Hardening Registration Number Protection Schemes against Reverse Code Engineering with Multithreaded Petri Nets

Thorsten Schneider
University of Hannover
FG Software Engineering
Welfengarten 1
30167 Hannover, Germany
<info@reverse-engineering.net>

Abstract

This paper proposes a new technique for hardening registration number protections by using multithreaded Petri nets. Using this technique one is able to prevent reverse code engineering attacks, which consist of protection scheme analysis and reengineering. We come to the conclusion that using such a technique leads at minimum to an enormous reverse code engineering and analysis process for the attacker and that the proposed technique is therefore an amelioration in registration number protection.

Keywords: *Reverse Code Engineering; Multithreaded Petri Nets; Software Protection; Registration Number Protection Schemes*

1. Introduction

Registration number protections require the user to enter a registration number to register a software application. However, in most cases such a protection can be defeated easily by an in-depth analysis of the disassembled code or by tracing the applications execution using dynamic disassembly (debugging) techniques. Even there still is heavy use of simple XOR encryption methods, several software applications use high optimized cryptographic algorithms to prove the given input for validity.

However, there are several approaches to deal even with such complicated calculations. Once the attacker identifies the algorithm routines, one approach is to use self-keygenning techniques, which produces always a correct serial number by turning the application against itself, which had been described by Webbit in detail [1]. Another approach is the common used code-ripping method, where the attacker extracts the relevant code segment and uses the extracted code to build up a keygenning application. Using self-keygenning or code-ripping, the attacker does not necessarily need to know exactly, how the algorithm works. The attacker just needs to be aware of initial settings, as register values.

For this paper we confine to the standard registration number, leaving out hardening methodologies as anti-disassembly or anti-debugging methods. For these Cerven defines 5 different types of such protection types [2]. Dealing with our proposal we use the simplest variant, which expects the input of a serial number and – if the serial number is correct – sets the state of the application to a registred status.

Against most used algorithms, which are based on sequential calculations, another method is to use graph based algorithms, which increase the analysis complexity. Breaking such a protection is hard for the analysing attacker, however by using again code-injection or self-keygenning methods, it is still possible to break such a protection whithin minutes.

In this paper we introduce a more complex graph based protection, based on Petri nets, which are dicotyledonous oriented multigraphs [3]. Main advantage of using Petri nets is that the attacker does not know the state of each node. He can only guess, bruteforce or reconstruct the Petri net, which can be a very complex task, as we will describe later. One more feature of a Petri net is that it can be executed parallelised, which again we will use and describe later.

Section 2 gives a short overview of Petri nets. As we confine to the proposed protection scheme, we reduce this section to a minimum. Section 3 describes on how to use Petri nets for registration checking processes, in section 4 we give a detailed example. Section 5 refers to the reachability problem, which is induced by the usage of Petri nets and is an important factor for increasing Petri net complexity. In section 6 we illustrate how to harden the proposed protection scheme, and finally in section 7 we give a short discussion of the proposed methods.

2. Petri Nets

This section gives a short introduction to Petri nets. Reader familiar with Petri nets can skip this section.

Petri nets originate from the early work of Carl Adam Petri [4]. Since then the research on Petri nets increased considerably . The use of Petri nets lead into a mathematical description of a system structure that can be then investigated analytically. Petri nets are divided in classical petri nets, which are directed dipartite graphs, and high level Petri nets [5]. The classical Petri net allows for the modelling of states, events, conditions, synchonisation, parallelism, choice, and iteraration.

