Load Profile-based Efficiency Metrics for Code Obfuscators

Marko Đuković

Schneider Electric DMS, Narodnog Fronta 25A, 21000 Novi Sad, Serbia, E-mail: marko.djukovic@schneider-electric-dms.com

Ervin Varga

University of Novi Sad, Faculty of Technical Sciences, Department of Power, Electronic and Telecommunication Engineering, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia, E-mails: evarga@uns.ac.rs, e.varga@ieee.org

Abstract: Code obfuscation techniques are gaining more attention and importance as ubiquitous computing becomes commonplace. The necessity to protect intellectual property rights as well as the need to prevent tampering with code running on all sorts of autonomous devices increases the demand for high quality code obfuscator tools. Currently, the principal quality metrics for evaluating code obfuscators mostly revolve around aspects such as security and functional correctness of the generated code. Nevertheless, power consumption plays a central role in handheld devices, as battery life and endurance is usually the bottleneck in achieving an acceptable level of user satisfaction with the system. Consequently, the criteria for choosing the right code obfuscator should be extended to also take into account the impact of obfuscation on the overall power consumption. This paper presents one viable approach for evaluating code obfuscators in regard to a power consumption level of the obfuscated code. The methodology is based upon load profiles. The performance of the solution has been tested using various commercial code obfuscators. The results show that there are significant differences in power consumption levels between original and obfuscated code. In order to select an obfuscation tool it is not enough to rely solely on non-power related attributes. High increase in a power consumption level may be totally inappropriate on mobile devices, despite the best obfuscation achieved with that particular tool. Accordingly, power consumption level should be incorporated into a set of quality metrics for code obfuscators.

Keywords: Energy-aware system; measurement techniques; protection mechanisms; obfuscation

1 Introduction

We are nowadays witnessing an extremely rapid development in the domain of information technologies (IT), which entails an additional level of electrical power dependency. This fact is especially important taking into account the proliferation of mobile solutions. The aforementioned trend is inevitably followed by higher power costs, partly due to increased number of data centers, server farms, etc. [1]. The IT sector today uses around 8% of the total produced electrical energy at the global level, and this tendency continues to rise [2].

One of the key challenges facing software engineering today is in relation to power consumption of executable code. Reducing load on batteries is not only pertinent in the case of handheld devices, but in general. There are already optimizing compilers for embedded software targeted to keep power consumption of running programs as low as possible (previously speed and memory consumption were the only key optimization goals for compilers). Naturally, any such optimization is a very complex task, as the code still needs to meet acceptable performance characteristics as well as utilize computing resources judiciously. Nevertheless, despite the sophistication level of a compiler in choosing the best possible route to generate energy-aware binaries, overall power reduction is still mainly dictated by the source code itself. In other words, it is impossible to leave every power consumption issues to the compiler. If the algorithm and its implementation are inherently power hungry then there is very little a compiler can do to mitigate the problem.

Another important aspect brought extensively into the foreground, particularly by the proliferation of the pervasive computing paradigm, is pertaining to the dependability and security of the software. As software is running on all sorts of devices they are more exposed to keen eyes of possibly malicious intent. Similarly, end users are also endangered by accepting and allowing various software to run on their devices. Consequently, much effort is invested to keep the software protected from reverse engineering and modification as well as to protect intellectual property rights. One technique to impede reverse engineering of software is code obfuscation. The basic idea is very simple. It is related to the way of rearranging the code to become incomprehensible for human readers, while guaranteeing semantic equivalence with original code (this is an imperative requirement). Of course, any such rearrangement inevitably changes the runtime execution characteristics of the original code. Among these is the power consumption level.

All in all, code obfuscation is one of the most popular techniques to protect programs from malicious code tampering and/or to prevent illegal appropriation of intellectual property rights. However, obscurity comes at the cost of memory, run-time and power consumption overhead. This research paper addresses the problem of estimating additional power consumption due to code obfuscation. Power consumption is a crucial problem for mobile devices, because of their limited power resources available as well as because of many power intensive sensors commonly installed. This paper presents a potential step forward in understanding the trade-off between higher level of code security and limited power resources.

To derive estimates and quantify the effects of obfuscation on power consumption an appropriate measurement and evaluation is needed. Without estimating the elevation in power consumption users might perceive degradation from the viewpoint of usability. If mobile devices need to be recharged more often, due to power hungry code, then the benefits of having higher security would be counterbalanced by the necessity to save power. Thus, users willing to fully leverage secure applications, to carry out sensitive e-commerce related actions, would be penalized. This is the reason that measures need to be taken to estimate and balance the obfuscation with power consumption.

2 Related Work

The analysis of how code obfuscation impacts power consumption is an uncharted research area. Papers dealing specifically with the influence of obfuscation techniques on power consumption do not exist. There are related works, but mainly for measuring electricity consumption in general, or for very specific targets. Essentially, the current research about the influence of software on power consumption is divided into several areas: code refactoring [3], lock free data structures [4], design patterns [5, 6] and web servers [7].

Anderson [8] presents a system whose purpose is to detect malware by analyzing dynamic power consumption patterns during run-time based on load profiles. This is possible as the signature of malware's power usage looks very different from the baseline power draw of a chip's standard operations [8]. The proposed solution is mainly useful in controlled environments (routers, switches, etc.), where the reference behavior of the system is known in advance. Our solution is targeted to code obfuscators, hence in some way expands the domain of load profile-based techniques for estimating power consumption. Tiwari in [9] presents one such measurement based approach for determining the power consumption rate at the granularity of CPU instructions. Research results were obtained for three architecturally different microprocessor types. The method and estimated results described in [9] were essentially the starting point for the purposes of our work. Paper [10] presents a methodology for power consumption estimation of embedded processors/systems. The work described in [11] elaborates about a model for energy and power estimation using constant parameters. Finally, [12] describes in detail the power consumption at the level of assembly instructions for advanced computer systems. The rest of the paper is organized in the following way: Section 2 presents an overview of the code obfuscation problem domain and its importance together with some brief theoretical background, Section 3 presents an overview of the architecture and methodology used for analyzing the power consumption of obfuscated code, Section 4 discusses the results obtained for various code obfuscators and Section 5 concludes the paper as well as details the future work of the authors.

3 Obfuscation and Problem Definition

The objective of this study was the identification of major problems related to energy efficiency as a consequence of code obfuscation, the most commonly used protection mechanism against reverse engineering. Recently, energy is becoming a very important factor in the design of advanced computer systems. Researchers invest considerable efforts in power conservation techniques [13], [14]. In this paper we have studied the effects of obfuscation on power consumption at the processor level for the .NET platform. For this purpose, we have developed the framework for analyzing and generating load profiles. These are energy profiles showing diagrams of electrical power consumption per instruction during the execution of an application.

