Method of Software Obfuscation Using Petri Nets

Dmitriy Dunaev, László Lengyel
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Magyar tudósok krt. 2, 1117, Budapest, Hungary
{dunaev, lengyel}@aut.bme.hu

Abstract. *Obfuscation, in general, is a technique that is used to intentionally make a program code harder to read and analyze for privacy or security purposes. To counteract reverse engineering and unauthorized program analysis, we have to consider obfuscation of a control flow graph since it describes all possible paths a program flow could take through a routine.*

This paper presents a control flow graph obfuscation method using multithreaded environment modelled with Perti nets. The focus is set on splitting a routine code to sections that are to be executed separately in different threads. We introduce a Petri net manager which is responsible for threads management, and describe the execution process of an obfuscated routine.

Keywords. Obfuscation, Petri nets, control flow graph.

1 Introduction

In the general approach, code obfuscation is a set of program transformations that make program code and/or program execution difficult to analyze [1]. Obfuscation hinders manual inspection of program internals and as a result protects against reverse engineering. It protects both storage and usage of keys, and it can hide certain properties such as a software fingerprint or a watermark, or even the location of a bug in case of an obfuscated patch. However, code obfuscation itself does not protect from code lifting or software piracy. It merely strengthens built-in protection mechanisms, e.g. against tampering or piracy [2].

The process of obfuscation can be defined and therefore approached in different ways. We consider obfuscation as a one-way process of original code transformation that results in adding some excessive functionality with the purpose of protecting software from unauthorized analysis and reverse engineering. This process is one-way, what means that there is no effective way to subsequently return to the original state [3].

Definition.

Let *TR* be a transformation process PR_1 => *TR* => PR_2, by which the PR_2 program is obtained from PR_1. We say that the process *TR* is obfuscating process if the following requirements are met:

1) Program PR_2, being obtained from PR_1, is significantly different from PR_1. However, it is runnable and has the same functionality as PR_1, so that Barack's functionality requirement holds true [4].
2) The program analysis, study of operation principles and reverse engineering of PR_2 is significantly more difficult and time consuming than in case of PR_1.
3) At any transformation of PR_1, the resulting PR_2 instance will be different.
4) There is no effective way to transform PR_2 back to the original PR_1.

Since the resulting code obtained after entangling transformations is always different, the obfuscating techniques can be used for prompt identification of copyright infringers, i.e. the buyers of legal software that are engaged in illegal distribution of purchased software copies. To utilize this idea, it is enough to calculate the checksum of every obfuscated program copy, and register it together with customer data in the relevant storage (database). Hereinafter, in case of illegal software distribution, it is enough to calculate the checksum of one illegal copy and compare it with information in a storage in order to identify the copyright infringer.

If we consider a software application, it can be represented at three levels (Figure 1):

- source code,
- some intermediate representation,
- machine code.

Source code obfuscation means taking the application source code and obscuring it, so prying eyes cannot view it in its native format. Actually, source code level obfuscation is less secure than intermediate or executable level techniques. This is primarily because code obfuscators cannot take advantage of implementation details that are not permitted by language compilers. Thus, such

obfuscators are restricted by the given programming language and by the given compiler. In addition, software protection models on source code level would not withstand attacks that combine static and dynamic analysis techniques [5].

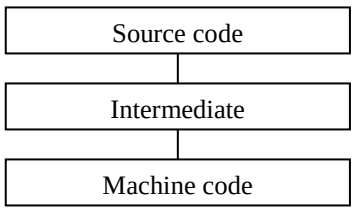

Figure 1. Possible levels of obfuscation

An advantage with intermediate level obfuscation is that it deals with a target platform independent intermediate code. In case such intermediate level obfuscator needs to be ported to another platform, one only needs to write a new translator for the specified processor. Intermediate code is usually a description of high-level statements with some simpler instructions that accurately represent the operations of the source code statements. An intermediate level obfuscation algorithm is described in [6].

