Deobfuscation of Computer Virus Malware Code with Value State Dependence Graph

Ivan Dychka[1[0000-0002-3446-3076]], Ihor Tereikovskyi[1[0000-0003-4621-9668]], Liudmyla Tereikovska[2[0000-0002-8830-0790]], Volodymyr Pogorelov[1[0000-0002-6100-1504]] and Shynar Mussiraliyeva[3[0000-0001-5794-3649]]

[1] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
[2] Kyiv National University of Construction and Architecture, Kyiv, Ukraine
[3] Al-Farabi Kazakh National University, Almaty, Kazakhstan
dychka@scs.ntu-kpi.kiev.ua, terejkowski@ukr.net, tereikovskal@ukr.net, volodymyr.pogorelov@gmail.com, mussiraliyevash@gmail.com

Abstract. This paper deals with improvement of malware protection efficiency. The analysis of applied scientific research on malware protection development has shown that improvement of the methods for deobfuscation of program code being analyzed is one of the main means of increasing efficiency of malware recognition. This paper demonstrates that the main drawback of the modern-day deobfuscation methods is that they are insufficiently adapted to the formalized presentation of the functional semantics of programs being tested. Based on the research results, we suggest that theoretical solutions which have been tried out in program code optimization procedures may be used for code deobfuscation. In the course of the study, we have developed a program code deobfuscation procedure utilizing a value state dependence graph. Utilization of the developed procedure was found to enable presentation of the functional semantics of the programs being tested in a graph form. As the result, identification of malware based on its execution semantics became possible. The paper shows that further research should focus on the development of a method for comparison of the value state dependence graph of the program being tested with corresponding graphs of security software and malware.

Keywords: deobfuscation, value state dependence graph, malware, code optimization.

1 Introduction

The results of research in the field of computer system security [12, 15] indicate that malware protection (MW) has been one of the most important and relevant problems in the field of data security over the last decade. The need to enhance malware protection is confirmed by a great number of well-known examples of computer system infection, which leads not only to the loss of functionality but also to the unauthorised use of the infected systems. For example, virus-infected computer

systems can send spam-messages without authorisation or participate in distributed DDoS attacks. The threat becomes even greater in the light of mainstreamification of web-oriented social networks which require the installation of potentially dangerous specialised software on the client computer to use them. Another aggravating factor is the possibility of computer system infection during a scheduled software update.

Along with that, the analysis of applied scientific research in the field of protection systems development [1–4, 12–15] shows that the main way of increasing their efficiency is to increase malware recognition accuracy. For this, most antivirus protection suites employ cutting-edge solutions in the field of data mining [12, 14, 15]. However, the use of these solutions is seriously complicated because the modern malware is created with extensive use of different techniques of program code distortion, making it impossible to form an input data set for the recognition system. One of the most common distortion techniques is program code obfuscation. By obfuscation, we mean translation of program code to a form, which preserves its functionality but complicates its analysis, understanding of the operation algorithm and modification in the event of decompilation. Therefore, this paper deals with the problematic of obfuscated program recognition for malware recognition.

2 Analysis of literature sources in the field of research

According to [1–4, 9, 10, 12–15], obfuscation is one of the most common techniques of program code protection in legitimate software and is used to prevent its illegal copying. Thus, the existence of obfuscated program code itself is not a sufficient indication of malware. Consequently, it is necessary to develop a deobfuscation procedure which must be executed before the program code is submitted to the recognition system. The main task of this procedure is to translate program code so that it can be examined and its functionality analysed. It is thereby concluded that it is reasonable to adapt the deobfuscation procedure to the common techniques of program code obfuscation: minimisation, meshing, and sealing. It is also determined that the following techniques are mainly used for program code obfuscation in web-oriented software:

- Replacement of carriage return characters with line feed characters.
- Replacement of multiple space characters with one space character.
- Replacement of multiple line feed characters with a line feed character.
- Replacement of comments with a line feed character or space characters.
- Declaration of a set of used variables.
- Call of undefined functions in conditional statements with false conditions.
- Encryption of names of variables and functions.
- Division of encrypted program code into visible and hidden parts.
- JavaScript script packaging into CSS.

In addition, development of deobfuscation methods is declared. For example, the method of minimised JavaScript code deobfuscation includes the following stages:

1. Detection of existing obfuscated JavaScript program code. For this purpose, representative examples of minimiser obfuscators outputs will be used.

