Metamorphic Malware Detection Using Code Metrics

Gerardo Canfora\textsuperscript{a}, Francesco Mercaldo\textsuperscript{a}, Corrado Aaron Visaggio\textsuperscript{a} & Paolo Di Notte\textsuperscript{a}

\textsuperscript{a} Department of Engineering, University of Sannio, Benevento, Italy

Published online: 16 Jul 2014.


To link to this article: http://dx.doi.org/10.1080/19393555.2014.931487
Metamorphic Malware Detection Using Code Metrics

Gerardo Canfora, Francesco Mercaldo, Corrado Aaron Visaggio, and Paolo Di Notte
Department of Engineering, University of Sannio, Benevento, Italy

ABSTRACT Malware is becoming more and more aggressive and new techniques are emerging to allow malicious code to evade detection by antiviruses. Metamorphic malware is a particularly insidious kind of virus that changes its form at each infection. In this article, a technique for detecting metamorphic viruses is proposed that is based on identifying specific features of the assembly code, such as the instructions that change the contents of the registers, the instructions that change the control flow, and the potential code fragmentation. Such features have been derived by the analysis of a large dataset of malware. The experimentation suggests that the proposed technique produces very high precision (over 97%) in recognizing metamorphic malware, and allows also for distinguishing among different families of malware.

KEYWORDS malware detection, metamorphic virus, static analysis, testing

1. INTRODUCTION

Malware is increasingly becoming more aggressive, as reports illustrate. Symantec analysts demonstrate that a malware called Reveton (aka Trojan.Ransomlock.G) infected 500,000 computers over a period of 18 days. The average loss per cyber crime incident is $197. In the last 12 months, it was estimated that 556 million users worldwide experienced some form of cyber crime (Security Response, 2013).

In 2011, customers reported more than 49,000 different threat families to the Microsoft Malware Protection Center. Many of these reported families were polymorphic versions of key threat families (Microsoft Security, 2012).

Metamorphic malware (Choucane, 2006) is a category of malicious programs particularly hard to detect, as it changes its form as it propagates: two replications of the same metamorphic virus will never have the same shape. Metamorphic viruses are born as an evolution of polymorphic viruses, to overcome the main limitation of this kind of virus. The polymorphic virus code must reside in memory to be executed; thus, an antivirus can detect a polymorphic virus by waiting for it to decipher and comparing its footprint with the ones contained in a database. The metamorphic malware mutates its code, producing different signatures of the same virus at each infection, in order to evade antiviruses. Metamorphic malware may also recompile itself using the same compiler of the system, thus allowing for integration of the virus into trusted code, which makes it even more difficult to identify. Currently, there is a strong interest in studying effective techniques for detecting metamorphic malware.
Canfora, Iannaccone, and Visaggio (2013) demonstrated that op-code occurrences could be effective to detect metamorphic malware. Relying on this study, we have implemented new heuristics that focus on specific forms of occurrence of specific op-codes, which are presented in this article.

Our method for the detection of metamorphic malware consists of extracting four metric-indexes, obtained through the static analysis of the assembly file of the candidate malware. The first metric computes the ratio between push and pop instructions, and the second one computes the ratio between the number of consecutive push instructions and the total number of push instructions. The third metric analyzes all the instructions that modify the control flow that contain a register whose value is assigned by the sequence of push and pop. The underlying idea is that virus writers and virus generators intensively use push/pop couples to modify the body of the malware code in the metamorphic virus, as these instructions help to hide the target of a jump or procedure call. The fourth metric counts the instructions that modify the sequential flow of control: a typical method to generate replications of a metamorphic malware is to decompose the control flow in smaller procedures and call them in a certain sequence instead of executing the original control flow. This technique is known as “fragmentation.” The assumption is that, in a trusted program, the number of times the procedures are recalled is smaller than in the metamorphic malware, but it shows a number of procedures that is, on average, greater than the number of procedures of a malware program.

This article investigates two research questions:

- RQ1: Is it possible to detect metamorphic malware through a classification based on these four metrics?
- RQ2: Is it possible to distinguish between different families of malware through a classification based on these four metrics?

The remaining of the paper is organized as follows: related work is discussed in next section, section 3 presents the proposed metrics for malware detection, section 4 discusses the experimentation, and conclusion and future works are given in the last section.

2. RELATED WORK

Techniques to detect metamorphic viruses rely on static or dynamic analysis.