However, Petri nets describing real processes tend to be complex and extremely large, which is especially a problem in bioinformatics when modelling pathway processes. Moreover, the classical Petri net does not allow for the modelling of data and time. To solve these problems, many extensions have been proposed. Three well-known extensions of the basic Petri net model are: (1) the extension with color to model data, (2) the extension with time, and (3) the extension with hierarchy to structure large models. A Petri net extended with color, time, and hierarchy is called a *high-level Petri net [6]*. In this paper we will use these extensions to model specific aspects. However, a formalization of these aspects is beyond the scope of this paper. For a more elaborate discussion on Petri net extensions and other kinds of high-level Petri nets, the reader is referred to [6-9]. One of the most interesting problems (*"The Five Chinese Sages Problem"*) solved with Petri nets has been given by Dijkstra [10].

3. Concept: Using Petri nets as registration checking routine

Petri nets are an ideal tool for modelling transitions. There are several research areas where the use of Petri net models comes handy. Examples are given in Bioinformatics (Biochemical Networks) [11], Software Performance Evaluation [12] or Queuing Nets [13]. As well Petri nets are used for representations of simple or complex algorithms. Since registration number routines correspondent to mathematical algorithms, they can be represented by Petri nets.

We use Petri net design for representing such algorithms, to improve software protection tasks and to harden the analysis mechanisms of the attackers attempt in understanding the underlying algorithm. By increasing the complexity of the resulting net the analysis becomes confusing and in the best case impossible. Additional we resort to the Petri net feature of parallelism, which replaces the common use of sequential algorithms with parallel running

processes. Using such parallelising feature, the attacker needs - for algorithm and code understanding - to dynamic disassemble (debug) all processes at the same time, which seems to be nearly impossible. Additional one feature, which comes handy for a registration number routine, is that the algorithm is not static, but highly dynamic since the different states are not known to the attacker.

It is not a secret that most protection approaches reduce to hide notorious conditional jumps from the analysis of a reverse code engineer. In most cases this can be simplified to the scheme *"bad guy / good guy"* jump. To prevent manipulation attempts, several techniques as anti-debugging and anti-disassembly methods have been introduced [2]. However, once an anti-technique has been defeated, a manipulation of such jumps becomes an easy task. This is where the use of Petri nets are useful – to obfuscate and to obscure the analysis process for the attacker.

4. Example: Using Petri nets as registration checking routine

We use a simple Petri net to realize a registration checking routine (see fig. 1). Starting the Petri net process with an initial marking of places $p_0,...,p_3$, all other places are invisible for any input settings. It is important that place p_7 has already a token before starting the net. Additional we need to add a lower priority setting to transition t_2, which results in preferring transition t_1 when both transitions (t_1 and t_2) are fired. Assuming that the Petri net is solved only correct, when the last fired transition is t_2, one solution of this Petri net is the setting: $\{p_0 = 1, p_1 = 0, p_2 = 1, p_3 = 1\}$. In other cases, transition t_2 will not be executed.

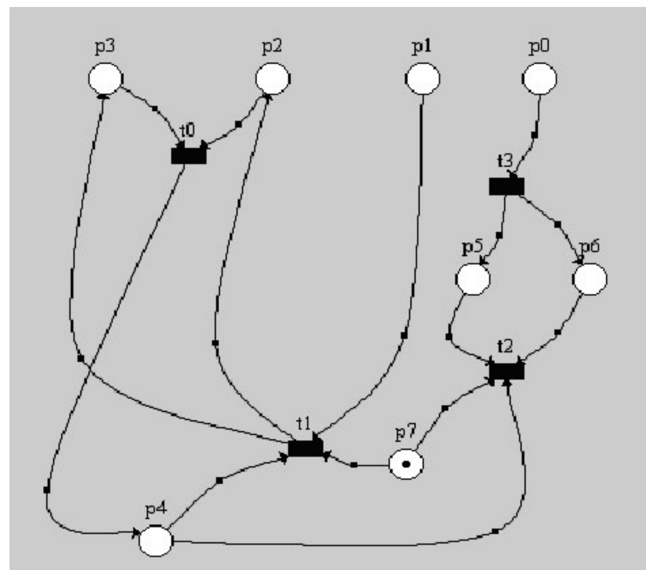

Fig.1: A simple Petri net for realizing a serial checking routine.