2. The start of minimised code review. For this, the minimised code is loaded into a string variable.

3. Division of the script into separate lines. For this, a line feed character is added after each semicolon in the obtained string variables.

4. Identification of names of the JavaScript functions used. For this, each line obtained on the previous stage is scanned. The declared variables found in the obfuscated code are passed to the alert() function to obtain the real names of JavaScript functions.

5. Security analysis of the functions found. It is proposed to use the results given in [10, 12, 15, 16] for this purpose.

The program code obfuscated using the method of executive process meshing, was found to be the most difficult to analyse. The one obfuscated with other methods is quite easy to interpret.

In addition, [1, 13] present the analysis of the main functionality of existing software designed for obfuscation/deobfuscation of web-oriented program code. It is determined that limited functionality of available deobfuscation instruments is primarily due to the imperfection of their mathematical support.

Talking about [1–4, 10, 12–15], one can claim that the result of use of the declared deobfuscation methods fails to reflect the objective of the obfuscated program execution. In other words, it does not reflect the formalised description of the program code execution sequence, which in its turn substantially complicated the analysis of its functional semantics. Consequently, the recognition of destructive properties indicative of malware becomes more complicated too.

[3] shows that obfuscation procedures used to hide malicious code elements employ the same techniques as those designed to protect program code against illegal copying.

Based on the analysis performed, we can claim that the main drawback of the deobfuscation methods available is that they are insufficiently adapted to the formalised presentation of functional semantics of the programs tested. In addition, a specific analogy is pointed out between the procedure of program code deobfuscation and the well-studied procedure for translation of high-level program code into executable code [11]. This suggests a possibility of correcting the mentioned drawback of the well-known deobfuscation methods by means of integrating theoretical solutions used in translator development into them. One of such solutions involves the presentation of program code as a value state dependence graph, which enables formalisation of program execution semantics.

Therefore, the objective of this study is to develop a program code deobfuscation approach utilising a value state dependence graph.

3 Formalization of obfuscation procedure

The logic of obfuscation procedure is to exclude most of the obvious connections from the program code, i.e. to transform the code so as to make investigation and modification of the obfuscated program more complicated and expensive than construction of a new algorithm [1–4]. At the same time, obfuscation procedure must be performed automatically, at a minimum estimated cost.

To provide an accurate definition of obfuscation process, we need to introduce the following terms: initial program code $PR1$, transformation process, $TR()$ and the set of algorithms $PR2_1...PR2_n$ arising as a result of transformation :

$$PR1 => TR(PR1) = \begin{cases} PR2_1 \\ PR2_2 \\ \dots \\ PR2_n \end{cases},$$

In this case, transformation function defines the obfuscation procedure if the following requirements are met: program code $PR2_1...PR2_n$ runs in the same way as program code $PR1$, program code $PR2_1...PR2_n$ is substantially different from program code $PR1$, application of the available reverse engineering algorithms on the program code $PR2_1...PR2_n$ fails, application of the available algorithms for program code $PR2_1...PR2_n$ detransformation into program code $PR1$ fails, each transformation procedure application on program code $PR1$ generates new program code $PR2_1...PR2_n$ with unpredictable structure specifics.

Let us consider the use of the procedure developed and formalise the main types of obfuscation algorithms. We should note that such algorithms are classified into two main groups according to [4, 5]. General (abstract) obfuscation algorithms are those, which are not associated with the specifics of programming language and can be applied even to the assembler code. It is considered more efficient to build the obfuscator based on the abstract algorithm of the procedure which uses all advantages of the specific software code [1–4].

Of abstract obfuscation algorithms, the Collberg's algorithm is the most generic one. While studying the types of obfuscation algorithms, it is reasonable to start with this general scheme and then analyse the methods which can be used during its application.