3.1 Obfuscation Problem

Thanks to rapidly growing popularity of internet technologies, software companies are facing an ever-increasing threat of theft of intellectual property rights. Code reverse engineering allows competitors to reveal important technological innovations, secrets and also to inflict some inestimable damage. Business logic comprehension is not conditioned on understanding the whole code. Consequently, reverse engineering entails development of new protection mechanisms, which usually revolve around: encryption, code morphing, security through obscurity and obfuscation. These techniques may also be used for protection against malwares, another growing problem in the software industry. Encryption is a very popular method of intellectual property protection against reverse engineering and it implies encryption of a byte code, so that the client is the only one who has the necessary key to decrypt and run the application. The problem with encryption is to find a safe way for exchanging keys.

Of all currently relevant principles, code obfuscation is the most commonly used. First concepts are mentioned in [15] and are pertaining to key exchange mechanisms. Risk of unauthorized code access, loss of intellectual property, finding software vulnerabilities and economic losses that individuals or companies may undergo are urgent problems nowadays. All programming languages can be obfuscated, but under the highest risk of being reverse engineered are software packages developed for the JVM (Java Virtual Machine) and .NET platforms. Unlike a native binary code, an intermediate byte code contains names of classes, methods and fields, thus disassemblers may easily generate almost identical code to the original. The previously mentioned platforms also utilize Just-In-Time (JIT) compiling, which allows byte code to be translated on-the-fly into a machine code. All in all, a high level of manageability of virtual machine code makes reverse engineering more feasible than in the case of a code produced by a compiler for the so-called non-managed languages (such as, C++ and C). In the latter case, obfuscation is often realized by using special macros, which perturb the source code before being given to the compiler.

Besides all good qualities, obfuscation has couple of disadvantages. The most considerable drawback is the fact that obfuscated applications contain many more executable instructions. Moreover, many transformations increase the execution time of programs, thus indirectly rising their power consumption needs.

3.2 Types of Obfuscation

There are several types of obfuscation (various obfuscation transformations are classified in detail in [16]), but we will list only those ones that are relevant in this study:

3.2.1 Lexical Obfuscation

Encompasses lexical changes in identifier names to hide their real meaning. Lexical obfuscation is a relatively weak type of protection, because an attacker can understand the meaning of the changed identifier from the context. It has the smallest impact on the energy profile. This type of obfuscation is usually not used independently, since there are tools that facilitate the understanding of lexically perturbed code.

3.2.2 Data Obfuscation

Used in situations when data itself needs to be protected. The data is modified in such a way that it is very hard to discern its value based upon static code analysis. This type of obfuscation affects the values and structures of data located in a program. The common transformations of this type includes: array and variable merging/splitting, data encoding, inheritance relation modification and variable reordering [17]. Array related manipulations are specifically elaborated in more detail in [18], [19]. Data obfuscation is very powerful in object-oriented systems, due to importance of understanding inheritance relations. A simple example of a data obfuscation is changing the value of one variable with an arbitrary number of new variables. Hence, the value of the original variable cannot be determined without combining the values of an arbitrary number of auxiliary variables. This similar idea can also be applied to classes [20]. An arbitrary number of classes may be fused together and replaced with one big class, and vice versa. If these two methods are used in tandem, an application would be notably changed and become extremely incomprehensible. It has to be noted, that this type of obfuscation has a big impact on the energy profile, thus will be further described in Section 5.

3.2.3 Control Obfuscation

Affects the flow of execution by altering it with irrelevant conditional statements. This results in reordering of methods, loops and statements. There are two broad categories of this type of obfuscation: opaque predicate and dynamic dispatch [21], [22], [23], [24], [25], [26], [27]. Analyzing such obfuscated code is enormously hard as there is no way to interrelate various program blocks with each other. Usually, aggressive control flow adjustments do negatively impact the performance of the application, as any kind of speculative execution optimizations at the CPU level are basically rendered unusable. This type of obfuscation considerable influences the power consumption, therefore it is also included in the study.

4 Architecture & Methodology

The proposed architecture depicted on Fig. 1 is comprised from 4 sub-modules denoted as S1, S2, S3 and S4, respectively. S1 represents the disassembling of the original source code (assembly), which is carried out by using the *OllyDbg* disassembler. S2 is hosting the 3 commercial obfuscators used in this study, and this is the place where obfuscation happens. The output from this phase is an input to S3, which is similar to S1 except that it works on obfuscated code. The outputs of S1 and S3 are eventually fed into S4, where load profiles are generated for various instruction sets (IS) using the *CP Generator* component. The load profiles are estimated data based upon static code analysis, and [12] explains the processor architecture together with accurate figures of how much power is needed by various instructions. As we are interested in obtaining relative power consumption differences, the concrete processor architecture is not so relevant to the study. The original and obfuscated assemblies are binary files with an extension ".dll" and/or ".exe", respectively. Output files from the S1 and S3 sub-modules (denoted as "Original IS" and "Obfuscated IS"), obtained after disassembling the input artifacts, are textual files with an extension ".txt". The executable component named as "CP Generator" was written in the C# language. The resulting load profiles (generated inside the S4 sub-module) are illustrated in Section 5.2.

On the other hand, it has to be emphasized that the proposed architecture has its limitations. For example, control obfuscations might alter static control flow such that it still has a very similar "dynamic" measured consumption. Nevertheless, the proposed approach may forecast a very different "static" power consumption. In this respect, the suggested architecture may be used to highlight the pertinent aspects of this novel metric, which is the aim of this paper. A combined "static/dynamic" analysis would give a much better picture of the actual power consumption.

