Learning Metamorphic Rules from Widening Control Flow Graphs

Marco Campion
Dipartimento di Informatica
University of Verona
Verona, Italy
email:marco.campion@univr.it

Mila Dalla Preda
Dipartimento di Informatica
University of Verona
Verona, Italy
email:mila.dallapreda@univr.it

Roberto Giacobazzi
Dipartimento di Informatica
University of Verona
Verona, Italy
email:roberto.giacobazzi@univr.it

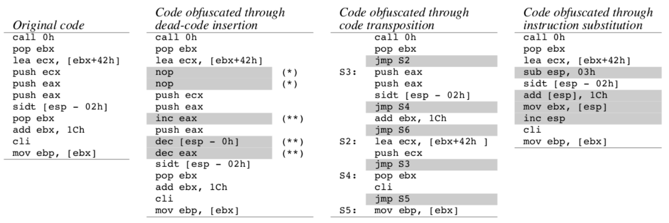
***Abstract*—Metamorphic malware are self-modifying programs which apply semantic preserving transformation rules to their own code in order to foil detection systems based on signature matching. Thus, a metamorphic malware is a malware equipped with a metamorphic engine that takes the malware, or parts of it, as input and morphs it at runtime to a syntactically different but semantically equivalent variant. Examples of code transformation rules used by the metamorphic engine are: dead code insertion, registers swap and substitution of small sequences of instructions with semantically equivalent ones. With the term metamorphic signature, we refer to an abstract program representation that ideally captures all the possible code variants that might be generated during the execution of a metamorphic program. In this paper, we consider the problem of automatically extracting metamorphic signatures from the analysis of metamorphic malware variants. For this purpose, we developed *MetaWDN*, a tool which takes as input a collection of simplified metamorphic code variants and extracts their control flow graphs. *MetaWDN* uses these graphs to build an approximated automaton, which over-approximates the considered code variants. Learning techniques are then applied in order to extract the code transformation rules used by the metamorphic engine to generate the considered code variants.**

***Keywords*—Static binary analysis; Metamorphic malware detection; Program semantics; Widening automata; Learning grammars.**

I. Introduction

Detecting and neutralizing computer malware, such as worms, viruses, trojans, and spyware is a major challenge in modern computer security, involving both sophisticated intrusion detection strategies and advanced code manipulation tools and methods. Traditional misuse malware detectors (also known as signature-based detectors) are typically syntactic in nature: they use pattern matching to compare the byte sequence comprising the body of the malware against a signature database [1]. Malware writers have responded by using a variety of techniques in order to avoid detection: encryption, oligomorphism with mutational decryption patterns, and polymorphism with different encryption methods for generating an endless sequence of decryption patterns are typical strategies for achieving malware diversification.

Metamorphism emerged in the last decade as an effective alternative strategy to foil misuse malware detectors. Metamorphic malware are self-modifying programs which iteratively apply code transformation rules that preserve the semantics of programs. These code transformations change the syntax of code in order to foil detection systems based on signature matching. These programs are equipped with a metamorphic engine that usually represents the 90% of the whole program code. This engine takes as input the malware and its own code and it produces at run time a syntactically different but semantically equivalent program. We call metamorphic variant the program variants generated by the metamorphic engine. At the assembly level these semantic preserving transformation include: semantic-nop/junk insertion, code permutation, register swap and substitution of equivalent sequences of instructions [2] (see Figure 1).