Execution of Collberg's algorithm can be conventionally divided into four main stages (Fig. 1):

- Loading of program code elements $PR1$
- Loading of libraries
- Cyclic execution of transformation procedure $TR()$ by isolating a code segment, which is repeated until the required level is reached or system resource is exceeded
- Program code generation $PR2_n$

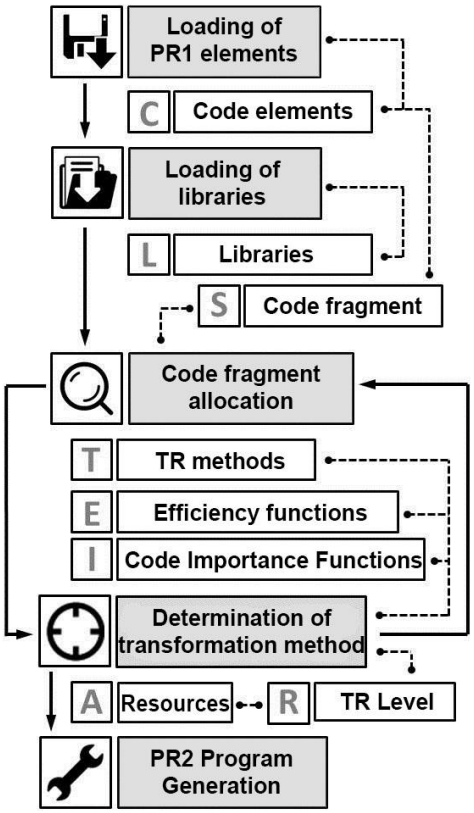

Fig. 1. Pattern of program code obfuscation based on Collberg's algorithm

The input of Collberg's algorithm thus includes:

- Source program code elements $PR1$ (C – Code)
- Standard libraries used in the program code $PR1$
- Methods of program code transformation (T – Transformation)
- The segment of the program code $PR1$ (S – Segment) subject to transformation
- A set of functions that define efficiency (E – Efficiency) of the transformation methods
- A set of functions defining the importance of code segment S
- Maximum acceptable cost of system resources that can be used for obfuscation (A – Accept Cost)
- A parameter indicating the required level of program code segment obfuscation (R – Require Obfuscation)

- Collberg's algorithm is a general pattern of the obfuscation process, while specialised algorithms are defined by obfuscation methods, which can be classified as follows: lexical obfuscation, data obfuscation, control flow obfuscation.

Lexical obfuscation is the simplest type of software protection. It involves code restructuring by deletion or replacement of the comments, deletion of the offsets that are helpful for visual scanning of code, replacement of the identifier (variable, array, function, procedure) names with random character sequences, as well as algorithm block repositioning. Lexical obfuscation enables the transformation of the program code into a form a programmer cannot analyse quickly and at a moderate cost of hardware resources. However, this method grants extremely low resistance against deobfuscation algorithms.

Data obfuscation, which involves the transformation of data structures, belongs to the group of more complex methods. Obfuscation methods can be divided into three subgroups:

The description of obfuscation subgroups shows that this group of methods requires much more hardware resources, but is more resistant to deobfuscation.

Control flow obfuscation is to obfuscate the sequence of program code execution. Algorithms of this method are based on the use of opaque predicates, i.e. predicates $P()$, the results of which are unknown. In this case, a predicate that always returns "true" is designated as $P(t)$, a predicate that always returns "false" is designated as $P(f)$, and a predicate that can return either is designated as $P(t, f)$.

Opaque predicates can be divided into: local, global, interprocedural.

The efficiency of control flow obfuscation algorithms primarily depends on opaque predicates, which must be sufficiently resistant and flexible in use. In terms of hardware requirements, other important parameters are the time of predicate execution and the number of operations performed during its use. Predicate functions, which aim to increase resistance to the static analysis-based deobfuscation algorithms, should be very similar to software functions.

The control flow obfuscation also includes the methods for computational obfuscation. The most efficient algorithm for computational obfuscation is known as the algorithm of cycle condition extension. Just like in the previous case, it is based on an opaque predicate that simulates influence on the number of cycle code executions. Another efficient pattern is the algorithm of library call elimination. If the software uses standard library functions, the operation principle of these program elements will be known, which can help in reverse engineering. Therefore, names of functions from standard libraries are also transformed in the course of obfuscation. One variety of this approach is to use a proprietary version of libraries (built through the transformation of standard libraries) in software. This technique does not affect the program execution time but increases program size significantly.

4 Deobfuscation procedure utilizing a value state dependence graph

Having analysed the available methods of computer virus code deobfuscation, we can claim that deobfuscation procedure is in many ways similar to the procedure of program code optimisation because it often involves incorporation of unnecessary operations and code structure distortion, that does not affect the functionality of the program but hinder the investigation of its operation algorithms. Like deobfuscation, optimisation is aimed at eliminating unnecessary nodes, therefore both can be assigned to the same type of processes on the technical level.