A virus that uses register swapping and instruction replacement can be detected by wildcards and half-byte scanning (Al Daoud, Jebril, & Zaqaibeh, 2008). This technique is effective only for simple metamorphic virus. Geometric detection is based on alterations of the file structure done by a virus (Konstantinou & Woltshusen, 2008; Šzör & Ferrie, n.d.). This method is prone to false positives (Šzör, 2005), whose rate can be reduced if applied in conjunction with virus infection markers.

Konstantinou and Woltshusen (2008) introduce a technique that checks if the disassembly code can be retrieved in existing virus disassembly codes. The technique is fast, but it has the limitations of a blacklist approach: what is known can be blocked, and what is unknown is not blocked.

Combining disassembling of the virus code with a finite state machine can be a powerful solution since it allows recording the order in which “suspicious” instructions are encountered. This technique is effective with obfuscated code, and its performances increase when combined with an emulator (Šzör & Ferrie, n.d.; Šzör, 2005). Emulators simulate the CPU and memory management system and use a virtual machine for executing malicious code (Konstantinou & Wolthusen, 2008; Šzör, 2005).

Several authors use heuristics for detecting virus. These techniques extract features from the code of the virus and run classification algorithms (Schmall, n.d.). Unfortunately, virus writers incorporate several anti-emulation techniques that allow the virus to escape the detection; metamorphic virus is one of the most widespread techniques. Another common anti-emulation technique consists of randomly inserting garbage instructions and dummy loops. This kind of anti-emulation technique leads the emulator to generate a high rate of false positives (Fiñones & Fernandez, 2006; Konstantinou & Wolthusen, 2008; Šzör & Ferrie, n.d.).

Christodorescu and Jha (2003) propose to use ALCFG (construction of Arbitrary Length of Control Flow Graph) matrices that provide a quantitative description of a program control flow graph. The authors compute a similarity metric to evaluate to which extent the pairs of metamorphic viruses belonging to the same sequence of infection are similar.

Wong and Stamp (2006) propose a technique based on Hidden Markov Models (HMM). The method can distinguish a virus generated with the Next Generation Virus Creation Kit from trusted programs, despite the fact that NGVCK virus shows a high degree of metamorphism. Toderici and Stamp (2013) combine HMM
Metamorphic Malware Detection

3. METRICS FOR MALWARE DETECTION

Our method to detect polymorphic viruses consists of extracting four metrics from the assembly code of the candidate malware and using them for program classification. The metrics have been defined by considering two aspects that are peculiar of a metamorphic malware:

- the fragmentation of the code, obtained with unconditional or conditional jumps; and
- the use of the instructions for handling the stack.

Trusted programs usually use these instructions for various purposes, such as preserving the content of the registers among procedure calls, or passing parameters to a procedure. The malware uses these instructions for obfuscating the business logic, to hide a portion of the code that is contaminated, or to hide values in the stack that will be resumed when needed.

The average distribution of assembly instructions in the trusted source code, as emerged from our previous investigation (Canfora et al., 2013), is different from the average distribution of the same assembly instructions in metamorphic malware. These findings were obtained by comparing the occurrence frequency of a subset of op-codes in trusted malware and in metamorphic virus. The metamorphic sample was generated with four virus generator engines, namely G2, Mpcgen, Ngvck, and Nrlg (Runwal et al., 2012; Saleh et al., 2011; Wong & Stamp, 2006; Sridhara & Stamp, 2013; Toderici & Stamp, 2013). Using the differences of instruction distributions, we defined a method for distinguishing: metamorphic viruses from no-malware programs; metamorphic viruses from no-metamorphic malware; no-malware from no-metamorphic malware; and no malware from any kind of malware used in the experimentation. In our investigation, we observed that push and pop instructions occur in trusted programs a smaller number of times than in malware. The average distributions in the malware generated with the Ngvck engine and with the Mpcgen engine are not the same. While in the Ngvck sample the number of push is equal to the number of pop, in
the Mpcgen sample the number of push is slightly different from the number of pop. Finally, the trusted programs show that the count of push instructions is significantly different from the count of pop instructions.

These considerations have motivated the definition of the four metrics derived by analyzing the number of push and pop instructions.