Original code
```
call 0h
pop ebx
lea ecx, [ebx+42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code obfuscated through dead-code insertion
```
call 0h
pop ebx
lea ecx, [ebx+42h]
nop                   (*)
nop                   (*)
push ecx
push eax
inc eax               (**)
push eax
dec [esp - 0h]        (**)
dec eax               (**)
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

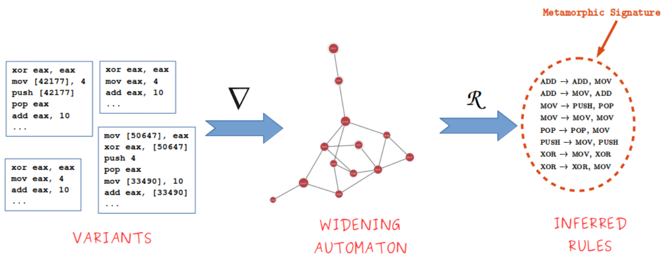
Code obfuscated through code transposition
```
      call 0h
      pop ebx
      jmp S2
S3:   push eax
      push eax
      sidt [esp - 02h]
      jmp S4
      add ebx, 1Ch
      jmp S6
S2:   lea ecx, [ebx+42h ]
      push ecx
      jmp S3
S4:   pop ebx
      cli
      jmp S5
S5:   mov ebp, [ebx]
```

Code obfuscated through instruction substitution
```
call 0h
pop ebx
lea ecx, [ebx+42h]
sub esp, 03h
sidt [esp - 02h]
add [esp], 1Ch
mov ebx, [esp]
inc esp
cli
mov ebp, [ebx]
```

Figure 1. Examples of semantic preserving rules transformation.

The large amount of possible metamorphic variants makes it impractical to maintain a signature set that is large enough to cover most or all of these variants, thus making standard signature-based detection ineffective. Heuristic techniques, on the other side, may be prone to false positives or false negatives. The key to identify these type of malicious programs consists in considering semantic program features and not purely syntactic program features, thus capturing code mutations while preserving the semantic intent [6]. For this reason, we would like to capture those semantic aspects that allow us to detect all the possible variants that can be generated by the metamorphic engine. We use the term metamorphic signature to refer to an abstract program representation that ideally captures all the possible code variants that might be generated during the execution of a metamorphic program. A metamorphic signature is therefore any (possibly decidable) approximation of the properties of code evolution.

The goal of this work is to statically extract a so called metamorphic signature, i.e., a signature of the metamorphic engine itself. In this setting, a metamorphic signature consists of a set of rewriting rules that the malware can use to change its code. These rules are represented as a pure context-free grammar in which each instruction is a terminal symbol and can be transformed into equivalent instructions following a production of the grammar. For this purpose, we built a tool, called *MetaWDN*, that takes as input simplified versions of the metamorphic code variants, embeds them in an over-approximating control flow graph (*widening*) and finally, it tries to *learn* from the control flow graph the rewriting rules used to generate each variant. The general structure of the tool is represented in Figure 2.



```
xor eax, eax
mov [42177], 4
push [42177]
pop eax
add eax, 10
...
```

```
xor eax, eax
mov eax, 4
add eax, 10
...
```

```
xor eax, eax
mov eax, 4
add eax, 10
...
```

```
mov [50647], eax
xor eax, [50647]
push 4
pop eax
mov [33490], 10
add eax, [33490]
...
```

```
ADD → ADD, MOV
ADD → MOV, ADD
MOV → PUSH, POP
MOV → MOV, MOV
POP → POP, MOV
PUSH → MOV, PUSH
XOR → MOV, XOR
XOR → XOR, MOV
```

Figure 2. Capturing the metamorphic signature.

In order to test the quality of the output on portions of code that are actual metamorphic variants of the same program, we have implemented a metamorphic engine. Our metamorphic engine takes as inputs a program written in an intermediate language very similar to the `x86` assembly and it randomly chooses the rewriting rules to apply in order to generate the metamorphic variants. The metamorphic rules implemented are a subset of ones used by the metamorphic malware MetaPHOR [14]. The metamorphic engine allows us to quickly generate numerous test sets, input them to the tool and check the quality of the results by comparing the rules inferred with those actually applied by the metamorphic engine.

The rest of this paper is organized as follows: in Section II we discuss some related work, Section III explains how the tool can be executed and how it works, in Section IV we present some results and consideration applied to one example and finally the paper ends with conclusion and future work in Section V.

