors. Anyway, it is possible to compare in both formats at the same time, giving the user the possibility to choose the format according to their situation and needs.

Chapter 4

Implementation of MMDect

The implementation details are divided into three basic steps.

- The first step, Translation (Section 4.1), explains how the original Intel syntax of the suspicious file is translated to the format used by the tool.
- The second step, Generation (Section 4.2), discusses the generation of new metamorphic versions.
- Finally, the third step, Detection (Section 4.3), covers matching all generated versions plus the original file against stored signatures.

Input and output are done via Python. For requests to Prolog, the pyswip library is used (Tekol 2007).

The generation of new programs is centralized in the generate_program function. From this module, the implementation in Prolog (generate_prolog) or the one in Python (generate_python) is called.

The comparison is centralized in the compare_program function, which also manages the calls to the implementation in Prolog and in Python.

Both generation and comparison functions can be called directly from the main or generate_and_compare_program, which connects the generate and compare modules.

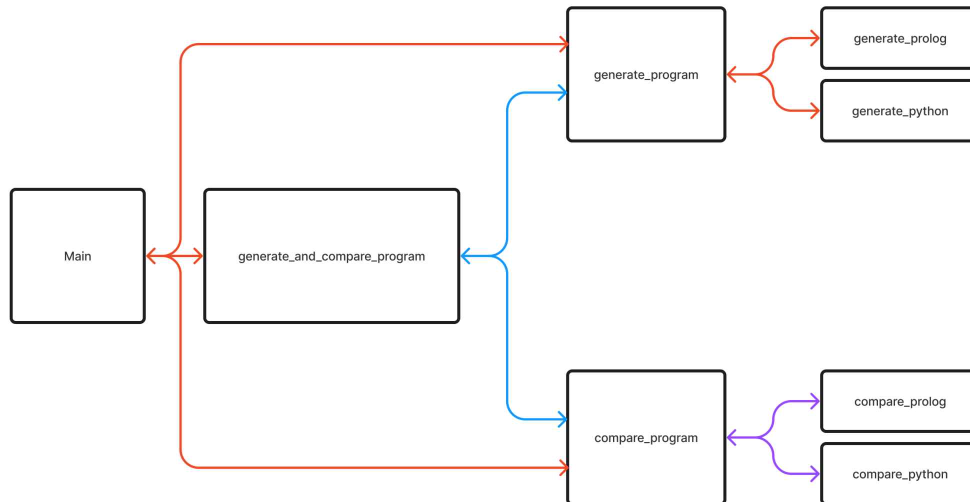

Figure 4.1: Function calls

Fig. 4.1 details the internal calls the program can make, exclusive OR calls (one or the other, but not both) are marked in red, AND calls (both are called) are marked in blue and OR calls (at least one of both) are marked in purple.

4.1 Step 1: Translation from Intel syntax

The intermediate language removes all comments in the code, isolates the header (all code before the entry point specified), and converts all instructions into a matrix. This format is directly adapted from the MetaSign project (Campion, Dalla Preda, and Giacobazzi 2021). This code can be found in its GitHub repository (Campion, Dalla Preda, and Giacobazzi 2020).

Table 4.1: Code converted to a matrix

Instruction line	Command	Argument 1 (optional)	Argument 2 (optional)
00	'mov'	'r12'	'0x6477737361702FFF'
01	'push'	'[123]'	
02	'push'	'12'	
03	'pop'	'r12'	

4.1.1 Translation for Python

In Python, the matrix mentioned above is directly used to generate and compare. This matrix is composed of a list of lists of a command + argument 1 + argument 2. For example, Table 4.1 shows the matrix corresponding to the code listed below.

```
1 # Intro code
2 .global _start
3 .text
4 _start:
5 mov r12, 0x6477737361702FFF
6 # Comment
7 push [123]
8 push 12
9 pop r12 # Comment
```

4.1.2 Translation for Prolog

On the other hand, under Prolog, it will suffer additional conversions starting from the previous matrix.