**Push/Pop Ratio:** this index computes the ratio between push and pop instructions. The formula is:

\[
\text{Push/Pop Ratio} = \frac{\text{Total Pop Number}}{\text{Total Push Number}}
\]

As an example, in the following snippet, the push/pop ratio is 1:

**Snippet 1:**
1. push cs
2. pop ax
3. dec ax
4. mov es,ax
5. mov bx,es:[3]
6. push cs
7. pop es

**Percentage of Consecutive Push:** the second index computes the ratio between consecutive push instructions before a pop and the total number of push instructions:

\[
\text{Percentage Consecutive Push} = \frac{\text{Number Consecutive Push}}{\text{Total Push Number}}
\]

Here, the term “consecutive” does not mean close push instructions with respect to all the other instructions, but only with respect to the pop instruction:

**Snippet 2:**
1. push eax
2. push esp
3. push eax
4. push ebx
5. push edi
6. call ebp
7. pop eax
8. popa
9. lea eax,[esp-80h]
10. loc_425E50
11. push 0
12. cmp esp,eax
13. jnz short loc_425E50
14. sub esp,0FFFFFF80h

**Snippet 3:**
1. push cs
2. pop ds

Let us consider the code snippets 2 and 3; the first is an example of malware generated by the Nrlg engine (Wong & Stamp, 2006), while the second is a trusted executable. Instructions 1 and 11 in snippet 2 and instruction 1 and 13 in snippet 3 indicates the first push, and instructions 2, 3, 4, and 5 indicate the consecutive push instructions, whose counting finishes when the presence of a pop is detected, which is instruction 7.

In code snippet 2, five consecutive push instructions are counted, while in the snippet 3 the metric is zero because the first push (instructions 1 and 13) is always followed by a pop instruction (instructions 2 and 14).

**Push/Pop in Jumps Sequence:** this index considers all the instructions that modify the control flow, i.e. unconditional and conditional jumps or procedure calls. The index considers the instructions that contain in the argument a register whose value is assigned by a push/pop sequence:

\[
\text{Control Flow Ratio} = \frac{\text{change influenced by Push/Pop}}{\text{total number Control Flow changes}}
\]

Let us consider the code snippet 4 generated by the G2 engine (Wong & Stamp, 2006):

**Snippet 4:**
1. mov di,offset encryptbuffer
2. mov si, offset ENCRYPT
3. push si
4. rep movsw
5. mov ax,offset endercrypt-encrypt+encryptbuffer
6. mov word ptr ds:[patchstart+1],ax
7. pop si
8. push offset endercrypt
9. mov byte ptr[offset endercrypt], 00C3h
10. xor byte ptr[offset xorpatch-encrypt+encryptbuffer], 0028h
11. push bx
12. call si

There are the shift instructions of register si on the stack in instructions 3 and 7. The register is then restored through
a pop. Subsequently, the register is directly used as the argument of the call instruction (instruction 12).

Potential Average Fragmentation of the Code: fragmentation refers to the use of instructions that modify the normal sequence flow of instructions; the term potential is used because it is not possible to know exactly how many times the jump and call instructions are executed using static analysis.

Considered the large number of procedures defined in the trusted file, the number of times in which they are called is much lower if compared to what happens in malware, where the number of procedures is significantly lower. The analysis considers the number of times in which each label is used by these instructions, which might alter the control flow; dividing by the number of lines of code normalizes these values. Subsequently we compute the size of each procedure, as the number of lines of code, and this value is normalized on the total number of procedures in the code:

\[
\text{Potential Average Code Fragmentation} = \sum_{i=1}^{N} \left( \frac{\text{calls to procedure } i - \text{th}}{\text{lines code number}} \right) \times \left( \frac{\text{number of lines of code for procedure } i - \text{th}}{\text{number of Procedure file}} \right)
\]

where \(i\) is the \(i\)-th procedure identified in the code, and \(\text{calls to procedure } i\)-th is the counting of all the instructions that change the control flow by targeting the \(i\)-th procedure.

4. EXPERIMENTATION

The aim of the study discussed in this section is to verify whether the four indexes are able to identify an application as trusted or malicious. The identification is carried out through classification algorithms. The primary objective is the recognition of a malware program; the information collected by the metrics will be also used to identify the family of the infection.

The study has three stages: (1) a comparison of descriptive statistics of the populations of programs, (2) hypothesis testing, in order to evaluate if the four indexes have different distributions for the populations of malware and trusted programs, and (3) a classification analysis aimed at assessing whether the indexes are able to correctly classify malware and trusted sources.

