International Journal of Software Engineering and Knowledge Engineering

World Scientific
www.worldscientific.com

PARAMETERIZATION AND EVALUATION OF INTERMEDIATE LEVEL OBFUSCATOR

DMITRIY DUNAEV

Department of Automation and Applied Informatics, Budapest University of Technology and Economics
H-1117, Magyar tudósok krt. 2., Budapest, Hungary
dunaev@aut.bme.hu

LÁSZLÓ LENGYEL

Department of Automation and Applied Informatics, Budapest University of Technology and Economics
H-1117, Magyar tudósok krt. 2., Budapest, Hungary
lengyel@aut.bme.hu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Obfuscation is a technology that secures software artifacts from reverse engineering by making its cost prohibitively high. Intermediate level obfuscator implements the defensive mechanisms inside the software, and owing to high potency and resilience, can successfully secure the sensitive software components. This paper provides an analysis and parameterization of the obfuscator, as well as a method of fine-tuning and evaluating obfuscating transformations in terms of potency, resilience and cost.

Keywords: Intermediate level obfuscation; software control flow; parameterization; complexity metrics; potency; resilience; infocommunications.

1. Introduction

Fast developments in infocommunications, ICT systems, computer networks and Internet technologies have created the necessity for researching in the areas of securing data.

ICT devices of the future, i.e. Internet of Things, are expected to perform a large variety of sensing and inference tasks, and to interpret and transform information rather than merely transmitting raw, uninterpreted data. They will be forming an increasingly integrated and global ecosystem for processing, storing, transporting information and managing content [1]. The appearance of large amounts of data sets (Big Data), which reflects the habits, thoughts, emotions and physiology of users, is creating an increasing urgency to the protection of sensitive code.

The objective of obfuscation techniques is to prevent, or at least to complicate, the interpretation, decoding, analysis, or reverse engineer of software. Such techniques relate to methods and apparatus for increasing the structural complexity of a program code. They are implemented by inserting, deleting, or permutating the identifiable information structures in the software. As a result, the difficulty of decompilation and reverse engineering highly increases [2].

Obfuscation can be also used to address privacy concerns. Paper [3] discusses an obfuscation-based approach that enables users to follow privacy-sensitive channels, while, at the same time, making it difficult for the microblogging service to find out their actual interests. Paper [4] presents a general-purpose obfuscator for polynomial-size circuits applying homomorphic encryption, and paper [5] describes the construction and usage of obfuscators for probabilistic programs. An example how adding obfuscation to programs can protect users from various privacy threats is described in [6].

The advantage of obfuscation is that it resides on implementing the defensive mechanisms inside the application software. An obfuscated application usually does not suffer from delays due to network limitations and does not require any hardware dongles.

Merits and demerits of different obfuscation techniques, and introduction to our obfuscation method can be found in [7]. We do obfuscating transformations using a target platform independent intermediate code. Such code is usually a description of the high-level statements with some simpler instructions that accurately represent the operations of the source code statements. This is important that the code is not executed in a real processor; it is only an internal representation of a program.

The advantage of intermediate level obfuscation is that we can create a target-independent infrastructure. This means that for each platform that needs to be supported we only have to write the “machine code to intermediate code” and “intermediate code to machine code” translators, and the obfuscator logic does not change. If we need to port our obfuscator to another platform, we only need to write another translator for a new processor.

The rest of this paper is organized as follows. In Section 2, we discuss the related work and justify the intermediate level obfuscation. In Section 3, we present the obfuscator designed and developed by us. We describe fine-tuning parameters of obfuscator and the used complexity metrics. We then show and justify the selection of obfuscator parameters, and provide qualitative evaluation of potency, resilience and cost of obfuscating transformations. Finally, in Section 4, we draw conclusions.

2. Related Work

The essence of obfuscation is to entangle the code and eliminate the majority of logical links in it or, in other words, to transform the code so that it becomes complex enough for analysis and unauthorized modification. A general method for obfuscating programs would solve many open problems in cryptography. However, Boaz Barak has presented families of functions that cannot be obfuscated, since there exists a predicate that cannot be computed from black-box access to a random function in the family, but can be computed from a non-black-box access to a circuit implementing any function in the family [8, 9]. A later paper of Goldwasser and Kalai [10] shows the impossibility and improbability of obfuscating more natural functionalities.