All over all, the Petri net given in fig. 1 is very easy to solve. In the given example one can easily find a correct key within a search space of $2^4 = 16$. Solving such a small Petri net can be done even by hand or by brute force within a very short time.

One solution to this problem is to increase the net complexity. Since current research is focussing more on the problem, on how to *reduce* net complexities, there are no algorithms described yet on how to automate the complexity increasing process. We use a simple

copying and linking approach to increase the complexity of the Petri net (see fig. 1) to receive a harder to understand and to attack Petri net (see fig. 2).

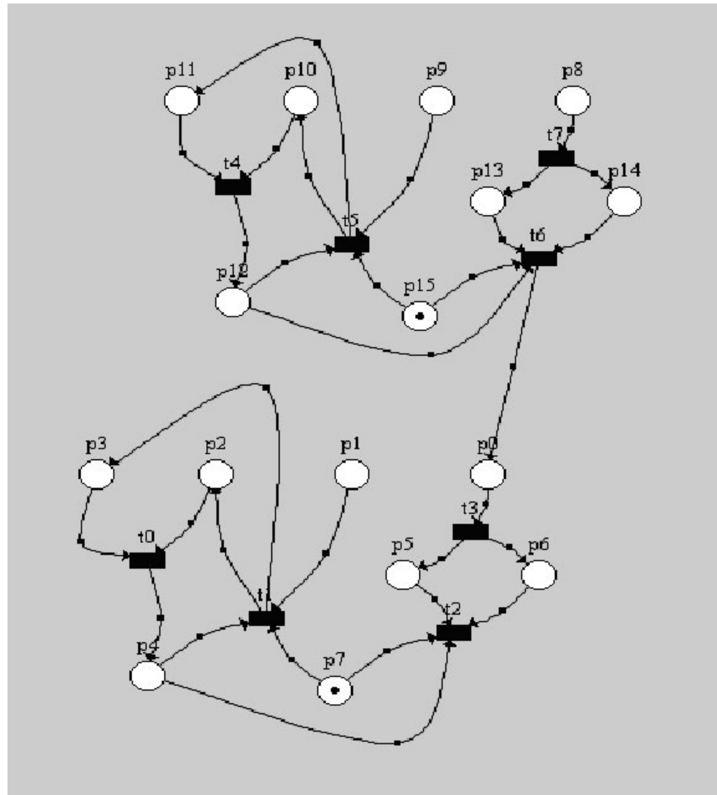

Fig. 2: Copying and linking the Petri net to increase the net complexity.

Even copying and linking seems to be an easy and appropriate option, some problems arise with increasing the net complexity. Since we are linking a new net to each place one can mark during initial marking, some places loose their marking characteristics. This means in short, that they can not be marked by initial marking in the future. Fortunatly there are new places available for marking after copying and linking the net. If one links the new net to place p_0, one has to take care about the functionality and reachability problem as well [14]. We will focus on the reachability problem in section 5. Using the example in fig. 2, the transition t_2 remains final reachable and the new places $\{p_8,...,p_{11}\}$ are available for an initial marking. In fig. 2 it is abviously that transition t_6 plays the role of transition t_2 from the original net. If transition t_2 is executed, t_6 will be executed as well. Using the initial marking $\{ p_1 = 0, p_2 = 1, p_3 = 1, p_8 = 1, p_9 = 0, p_{10} = 1, p_{11} = 1\}$ we should be able to get the same result as for the Petri net in fig. 1.

Using such an improvement, the key length grows up to 7 bits, which increases the search space for brute forcing to 128 keys. Using a similar linking for p_1, p_2 and p_3, one is able to increase again the search space for bruteforce attacks to 16 bit, leading into 2^{16} keys.