As an internal representation for deobfuscation process we propose to use the Value State Dependence Graph. This graph does not use assignments; the control flow is used only to determine the corresponding operation values, and dependencies are explicit, as well as the conditions for their existence.

In the terms of graph theory, a value state dependence graph (VSDG) can be defined as an oriented designated hierarchical graph $G(T, E, l, S, S_0, S_\infty)$, which consists of functional elements.

These elements include the following:

- Transitions T are the nodes that correspond to operations.
- Places S are the nodes that correspond to the results of operations.
- Edges E the are operation result dependencies.
- Labelling function l corresponds to each branching operation.
- Arguments S_0 indicate the places wherein the function input arguments are located.
- Results S_∞ indicate the places wherein function output is located.

Each place and each graph edge are typeable by value or state. Edge type is defined by endpoints: The state edge is an edge with a state place being its end-point, and the value edge is the one with a value place being its end-point. Transitions represent VSDG operations effected by the labelling function via the corresponding operator. Transition T's input I_T is a place linked to the branch with an edge. A transition may be considered a place consumer.

In a similar way, a place is called transition T's output O_T is a place with an edge leading from the transition thereto. In this case, a transition may be considered a place producer. A set of transition T's inputs is called transition operands or simply inputs, while a set of transition T's outputs is called transition results or outputs OS_T.

While constructing a VSDG for deobfuscation of potentially malicious software (SW) and code optimisation, the following requirements must be fulfilled:

- Acyclicity: VSDG must not use graph theoretical codes
- Node arity: each place must have a unique producer (i.e. a distinct edge $E \in T \times S$ must exist)
- Linear use of states: states must act as consumers not more than once

It is important to note, that VSDG edges must be of the same type, and nodes are described by the following set of simultaneous equations:

$$\begin{cases} IS_N = IS_T \\ OS_N = OS_T \end{cases},$$

Input nodes are subject to additional conditions:

$$\begin{cases} IS_T = \emptyset \\ OS_T = S_0 \end{cases},$$

Similarly, the following conditions are true for output nodes:

$$\begin{cases} OS_T = \emptyset \\ IS_T = S_\infty \end{cases}.$$

Nodes, which are used in VSDG, can be divided into three types:

- Calculation nodes
- γ-nodes
- Complex nodes

Calculation nodes simulate simple low-level operations. In turn, they can be subdivided into the following types:

- Value nodes (contain input and output values without additional action)
- Constant nodes (similar to value nodes, but don't have inputs)

State nodes have mixed inputs and outputs and represent operations as additional actions, such as load or store.

γ-nodes are used to express conditional behaviour in VSDG; they perform multiplexing between two sets of operands t and f, which act as predicate functions, based on input predicate p. Operands of both sets, as well as the result of a γ-node execution, shall be of the same type to perform this operation. Characteristically, γ-nodes are the only type of nodes in VSDG that demonstrate the inconsistent behaviour.

Complex nodes are also called regions. A region contains a distinct graph G' and can be substituted with this graph. Characteristically, this graph can contain its own regions; therefore regions, being a separate type of nodes in VSDG, form hierarchic structures. During code deobfuscation and optimisation, regions may transfer between external and internal regions under certain conditions. However, in this case nesting the property should be kept in mind. A nesting property places a restriction on edges: they must connect nodes only within one region or with a child region. A separate type of complex nodes is θ-nodes. θ-nodes are used on VSDG only for cycles simulation.

It should be pointed out, that a VSDG, to a certain extent, reflects semantic properties of the program being tested, which are related to the use of computational resources of the computer system. Owing to this capability, application of VSDG is promising for semantic analysis of obfuscated software based on the comparison of

the tested program's graph with the corresponding graphs of malware and security software. The development of a relevant comparison method will be tackled in further studies.

5 Conclusion

The analysis of applied scientific research on malware protection development has shown that improvement of the methods for deobfuscation of program code being analysed is one of the main means of increasing efficiency of malware recognition. This paper demonstrates that the main drawback of the modern-day deobfuscation methods is that they are insufficiently adapted to the formalised presentation of the functional semantics of programs being tested. An analogy between the procedures for deobfuscation and program code optimisation has also been identified. Based on the research results, we suggest that theoretical solutions which have been tried out in program code optimisation procedures may be used for code deobfuscation. In the course of the study, we have developed a program code deobfuscation procedure utilising a value state dependence graph. Utilisation of the developed procedure was found to enable presentation of the functional semantics of the programs being tested in a graph form. As the result, identification of malware based on its execution semantics became possible. The paper shows that further research should focus on the development of a method for comparison of the value state dependence graph of the program being tested with corresponding graphs of security software and malware.