In this paper, we introduce a method combining different aforementioned approaches. To counteract reverse engineering and unauthorized program analysis, we have to consider entangling the control-flow graph (CFG), which is a graph of the different possible paths program flow could take through a routine. To do this we propose a method based on Petri nets.

The rest of this paper is organized as follows. In Section 2, we introduce the formal definitions of a control flow graph and a Petri net graph. With the help of an example, we show how Petri net graph can be used for obfuscation modeling. We handle the problem of switching threads and context management by introducing a Petri net manager. Finally, in Section 3 we draw the conclusions, point out problems and outline the further work.

2 Contribution

A control flow graph is a data structure usually built on top of the intermediate code representation abstracting the control flow behavior of a routine. The CFG is a directed graph where the vertices represent basic blocks and edges represent possible transfer of control flow from one basic block to another.

The formal definition of CFG is the following.

Definition.
$G=(V, E, start, stop)$ is a control flow graph $\Leftrightarrow$
1) (V, E) – directed graph
2) $start \in G.V, stop \in G.V$
3) $|in(start)| = |out(stop)| = \emptyset$
4) $\forall v \in G.V\ start \rightarrow^* v \rightarrow^* stop$

The main problem of CFG is that it is essential to many static analysis tools. For example, analysis and optimization tools usually use such graph property as reachability. If a block/subgraph is not connected from the subgraph containing the entry block, that block is unreachable during any execution, and so is the unreachable code; that is, it can be safely removed (such code is called dead). If the exit block is unreachable from the entry block, it indicates an infinite loop (not all infinite loops are detectable, of course). However, dead code and some infinite loops are possible even if the programmer did not explicitly code that way: optimizations like constant propagation and constant folding followed by jump threading could collapse multiple basic blocks into one, and by that cause edges to be removed from a CFG.

Therefore, we can conclude that a CFG should be properly entangled in order to counteract reverse engineering attempts.

Perti net is a directed, bipartite graph in which nodes are either “places” (represented by circles) or “transitions” (represented by horizontal lines or rectangles), invented by Carl Adam Petri [7, 8]. Petri Nets provide an elegant and mathematically rigorous modelling framework for dynamic and discrete event systems. A Petri net is marked by placing “tokens” on places. When all the places with arcs to a transition (its input places) have a token, the transition “fires”, removing a token from each input place and adding a token to each place pointed to by the transition (its output places).

Petri nets are widely used to model concurrent systems and network protocols [9, 10]. We will use them to obfuscate a CFG of a routine.

Definition.
A Petri net graph is a 3-tuple (S, T, W), where:
1) S is a finite set of places
2) T is a finite set of transitions
3) S and T are disjoint, i.e. no object can be both a place and a transition
4) $W: (S \times T) \cup (T \times S) \rightarrow N$ is a multiset of arcs, i.e. it assigns to each arc a non-negative integer arc multiplicity.

In the presented method, a code of a routine is divided into code sections that will be executed separately in different threads. Each section is executed when the appropriate Petri net transition fires.

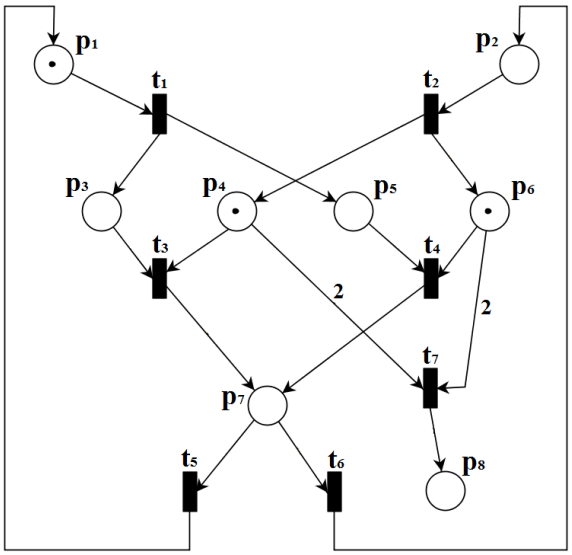

Figure 2. An example of Petri net used for CFG obfuscation[1]