The data set consisted of 400 trusted programs and 400 metamorphic malware generated with four different Virus Generator Kits: G2, Ngvck, Mpcgen, and Nrlg. Two control groups were used. The first control group included popular trusted programs. The second control group was generated by mixing the malware codes of the data set with a percentage of trusted code equal to 0 (only malware code), 10%, 20%, and 40%. A code-cleaning phase was needed because the assembly files generated from these tools may contain comments, blanks and very different formatting styles.

Hypothesis testing was aimed at rejecting the following null hypothesis:

\[H_0: \text{malware and trusted sources have similar values of the four indexes.}\]

The null hypothesis was tested with Mann-Whitney (with the p-level was fixed to 0.05) and with Kolmogorov-Smirnov Test (with the p-level fixed to 0.01). Two different tests have been used in order to enforce the conclusions validity. As a matter of fact, we assume valid the results when the null hypothesis is rejected by both the tests.

Six algorithms of classification were used: J48, LadTree, NBTree, RandomForest, RandomTree, and RepTree. These algorithms were applied to the four indexes both for distinguishing trusted programs from malware and for distinguishing trusted programs from malware integrated with the a portion of trusted code. The classification analysis was accomplished by using Weka (University of Waikato, n.d.).

4.1. Data Analysis

The analysis of data comprises three stages: (1) a comparison of descriptive statistics, (2) hypothesis testing, and (3) a classification analysis. The next subsections discuss the results of each stage of analysis.

4.1.1. Descriptive Statistics

Comparing the descriptive statistics (Figure 1) suggests that the index push/pop is the most effective for the distinction between malware and trusted sources. The distribution of malware is in fact well separated from the one of trusted sources, in particular the malware generated by G2, Mpcgen and Nrlg differ greatly with respect to trusted sources. The Ngvck family represents the only exception, as it is different from the other families but also from the trusted program.

The analysis of the distributions of consecutive push index (Figure 2) shows a separation that is not as clear as...
for the push/pop index, although it shows a strong separation between infected and harmless sources. In fact the distributions of all the families of metamorphic malware are much lower than the distribution of trusted programs.

The push/pop in jumps sequences index (Figure 3) shows a greater separation for the identification of malware generated by G2, Mpcgen, and Ngvck. In fact, unlike G2 files, the malwares generated by Mpcgen, do not include in their structure the sequence of push/pop used to modify the control flow. As a result, while G2 files have values much greater than the one of trusted programs, Nrlg values are not very different from the values of trusted sources.

The medians of Nrlg and trusted files (Figure 3) are very close to each other, but they are distant from those of the other families; this happens because the values of Mpcgen have null contribution as this family viruses do not usually have in their structure the analyzed sequence.

Potential code fragmentation index (Figure 4) shows a clear distinction between harmless sources and malware, although some outliers make higher the averages. In particular, while all the metamorphic samples show a similar number of outliers, this does not happen with the trusted sample. It emerged that trusted code shows a very varying levels of fragmentation, as the corresponding box plot in Figure 4 demonstrates. The box-plots shows how trusted sources differ from the malicious ones. In particular, the medians of the sources generated by G2 and Mpcgen are distinct from the trusted sources.
Remark 1: The distributions of the four indexes for the populations analyzed seem to be very different, thus the four indexes could be used for discriminating a malware application from a trusted application.

4.1.2. Hypothesis Testing

The qualitative results are confirmed by the results of the hypothesis testing. For each metric (Push/Pop, Consecutive Push, Push/Pop jump sequence, and Code Fragmentation), we tested the $H_0$ hypothesis for comparing the following pairs of samples: G2-Mpcgen; G2-Ngvck; G2-Nrlg; G2-Trusted; Mpcgen-Ngvck; Mpcgen-Nrlg; Mpcgen-Trusted; Ngvck-Nrlg; Ngvck-Trusted; Nrlg-Trusted; Malware-Trusted.

The hypothesis $H_0$ can be rejected for the four indexes analyzed in every considered data set, when the populations are malware and trusted. This means that there is statistical evidence that these indexes are potential candidates for correctly classifying malware and trusted applications. The null hypothesis cannot be rejected for some cases when comparing malware of different families. These cases have been reported in the Tables 1 and 2, which show only the
samples that have values exceeding a p-level of 0.05 for test of Mann-Whitney and at a p-level greater than 0.01 with regard to the Kolmogorov test.

