MMDect: Metamorphic Malware Detection Using Logic Programming

Luciana C. Fidilio-Allende and Joaquín Arias

Universidad Rey Juan Carlos, Móstoles, Spain

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, e.g., by improving signature-based static analysis to detect or classify malware based on patterns of code. However, current tools based on static analysis are vulnerable to the code changes used in metamorphic malware.
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 and a (real) use case. 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.

1 Introduction

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.

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 modifying comments, the file's name, or function names [7].

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 [8].

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 [6].

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 [4], based on Scalable Analysis Framework for ECMAScript [5], or the experimental tool SAFE [3], 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 malware can change its code, making it impossible to detect with heuristic analysis.

A more recent application of abstract interpretation to metamorphic malware, in this case, focused on the infer of their signatures and transformations, is MetaSign [2]. This solution uses the same transformations the malware uses to change its code to get all possible iterations of this software. If the malware has used the same set of rules to mutate as the tool, this can generate the original code giving it as input one iteration of it. However, this project didn't provide a direct implementation of software analysis.

We have created a tool, MMDect, which applies two separate steps, namely generation and comparison, to establish the practicality of utilizing metamorphic transformations for software analysis. 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`.

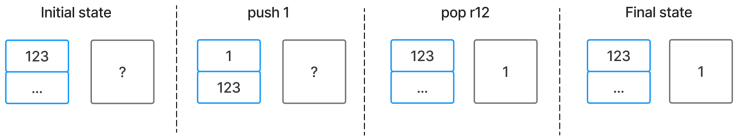

Fig. 1: Behavior of instructions $push(1); pop(12);$.

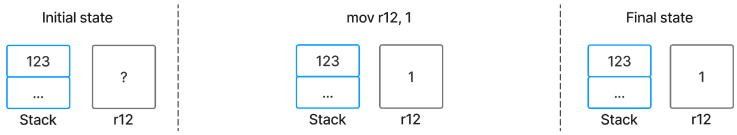

Fig. 2: Behavior of instruction $mov(r12, 1);$.

In Section 1 it is explained the context and aims of this project, along with research in a similar area, including the work on which this technique is based. Section 2 explains the theory on which MMDect is based more in-depth. Section 3 presents an extensive description of the implementation of the tool. Finally, Section 4 details the evaluation process, by testing MDDect with four specific scenarios and a real case.

2 Background: Metamorphic rules as functors

Metamorphic malware, like any malware, is a program with potentially malicious intentions. However, the distinction between metamorphic malware and regular malware is that to fool 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.

To make these changes, it 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. 1 shows the behavior of the instructions on the left, $push(1); pop(r12);$, while Fig. 2 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.

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 project, we have treated metamorphic rules as functions, in the case of Python, and as methods in the case of Prolog.

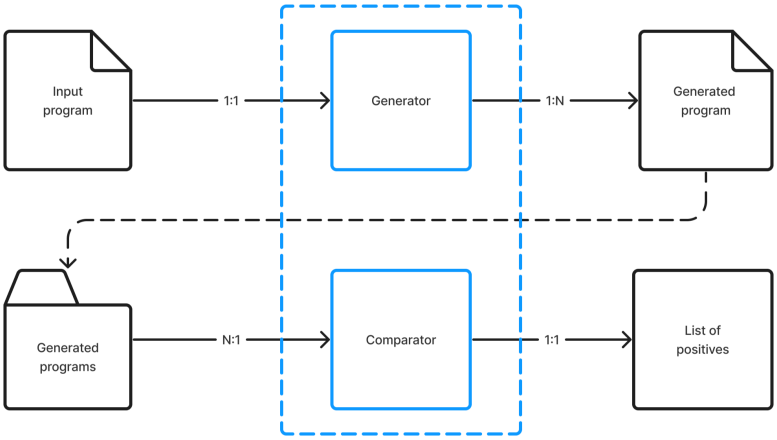

Fig. 3: General structure of inputs and outputs

3 Description of MMDect

This Section describes the three basic steps of MMDect's functionality. First, Subsection 3.1 explains how the original Intel syntax of the suspicious file is translated to the format used by the tool. Subsection 3.2 discusses the generation of new metamorphic versions. Finally, Subsection 3.3 covers matching all generated versions plus the original file against stored signatures.

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, 15 of which are inherited from MetaSign [2]. 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.

The defined set of rules can be found in the file rules.txt in the tool's repository, where the ones with names starting with "G" are inherited from MetaSign [2] and the rest are added to highlight the expressiveness of MMDect.

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.

- 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 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.