- From a list of lists of strings to a list of lists of a command and a list of its arguments expressed as atoms:

$$[[\text{“}i\text{”}, \text{“}a1\text{”}, \text{“}a2\text{”}][\text{“}i\text{”}, \text{“}a\text{”}]] \Rightarrow [[i, [a1, a2]], [i, [a]]]$$

- From the previous list to a list of functors:

$$[[i, [a1, a2]], [i, [a]] \Rightarrow [i(a1, a2), i(a)]$$

The parsing from the matrix used in Python to the list of functors used in Prolog is done with Definite Clause Grammars, although an alternative implementation in classic Prolog is also included.

In this case, the instruction grammar used is composed of a command and arguments, simulating the general structure of the original assembly language that is described in Fig. 4.2.

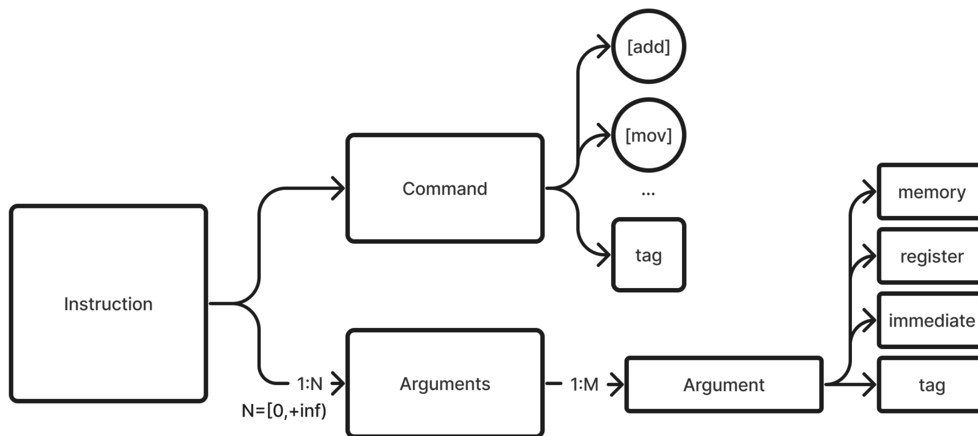

Figure 4.2: Instruction grammar

This structure is translated into the code in Fig. A.3 and then, the code in Fig. A.4 invokes the grammar:

- First, in lines 1–6, we call the parse predicate, which handles the two steps of parsing.
- To convert the matrix into functors with typed arguments, we first use the previous grammar. The call is made in the "to_atoms" predicate, defined in lines 7–9.
- In lines 10–12, the predicate involving the second step is defined, i.e. to convert the matrix into functors employing the "to_functors" predicate.
- The predicate "to_functors" manages the matrix and calls the predicate "to_functor" (lines 13–18). The last predicate converts a list defining an instruction and its arguments into a functor, using one of the three predicates depending on the number of arguments of the instruction.

4.2 Step 2: Generate new metamorphic programs

Generation is handled by the generate_program function in modeHandler.py. This function will redirect the generation to Prolog or Python relevant modules as specified.

4.2.1 Python implementation

A rule in Python is defined in about ten to twenty complex lines divided into two functions depending on its complexity. Each rule has a check function and a code change function.

The check internal function checks that the rule can be applied by verifying the type of its arguments, as the command types are already confirmed by the apply_rules function.

The code change function replaces the left side of the rule with the corresponding right side. An example of two rules is described in Fig. 4.3.

4.2.2 Prolog implementation

Rules in Prolog are defined in a maximum of four simple lines, and only one line if they do not involve operation instructions. In this implementation, a rule is defined in the following way:

```
1 rule(g1, [push(Imm),pop(Reg)], [mov(Reg,Imm)]) :-
2     imm(Imm), reg(Reg).
3
4 rule(g7, [mov(mem(Mem), imm(Imm)), Opi], [Opo]) :-
5     operation(Opi, O, [reg(Reg),mem(Mem)]),
6     operation(Opo, O, [reg(Reg),imm(Imm)]).
```