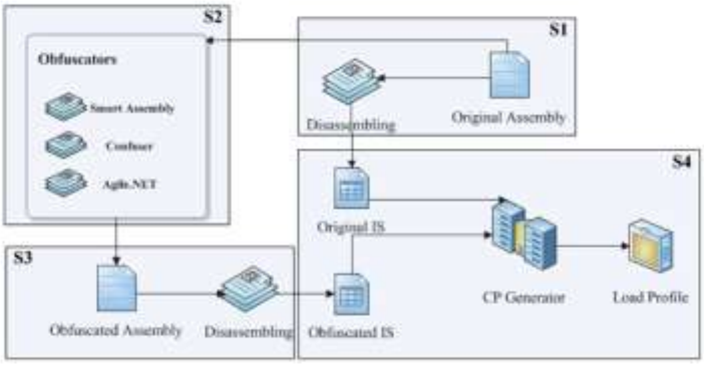
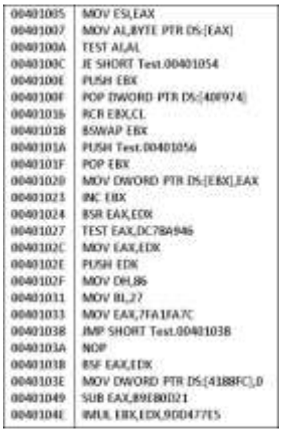


Figure 1

The architecture of the measurement scheme

5 Performance Evaluation

This section presents the evaluation results for three commercial obfuscators: the Agile.NET, Confuser and the Smart Assembly. The inputs were a simple matrix multiplication program and a multi-agent system instance. This section is divided into 2 parts: the first part contemplates the important obfuscation techniques (illustrated with small snippets) [28], while the second part provides a comparative view of obtained results for each obfuscator using the previously mentioned inputs. Lexical obfuscation is denoted as L1, lexical and data obfuscation as L2, while all three types as L3.



```
00401005 | MOV ESI,EAX
00401007 | MOV AL,BYTE PTR DS:[EAX]
0040100A | TEST AL,AL
0040100C | JE SHORT Test.00401054
0040100E | PUSH EBX
0040100F | POP DWORD PTR DS:[40F974]
00401016 | RCR EBX,CL
00401018 | BSWAP EBX
0040101A | PUSH Test.00401056
0040101F | POP EBX
00401020 | MOV DWORD PTR DS:[EBX],EAX
00401023 | INC EBX
00401024 | BSR EAX,EDX
00401027 | TEST EAX,DC78A946
0040102C | MOV EAX,EDX
0040102E | PUSH EDX
0040102F | MOV DH,86
00401031 | MOV BL,27
00401033 | MOV EAX,7FA1FA7C
00401038 | JMP SHORT Test.0040103B
0040103A | NOP
0040103B | BSF EAX,EDX
0040103E | MOV DWORD PTR DS:[4188FC],0
00401049 | SUB EAX,89E80D21
0040104E | IMUL EBX,EDX,9DD477E5
```

Figure 2

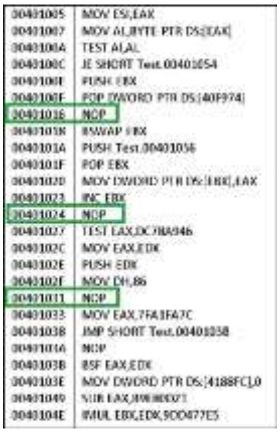
Original source code in assembly language

5.1 Obfuscation Techniques

Part of the source code used to exemplify different obfuscation techniques is shown in Fig. 2.

5.1.1 Insertion of Dead (Garbage) Code

Represents a very simple and frequently used technique based upon the insertion of specific assembly instructions. The only purpose of these are to change the code's size and shape while retaining the original functionality [29]. These extra instructions are totally insignificant and the most famous among them is NOP (No Operation). Fig. 3 shows an example of the code after insertion of these superfluous instructions.



```
00401005  MOV ESI,EAX
00401007  MOV AL,BYTE PTR DS:[EAX]
0040100A  TEST AL,AL
0040100C  JE SHORT Test.00401054
0040100E  PUSH EBX
0040100F  POP DWORD PTR DS:[40F974]
00401016  NOP
00401018  BSWAP EBX
0040101A  PUSH Test.00401056
0040101F  POP EBX
00401020  MOV DWORD PTR DS:[EBX],EAX
00401023  INC EBX
00401024  NOP
00401027  TEST EAX,DC7BA946
0040102C  MOV EAX,EDX
0040102E  PUSH EDX
0040102F  MOV DH,86
00401031  NOP
00401033  MOV EAX,7FA1FA7C
00401038  JMP SHORT Test.00401038
0040103A  NOP
0040103B  BSF EAX,EDX
0040103E  MOV DWORD PTR DS:[4188FC],0
00401049  SUB EAX,[illegible]
0040104E  IMUL EBX,EDX,9DD477E5
```

Figure 3

NOP operation as a dead code

Dead code may also be manually inserted by combining certain assembly instructions. The most trivial example revolving around an increase/decrease of the value inside the CX register is given in Table 1.

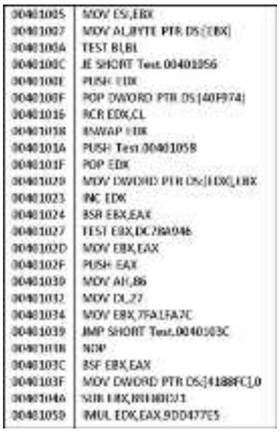
Table 1

Dead code manipulation examples

Commands	Explanation
SUB CX, 2 INC CX INC CX	A subtraction of 2 followed by a double increment does not change the value inside the CX register.
PUSH CX POP CX	The value of CX remains unchanged, because the CX register receives a previously saved value.

5.1.2 Register or Variable Reassignment

Represents a very simple and popular method based on switching registers of different types. An example of this kind of technique is presented in Fig. 4. In this example, the following switches have been carried out: EAX –> EBX, AL–>BL, EBX–>EDX, BL–>DL, DH–>AH, EDX–>EAX.



```
00401005  MOV ESI,EBX
00401007  MOV AL,BYTE PTR DS:[EBX]
0040100A  TEST BL,BL
0040100C  JE SHORT Test.00401056
0040100E  PUSH EDX
0040100F  POP DWORD PTR DS:[40F974]
00401016  RCR EDX,CL
00401018  BSWAP EDX
0040101A  PUSH Test.0040105B
0040101F  POP EDX
00401020  MOV DWORD PTR DS:[EDX],EBX
00401023  INC EDX
00401024  BSR EBX,EAX
00401027  TEST EBX,DC7BA946
0040102D  MOV EBX,EAX
0040102F  PUSH EAX
00401030  MOV AH,86
00401032  MOV DL,27
00401034  MOV EBX,7FA1FA7C
00401039  JMP SHORT Test.0040103C
0040103B  NOP
0040103C  BSF EBX,EAX
0040103F  MOV DWORD PTR DS:[4188FC],0
0040104A  SUB EBX,[illegible]
00401050  IMUL EDX,EAX,9DD477E5
```

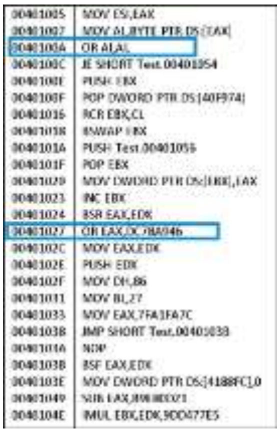
Figure 4
Register reassignment

5.1.3 Subroutine Reordering

Embodies a bit more complex scheme by randomly altering the order of execution of subroutines. This technique has factorial number of variants in regard to the number of subroutines [30]. For example, if the code contains 10 subroutines then the obfuscator may generate 10! variants of the original code.