II. Related Work

In [3] the authors propose a malware detector scheme based on the detection of suspicious system call sequences. In particular, they consider only a reduction (subgraph) of the control flow graph of the program, which contains only the nodes that represent certain system calls and finally, they check if this subgraph has some known malicious system call sequences.

In [8] the authors describe a system of malware detection based on containment and unification of languages. The malicious code and the possible infected program are modeled as an automaton with unresolved symbols and placeholders for registers dealing with certain types of obfuscation. In this configuration, a program exhibits malicious behavior if the intersection between the malware's automaton language and the one of the program is not empty.

In [4] the authors specify malicious behavior through a Linear Temporal Logic (LTL) formula and then use the SPIN model checker to check if this property is satisfied by the control flow graph of a suspicious program.

In [5] the authors introduce a new Computation Tree Predicate Logic (CTPL) temporal logic, which is an extension of the logic CTL, which takes into account the quantification of the registers, allowing a natural presentation of malicious patterns.

In [9] they describe a malicious behavior model through a template, that is a generalization of the malicious code that expresses the malicious intent excluding the details of the implementation. The idea is that the template does not distinguish between irrelevant variants of the same malware obtained through obfuscation processes. For example, a template will use symbolic variables / constants to handle the renaming of variables and registers, and will be related to the malware control flow graph in order to handle code reordering. Finally, they propose an algorithm that checks if a program presents the behavior as a template, using a process of unification between the variables / constants of the program and the symbolic variables / constants of the malware.

In [7] Dalla Preda et al. consider the problem of automatically extracting metamorphic signatures from metamorphic code. They introduced a semantic for self-modifying code, called phase semantics, and prove its correctness by proving that it is an abstract interpretation of standard trace semantics. Phase semantics precisely models the metamorphic behavior of the code, providing a set of program traces which correspond to the possible evolution of the metamorphic code during execution. They therefore demonstrate that metamorphic signatures can be automatically extracted by abstract interpretation of phase semantics. In particular, they introduce the notion of regular metamorphism, in which the invariants of phase semantics can be modeled as a Finite State Automata (FSA) representing the code structure of all possible metamorphic changes of a metamorphic code.

In [10], the authors propose to model the behavior of a metamorphic engine of a malicious program, with rewriting systems also called term-rewriting systems and to formalize the problem of constructing a normalizer for rewrite systems (called NCP) that is able to reduce to the same normal form, variants of malware generated by the same metamorphic engine. From this problem, they propose a possible solution by building a normalizer on a set of rules that maintain three properties: termination, confluence and preservation of equivalence.

All these approaches provide a model of the metamorphic behavior that is based on the knowledge of the metamorphic transformations, i.e., obfuscations, that malware typically use. By knowing how the code mutates, it is possible to specify suitable (semantics-based) equivalence relations which trace code evolution and detect malware. This knowledge is typically the result of a time and cost consuming tracking analysis, based on emulation and heuristics, which requires intensive human interaction in order to achieve an abstract specification of code features that are common to the malware variants obtained through various obfuscations and mutations.

In this paper, we aim at defining an automatic technique for the extraction of a metamorphic signature that does not need any a priori knowledge of the code transformation rules used by the metamorphic engine.

III. *MetaWDN* Tool

MetaWDN is a program written in *Python 3* language that allows us to automatically generate a set of variants starting from a given input program. Next, *MetaWDN* compacts them all together through the widening operator and then it tries to automatically derive the rewriting rules used to generate them. Depending on the execution parameters, the tool can be executed in one of the following ways (Figure 3):

- execution of the metamorphic engine to generate a desired number of variants starting from a set of instructions (which will be the starting program) written on an input text file (①);
- computing the widening between a set of variants given as inputs in order to build an unique abstract representation of the considered metamorphic variants (②);
- inferring the rewriting rules from the program representation obtained through the widening process (② → ③);
- finally, you can run all the operation above (① → ② → ③).