Figure 2 shows an example of a Petri net that can be used for protecting routines from analysis and reverse engineering [11]. The illustrated net contains transitions $t_1 \ldots t_7$ and places $p_1 \ldots p_7$. The transition t_7 fires in a single case – when places p_4 and p_6 contain two tokens both. The places p_4 and p_6 can obtain two tokens at one of two possible sequences of transitions:

$$t_1 \rightarrow t_3 \rightarrow t_4 \rightarrow t_6 \rightarrow t_2 \rightarrow t_6 \rightarrow t_2 \qquad (1)$$

$$t_1 \rightarrow t_4 \rightarrow t_3 \rightarrow t_6 \rightarrow t_2 \rightarrow t_6 \rightarrow t_2 \qquad (2)$$

In all other sequences, the transition t_7 will not fire.

We propose that $t_1 \ldots t_7$ represent some sections of a routine code, and the sequence of execution of these sections of code is important. We further propose that places $p_1 \ldots p_7$ correspond to certain sets of input data. Here we assume that the code sections are executed in separate threads, and the execution sequence is managed by synchronization mechanisms of an operating system. Suppose we know the maximum execution time of each code section; let us denote it by T^i_{max}. We assume that if execution time of i-th code section exceeds T^i_{max}, the sequence of transitions firing changes and consequently t_7 will not fire. Thus, putting a breakpoint in one of the above code sections will change the sequence of transitions and, therefore, reverse engineering of such routine becomes a non-trivial task.

It should be emphasized that for runnability of obfuscated routine, we need to make sure that the context that working threads are dealing with does not change while switching threads. By context we understand the following: register values, stack variables, values of flag registers, and values in global memory segments. Consequently, switching between threads must be completely transparent, and must not introduce any changes to the context. Prologue code and similar epilogue code is needed just for this purpose. Prologue code restores the context that has been saved by epilogue code of the previous code section (Figure 3). Petri net manager by-turn is responsible for controlling and activating threads.

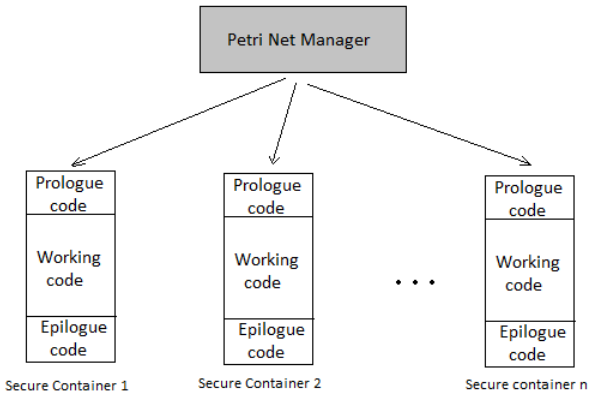

Figure 3. A structure chart of an obfuscated routine using Petri net.

Thus, the execution of obfuscated CFG with Petri nets may look as described below.

1) Petri net manager receives control over the routine execution.
2) Petri net manager carries out initialization phase:
   - saves initial context,
   - marks the net by setting up initial values to places (each place can contain a fixed number of tokens),
   - starts as many threads as the number of available transitions
   - suspends all started threads.
3) Petri net manager "starts" transitions, which contain *WaitForMultipleObjects* function call. Preceding timing functions will determine which of the transitions should fire first. Once the transition fires, all other transitions are "blocked", i.e. cannot fire until the next Petri net manager call.
4) As a next step, the control is transferred to a secure container of corresponding transition (Figure 3), namely the corresponding thread is activated from suspended state.
5) The activated thread restores the context, executes the working code, saves the context and transfers the control back to Petri net manager.
6) Petri net manager moves the tokens and restarts transitions.
7) The process is repeated until the transition containing the last piece of code is fired. For

[1] The figure is made by PIPE2 – an open-source platform-independent tool for creating and analyzing Petri nets. The project webpage: http://pipe2.sourceforge.net/

Petri net in Figure 2 this would be the transition t_7. We call it "the last transition".