5.1.4 Instruction Substitution

This method uses as input a library of equivalent instructions. The main idea is to replace one sequence of instructions with another one without changing the original functionality. This method has a high impact on the code's signature, it is very hard to reverse engineer it, especially in the case when the previously mentioned library is not known. Fig. 5 shows an example for instructions XOR and SUB. Evidently, it is hard to notice what is going on in the altered code.

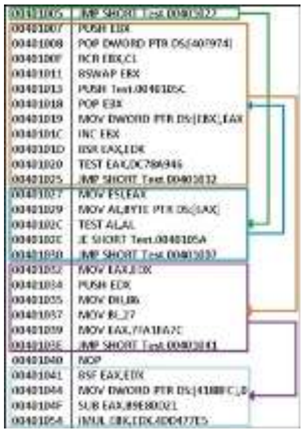


```
00401005 | MOV ESI,EAX
00401007 | MOV AL,BYTE PTR DS:[EAX]
0040100A | OR AL,AL
0040100C | JE SHORT Test.00401054
0040100E | PUSH EBX
0040100F | POP DWORD PTR DS:[40F974]
00401016 | RCR EBX,CL
00401018 | BSWAP EBX
0040101A | PUSH Test.00401055
0040101F | POP EBX
00401020 | MOV DWORD PTR DS:[EBX],EAX
00401023 | INC EBX
00401024 | BSR EAX,EDX
00401027 | OR EAX,DC78A946
0040102C | MOV EAX,EDX
0040102E | PUSH EDX
0040102F | MOV DH,86
00401031 | MOV BL,27
00401033 | MOV EAX,7FA1FA7C
00401038 | JMP SHORT Test.0040103B
0040103A | NOP
0040103B | BSF EAX,EDX
0040103E | MOV DWORD PTR DS:[4188FC],0
00401049 | SUB EAX,89E80D21
0040104E | IMUL EBX,EDX,9DD477E5
```

Figure 5
Instruction substitution

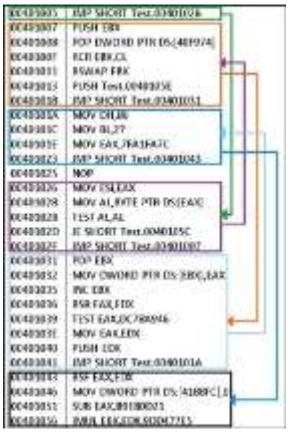
5.1.5 Code Transposition

There are two commonly used approaches. The first one is based on a random distribution of instructions and insertion of unconditional branches and/or jumps to retain the original flow of control [31]. The second one relies on the exchange of independent instructions and replacement of these with new ones. This approach is difficult to implement, since it is not easy to find such independent sets of instructions. Figs. 6 and 7 show a test code for both approaches.



```
00401005 | JMP SHORT Test.00401027
00401007 | PUSH EBX
00401008 | POP DWORD PTR DS:[40F974]
0040100F | RCR EBX,CL
00401011 | BSWAP EBX
00401013 | PUSH Test.0040105C
00401018 | POP EBX
00401019 | MOV DWORD PTR DS:[EBX],EAX
0040101C | INC EBX
0040101D | BSR EAX,EDX
00401020 | TEST EAX,DC78A946
00401025 | JMP SHORT Test.00401032
00401027 | MOV ESI,EAX
00401029 | MOV AL,BYTE PTR DS:[EAX]
0040102C | TEST AL,AL
0040102E | JE SHORT Test.0040105A
00401030 | JMP SHORT Test.00401007
00401032 | MOV EAX,EDX
00401034 | PUSH EDX
00401035 | MOV DH,86
00401037 | MOV BL,27
00401039 | MOV EAX,7FA1FA7C
0040103E | JMP SHORT Test.00401041
00401040 | NOP
00401041 | BSF EAX,EDX
00401044 | MOV DWORD PTR DS:[4188FC],0
0040104F | SUB EAX,89E80D21
00401054 | IMUL EBX,EDX,9DD477E5
```

Figure 6
Code transposition – unconditional branches



```
00401005   JMP SHORT Test.0040102B
00401007   PUSH EBX
00401008   POP DWORD PTR DS:[40F974]
0040100F   RCR EBX,CL
00401011   BSWAP EBX
00401013   PUSH Test.0040105E
00401018   JMP SHORT Test.00401031
0040101A   MOV DH,36
0040101C   MOV BL,27
0040101E   MOV EAX,7FA1F47C
00401023   JMP SHORT Test.00401043
00401025   NOP
00401026   MOV ESI,EAX
00401028   MOV AL,BYTE PTR DS:[EAX]
0040102B   TEST AL,AL
0040102D   JE SHORT Test.0040105C
0040102F   JMP SHORT Test.00401007
00401031   POP EBX
00401032   MOV DWORD PTR DS:[EBX],EAX
00401035   INC EBX
00401036   BSR EAX,EDX
00401039   TEST EAX,0C7B8946
0040103E   MOV EAX,EDX
00401040   PUSH EDX
00401041   JMP SHORT Test.0040101A
00401043   BSF EAX,EDX
00401046   MOV DWORD PTR DS:[4188FC],E
00401051   SUB EAX,B0100021
00401056   IMUL EBX,EDX,9D0477E5
```

Figure 7

Code transposition – independent instructions

5.2 Matrix Multiplication - Results

Here, we present the results of measuring 3 parameters (obfuscated code size, average power consumption per instruction and number of executed instructions) of obfuscated code as well as visually show their comparative values. Fig. 8 shows the load profile of the non-obfuscated matrix multiplication test program.

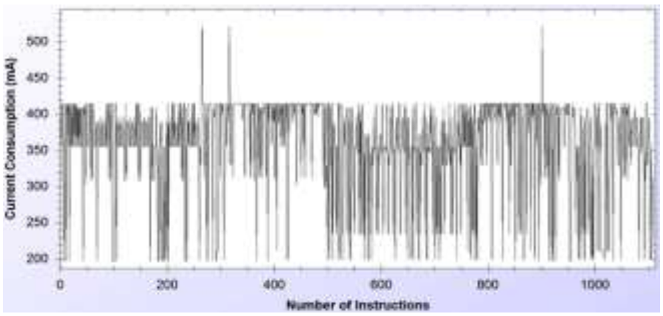


Figure 8

Non-obfuscated load profile of the matrix multiplication program

Case L1: comparative view of load profiles obtained by applying the L1 obfuscation technique is shown in Fig. 9. Evidently, the output generated by the Confuser considerably differs from the original version of the code. The average consumption per instruction has not changed too much (~3 mA), but the number of executed instructions have increased 4.5 times. This resulted in an elevated power consumption level while running the obfuscated code. Agile.NET has raised the code size by 35%, while the average consumption per instruction jumped by almost 14 mA. Smart Assembly produced the best results here. It reduced the number of instructions thankfully to the optimization applied when using the instruction substitution technique. Fig. 10 graphically shows the measured parameters.