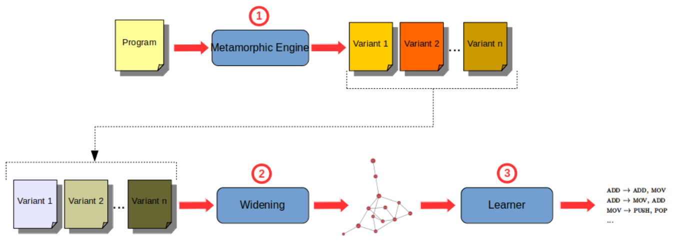


Figure 3. Phase of execution of *MetaWDN*.

The tool takes as input programs written in an intermediate language very similar to the language used by MetaPHOR [14], both with the aim of simplifying and abstracting the `x86` assembly language. Therefore, the input is an extremely simplified version compared to the code that can be found in any executable. You can use the classic instructions of the `x86` assembly code with *Intel* syntax like: data manipulation (`mov, push, pop, lea`), mathematical expressions (`add, sub, and, xor, or`), jumps (`je, jne, jl, jle, jg, jge, jmp, call`), etc. There are three kinds of operands: registers (`eax, ebx, ecx, edx, esp, ebp, esi, edi`), immediate values and memory values (decimal number or register between square brackets, for example `[77382]`). For jump instructions, the memory value to which the instruction can jump corresponds to the line number where the target instruction is located (the first line starts from zero). Analogously, for function calls we have that in the instruction `call` the value of the operand corresponds to the line number of the first statement of the function. Each function (including function `main`) must end with the instruction `ret`.

A. The Metamorphic Engine

The tool can be executed as a metamorphic engine: it takes as input a text file containing a program written in the `x86` intermediate language and the number of variants to be generated. The implemented rewriting rules are instructions transformation that preserve the semantics, e.g., `mov` → `push, pop` which expands the instruction `mov` in two instructions `push` and `pop`. A rewriting rule could be applied either in expansion (following the rule from left to right) or in reduction (right to left). After reading the file, the metamorphic engine randomly selects: the rewriting rule to apply, the line of the program where to apply the rule, and whether to apply the rule as expansion or reduction. If it is not possible to apply the rewriting rule to the selected instruction, the following instruction is considered and if it is not possible to apply the rule to the whole file then another rewriting rule is selected randomly. The implemented rewriting rules are a subset of the rules used by the `MetaPHOR` metamorphic engine [14].

B. Widening Control Flow Graphs

Each metamorphic variant is represented as a Control Flow Graph (CFG). Each node of the CFG contains one instruction that is abstracted according to an abstraction function that removes details usually modified by the metamorphic transformations. In particular, *MetaWDN* abstracts instructions by eliminating the operands, so, e.g., the instruction `mov eax, 4` is abstracted in `mov`. In the CFG representation of programs the vertices contain the instructions to be executed, and the edges represent possible control flow. For our purposes, it is convenient to consider a dual representation where vertices correspond to program locations and abstract instructions label edges. The resulting representation is isomorphic to FSA over an alphabet of instructions [13]. For this reason we use the terms CFG and automaton interchangeably. In order to compact the CFG of the metamorphic variants into an unique representation we use a widening operator. This allows us to obtain an unique representation that contains all the seen metamorphic varinats but that also generalizes the considered mutations. Given the equivalence between CFG and FSA, we can use the widening operator for FSA defined in [13]. To this end, we have to to compute the language of each node of the CFG. According to [13], we define the language of length `N` of a node of a CFG as the set of all the strings of length less or equal than `N` that are reachable from the considered node.

Example III.1. Consider the following program `P`, where the numbers on the left correspond to line numbers:

```
0: mov eax, 1      4: jmp 1
1: cmp eax, 1000   5: ret
2: jge 5           6: add eax, 1
3: call 6          7: ret
```

the CFG is represented in Figure 4. The alphabet of the

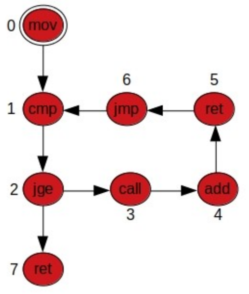


Figure 4. CFG of Example III.1.