In our approach, obfuscation is understood as a program transformation technique, which attempts to convolute the low-level semantics of routines, and aims to counteract the reverse engineering. We have shown in [7] that by restricting ourselves to automatic

generation of additional fake operations, we cannot guarantee the absence of effectively optimized deobfuscation algorithm.

The solution lies in a global fake context. With respect to a routine, we define two contexts: local and global. Local context is private to a particular routine and expires (disappears) when the routine execution is finished. An example of such context is local variables stored on the local stack. Global context may be shared across routines and does not expire immediately after a routine execution. It can be composed from different global parameters, such as pointers to memory buffers, control flow graph parameters, and initializing values, provided as input to a routine. The presence of fake global context has direct influence on the obfuscator complexity.

A general approach to intermediate level obfuscation and a bird-eye view of an obfuscation algorithm is presented in [11]; a technique of machine code translation to intermediate representation is discussed in [12]. In this paper, we are to present an implemented prototype with one module, which supports x86 platform executables.

3. Contribution

Intermediate level obfuscation method, being platform-independent, offers important advantages with respect to cost, configurability and portability. For intermediate representation, we use a three-address code (TAC), since TAC is not specific to a language being implemented (unlike P-code for Pascal and Bytecode for Java). In addition, the TAC instruction set is sufficient in translation of assembly code [13].

Fig. 1 shows the high-scale structure of obfuscator. During the development phase, we have separated platform-specific and platform-independent components to obtain the module structure. Data exchange between TAC component and platform-specific modules is standardized (XML). In this case, the platform-specific modules turn out to be just plug-ins for the TAC component. During the last 12 months, x32 and x64 modules have been developed and tested. Consequently, one can introduce new additional platform-specific models if needed.

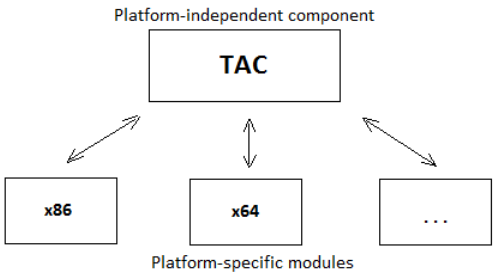

Fig. 1. General structure of IL Obfuscator.

The task of a platform-specific module is a translation of low-level representation (machine code or assembler code) to TAC, and backwards – generation of low-level representation from TAC. The platform-specific modules should incorporate appropriate platform-specific obfuscation techniques, e.g. polymorphic code generation. Polymorphic

code generator can be implemented as a part of a platform-specific module, or as a standalone application.

3.1. *Description of fine-tuning parameters*

Since the obfuscation logic does not change between supported platforms and software types, we have to provide the acceptable level of adjustment and parameterization to a specific need.

To provide the most possible user control and tunability of the obfuscation process, we have developed an IL Obfuscator tool [7,11]. The choice of parameters was driven by theoretical research and empirical evidence obtained during development of IL Obfuscator. Some of the parameters define the code transformation process, and others adjust the specific steps, providing the best possible result regarding potency and resilience. Now we present the parameters, which define the obfuscating process and give their brief description.

Multiple Obfuscation enables running the obfuscation process for multiple times, and the user can define which steps should be performed in each run. This option is intended to increase the deobfuscation resistance by multiple running of different code transformations, and by that, increasing the complexity of the obfuscated code. Applying multiple obfuscation results in excessive need for resources, yet the complexity increases greatly. Therefore, we can recommend using multiple obfuscation only when there is a need of high-level deobfuscation resistance.

Global maximum/minimum values define the most and least reasonable values in the code context. These values are used during generation of fake input parameters, fake conditional jumps, fake instructions, and provide the conformity of obfuscated code with the original one.

The IL Obfuscator also provides a number of parameters for fine-tuning the obfuscation quality. We should note, however, that the better quality of obfuscation we want to achieve, the greater is the requirement for additional resources.

Percentage of fake local variables specifies the ratio of fake local variables to be created compared to original ones. The number of original variables is taken as 100%.

Number of fake input parameters specifies the number of fake input parameters to be created during the obfuscation process.

Unconditional meshing probability specifies the probability of substituting an unconditional jump by a sequence of conditions that verify the fake input parameters.

Conditional meshing probability specifies the probability of substituting a conditional jump by a series of consequent conditions equivalent to the original one.

Double meshing specifies whether conditional meshing should be applied to fake conditional jumps.

Fake conditional jump probability specifies the probability of creating a conditional jump that verifies whether valid fake input parameters were provided. Turning this option on will expand the control flow graph and make it irreducible.