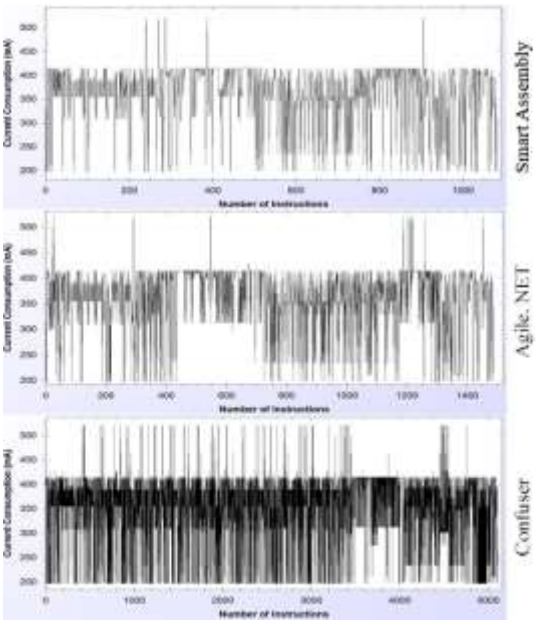


Figure 9

Comparative view of load profiles after L1 obfuscation cycle for all obfuscators

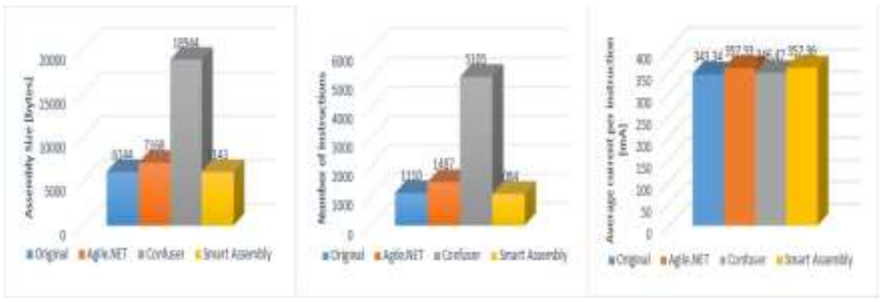


Figure 10

Graphical view for measured parameters after L1 obfuscation for original and obfuscated profiles

Case L2: this obfuscation cycle produced totally different load profiles, as it is obvious from Fig. 11.

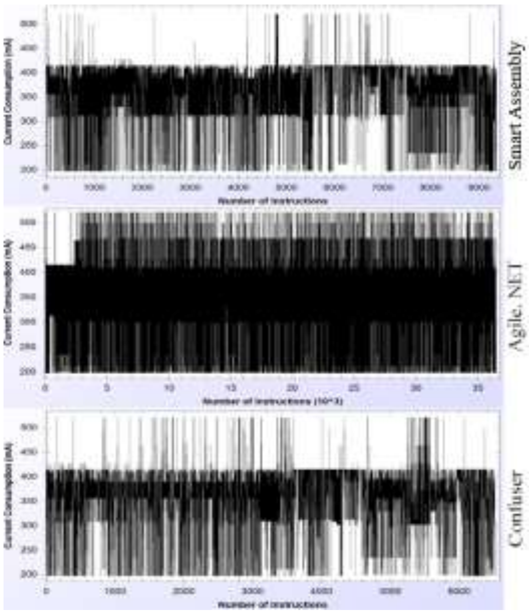


Figure 11

Comparative view of load profiles after L2 obfuscation cycle for all obfuscators

Agile.NET has produced a code with the biggest power demand. Despite the fact that the average power consumption per instruction remained the same, the code base has increased by a factor of 35 (huge amount of dead code, conditional branching instructions and jumps). This has caused an increase in power consumptions. Fig. 12 graphically shows the measured parameters for this cycle.

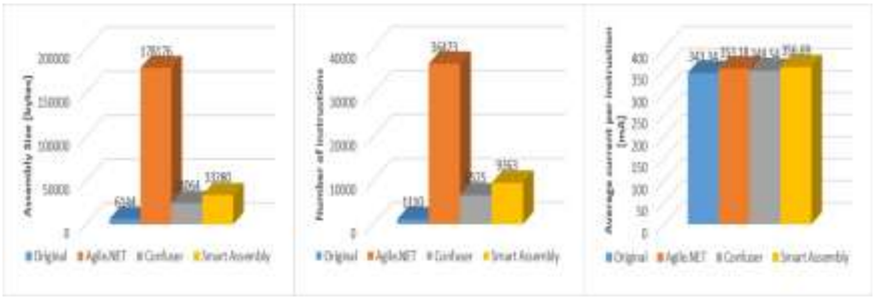


Figure 12

Graphical view for measured parameters after L2 obfuscation for original and obfuscated profiles

Case L3: the biggest changes have been observed in this cycle as depicted in Fig. 13. Confuser and Agile.NET have utilized extremely huge instruction sets, while at the same time the power consumption per instruction has also raised. This effect is best visible in the case of the Confuser, which is around 57 mA. The number of instructions are increased due to very large amount of dead code as well as high level of instruction substitutions and code transpositions. Smart Assembly gave the best results here, where the load profile is very similar to the original code with a higher consumption level due to dead code. Fig. 14 graphically shows the measured parameters for this cycle.