Unfortunatly, one can not just link to the place p_1, because of the reachability problem for transition t_2, we need its empty state. If one links the Petri net for p_1 similar to other places, then the empty state is reachable from the linked net by 15 (16 - 1) different combinations. To solve this problem, one can use a modified linked segment (see fig. 3). Here, for the unreachability of t_{11} by initial marking, one has to put the tokens to the places $\{p_{16} = 1, p_{17} = 1, p_{18} = 0, p_{19} = 1\}$. Additional, transition t_{11} receives lowest priority.

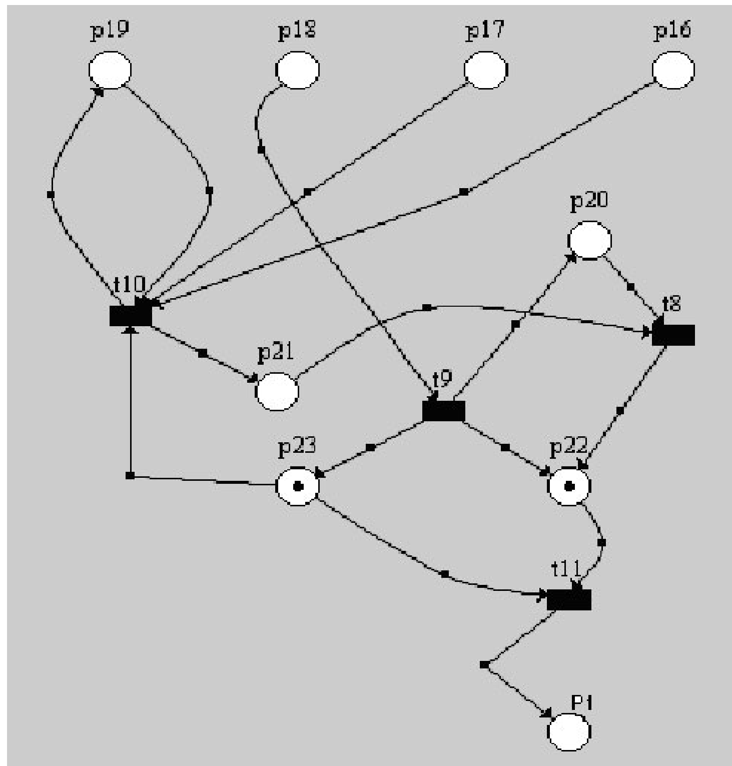

Fig. 3: Linked Petri net as variant.

5. Difficulty of the Reachability Problem

Dealing with complexity increasing manipulations of Petri nets, as described in the section before, one must be aware about the reachability problem, to keep the Petri net solvable at all and to keep its original functionality. The reachability problem – a reduced variant of the liveness problem – is defined by Handorean by the following question [5]:

> *“Given a marked Petri net, m_0 beeing the initial marking and a marking m’ – is m’ reachable from m_0 ?”*

As far as problems of subsets and equalities for ensemble reachabilities of Petri nets are undecidable then maybe the reachability problem is undecidable too. To solve this problem several approaches are introduced by Mayr [15] and Sivaraman [14]. Other algorithms had been shown by Jancar [16] to be definitive wrong for solving such problems. One basic solution technique is to build a finite representation for the reachability set of a Petri net.

> *"As we can see, with reachability tree we can solve problems of safety, preservation and coverage. Regrettably, in common case we can't use it for solveing problems of B reachability and activity [...]" [17].*

Handorean [5] describes in a showcase how to build the reachability tree of a Petri net which also allows to build the entire state-space (see fig. 4) and how to make the representation finite:

1. define Ω number of tokens in a place then it is "too big" (plays the role of infinite)
2. when a new marking is equal to another marking on the path from the root node, we add it as a terminal node
3. a new marking x is grater than a marking y on the path from root, the components of y which are strictly greater than the corresponding components are replaced by Ω (if $x>y$ then whatever is reachable from y is reachable from x too)

This results into the following definitions:

1. If the Petri net is *k*-bounded (max *k* tokens in a place), the reachable state-space is finite.
2. If the Petri net is conservative and let *k* being the number of tokens in the net, the reachable state space is finite (there is a finite number of ways we can partition *k* tokens among *n* places).
3. If Ω is anywhere in the reachability tree, the reachability set is not finite (and therefore cannot be bounded or conservative)
4. If the reachability problem is solvable (possibly at a high cost) then the liveness problem is solvable.