where the first argument is its name, the second is the left side of the rule, and the last argument is the right side. As we can see, defining a rule in Prolog is much simpler than doing it in Python.

```
def _rule1_check(self, program_line: int, program: Program) -> bool:
    return (is_immediate(program, program.operand(program_line, 1))) and (
        is_register(program.operand(program_line + 1, 1)))

def rule1(self, program_line: int, program: Program) -> bool:
    """
    PUSH Imm / POP Reg <--> MOV Reg,Imm
    :param program: program to modify
    :param program_line: line where the rule should be applied
    :return: True if rule can be applied, False if else
    """
    if self._rule1_check(program_line, program):
            new_line = ["mov", program.operand(program_line + 1, 1),
                program.operand(program_line, 1)]
            program.delete(program_line)
            program.delete(program_line)
            program.insert_to_instructions(program_line, new_line)
            return True
        else:
            return False

def _rule7_check(self, program_line: int, program: Program) -> bool:
    return (is_memory_address(program.operand(program_line, 1))) and (
        is_immediate(program, program.operand(program_line, 2))) and (
               is_register(program.operand(program_line + 1, 1))) and (
               is_memory_address(program.operand(program_line + 1, 2))) and
                   (
                   program.operand(program_line, 1) ==
                       program.operand(program_line + 1, 2))

def rule7(self, program_line: int, program: Program) -> bool:
    """
    MOV Mem,Imm / OP Reg,Mem <--> OP Reg,Imm
    :param program: program to modify
    :param program_line: line where the rule should be applied
    :return: True if rule can be applied, False if else
    """
    if self._rule7_check(program_line, program):
        new_line = [program.instruction(program_line + 1),
            program.operand(program_line + 1, 1),
                    program.operand(program_line, 2)]
        program.delete(program_line)
        program.delete(program_line)
        program.insert_to_instructions(program_line, new_line)
        return True
    else:
        return False
```

Figure 4.3: Code of rules under Python

4.3 Step 3: Detect malware using signatures

The detection function is handled by the compare_program in modeHandler.py. This function will redirect the comparison to Prolog or Python relevant modules as specified.

4.3.1 Python implementation

The Python module is implemented in regexHandler.py.

A signature in regex follows the structure specified in the subsections hereunder.

General structure

Common signatures start with the symbol ".*(" and end with ").*". The first symbol, ".*", means that the match can happen either at the start or middle of the code, while the second one means that the last instruction that matches can be at the middle or the end of the code.

Each instruction is surrounded by a "\[" and "\]" set of symbols. Between instructions either the set ", " can be put if the instructions must be contiguous, or ".*" if it is not a requirement to match the signature. An instruction is composed by its command and its arguments, each of them surrounded by the "'" symbol set. Between the command and the first argument, and between arguments, the set of symbols ", " or ",\s*" must be put. The second one guarantees the good operation of the rule by including more types of space (or the lack of one).

The types of arguments are written directly without specifying their type. Memory addresses must be between "\[" and "\]". An example of a signature would be:

```
1    .*(\['mov',\s*'r12',\s*'0x6477737361702FFF'\].*\['shr',\s*'r12',\s*'8'\]).*
```

Recommendations

Variables and wildcards are defined following the same rules as regular expressions. Even so, there are certain recommended patterns to follow depending on the type of argument:

- Wildcarding a register: '((r\w\w?)|(e\w\w))'
- Immediates: it is recommended to give more than one number format. For example, for the number 0xFFFFFFFF6374652F, it is recommended to use '(0xFFFFFFFF6374652F|18446744071083156783)' instead.
- Wildcarding a memory address: '\[\w+\]'

4.3.2 Prolog implementation

Both the generation and detection of malware in Prolog are implemented in the same file: dcg_rules.pl in the case of DCG Prolog and rules.pl in the case of classic Prolog.

The following subsections are a guide to the implementation of new signatures in intermediate languages.