CFG of P is {mov, cmp, jge, call, jmp, add, ret}, and the language of length 2 recognized by the nodes is:

```
lang(0) = {(mov),(mov,cmp)}
lang(1) = {(cmp),(cmp,jge)}
lang(2) = {(jge),(jge,call),(jge,ret)}
lang(3) = {(call),(call,add)}
lang(4) = {(add),(add,ret)}
lang(5) = {(ret),(ret,jmp)}
lang(6) = {(jmp),(jmp,cmp)}
lang(7) = {(ret)}
```

Consider a set of code variants $V_1V_2\dots V_n$ generated from the initial program P. The widening operator $\bigtriangledown$ is defined as:

$$W_0 = \alpha(P) \qquad W_{i+1} = W_i \bigtriangledown_k (W_i \cup \alpha(V_i))$$

where W_i with $i \geqslant 0$ is the widening CFG at step i (the initial widening W_0 is the CFG of the program itself), α is the abstraction function that eliminates the operands of instructions, and k is the length of the language of nodes. Briefly, the widening operator merges all the nodes with the same language of length k.

C. Learning Rewriting Rules

The section of the tool that infers the transformation rules is called learner. The learning algorithm implemented in *MetaWDN* is a simplified version of the algorithm proposed in [16] for learning pure grammars from a set of words. The general problem of inferring rewriting rules from a set of positive examples, i.e., from a positive set, can be transformed into the general problem of inferring a grammar starting from a set of strings belonging to a language. In particular, we try to infer a grammar that is able to generate at least all the strings given as input to the algorithm and belonging to the language to be studied. In our case, the language to be learned includes all the possible variants generated by the unknown metamorphic engine, while the grammar we want to infer corresponds to the set of rewriting rules used by the metamorphic engine to generate the metamorphic variants given in input to the positive set. Pure grammars [11] have been chosen as a formal representation for the rewriting rules, because they do not present terminal symbols but all the symbols are considered as non-terminals. In fact, the metamorphic transformation rules are all instructions of the same type, that is, they can be transformed into other instructions by applying the correct production. More in details, since the general problem of learning pure grammars from a positive set is undecidable [12], we move to the formalism of k-uniform pure context-free grammars [11] where, each production has the left part with one letter of the alphabet while the right part has at most k symbols of the alphabet. All these restrictions, of course, will lead to a loss of precision in the rules inferred by the tool as it will only be possible to infer productions of the form $\{x \to y \mid |x| = 1, |y| \leqslant k\}$. In our learning algorithm, the constant k is always set to 2 since the rewriting rules implemented in the tool have the right part of length 2. The learning algorithm takes a CFG as input and operates in three phases:

1. it builds the positive set;
2. it learns the rewriting rules;
3. finally, it eliminates the spurious inferred rules.

The positive set consists of a set of code variants where all the instructions are abstracted (no operands). This set is built in the widening phase and will be the input for inferring the rewriting rules. The length min of the smallest variant is calculated, i.e., the variant with the fewest instructions. Then, all the paths of length min of the graph that go from a root node (the first instruction of a variant, those drawn with the double circle) to the final node (the ret instruction) are visited. For each path found, the set of instructions related to the visited nodes are inserted in the positive set. During this process every time that we visit an edge we mark it. When the path of length min has been found, if all the edges are marked then the search is interrupted without visiting other paths. Otherwise the variable min is incremented.