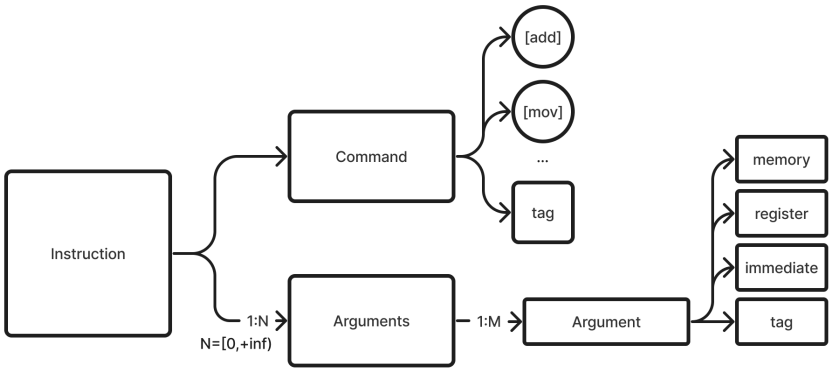

Fig. 4: Instruction grammar

Table 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'	

Input and output are done via Python and for requests to Prolog, the pyswip library is used [10].

3.1 Step 1: Translation from Intel syntax

To handle the code more comfortably, we translate the original Intel syntax into an intermediate syntax.

A code in Intel can be described using the structure described in Fig. 4:

The intermediate language removes all comments in the code, isolates the header (all code before the "_start:" tag such as .data), and converts all instructions into a matrix. This format is directly adapted from the MetaSign project [2]. This code can be found in its GitHub repository [1].

This matrix is composed of a list of lists of a command + argument 1 + argument 2. For example, Table 1 shows the matrix corresponding to the code listed below that is used under Python implementation:

```
# Intro code
.global _start
.text
_start:
mov r12, 0x6477737361702FFF
# Comment
push [123]
push 12
pop r12 # Comment
```

```
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) }.
```

Fig. 5: DCG grammar corresponding to Intel syntax

On the other hand, under Prolog it will suffer additional conversions:

- 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.

DCG, or Definite Clause Grammars, is a set of context-free grammars that can be executed. In this case, the instruction grammar is composed of a command and arguments, simulating the general structure of the original assembly language that was described in Fig. 4.

This structure is translated into the code in Fig. 5 and then, the code in Fig. 6 invokes the grammar:

- First, in lines 1–6, we call the parse method, 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” method, defined in lines 7–9.
- In lines 10–12, the method involving the second step is defined, i.e. to convert the matrix into functors employing the “to_functors” method.

```
%[["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.
```

Fig. 6: Code converted to a matrix

– The method "to_functors" manages the matrix and calls the method "to_functor" (lines 13–18). The last method converts a list defining an instruction and its arguments into a functor, using one of the three methods depending on the number of arguments of the instruction.

3.2 Step 2: Generate new metamorphic programs

This step 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. 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):

```
["mov", "[123]", "0x6477737361702FFF"],
["push", "[123]"],
["push", "12"],
["pop", "r12"],
["push", "r12"],
["mov", "r13", "13"],
["mov", "r12", "0xFFFFFFFF6374652F"],
["xor", "rax", "rax"]
```

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

```
["push", "7239381865414537215"],
["mov", "r12", "12"],
["push", "r12"],
["mov", "r13", "13"],
["mov", "r12", "184467440710831567 83"],
["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, $push(12); pop(r12) \Longleftrightarrow mov(r12, 12)$, MMDect applies the following transformation:

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

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

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

Note that one of the possible variations is the one generated by the implementation under Python. Generation is handled by the generate_program function in modeHandler.py. This function will redirect the generation to Prolog or Python relevant methods as specified.

Rule definition A rule in Python is defined in about ten to twenty complex lines divided into two functions depending on its complexity. However, rules in Prolog are defined in a maximum of four simple lines, and only one line if they do not involve operation instructions.

In Python, 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. 7.

In Prolog, a rule is defined in the following way:

```
rule(g1, [push(Imm),pop(Reg)], [mov(Reg,Imm)]) :-
   imm(Imm), reg(Reg).

rule(g7, [mov(mem(Mem), imm(Imm)), Opi], [Opo]) :-
   operation(Opi, 0, [reg(Reg),mem(Mem)]),
   operation(Opo, 0, [reg(Reg),imm(Imm)]).
```

[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.

```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
```

Fig. 7: Code of rules under Python

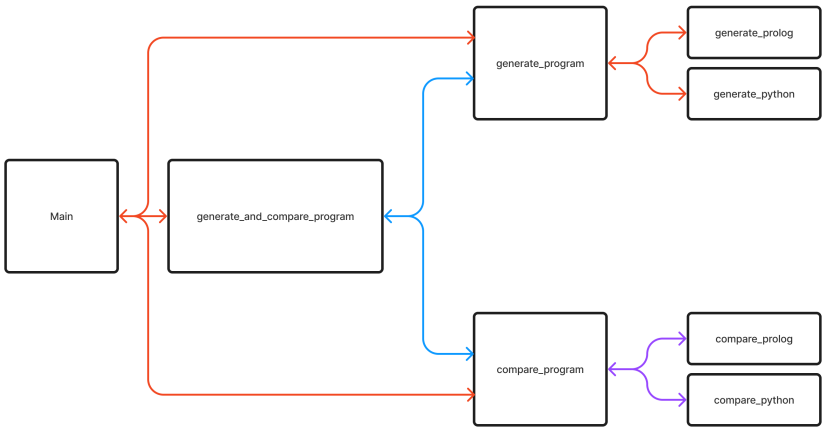

Fig. 8: Function calls

where the first argument is its name, the second is the left side of the rule, and the last argument is the right side.

3.3 Step 3: Detect 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:

```
.*('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))
```

Writing the rules in regex 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.

However, 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 the situation.

The comparison is centralized in the compare_program function, which works similarly to the compress function.

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

Fig. 8 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 Evaluation

To install the Python version, it is needed *Python3* 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 Python* [9] 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/`).

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, whereas the use case is a real program.

Example 1: Basic rule appliance

This example tests the capacity to apply rules and generate metamorphic programs. 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 $popr12; pushr12$.

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

Example 2: Comments and fake jumps

This example tests the effectiveness of the tool against comments in the code and simple fake jumps. A fake jump consists of adding an if-else where the if

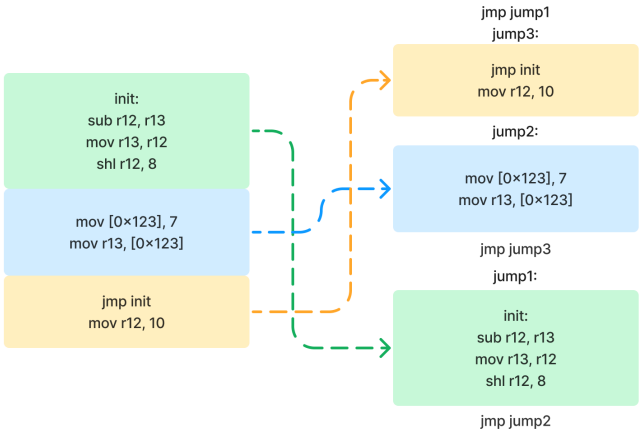

Fig. 9: Additional jumps transformation

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 doesn't, 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.

Example 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 actual values in others.

Example 4: Additional jumps

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. 9. 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.

Table 2: 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**

Real use case

The use case is available at https://github.com/Lu-all/MMDect/blob/master/examples/example_programs/uc_passwddump.txt. This program or artifact 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.

4.1 Analysis of the Evaluation

After testing the five cases, the tool (under Prolog) only fails in the fourth case, as an additional technique is needed to detect this type of change (see last column in Table 2). This technique is explained in more detail in Section 5.

In addition, Table 2 shows that the Python version could not detect the first example, nor the signatures related to *etc/passwd*. The Prolog version reports similarly equal 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).

Achievements

The results of the tests performed indicate that it is possible to use metamorphic rules to detect metamorphic malware. Moreover, it can be a perfect complement to other types of analysis, as well as a complete replacement for classic signature-based analysis.

Limitations

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.

Code lines reordering and additional jumps As we can see in the previous results, detecting this type of transformation would need a different technique, which is contemplated as future work.

Rules in Python An interesting idea is to implement the functionality achieved in Prolog in Python. One way to achieve this is to implement a Breadth/Depth First Search algorithm, which applies or ignores applicable rules at each step until the end of the program is reached or all signatures have been detected. However, this approach would add too much complexity, so other routes of investigation are recommended.

5 Conclusions

Although malware is a bigger 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.

To test the extent to which it was able to detect these mutations, MMDect was tested with four different scenarios and a real case. 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. However, MMDect, like any other solution, can be further developed.

This work opens new lines of research and we propose to implement the following functionalities to improve MMDect.

- BFS/DFS algorithm in Python: Applying rules as possible actions and signatures as goals in graph search algorithms.
- Code lines reordering and additional jumps limitations: To overcome these limitations, 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, ...) joined with the call graph and its relation with the jumps performed.

References

1. Campion, M., Dalla Preda, M., Giacobazzi, R.: Labspy-univr/metasign: Metasign - metamorphic engine, widening cfg, lerning rewriting rules (Aug 2020), https://github.com/LabSPY-univr/MetaSign
2. Campion, M., Dalla Preda, M., Giacobazzi, R.: Learning metamorphic malware signatures from samples. Journal of Computer Virology and Hacking Techniques pp. 1–17 (2021)
3. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th USENIX Security Symposium (USENIX Security 03) (2003)
4. Jordan, A., Gauthier, F., Hassanshahi, B., Zhao, D.: Safe-pdf: Robust detection of javascript pdf malware using abstract interpretation. arXiv preprint arXiv:1810.12490 (2018)
5. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In: FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages. p. 96. Citeseer (2012)
6. Quertier, T., Marais, B., Morucci, S., Fournel, B.: Merlin–malware evasion with reinforcement learning. arXiv preprint arXiv:2203.12980 (2022)
7. Roberts, C.: How to bypass anti-virus to run mimikatz (Nov 2022), https://www.blackhillsinfosec.com/bypass-anti-virus-run-mimikatz
8. Singh, J., Singh, J.: A survey on machine learning-based malware detection in executable files. Journal of Systems Architecture **112**, 101861 (2021)
9. SWI Prolog: Swi prolog official website, https://www.swi-prolog.org/
10. Tekol, Y.: Pyswip (May 2007), https://pypi.org/project/pyswip/