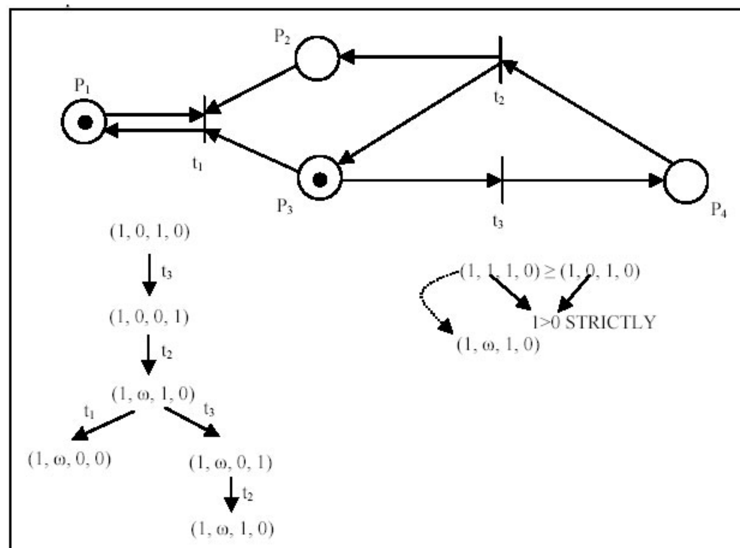

Fig. 4: Building the reachability tree of a Petri net (Image Source [5])

6. Saying "Foo on You!" to the attacker

After building a working Petri net one is able to use this for registration number checking routines. However, the weakness of such a protection is obvious. So far, the handling of the Petri net is sequential. It is an easy task for the attacker to trace the protection scheme and to rebuild the Petri net for further investigation. One way to increase the analysis complexity is to parallelise the Petri net by using multithreading technology. Assumed, that each place or transition receives one thread, and assumed as well that we use absolute parallelism for the Petri net; the attacker needs to debug and trace all threads (transitions) simultaneously. This is nearly impossible, but writing a special tool for handling such a problem might still be possible – considering an enormous investment of time for developing such a tool.

Next, one is able to add hundreds of fictitous transitions with very low priority levels. This results in a very large and complex Petri net (see fig. 4 and 5), which can end in a tunnel of horror for the attacker. We agree that such hardening is inefficient for memory resources and performance. However, reverse code engineering such a protection scheme becomes a pain at all.

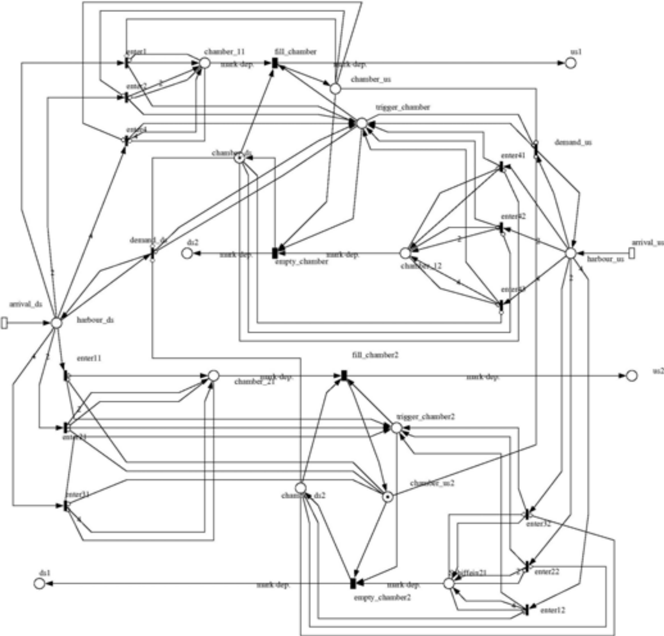

Fig. 4: A more complex Petri net example. This net is harder to understand.