Given a couple of code variants (V_i, V_j) with $|V_i| < |V_j|$, the idea of the learning algorithm is to add a production rule r of the form $V_i \xrightarrow{r} V_j$. The rewriting rule r is inferred through simplification rules between the two variants (V_i, V_j). There are three kinds of simplification rules:

- top simplification: compare the first instruction of V_i and V_j and delete them if they are the same. This process continues until two different instructions are encountered: in this case, if $|V_i| > 1$ then the comparison restarts from the last instruction of V_i and V_j, otherwise ($|V_i| = 1$) the rule is added to the set of inferred rules;
- bottom simplification: it is similar to the previous one, but starts from the last instruction;
- top and bottom simplification: compare the first instruction of V_i and V_j and, if they are equal, it deletes them and starts again but from the bottom instruction of V_i and V_j.

The algorithm applies the top simplification repeatedly until a rule is added to the set of inferred rules and then it starts back with bottom simplification and finally, with top and bottom simplification.

Example III.2. Let us consider the following simple code variants: $xor, mov, push$ and $xor, push, pop, push$. After applying two times the top simplification we get

$$\cancel{\text{xor}},\text{mov},\cancel{\text{push}} \to \cancel{\text{xor}},\text{push},\text{pop},\cancel{\text{push}}$$

Since the left part is of length 1 then the rule mov $\to$ push,pop is added to the set of inferred rules. With the other two kinds of simplification we get the same rewriting rule.

After the simplification phase, the algorithm has produced a set of rewriting rules of the form: $\{x \to y \mid |x| = 1, |y| \leqslant k\}$. However, most of these rules are superfluous since they can be generated by other rules of the set. The elimination algorithm tries to reduce the right part of each rewriting rule by applying all rewriting rules inferred in the reduction form (from right to left). If at the end of this procedure, the rule is reduced to another rule already in the inferred set, then that rule can be eliminated.

Example III.3. Let us suppose that there are two rewriting rules inferred by the learning algorithm:

$$1)\ \text{mov} \to \text{mov},\text{mov}$$

$$2)\ \text{mov} \to \text{push},\text{pop}$$

Now suppose that the following rewriting rule is produced: mov $\to$ push,pop,mov,mov. This rule is spurious since:

$$\text{push},\text{pop},\underline{\text{mov}},\underline{\text{mov}} \overset{1)}{\Rightarrow} \underline{\text{push}},\underline{\text{pop}},\text{mov} \overset{2)}{\Rightarrow} \text{mov},\text{mov}$$

IV. Case Studies

In the following section, we present some results and considerations applied to a program of 21 instructions:

```
0: mov [ebp], [esp]    11: mov eax, ebx
1: sub ebp, 4          12: push eax
2: push 100            13: pop [440303]
3: pop ecx             14: pop [443905]
4: cmp eax, exc        15: xor eax, 0
5: xor eax, 0          16: xor eax, eax
6: test eax, eax       17: nop
7: mov eax, 4          18: test eax, 0
8: sub eax, 1          19: xor eax, 0
9: cmp eax, ebx        20: ret
10: nop
```

We have used *MetaWDN* to generate 50 variants of this program. Next we have randomly selected a subset of 25 code variants that are obtained by applying all the rewriting rules implemented in *MetaWDN*. This subset is provided as input to the widening process (with language length sets to 2) and next to the learning process. The final graph of the widening is shown in the Figure 5. The rewriting rules inferred by the tool is the empty set. This looks like a mistake, however, by looking more carefully at the possible paths of the graph we observe that all paths from any root node to the `ret` node, starting from the minimum length (the smallest variant in terms of instructions), are already visited. For this reason, the set of positive examples contains all code variants of the same length and therefore it is not possible to infer any rewriting rule. This result is caused by the numerous spurious variants inserted by the widening process that agglomerates the nodes with the same language of length 2. In fact, due to