For instance, H₀ cannot be rejected for the Push/Pop index, when comparing G2 family and MpcGen family with 10% and 20% of trusted code, when running the Mann-Whitney test, as the p-values are, respectively, 0.33 and 0.92, i.e., greater than 0.05.

Remark 2: All the metrics are able to distinguish malware from trusted code with statistical significance for both the tests, with the only exception of Code fragmentation (for which only Mann-Whitney test exhibits statistical significance). In several cases it is also possible to distinguish the generator kit used.
4.1.3. Classification Analysis

In the classification phase, we trained a set of classifiers and evaluated their ability to discriminate between trusted and malware. In the training phase, we used a set of labeled applications, \( T = (A_i, l_i) \), where \( A_i \) denotes an application and \( l_i \in \{\text{trusted, malware}\} \) is the label. For each \( A_i \), we built a feature vector \( F \in \mathbb{R}^y \), where \( y \) is the number of the feature used in training phase \((1 \leq y \leq 4)\). In particular, we performed one classification with all the four features \((y = 4)\).

We used k-fold cross-validation: the entire dataset was randomly partitioned into \( k \) subsets. A single subset of data was retained as the validation data for testing the model while the remaining \( k-1 \) subsets of data were used as training data. We repeated the process for \( k \) times, each of the \( k \) subsets of data has been used once as validation data. To obtain a single estimate we caught the average of the \( k \) results from the folds. Specifically, we performed a 10-fold cross validation \((k = 10)\).

The results of the classification analysis are shown in Table 3. Three metrics were used to evaluate the classification results: precision, recall, and ROC area. The recall is computed as the proportion of examples that were assigned to class X among all examples that truly belong to the class, that is, how much part of the class was captured. The precision is computed as the proportion of the examples that truly belong to class X among all those were assigned to the class.

The ROC Area is the area under the ROC curve (AUC); it is defined as the probability that a randomly chosen positive instance is ranked above randomly chosen negative one.

Regarding precision, all the classification algorithms return a value greater than 0.97. With regards to recall, all the algorithms return a value greater than 0.97. With regards to ROC area, all the algorithms return a value greater than 0.99. Precision, recall and ROC area decrease when malware is mixed with portions of trusted code, but in the worst case the values keep next to 0.8.

Remark 3: The classification analysis suggests that the indexes are robust to detect the malware, with any percentage of added trusted code.

In summary, the experiment shows that the four indices could be effective both to distinguishing metamorphic malware from trusted programs and to distinguish different families of metamorphic malware.

<table>
<thead>
<tr>
<th>Classification algorithm</th>
<th>file</th>
<th>Precision</th>
<th>Recall</th>
<th>RocArea</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48 0%</td>
<td></td>
<td>0.990</td>
<td>0.990</td>
<td>0.997</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>0.919</td>
<td>0.919</td>
<td>0.987</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>0.932</td>
<td>0.933</td>
<td>0.960</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>0.838</td>
<td>0.840</td>
<td>0.912</td>
</tr>
<tr>
<td>LADTREE 0%</td>
<td></td>
<td>0.993</td>
<td>0.993</td>
<td>0.999</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>0.923</td>
<td>0.921</td>
<td>0.989</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>0.896</td>
<td>0.896</td>
<td>0.977</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>0.816</td>
<td>0.814</td>
<td>0.959</td>
</tr>
<tr>
<td>RANDOMTREE 0%</td>
<td></td>
<td>0.986</td>
<td>0.986</td>
<td>0.994</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>0.943</td>
<td>0.942</td>
<td>0.962</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>0.916</td>
<td>0.915</td>
<td>0.945</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>0.855</td>
<td>0.855</td>
<td>0.906</td>
</tr>
<tr>
<td>NBTREE 0%</td>
<td></td>
<td>0.995</td>
<td>0.955</td>
<td>0.998</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>0.919</td>
<td>0.919</td>
<td>0.987</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>0.914</td>
<td>0.914</td>
<td>0.976</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>0.852</td>
<td>0.852</td>
<td>0.976</td>
</tr>
<tr>
<td>RANDOMFOREST 0%</td>
<td></td>
<td>0.993</td>
<td>0.993</td>
<td>0.998</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>0.954</td>
<td>0.954</td>
<td>0.993</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>0.940</td>
<td>0.940</td>
<td>0.998</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>0.872</td>
<td>0.872</td>
<td>0.871</td>
</tr>
<tr>
<td>REPTREE 0%</td>
<td></td>
<td>0.989</td>
<td>0.989</td>
<td>0.995</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>0.917</td>
<td>0.915</td>
<td>0.978</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>0.910</td>
<td>0.910</td>
<td>0.968</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>0.823</td>
<td>0.824</td>
<td>0.935</td>
</tr>
</tbody>
</table>