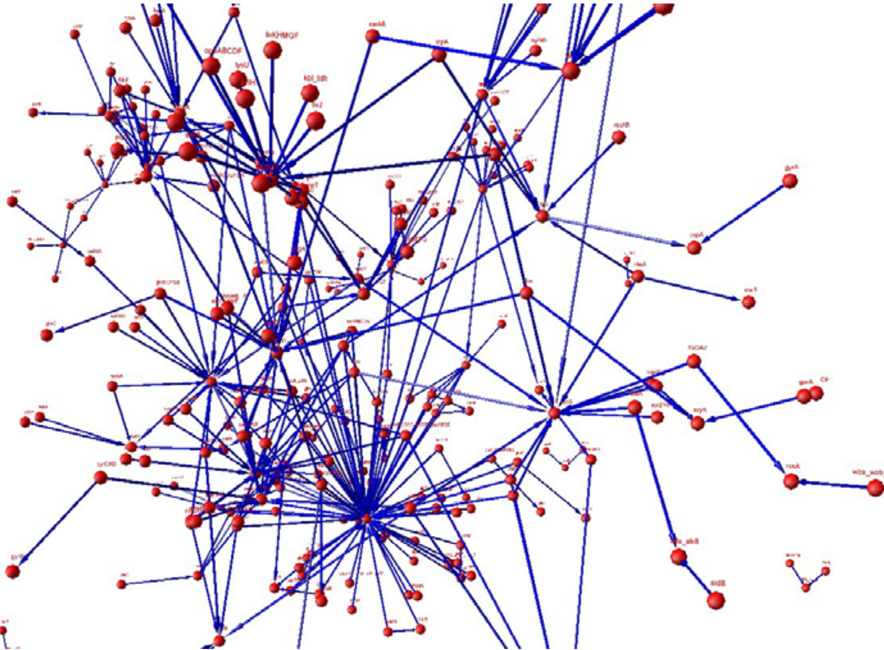

Fig. 5: A more complex 3-dimensional Petri net example showing places (red) and arcs (blue). It is obvious that the understanding of such a network gets confusing.

7. Discussion

Obviously standard methods of breaking this protection scheme do not work here. Unlike for consecutive protection schemes, an attacking reverse code engineer never can guess which transition is decisive, since upon wrong initial marking the decisive transition is not accessible. On the other hand, the attacker may try to execute all transitions - if he finds all places in any way. In our given example the attacker needs to try only 20 transitions instead of 2^{16} possible keys - he just has to put tokens in *all* places. It has not to be forgotten that our decisive transition has lowest priority - all tokens from input places disappear faster then one can check them. The attacker could somehow find the transition with the lowest priority - for this we can enter thousand fictitious transitions with low priority levels and similar. However, we agree that this might be inefficient for memory resources and performance.

Dynamic Disassembly (Debugging) such an application becomes complicated since the attacker needs to debug all threads (transitions) simultaneously, which is nearly impossible. Additional, if one somehow uses protection code (anti-disassembly or anti-debugging) in different places of the application, one is able to protect from statically attacks as well.

Even it is possible to solve the reachability problem, solving large Petri nets might be time consuming and inefficient to solve. This is one advantage for the suggested protection technique. One solution in preventing even high computation on such nets might be using unsolvable problems like given 2 Petri nets with M_1 and M_2, where a possible question could be if $R(M_1)$ is a subset of $R(M_2)$ or $R(M_1)$ is equal $R(M_2)$. An arising problem might be, that one gets an unsolvable net which even prevents to work correct.