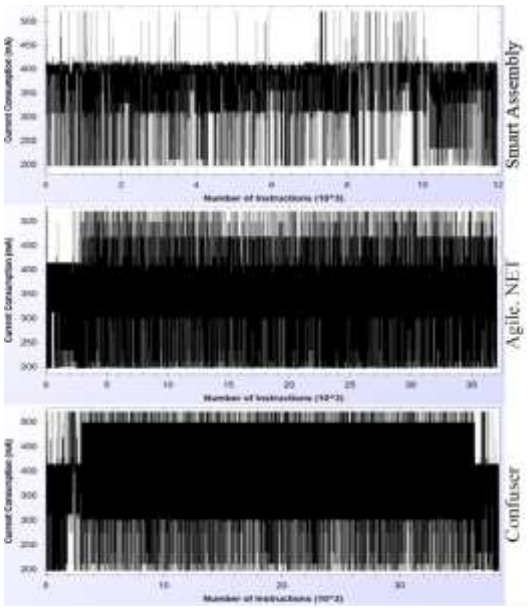


Figure 13

Comparative view of load profiles after L3 obfuscation cycle for all obfuscators

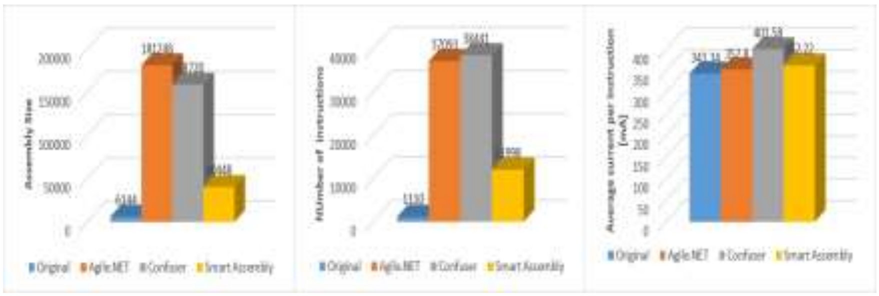


Figure 14

Graphical view for measured parameters after L2 obfuscation for original and obfuscated profiles

Based upon the presented results it is quite straightforward to qualify and classify code obfuscators in regard to their associated load profile, i.e. profile resulted by executing the corresponding obfuscated code. Apparently, a load profile nicely summarizes all the various effects on the code (number of instructions, type of instructions used, execution time, etc.), which would be quite hard to judge in advance just considering each of these effects independently.

5.3 Agents in a Proprietary Multi-Agent System

The associated energy consumption problems due to obfuscation are nicely illustrated in the following multi-agent system case study. The benefits of leveraging a multi-agent system in an electrical power distribution network is best reflected in an increased information exchange and processing capabilities of the network. Power networks are radial by nature, where a consumer may be conveniently represented by an agent situated inside a hierarchically organized structure. An example of simple power network is shown in Fig. 15.

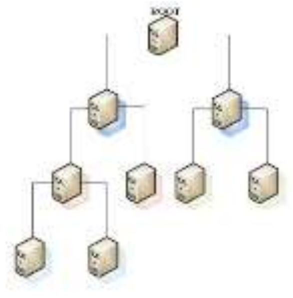


Figure 15

Sample power distribution network

The agents are grouped by zones, which are themselves organized in a hierarchical fashion [33]. One straightforward and simple method for creating such a zone hierarchy is to just follow the network's topology. Agents inside a zone actively exchange various operational data, like voltage levels, load flow, etc. Fig. 16 depicts one example of mapping agents to a hierarchy of zones.

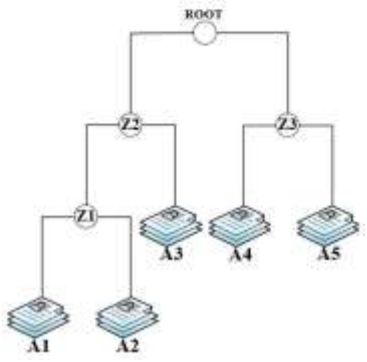


Figure 16

Power distribution network modeled with zones

The example shown in Fig. 16 contains 5 agents and 4 communication zones including the root zone. Agents A1 and A2 belong to the same zone Z1 and as such behave like equal peers. Zone Z2 aggregates zone Z1 and agent A3. In order for agent A3 to communicate with any agent from zone Z1 it needs to send messages toward Z1 zone's representative (it might be either agent from zone Z1). Fig. 17 shows the load profile of the non-obfuscated agent's control program.

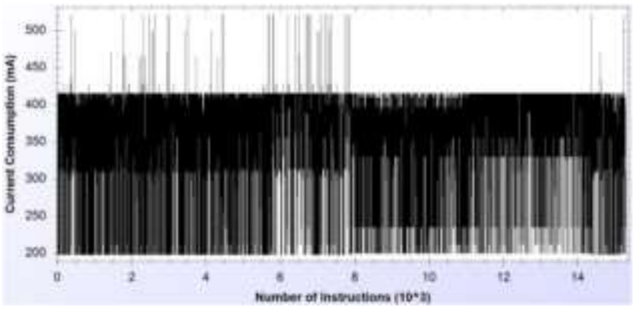


Figure 17
Non-obfuscated load profile of the agent's control program

Case L1: comparative view of load profiles got by applying the L1 obfuscation technique is shown in Fig. 18. Apparently, all outputs are similar by the number of instructions. Fig. 19 graphically shows the measured parameters.

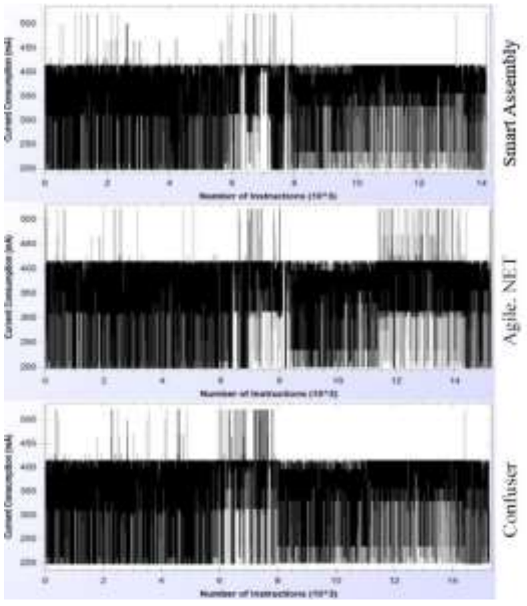


Figure 18
Comparative view of load profiles after L1 obfuscation cycle for all obfuscators

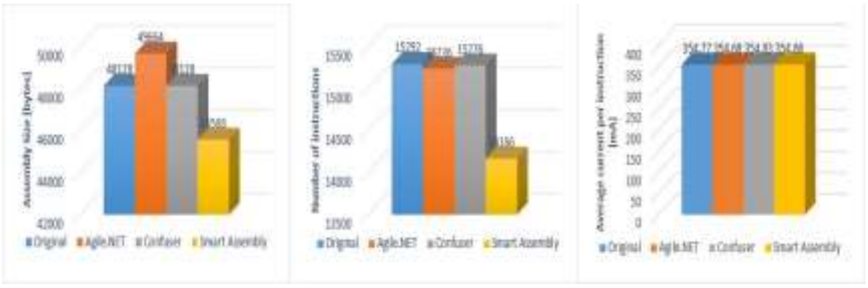


Figure 19

Graphical view for measured parameters after L1 obfuscation for original and obfuscated profiles

Case L2: as in the case of the matrix multiplication, this obfuscation cycle produced totally different load profiles (see Fig 20). The load profile, which resulted after leveraging Agile.NET, clearly emphasizes the fact that the number of instructions has doubled compared to the original case.