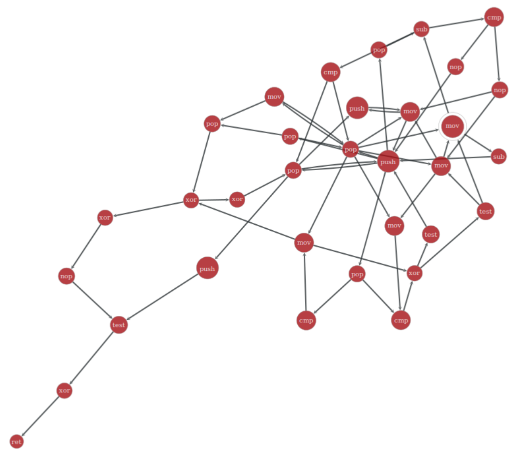


Figure 5. Graph obtained by the widening operator with length sets to 2.

the numerous cycles, i.e., regularities inserted by the widening, a path from the root to the end node of length less than any true variant is "increased" until reaching the minimum length (in this example equal to 20) thus creating a spurious variant.

If we increase the level of precision of the widening by setting the parameter of the language length to 3, we obtain the graph in Figure 6 with the following rewriting rules inferred:

```
cmp -> ['cmp', 'mov']      mov -> ['push', 'mov']
mov -> ['push', 'pop']     mov -> ['mov', 'push']
mov -> ['pop', 'mov']      nop -> ['pop', 'push']
nop -> ['pop', 'mov']      nop -> ['nop', 'mov']
pop -> ['pop', 'push']     pop -> ['pop', 'mov']
pop -> ['mov', 'pop']      pop -> ['nop', 'mov']
push -> ['mov', 'push']    sub -> ['mov', 'sub']
test -> ['test', 'mov']    xor -> ['mov', 'xor']
```

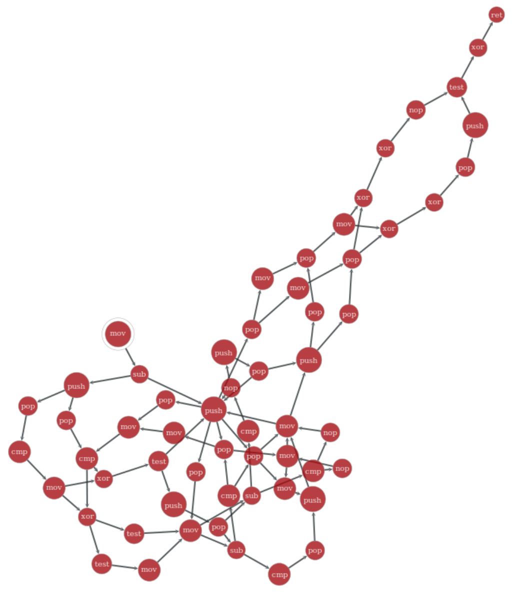


Figure 6. Graph obtained by the widening operator with length sets to 3.

Clearly, by increasing the length of the widening language we obtain a graph with more nodes but more precise. In fact, in this case it is possible to infer the rewriting rules even if there are numerous spurious rules still due to the presence of spurious paths induced by the widening.

If we increase the level of precision of the widening, setting the length of the language to 4 we obtain the following rewriting rules inferred:

```
cmp -> ['cmp', 'mov']     cmp -> ['mov', 'cmp']
mov -> ['push', 'pop']    mov -> ['mov', 'mov']
nop -> ['nop', 'mov']     nop -> ['pop', 'push']
pop -> ['pop', 'mov']     pop -> ['mov', 'pop']
push -> ['mov', 'push']   sub -> ['mov', 'sub']
test -> ['test', 'mov']   xor -> ['mov', 'xor']
```

Thanks to the greater precision of the widening, this time the inferred rules are more precise and they represent an acceptable result. Moreover, with a language length equal to 5 the same rules are still obtained.