One additional attacking method could be in writing a specialised tool, which watches each transition and rebuilds the Petri net.

8. Conclusions and Future Work

In conclusion the main goal of any protection scheme is hiding some information from other people. There is no protection which reaches its main goal. However, the proposed technique is a fine grained method to increase the complexity of protection in a significant way. Further investigations should focus on enhancing the algorithm and its implementation and to run several attacks on the protection scheme. As well intensive research is necessary on how to increase Petri net complexities regarding to reachability problems.

Acknowledgement

The author thank Robert Airapetyan for his prior concept in the field of software protection and Petri nets.

Thorsten Schneider *received his diploma in medical informatics from the University of Heidelberg in 2002, and the Doctor rerum medicarum with Magna cum laude from the Free University of Berlin (Charité Universitätsmedizin Campus Benjamin Franklin) in 2004. He was scientific assistant at the bioinformatics faculty at the University of Bielefeld from 2003 to 2004 and is currently scientific assistant for software engineering at the University of Hannover. He is a member of the Center for Space Medicine Berlin (ZWMB). His current research interests include reverse code engineering, experimental and empirical software engineering and time series analysis. He is working on his postdoctoral thesis in the field of reverse code engineering, watermarking, obfuscation, decompilation and software protection. He maintains and administrates several reverse code engineering websites, including the Reverse Code Engineering Portal (Anticrack) (http://www.reverse-engineering.net), the Reverse Engineering Academy (http://www.reverser-course.de) and the crackmes website system (http://www.crackmes.de).*

References

1. Webbit K: **Keygen Injection**. *CodeBreakers-Journal* 2004, **1**(2).
2. Cerven P: **Crackproof Your Software**: No Starch Press; 2002.
3. Peterson JL: **Petri Net Theory and the Modelling of Systems**: Prentice Hall PTR; 1981.
4. Petri CA: **Kommunikation mit Automaten**. *PhD.* Bonn: University of Bonn, Germany; 1962.
5. Handorean R: **Petrinets - Notes**. *Available at http://userfscecwustledu/~cs576/Notes/Petrinets.pdf* 2003.
6. Jensen K: **Colored Petri Nets. Basic Concepts, Analysis Methods and Practical use.** *EATCS Monographs on Theoretical Computer Science* 1996.
7. Murata T: **Petri Nets: Properties, Analysis and Applications**. *Proceedings of the IEEE* 1989, **77**(4):541-580.
8. Van der Aalst WMP: **Putting Petri Nets to Work in Industry**. *Computers in Industry* 1994, **25**(1):45-54.
9. Van Hee KM: **Information System Engineering: A Formal Approach**: Cambridge University Press; 1994.
10. Dijkstra EW: **Co-Operating Sequential Processes**. *Programming Languages* 1968:43-112.
11. Popova-Zeugmann L, Heiner M, Koch I: **Modelling and Analysis of Biochemical Networks with Time Petri Nets**. In: *Workshop Concurrency, Specification & Programming'2004: Sept. 24 - 26 2004; Caputh*: Informatik-Berichte der HUB Nr. 170; 2004: 136-143.
12. Becker M, Twele L, Szczerbicka H: **Software Performance Evaluation with Generalized Stochastic Petri Nets**. *Performance Evaluation, Special Issue on Performance Validation of Software Systems* 2002.
13. Becker M, Szczerbicka H: **Combined Modeling with Generalized Stochastic Petri Nets including Queuing Nets**. In: *14th UK Computer and Telecommunications Performance, Engineering Workshop: 1998*; 1998: 48-62.
14. Sivaraman E: **An Approach for Solving the General Petri Net Reachability Problem - Duality Theory and Applications**. *Available at: http://wwwokstateedu/cocim/members/eswar/duality.pdf* 2004.
15. Mayr EW: **An algorithm for the general Petri net reachability problem**. In: *13th Annual ACM Symposium on Theory of Computing: May 11-13 1981; Milwaukee, Wisconsin, USA*: ACM; 1981: 238-256.
16. Jancar P: **Bouzinae's algorithm for the Petri net reachability problem is incorrect**. *Petri Nets Newsletter* 2000:1-6.
17. Peterson JL: **Petri Net Theory and The Modeling of Systems.**: Prentice Hall PTR; 1981.