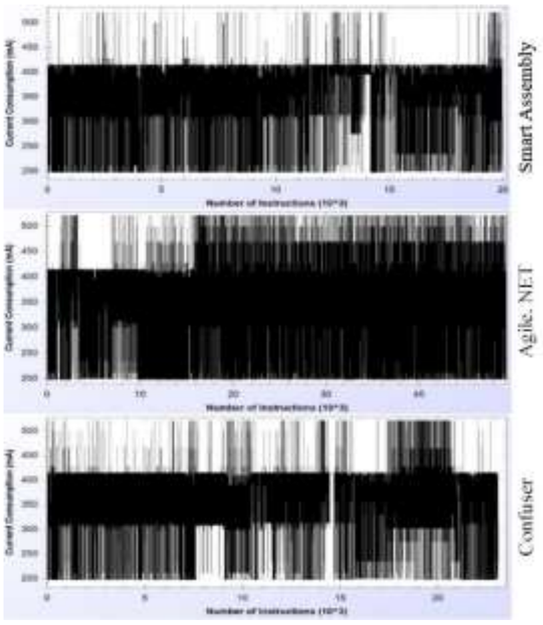


Figure 20

Comparative view of load profiles after L2 obfuscation cycle for all obfuscators

Fig. 21 graphically shows the measured parameters for this cycle.

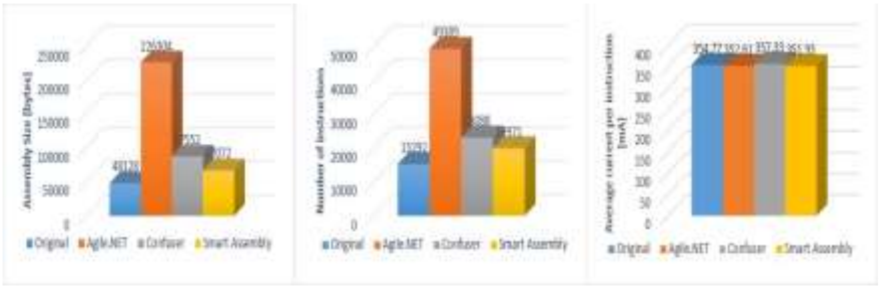


Figure 21
Graphical view for measured parameters after L2 obfuscation for original and obfuscated profiles

Case L3: the biggest difference was observed in the L3 cycle as depicted in Fig. 22. Agile.NET has utilized huge instruction sets, while the power consumption per instruction for Confuser raised almost up to 400 mA. Smart Assembly again gave the best results here, where the load profile is very similar to the original one (a higher consumption level is due to the presence of dead code). Fig. 23 graphically shows the measured parameters for this cycle.

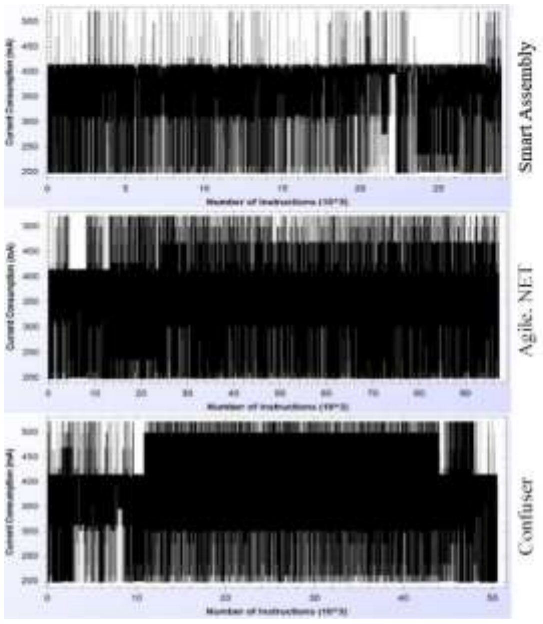


Figure 22
Comparative view of load profiles after L3 obfuscation cycle for all obfuscators

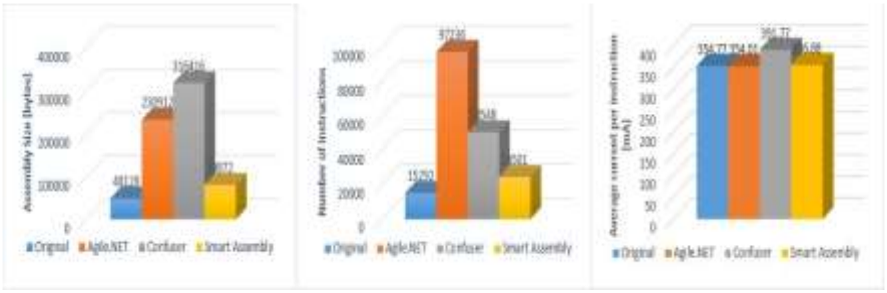


Figure 23
Graphical view for measured parameters after L3 obfuscation for original and obfuscated profiles

Conclusions and Future Work

This paper shows that load profile based classification of code obfuscators is assuredly a viable method. The power consumption pattern of an application represents its unique signature (print), which might be used as a quality metric for judging its energy efficiency. This aspect is gaining much popularity nowadays, especially with the introduction of new quality attributes such as sustainability. It is not anymore just enough to have high performing and secure applications. Energy considerations need to be brought into a foreground, as customer satisfaction will surely depend upon how long the battery on his/her mobile device will last while running programs. On the other hand, it is quite obvious that ubiquitous computing and the proliferation of code on remote devices requires well thought out mechanisms and technologies to protect and secure code. This is not only for the benefit of protecting intellectual properties, but also to save customers from running bogus code, which might inflict undesired damages. Obfuscation, is just one although very important, way to achieve this goal.

This paper has presented a novel load profile based power consumption metric to score the efficiency of code obfuscators. Using this metric it is now possible to analyze and exactly express how code obfuscation impacts power consumption.

The paper gives an overview and explanation of common types and techniques of obfuscation. These are all interrelated and their impact on power consumption explained.

The study includes evaluation results for 3 commercial obfuscators: Agile.NET, Confuser and Smart Assembly. To obtain experimental results a custom built measurement architecture was implemented based on static code analysis.

The chosen static code analysis approach might occasionally produce false results. Our future work is related to extend the framework to include dynamic measurements, too. This would definitely result in a much higher accuracy during evaluation of the various obfuscators.