4.2. Experimental Threats

This section discusses the main limitations of our study. The external validity of conclusions should be strengthened by enlarging the experimental and control samples; currently, each one includes 400 units. This threat is mitigated by the fact that each engine produces a “family” of metamorphic viruses; that is, certain design characteristics tend to be common to many codes generated with the same engine. This entails that there is a high probability that even enlarging the size of the sample, the results should remain the same.

Another way to enlarge the sample is to use additional generators. This is a more difficult task, as metamorphic generators are not easily available. However, the four engines used in the experimentation are well representative of the existing engines for generating metamorphic viruses. In fact they are diffusely used in many research projects.
papers (Runwal et al., 2012; Saleh et al., 2011; Wong & Stamp, 2006; Sridhara & Stamp, 2013; Toderici & Stamp, 2013).

We did not identify threats to conclusions validity, as the observed variables exactly represent the object of our observation: in fact, the results of the analysis is binary (the code is malicious or not) and the benign or malicious nature of the analyzed code is clearly and unquestionably determined. Neither did we identify threats to internal validity, as the methods and the software used for the data analysis are widely accepted by the scientific community and largely employed in a number of papers.

Construction validity is not threatened as the independent variables straightforward represent the observed characteristics of the codes under analysis.

5. CONCLUSION AND FUTURE WORK

Current solutions for detecting metamorphic malware are still ineffective. The proposed method extracts four indexes from the assembly code of the candidate program that measure static properties of the code. The method is extremely lightweight. The results obtained show a very high precision in distinguishing metamorphic malware from trusted programs. In addition to the high precision, the virus generator kit of origin is almost always identified. This secondary aspect is not to be underestimated; in fact, the correct identification of the family of viruses allows the use of an adequate countermeasure. Among the algorithms used in the classification accuracy rate and the best results show a precision of 0.99 and a recall of 0.99. The point of strength of the proposed metrics is that they can be obtained straightforward, thus they can be implemented easily into an antimalware system.

The experimentation suggests that the research direction is promising. A possible future development is the expansion of the population including other types of metamorphic malware. Also another development is the integration of the proposed solution in a dynamic system, in order to obtain a hybrid system. In such a solution the dynamic techniques serve for an initial choice of potential partners of infected code, while the proposed solution would focus on detailed analysis of the code identified.

REFERENCES


BIOGRAPHIES

Gerardo Canfora is a full professor of computer science in the Faculty of Engineering and the Director of the Research Centre on Software Technology (RCOST) at the University of Sannio in Benevento, Italy. He serves on the program committees of a number of international conferences. He is a program cochair of the 2015 International Conference on Software Engineering and was a program co-chair of the 1997 International Workshop on Program Comprehension, of the 2001 International Conference on Software Maintenance, and of the 2004 European Conference on Software Maintenance and Reengineering. Also, he was the general chair of the 2003 European Conference on Software Maintenance and Reengineering. His research interests include software maintenance, program comprehension, reverse engineering, workflow management, metrics, and experimental software engineering. He serves on the editorial board of the IEEE Transactions on Software Engineering and the Journal of Software Maintenance and Evolution. He is a member of the IEEE and the IEEE Computer Society.

Francesco Mercaldo received his Master Degree in Computer Engineering from the University of Sannio, with a thesis in software testing entitled “Design and implementation of a system for the identification of anomalies of alteration of databases.” He is currently a student enrolled in the first year of the PhD program in Information Engineering at the Engineering Department of the University of Sannio. Research areas of his PhD will be software testing, verification, and validation, with the emphasis on the application of empirical methods.

Corrado Aaron Visaggio is assistant professor of Database and Software Security in the BsC and MsC in Computer Engineering at the University of Sannio, Italy. He obtained his PhD in Computer Engineering at the Engineering Department of the University of Sannio, Italy. His research interests include empirical software engineering, software security, and data privacy. He is author of about 40 papers published in international journals, international and national conference’s proceedings, and books. He serves in many program committees and editorial boards of international conferences and journals.

Paolo Di Notte has a BSc in Computer Engineering. He is a junior free consultant in security engineering.

Metamorphic Malware Detection