Basic types

Prolog signatures are a list of functors. Each functor is composed in the same way that instructions, i.e. instruction(type(argument1), type(argument2)). For example, "shr r12, 8" will be "shr(reg(r12), imm('8'))". The definition of each type of argument follows this pattern:

- Register / reg(register): r12 <–> reg(r12)
- Immediate / imm(immediate): 8 <–> imm('8')
- Memory / mem('address'): [r12] <–> mem('r12')
- Tag / tag(name): close_file <–> tag(close_file)

Variables

A variable can be defined as "V" or "_V" (in this tool, values of variables defined in signatures will not be displayed). For example:

```
1       mov(reg(_Reg),imm('0x6477737361702FFF')), shr(reg(_Reg),imm('8'))
```

In this code, "_Reg" must have the same value in both occurrences. If the first "_Reg" is assigned the value r12, the second line must be "shr(reg(r12),imm('8'))" to validate the signature.

Wildcards

A wildcard whose value will not necessarily be repeated later in the code can be defined as "_". For example:

```
1  mov(reg(_),imm('0x6477737361702FFF')), shr(reg(_),imm('8'))
```

In this other code, "_" doesn't have to have the same value in both instances. If the first "_" is assigned the value "r12", the second line could be "shr(reg(r14),imm('8'))" and the signature would still apply. Not only instructions or arguments can be wildcarded, but it is also possible to indicate that zero or more lines may appear between two instructions by using "_", as seen in the signature uc_etc-passwd-wildcard.prologsign.

4.4 Installation and Usage

In order to use the tool, regardless of the operating system used, since it has a unique multi-platform code, it is needed *Python 3.6 or higher* and pip-installable libraries, mainly *colorama*. To run the Prolog module, it is required the *pyswip* library, available through the normal pip installation. It is also necessary to install *SWI Prolog 8.2 or higher* (SWI Prolog) on the machine where the program will be executed. To do this, follow the instructions on the official page of SWI Prolog (https://www.swi-prolog.org/).

The program analyzed must be written in Intel format in case of assembly code and must have all sections that are not code before its entry point.

In the following subsections, a guide to its usage and how to add new rules, signatures, and syntaxes will be provided.

4.4.1 Usage

MMDect is presented as a command line tool because it is the best option when using it in an automated way, compared to a graphical version. Nonetheless, having this format does not imply a greater difficulty of use, since it has a help section accessible at any time using the usual flag "-h" and a comprehensive guide in its repository.

The more basic usage would be executing "python mmdect.py", and it would do a test execution using a default example located at examples/passwddump.txt.

The tool has the following options, specified as arguments:

- -h or –help to display options.
- -v or –verbose to show output (true by default).
- -a or –att_syntax to write the output file in ATT syntax (Intel syntax is selected by default).
- -m or –mode to specify mode between generate-only (only execute generation module), compare-only (only execute comparison module, or both (execute both modules). "Both" mode is selected by default.
- -p or –python to execute generate, compare, or both in Python instead of Prolog (default value is none).
- -f or –file to specify the input file. If not specified, it will use examples/passwd-dump.txt as input.
- -o or –output to specify the name of the output file. If not specified, it will be < file >-generated.< extension >.
- -O or –positives_output to write positives to a file. If not specified, positives will be printed in standard output (even in silent mode).

- -s or –signatures to specify the path of the signatures parent directory, which also enables compare step. Rules for prolog calculation should have '.prologsign' extension, while rules extension for comparison in Python must be '.txt'. Python rules can be in a regex format.
- -c or –compare-both to compare both Regex and Prolog signatures (overwrites -p python in comparison).
- -M or –multiple_input to input multiple files, giving the path to the directory (it uses recursion) in the -f parameter.
- -P or –prolog to specify the use of DCG (dcg) or classic Prolog (classic). By default, DCG is used.

4.4.2 Rule definition

New rules are added in dcg_rules.pl if DCG version is used, rules.pl if classic Prolog version is used, or rulesHandler.py if the Python version is used. Instructions and examples on how to add a rule can be found in Subsection 4.2.

4.4.3 Signature definition

New signatures are added in the path specified by the "–signatures" argument. Intermediate language signatures have the extension ".prologsign" (but can be written as a regular plain text file, the extension is used only for simplicity of implementation). Regex signatures have ".TXT" extension.