Attachment

The following source code has been provided by Robert Airapetyan, Polytechnical University of Odessa.

```
; *******************************************************************
.586p
.model          flat,stdcall
option          casemap:none
include         \masm32\include\windows.inc
include         \masm32\include\user32.inc
include         \masm32\include\kernel32.inc
includelib      \masm32\lib\kernel32.lib
includelib      \masm32\lib\user32.lib
; *******************************************************************
                .data
key             db  00h
P               db  0,0,0,0,0,0,0,1,0,0
box_title       db  "Congratulations!",0
box_mes         db  "You've crack this easy one...
                    But what you say if there was 1000 threads?",0
box_title2      db  "Shit...",0
box_mes2        db  "Invalid key",0
; *******************************************************************
               .data?
ThreadId        dd                ?
; *******************************************************************
                .code
                CT                MACRO StartAddress
                push              ThreadId
                push              EBX
                push              EBX
                push              StartAddress
                push              EBX
                push              EBX
                call              CreateThread
                ENDM
_start:
                xor               EBX,EBX
                call              byte2bit
                CT                _T6
               ;invoke SetThreadPriority, EAX, THREAD_PRIORITY_ABOVE_NORMAL
                CT                _T2
                CT                _T3
                CT                _T1
                CT                _T4
                CT                _T5
               ;invoke SetThreadPriority, EAX, THREAD_PRIORITY_LOWEST

                CT                _T7
                jmp               $
_T3:
                mov               AL,P[4]
                add               AL,P[5]
                dec               AL
                dec               AL
                jnz               _T3
                mov               P[6],1
                mov               P[4],AL
                mov               P[5],AL
                jmp               _T3
```

```
_T1:
                mov             AL,P[3]
                dec             AL
                jnz             _T1
                mov             P[3],AL    ; 0
                mov             P[4],1
                jmp             _T1
_T4:
                mov             AL,P[0]
                dec             AL
                jnz             _T4
                mov             P[8],1
                mov             P[9],1
                mov             P[0],AL
                jmp             _T4
_T5:
                mov             ECX,0FFFFFFh
                loop            $       ; delay
                mov             AL,P[8]
                add             AL,P[9]
                add             AL,P[7]
                add             AL,P[6]
                sub             AL,4
                jnz             _T5

                invoke   MessageBox, 40h, addr box_mes, addr box_title, EBX
                jmp             _fin
_T2:
                mov             AL,P[2]
                dec             AL
                jnz             _T2
                mov             P[5],1
                mov             P[2],AL
                jmp             _T2

_T6:
                mov             AL,P[6]
                add             AL,P[1]
                add             AL,P[7]
                sub             AL,3
                jnz             _T6
                mov             P[3],1
                mov             P[2],1
                mov             P[1],1
                mov             P[7],AL
                jmp             _T6
_T7:
                mov             ECX,0FFFFFFFh
                loop            $
_patch:
                invoke MessageBox, 40h, addr box_mes2, addr box_title2, EBX
_fin:
                Invoke ExitProcess, EBX
byte2bit:
                mov             AL, byte ptr [key]
                push            EAX
                shr             AL, 3
                mov             P[3],AL
                pop             EAX
                push            EAX
                shr             AL, 2
                and             AL,1
                mov             P[2],AL
```

```
pop             EAX
push            EAX
shr             AL, 1
and             AL,1
mov             P[1],AL
pop             EAX
and             AL,1
mov             P[0],AL
ret
end             _start
```