References

1. Yadegari Babak N. (2016), Automatic deobfuscation and reverse engineering of obfuscated code, PhD Thesis, The University of Arizona, Tucson, USA, 22.09.2016, 200 p.
2. Xu W., Zhang F., Zhu S. (2002), The power of obfuscation techniques in malicious JavaScript code: A measurement study, 7th International Conference. Malicious and Unwanted Software, 2012, 8 p., DOI: 10.1109.
3. Ming, J., Xin, Z., Lan, P., etc. (2017), Impeding behaviour-based malware analysis via replacement attacks to malware specifications, Springer [Electronic], Sept. 2017, pp. 1–13, available at: https://link.springer.com/article/10.1007/s11416-016-0281-3.
4. Chad Robertson (2012), PDF Obfuscation, A Primer [Electronic], SANS Institute Reading Room site, No. 1, 2012, pp. 1–38, available at: https://www.bing.com.
5. Singh A. (2009), Identifying malicious code through reverse engineering, Springer, New York, 2009, 196 p.
6. Udupa Sh.K., Debray S.K., Madou M. (2005), Deobfuscation: Reverse Engineering Obfuscated Code, 12th Working Conference on Reverse Engineering (WCRE'05), 2005, No. 13, pp. 1–10.

7. Lawrence A.C. (2008), Optimising compilation with the value state dependence graph, University of Cambridge, Great Britain, 183 p. (Cambridge CB3 0FD)
8. Reissmann Nico (2012), Utilising the Value State Dependence Graph for Haskell, University of Gothenburg, Göteborg, Sweden, May 2012, 68 p.
9. Zhengbing H., Dychka I.A., Onai M., Bartkoviak A. (2016). The Analysis and Investigation of Multiplicative Inverse Searching Methods in the Ring of Integers Modulo M, Intelligent Systems and Applications, 2016, 11, pp. 9-18.
10. Zhengbing H., Tereykovskiy I., Tereykovska L., Pogorelov V. (2017), Determination of Structural Parameters of Multilayer Perceptron Designed to Estimate Parameters of Technical Systems, Intelligent Systems and Applications, 2017, 10, pp. 57-62.
11. Pogorelov V.V., Marchenko O.I. (2016), Ohlyad vnutrishnikh form predstavlennya prohramy dlya translyatsiyi z protsedurnykh mov prohramuvannya u funktsional'ni movy [Review of internal program presentation forms for translation from procedural programming languages to functional languages], Scientific Magazine "Computer Integrated Technologies: Education, Science, Production", 2016, No. 23, pp. 85-92.
12. Kushnarev M. V. (2016), Metody i modeli raspoznavaniya vredonosnyh programm na osnove iskusstvennyh immunnyh sistem [Methods and models of malware recognition based on artificial immune systems], Thesis of Candidate of Technical Sciences, Specialty 05.13.23 – Atrificial intelligence systems and tools, Kharkiv, Ukraine, 2016, 164 p.
13. Unhul V. V. (2016), Analysis and development of methods of scripts deobfuscation to identify threats to information computer sustainability, International Scientific Magazine, vol. 2, No. 6, 2016, pp. 19-27.
14. Petrov S.A. (2013), Building adaptive security system based on multi-agent system, Materials of the second international research and practice conference, Westwood, Canada, vol. 2, 2013, pp. 196-201.
15. Z. Hu, S. Gnatyuk, O. Koval, V. Gnatyuk, S. Bondarovets, «Anomaly Detection System in Secure Cloud Computing Environment», International Journal of Computer Network and Information Security, Vol. 9, № 4, pp. 10-21, 2017.
16. Z. Hu, V. Gnatyuk, V. Sydorenko, R. Odarchenko, S. Gnatyuk, «Method for Cyberincidents Network-Centric Monitoring in Critical Information Infrastructure», International Journal of Computer Network and Information Security, Vol. 9, № 6, pp. 30-43, 2017.