The first line of both signatures should contain the name of the signature. Prolog signatures are multiline and require separate files for each signature, whereas multiple regex signatures can be stored in the same file. In the latter case, only one name is specified in the first line, and each signature is associated with a specific line.

The structure of signatures used by Python implementation is detailed in Subsection 4.3.1, while Subsection 4.3.2 extends the implementation of the signatures used by the Prolog implementation.

4.4.4 Grammar definition

Adding a new grammar would require adapting the translation step of the tool to adapt the syntax of the new language to the intermediate language used. In order to do so, it is necessary to change the input and output handler, and the grammar defined to do the translation to the format Prolog uses. Additionally, it would be necessary to add new signatures and rules applicable to the new syntax.

Chapter 5

Evaluation

As mentioned before, MMDect and the examples used in the evaluation are available at `https://github.com/Lu-all/MMDect`. The first four examples are designed to test specific capabilities based on typical transformations used by metamorphic malware explained in detail in Section 2.1, whereas the use case is a real program. In addition to that, a program that generates programs that may or may not contain malicious code fragments has been used to test MMDect's detection capability with non-crafted examples. More details about the examples can be found in Section 5.1.

The analysis of the results of the cases is displayed in Section 5.2, and the limitations detected by these tests are described in Section 5.3.

5.1 Examples

The first four examples are crafted using popular transformations used by malware. These transformations are explained in detail in Section 2.1. Each subsection from Subsection 5.1.1 to 5.1.4 is dedicated to one transformation. Subsection 5.1.5 is destined for the real use case used to test the tool against a real program. Finally, a program that generates random files to test the tool was used, as shown in Subsection 5.1.6.

5.1.1 Basic rule appliance

This example tests the capacity to apply rules and generate metamorphic programs, reversing the transformation consisting in replacing sets of instructions with equivalent ones.

The input program will be basic_rule_appliance.txt. To pass this test, the program should detect a signature made of the first four lines of code after applying a rule on "pop r12; push r12".

```
mov(mem('123'),imm(_)),
push(mem('123')),
push(imm(_)),
mov(reg(r13),imm(_))
```

5.1.2 Comments and fake jumps

This example tests the effectiveness of the tool against comments in the code and simple fake jumps. In addition to detecting malware even if comments in random lines of code were added, it also includes the transformation technique of adding fake jumps. A fake jump consists of adding an if-else where the if and else clauses do the same thing. For example, in the following instructions, if r13 contains the value 10, it will jump to the “start_code” tag, but if it is not, it will jump to that tag anyway.

```
cmp r13, 10
je start_code
jne start_code
```

The input program containing the fake jump and some comments included in the code will be comments.txt The signature to detect this program will be the same input but without comments and a direct jump.

5.1.3 Immediates

Some metamorphic malware changes unimportant immediates or strings to fool analyzers. A similar technique is to change the format used, for example, changing the format of a number from decimal to hexadecimal. The goal of this test is to prove the ability to detect a program without depending on a specific number or format. The input program will be immediates.txt. The signature used to test will be the same program, but changing the format of some numbers and using variables instead of constants in others.

5.1.4 Additional jumps

This case tests the capacity of the tool to detect malware that has used additional jumps to avoid being detected. The input program will be additional_jumps.txt, while the signature to detect it is additional_jumps.prologsign, made from the original program without jumps.

5.1.5 Real use case

The use case defined is a program or artifact that prints a prefixed file, in this case, “/etc/passwd”, to stdout, with the privileges of the program or user that executes it. Because of this last feature, it can be used to compromise the confidentiality of a file with privilege escalation in combination with other elements.