8) When the last transition is fired and the corresponding thread finishes execution, Petri net manager frees any allocated resources and control is transferred to the subsequent code.

It should be particularly noted that the presented method can be applied not only to the complete software application, but also to some critical code sections or subroutines. Even if there is no possibility to obfuscate the complete program, it can be done with respect to lower-level subroutines.

3 Conclusions and Future Work

It has been proved by Boaz Barack in his works [4,12] that universal obfuscator does not exist, since there exists a class of programs for which the virtual black box property is not feasible. However, if the obfuscated program does not belong to the Barack's class of non-obfuscateable programs, then the reverse engineering would not be trivial, because the entangled operational logic can be implemented with still high level of complexity.

In the paper, we have presented a method of obfuscation based on Petri nets. The method can be used to protect software from unauthorized analysis and modification, and consequently to prevent its reverse engineering. We have described the step-by-step execution process of obfuscated code, showing that this technique can be used as a part of a software protection utility. The main disadvantage of this method is its platform- and system-dependence.

The implementation of the above-described approach presents problems that still need to be solved, such as:

- timings;
- synchronization of threads;
- considerable execution slowdown.

Consequently, another issue to be solved is a possibility of violating timings in real-time sensitive applications, or in some cases introducing problems with concurrent accesses to local variables or I/O subsystem.

The presented method is system-dependent in its implementation, and therefore cannot be named universal. However, we find the presented idea promising, since involving Petri nets into obfuscation can significantly complicate the reverse engineering of protected code.

Future work includes, but is not limited to, solving the aforementioned problems with timings and synchronization and working out in details methods of interaction between Petri net manager and program threads.

4 Acknowledgments

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

References

[1] Collberg, C.S.; Thomborson, C. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, vol. 28, pp. 735–746, August 2002.

[2] Sorokina, S.I.; Tihonov, A.Ju.; Scherbakov, A.Ju. *Programming of drivers and secure systems.* BHV-Petersburg Press, ISBN 5-94157-263-8, St. Petersburg, Russia, 2003. In Russian.

[3] Dunaev, D. Obfuscation for protecting software from analysis and modification. In *Proceedings of the Automation and Applied Computer Science Workshop 2011 (AACS'11)*, pages 290-296, Budapest, Hungary, June, 2011.

[4] Barak, B.; Goldreich, O.; Impagliazzo, R.; Rudich, S.; Sahai, A.; Vadhan, S.; Yang, K. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference*, Santa Barbara, California, USA. LNCS, Vol. 2139, 2001.

[5] Madou M.; Anckaert, B.; De Sutter, B.; De Bosschere, K. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital rights management*, pages 75-82. ACM, 2005.

[6] Dunaev D.; Lengyel L. Overview of an Obfuscation Algorithm. In *Proceedings of the International Conference on Computer Science and Information Technologies (CSIT'2012)*, pages 36-38, Lvov, Ukraine, November, 2012.

[7] Petri, C.A., *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. In German.

[8] Petri, C.A. Fundamentals of a theory of asynchronous information flow. In *Proceedings of IFIP Congress 62*, pages 386-390, Amsterdam, North Holland Publ. Comp., 1963.

[9] Lakos Ch.; Lamp J.; Keen Ch.; Marriott B. Modelling network protocols with object Petri nets. In *Proceedings of Workshop on Petri Nets*

Applied to Protocols, pages 31-42, Springer-Verlag, 1995.

[10] Jensen, K.; Kristensen, L.M.; Wells L. Coloured petri nets and CPN tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.* 9, 3, pages 213-254, 2007.

[11] Dingle, N.J.; Knottenbelt, W.J.; Suto, T., PIPE2: a tool for the performance evaluation of generalized stochastic Petri Nets. *SIGMETRICS Perform. Eval. Rev.* 36, 4 (March 2009), pages 34-39.

[12] Barak, B. *Non-black-box techniques in cryptography*. PhD thesis, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, January 6, 2004.