V. Conclusion and Future Work

In this work we tried to capture the behavior of the metamorphic engine itself, namely we tried to find a set of rules that allow us to predict possible mutations of code variants starting from a set of examples. To this end, we presented the tool *MetaWDN* that has three main functions: metamorphic engine, widening of code variants and learning of rewriting rules. Thanks to the metamorphic engine, it is possible to quickly generate numerous variants in an intermediate language similar to `x86`. These variants are created by randomly applying rewriting rules implemented in the tool. The goal is to capture, starting from a subset of these code variants, the rewriting rules used by the metamorphic engine to generate them. Starting from the set of code variants, *MetaWDN* uses a widening operator to generate a graph that approximates all the variants of the set. Rewriting rules are then represented as productions of a k-uniform pure context-free grammar. From the learning algorithm and the elimination of superfluous rewriting rules algorithm, it is possible to obtain a set of rules that describes, in an approximate way, the possible evolution of code variants. The experimental results show us how the choice of the language length parameter of the widening operator affects the precision of the learned rules. The lower the value is, the more the nodes will be joined together because they will be more likely to present the same language. In this case the presence of spurious paths will be higher therefore there will be less precision in the results inferred by the learner. On the contrary, the higher the length of the language is and the greater is the precision of the graph. This means that the widening graph presents fewer spurious paths and therefore it allows us to infer more precise rewriting rules. Of course, the increase in precision comes at a cost in terms of time execution and memory consumption.

As a priority of future work, we will try to apply this tool to a set of real malware variants. In this work only one level of abstraction on the instructions has been considered, that is, the one that does not consider the operands. It would be interesting to consider different abstractions, assigning, for example, to the operands symbolic values such as those of [15]. Finally, an implementation of new rewriting rules in the tool and a new learner should be considered as a future work. The new learner needs to be able to learn, in an approximate way, more complex rewriting rules in order to catch more sophisticated metamorphic engine.

Acknowledgment

This paper has been supported by the grant PRIN2017 (code: 201784YSZ5) by MIUR Italy.

References

[1] P. Szr, "The Art of Computer Virus Research and Defense", Addison-Wesley Professional, Boston, MA, USA, 2005.
[2] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware", IEEE Security and Privacy, vol. 5, no. 2, pp. 4654, 2007.
[3] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs", Symposium on Requirements Engineering for Information Security, vo. 2001, no. 79, pp. 184-189, 2001.
[4] P. Singh, and A. Lakhotia, "Static verification of worm and virus behaviour in binary executables using model checking", IEEE Systems, Man and Cybernetics Society Information Assurance Workshop, pp. 298-300, 2003.
[5] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking", International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 174-187, 2005.
[6] M. Dalla Preda, "The grand challenge in metamorphic analysis", International Conference on Information Systems, Technology and Management, vol. 285, pp. 439-444, 2012.
[7] M. Dalla Preda, R. Giacobazzi, and S. Debray, "Unveiling metamorphism by abstract interpretation of code properties", Theoretical Computer Science, vo. 577, pp. 74-97, 2015.
[8] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns", Symposium on USENIX Security, 2003.
[9] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection", IEEE Symposium on Security and Privacy, pp. 32-46, 2005.
[10] A. Walestein, R. Mathur, M. R. Chouchane, and A. Lakhotia, "Constructing malware normalizers using term rewriting", Journal in Computer Virology, vo. 4, no. 4, pp. 307-322, 2008.
[11] H. A. Maurer, A. Salornaa, and D. Wood, "Pure grammars", Inform. Control, vo. 44, pp. 47-72, 1980.
[12] T. Koshiba, E. Mkinen, and Y. Takada, "Inferring pure context-free languages from positive data", Journal in Acta Cybernetica, vo. 14, no. 3, pp. 469-477, 2000.
[13] V. D'Silva, "Widening for automata", Diploma thesis, Institut Fur Informatick, Universitat Zurich, 2006.
[14] P. Beaucamps, "Advanced Metamorphic Techniques in Computer Viruses", International Conference on Computer, Electrical, Systems Science, and Engineering, 2007.
[15] A. Lakhotia, M. Dalla Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic Juice", In PPREW@ POPL, 2013.
[16] C. Higuera, "Grammatical inference: learning automata and grammars", Cambridge University Press, 2010.