In this case, three signatures are defined. The first one is positive when a program prints to standard output:

```
mov(reg(rdx), reg(rax)),
mov(reg(Reg),imm('0x1')),
mov(reg(rdi),imm('0x1')),
mov(reg(rsi),reg(rsp)),
mov(reg(rax),imm('0x1')),
syscall
```

The second one, when a program puts “/etc/passwd” in the stack:

```
mov(reg(_Reg),imm('0x6477737361702FFF')),
shr(reg(_Reg),imm('8')),
push(reg(_Reg)),
mov(reg(_Reg),imm('0xFFFFFFFF6374652F')),
shl(reg(_Reg),imm('32')),
push(reg(_Reg))
```

The third is the same as the second, but tests for variable lines:

```
mov(reg(_Reg),imm('0x6477737361702FFF')),
_,
push(reg(_Reg)),
mov(reg(_Reg),imm('0xFFFFFFFF6374652F')),
_,
push(reg(_Reg))
```

The three signatures have their equivalent in Regex format in the corresponding directory.

5.1.6 Randomly generated programs

Furthermore, apart from the aforementioned tests, a program was created to generate randomized tests. This program (test_generator.py) can be found in the examples directory of the tool. These test programs consisted of a permutation of common code segments and malicious ones, with their order randomized.

Table 5.1: Comparison between Python and Prolog

	Time(ms)		# of versions		Detection	
	Python	Prolog	Python	Prolog	Python	Prolog
Basic rule appliance	**151**	206	1	**6**	No	**Yes**
Comments and fake jumps	**125**	166	1	**2**	Yes	Yes
Immediates	164	**150**	1	1	Yes	Yes
Additional jumps	150	**148**	1	1	No	No
Real use case	**153**	496	1	**32**	1/3	**Yes**

5.2 Analysis of the Evaluation

After testing the five cases, the tool only fails in both Prolog and Python executions in the fourth case, as an additional technique is needed to detect this type of change (see the last column in Table 5.1). This technique is explained in more detail in Section 6.

In addition, Table 5.1 shows that the Python version could not detect the first example, nor the signatures related to *etc/passwd*. The Prolog version reports similar execution times to Python, considering that the former generates more versions. We can observe this fact in the scenarios where only one version is generated (examples Immediates, and Additional jumps).

After conducting extensive testing using various randomly generated programs with the method shown in Subsection 5.1.6, all the obtained results aligned with the aforementioned scenarios. Not a single false positive was produced.

5.2.1 Achievements

The results of the tests performed indicate that the tool has successfully identified most of the malware. Our proposed objectives have been accomplished, leading to the successful development of a heuristic static analyzer capable of detecting metamorphic malware.

This work significantly enhances the traditional antivirus model by enabling the detection of malicious programs, even in the presence of various types of transformations. It represents an overall improvement in this field.

Yet, even after obtaining such optimistic results, there are also new opportunities for improvement, which are discussed in Section 6.1.

5.3 Limitations

The tests carried out allowed us not only to confirm the effectiveness of the tool but also to identify areas for improvement and devise solutions to turn those deficiencies into improvements. The limitations mentioned below can be remedied by developing new techniques and combining the results with existing tools, thus following a philosophy of continuous improvement.

5.3.1 Static analysis and zero-days

As with any signature-based detection, to be capable of recognizing malware it must have a signature for it. A zero-day threat does not have a signature yet, so it is not possible to detect this type of attack. In order to do so, it is necessary to have AI or behavioral-based detection, among other alternatives. It is also necessary to have a broad set of known rules. Even though, this last limitation can be circumvented by the use of MetaSign (Campion, Dalla Preda, and Giacobazzi 2021), as the goal of this work was to know and learn new rules.

5.3.2 Code lines reordering and additional jumps

MMDect was not capable of detecting malware in the case explained in Subsection 5.1.4. This type of transformation would require a very different approach. The technique needed would involve identifying and reordering blocks of codes, together with the elimination of unnecessary jumps after this process. Anyhow, a module that performs this task can be developed and implemented in the tool to increase its detection rate.

Chapter 6

Conclusions

Although malware is a more significant menace than ever, the same impetus continues to be given to research and development of new techniques to counteract it. Individually, all methods have weaknesses, but the combination of their strengths can become a difficult obstacle for malicious actors to circumvent.