Load profiles are very hard to be spoofed by malware. Classical signature based malware detection methods may be thwarted by various polymorphic packers. This is not the case with a load profile. Malware cannot even detect whether it is running under supervision or not from the viewpoint of its energy consumption. Although static analysis of executable code to detect a malware is a promising technique [32], load profiles represent a perfect side-channel to watch out for changes in behavior. There is no way to alter the original code without disturbing its load profile.

Besides detecting unusual changes in the power consumption due to software changes, load profiles may also be used to detect failing hardware. This is especially interesting in highly distributed environments.

References

[1] T. Guelzim, M. Obaidat: Chapter 8 – Green Computing and Communication Architecture, Handbook of Green Information and Communication Systems, Academic Press, 2013

[2] H. T. Mouftah, B. Kantarci: Energy Efficient Cloud Computing – A Green Migration of Traditional IT, Handbook of Green Information and Communication Systems, Academic Press, 2013, pp. 295-330

[3] C. Sahin, L. Pollock, J. Clause: How do Code Refactorings Affect Energy Usage?, Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement, 2014

[4] N. Hunt, P. Sandhu, L. Ceze: Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures, Proceedings of the 15th Workshop on Int. between Compilers and Computer Architectures, 2011, pp. 63-70

[5] C. Sahin, F. Cayci, I. L. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh: Initial Explorations on Design Pattern Energy Usage, Proc. of the First International Workshop on Green and Sustainable Software, 2012, pp. 55-61

[6] S. Christian Bunse: On Energy Consumption of Design Patterns, Proc. of the 2nd Workshop on Energy Aware Soft. Engineering and Development, 2013, pp. 7-8

[7] I. Manotas, C. Sahin, J. Clause, L. Pollock, K. Winbladh: Investigating the impact of Web Servers on Web Application Energy Usage, Proc. of the Second Inter. Workshop on Green and Sustainable Software, 2013

[8] M. Anderson: Rooting Out Malware with a Side Channel Chip Defense System, IEEE Spectrum, Jan. 2015

[9] V. Tiwari, S. Malik, A. Wolfe: Instruction Level Power Analysis and Optimization of Software, IEEE Trans. Very Large Scale Integration (VLSI Systems), 1996, pp. 326-328

[10] S. Nikolaidis, Th. Laopoluos: Instruction Level Power Consumption Estimation of Embedded Processors for Low Power Application, CIEEE Conf. Intelligent Data Acquisition and Advanced Computing Systems, Vol. 24, 2002, pp. 133-137

[11] J. T. Russel, M. F. Jacome: Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors, Proc. Conf. Computer Design, 1998, pp. 328-333

[12] V. Tiwari, S. Malik, A. Wolfe: Power Analysis of Embedded Software: A First Step Towards Software Power Minimization, IEEE Trans. Very Large Scale Integration (VLSI), 1994, pp. 437-445

[13] L. Benini, G. D. Micheli: System-Level Power Optimization: Techniques and Tools, ACM Trans. Design Autom. Elect. System, 2000, Vol. 5, No. 2, pp. 115-192

[14] H. Zeng, C. S. Ellis, A. R. Lebeck, A. Vahdat: ECO Systems: Managing Energy as a First Class Operating System Resource, Proc. Int. Conf. Architectural Support Programming Languages System, 2002, pp. 123-132

[15] W. Diffie, M. Hellman: New Directions in Cryptography, IEEE Trans. Information Theory, Vol. 22, No. 6, pp. 644-654

[16] C. S. Colberg, C. Thomborson: Watermarking, Tamper-Proofing and Obfuscation Tools for Software Protection, IEEE Trans. Software Eng., 2002, pp. 735-746

[17] J. T. Chan, W. Yang: Advanced Obfuscation Techniques for JAVA bytecode, The Journal of System and Software (Elsevier - 2004), Vol. 71, No. 2, pp. 1-11

[18] S. Drape: Generalizing the Array Split Obfuscation, The Journal of Information Sciences (Elsevier - 2006), Vol. 177, No. 1, pp. 202-219

[19] S. Praveen, P. S. Lal: Array Data Transformation for Secure Code Obfuscation, Proc. World Academy of Science, Engineering and Technology, 2007, Vol. 21

[20] C. S. Colberg, C. Thomborson, D. Low: Breaking Abstraction and Unstructuring Data Structures, IEEE Conf. Computer Languages, 1998a, pp. 28-38

[21] M. Sosonkin, G. Naumovich, N. Memon: Obfuscation of Design Intent in Object Oriented Applications, Proc. Digital Rights Management, 2003, pp. 142-153

[22] C. Wang, J. Hill, J. C. Knightm J. W. Davidson: Protection of Software-Based Survivability Mechanisms, Proc. Dependable System and Network, IEEE Computer Society - 2001, pp. 193-202

[23] S. Chow, Y. Gu, H. Johnson, V. Zakharov: An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs, Int. Conf. Information Security, 2001, pp. 144-155

[24] T. Toyofuku, T. Tabata, K. Sakurai: Program Obfuscation Scheme Using Random Numbers to Complete Control Flow, Proc. International Workshop Security in Ubiquitous Comp. Systems, 2005, Vol. 3823, pp. 916-925

[25] C. Collberg, C. Thomborson, D. Low: Manufacturing Cheap, Resilient and Stealth Opaque Constructs, ACM Symposium on Principles Programming Languages, 1998b, pp. 184-196

[26] A. Venkatraj: Program Obfuscation, Dep. of Computer Science Arizona University, 2003

[27] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, Y. Zhang: Experience with Software Watermarking, IEEE Proc. Comp. Security Applications, ACSAC - 2000, pp. 308-316

[28] S. Drape, A. Majumdor, C. Thomborson: Slicing Obfuscation - Design, Correctness and Evaluation, ACM Proc. Digital Rights Management, 2007, pp. 70-81

[29] I. You, K. Yim: Malware Obfuscation Techniques - A Brief Survey, IEEE International Conference on Broadband, Wireless Computing, Communication and Application, 2010, pp. 297-300

[30] A. Balakrishnan, C. Schulze: Code Obfuscation Literature Survey, Computer Science - University of Wisconsin, 2005, USA

[31] W. Wong, M. Stamp: Hunting for Metamorphic Engines, Journal in Computer Virology, 2006, Vol. 2, pp. 211-229

[32] M. Christodorescu, S. Jha: Static Analysis of Executable to Detect Malicious Patterns, Proc. of Conference on USENIX Security Symposium, 2003, Vol. 1, pp. 169-186

[33] V. Renesse, K. P. Birman, W. Vogels: Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management and Data Mining, ACM Trans. Computer Systems, Vol. 21, pp. 164-206, 2003