We have developed a tool that can be used to detect metamorphic malware using static analysis, which can be the perfect complement to more costly techniques. This approach has all the advantages of traditional static analysis, as it does not require prior training, it does not need to execute potential malware and it is cost-friendly. Moreover, it also covers some deficiencies of traditional heuristic analysis, as it can detect malicious programs even if they have used a diverse group of mutations.

A high-level approach to the design of the tool can be found in Chapter 3, while some insight into its implementation is detailed in Chapter 4.

To test the extent to which it was able to detect these mutations, MMDect was tested with four different scenarios, a real case, and randomly generated programs. This evaluation can be found in Chapter 5. The results were very positive, as it can detect not only metamorphic rule-based mutations but also certain types of fake jumps, comment variations, and immediate format changes. In the following section, it is discussed some lines of research that would improve even more the results obtained.

6.1 Future work

MMDect is a proof of concept and, of course, can be further developed. Nevertheless, this work opens new lines of research and we propose to implement the following functionalities to improve it.

First, code lines reordering to capture additional jumps. This functionality overcomes the limitation mentioned in the example with additional jumps (Section 5.1.4). It is recommended to implement a technique that reorganizes the lines of code in such a way that it does not affect the overall operation of the program. To do this, it is necessary to take into account the state of the variables (created, accessed, modified, among others) joined with the call graph and its relation with the jumps performed. After this process, it is possible to detect which jumps are unnecessary and remove them.

Second, combine MetaSign capacity of learning rules and MMDect detection. As these features would reduce the dependency on known rules, it would be a good idea to add an intermediate plugin that translates the final result of MetaSign executions into rules expressed in a format that can be read by MMDect (either in Prolog or in Regex format), combining the functionality of learning new metamorphic rules programmed in MetaSign with the detection method used by MMDect.

Third, an interesting option would be integrating a decompiler technology in the first step of the tool, adding the possibility of receiving as input an executable file instead of an Intel assembly code.

Finally, we would like to improve efficiency by, for example, implementing parallelism techniques to improve tool execution times.

Bibliography

Campion, M., Dalla Preda, M., and Giacobazzi, R. (2020). **LabSPY-UNIVR/MetaSign: MetaSign - metamorphic engine, widening CFG, Lerning rewriting rules**. URL: https://github.com/LabSPY-univr/MetaSign.

Campion, M., Dalla Preda, M., and Giacobazzi, R. (2021). **Learning metamorphic malware signatures from samples**. In: *Journal of Computer Virology and Hacking Techniques*, pp. 1–17.

Christodorescu, M. and Jha, S. (2003). **Static analysis of executables to detect malicious patterns**. In: *12th USENIX Security Symposium (USENIX Security 03)*.

Jordan, A., Gauthier, F., Hassanshahi, B., and Zhao, D. (2018). **Safe-pdf: Robust detection of javascript pdf malware using abstract interpretation**. In: *arXiv preprint arXiv:1810.12490*.

Lee, H., Won, S., Jin, J., Cho, J., and Ryu, S. (2012). **SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript**. In: *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, p. 96.

Quertier, T., Marais, B., Morucci, S., and Fournel, B. (2022). **MERLIN–Malware Evasion with Reinforcement LearnINg**. In: *arXiv preprint arXiv:2203.12980*.

Roberts, C. (2022). **How to bypass anti-virus to run Mimikatz**. URL: https://www.blackhillsinfosec.com/bypass-anti-virus-run-mimikatz.

Singh, J. and Singh, J. (2021). **A survey on machine learning-based malware detection in executable files**. In: *Journal of Systems Architecture* 112, p. 101861.

SWI Prolog (n.d.). **SWI Prolog official website**. URL: https://www.swi-prolog.org/.

Tekol, Y. (2007). **Pyswip**. URL: https://pypi.org/project/pyswip/.

Appendix A

Appendix

```
1  instruction([C]) -->
2  command(C).
3  instruction([C|[A]]) -->
4  command(C), arguments(A).
5  argument(A) --> register(A),!.
6  argument(A) --> memory(A),!.
7  argument(A) --> immediate(A),!.
8  argument(A) --> tags(A).
9  arguments([A]) --> argument(A).
10 arguments([A|As]) --> argument(A), arguments(As).
11 command(add) --> ["add"].
12 command(tag(A)) --> [T],
13    { nonvar(A), atom(A), atom_string(A,T1), string_concat(T1,":",T) }.
14 command(tag(A)) --> [T],
15    { nonvar(T), string(T), string_concat(T1,":",T), atom_string(A,T1) }.
```

Figure A.1: DCG grammar corresponding to Intel syntax

```
%[["i", "a1", "a2"],["i","a"],["i"]] --> [i(a1,a2), i(a), i]
parse(List, Program) :-
  nonvar(List), to_atoms(List, Flat), to_functors(Flat,Program), !.
%[["i", "a1", "a2"]["i","a"]] <-- [i(a1,a2), i(a)]
parse(List, Program) :-
  nonvar(Program), to_functors(Flat, Program), to_atoms(List, Flat), !.
% [["i", "a1", "a2"]["i","a"]] <--> [[i,[a1,a2]], [i,[a]]]
to_atoms([], []).
to_atoms([S|Ss], [A|As]) :- phrase(instruction(A), S), to_atoms(Ss, As).
% [[i, [a1, a2]], [i, [a]] <--> [i(a1,a2), i(a)]
to_functors([], []).
to_functors([X|Xs], [Y|Ys]) :- to_functor(X, Y), to_functors(Xs, Ys).
% [i, [a1, a2]] <--> i(a1,a2)
to_functor([],[]).
to_functor([tag(T)], tag(T)):- !.
to_functor([X|Xa], Y) :- [A] = Xa, [Xa1, Xa2] = A, Y =..[X, Xa1, Xa2].
to_functor([X|Xa], Y) :- [A] = Xa, [Xa1] = A, Y =..[X, Xa1].
to_functor(X, Y) :- Y =..X.
```

Figure A.2: Code converted to a matrix

```
instruction([C]) -->
command(C).
instruction([C|[A]]) -->
command(C), arguments(A).
argument(A) --> register(A),!.
argument(A) --> memory(A),!.
argument(A) --> immediate(A),!.
argument(A) --> tags(A).
arguments([A]) --> argument(A).
arguments([A|As]) --> argument(A), arguments(As).
command(add) --> ["add"].
command(tag(A)) --> [T],
   { nonvar(A), atom(A), atom_string(A,T1), string_concat(T1,":",T) }.
command(tag(A)) --> [T],
   { nonvar(T), string(T), string_concat(T1,":",T), atom_string(A,T1) }.
```

Figure A.3: DCG grammar corresponding to Intel syntax

```
%[["i", "a1", "a2"],["i","a"],["i"]] --> [i(a1,a2), i(a), i]
parse(List, Program) :-
  nonvar(List), to_atoms(List, Flat), to_functors(Flat,Program), !.
%[["i", "a1", "a2"]["i","a"]] <-- [i(a1,a2), i(a)]
parse(List, Program) :-
  nonvar(Program), to_functors(Flat, Program), to_atoms(List, Flat), !.
% [["i", "a1", "a2"]["i","a"]] <--> [[i,[a1,a2]], [i,[a]]]
to_atoms([], []).
to_atoms([S|Ss], [A|As]) :- phrase(instruction(A), S), to_atoms(Ss, As).
% [[i, [a1, a2]], [i, [a]] <--> [i(a1,a2), i(a)]
to_functors([], []).
to_functors([X|Xs], [Y|Ys]) :- to_functor(X, Y), to_functors(Xs, Ys).
% [i, [a1, a2]] <--> i(a1,a2)
to_functor([],[]).
to_functor([tag(T)], tag(T)):- !.
to_functor([X|Xa], Y) :- [A] = Xa, [Xa1, Xa2] = A, Y =..[X, Xa1, Xa2].
to_functor([X|Xa], Y) :- [A] = Xa, [Xa1] = A, Y =..[X, Xa1].
to_functor(X, Y) :- Y =..X.
```

Figure A.4: Code converted to a matrix