Minimum number of instructions per basic block specifies the least possible number of instructions in a single basic block. This option can conceal the vulnerable code structures.

Random number generator is used at all steps for setting probabilities, defining constants, selecting relational operators, setting up intervals, etc. By that we assure different output at each run of IL Obfuscator, and therefore the analysis of several obfuscated programs will give no unilateral advantage to a reverse engineer.

3.2. *Complexity metrics*

To guide the process of evaluation of obfuscating transformations, we need to characterize the obfuscation process. For that we use complexity metrics that cover both control flow and data flow complexities.

Cyclomatic complexity (McCabe metric) [14] is an indication of the number of paths through a function. A path is a legal sequence of statements from the start of a function to its end. The lower the number, the less complex the code is. We calculate cyclomatic complexity with the following formula: $V(G)=e-n+p$, where G stands for the given control flow graph, e and n denote the number of edges and notes respectively, and p stands for the number of connected components.

Language complexity (Halstead metric) [15] is a quantitative measure based on the number of operators and operands present in the code. From it, we can derive several submetrics, such as:

- *Program length*, which is the sum of occurrences of operators and occurrences of operands. Calculation: $N=N_1+N_2$, where N_1 stands for the number of occurrences of operators, and N_2 is the number of occurrences of operands.
- *Program vocabulary*, which is the sum of the number of distinct operators and the number of distinct operands. Calculation: $n=n_1+n_2$, where n_1 and n_2 stand for the number of distinct operators and operands, respectively.
- *Volume*, which can be interpreted as "the number of comparisons needed to write the program" or "the number of bits required to code the program" [16]. We calculate it by the formula $V=N*log_2n$, where N denotes the program length, and n denotes the vocabulary.
- *Difficulty*, which is an indicative measure of code readability. Calculation: $D=(n_1/2)*(N_2/n_2)$, where n_1 denotes the number of distinct operators, N_2 denotes the number of occurrences of operands, and n_2 denotes the number of distinct operands.
- *Effort*, which is a measure of "the number of elementary mental discriminations" [16]. Following that definition, we calculate effort as the product of the difficulty and the volume, that is $E=D*V$.

Data flow complexity (Elshoff metric) [17] is based on the number of variables referenced but not defined in a basic block. The Elshoff's metric in terms of the routine can be achieved by adding the data flow complexities regarding each basic block of the

routine. The latter can be calculated by the following formula: $C=R-D$, where R stands for the number of referenced variables, and D stands for the number of defined variables.

Oviedo complexity metric [18] provides an overall code complexity based on both control flow complexity and data flow complexity. Considering that, we calculate Oviedo complexity metric by the following formula: $C=a*CF+b*DF$, where CF is the control flow complexity; DF is the data flow complexity; a, b are weighting factors. We assume a and b to be equal 1, since we do not weight up either of them.

Decisional complexity (McClure metric) [19] is the sum of the number of comparisons and the number of control variables referenced in the routine code. The used formula just reflects the definition: $D=C+V$, where C denotes the number of comparisons in the code, and V denotes the number of control variables referenced in the code.

Data complexity (Chapin metric) [20] is a quantitative measure of information being used in a method. The claim is that the more variables we have, the harder it is to understand, modify, etc. For quantitative determination of data complexity, we have used the following formula: $D=P+2M+3C+T/2$, where P denotes the number of inputs and global variables; M denotes the number of modified and new (declared) variables in the code; C denotes the number of variables that were used in determining control flow direction; T denotes the number of unused variables.

3.3. *Parameterization of obfuscating transformations*

Let us use a well-known least common multiple (LCM) algorithm, since it features the majority of programming techniques in the list below, that is: integer data types; arithmetic operations, such as basic assignment, modulo division, multiplication, division; relational operations; repetition structures; multiple functions; external function calls.

The original C code listing for the LCM algorithm is shown in Fig. 2. The algorithm determines the least common multiple of two integer numbers, both of which are to be provided by the user. There is no input validation and we suppose that both numbers are non-negative integers.

```
int LCM(int num1, int num2) {
    int x,y,r;
    x = num1;
    y = num2;
    do {
        r = x % y;
        x = y;
        y = r;
    }
    while (r > 0);
    return (num1 * num2) / x;
}
```

```
int main() {
    int num1,num2,lcm;
    scanf("%d",&num1);
    scanf("%d",&num2);
    lcm = LCM(num1,num2);
    printf("%d",lcm);
    return 0;
}
```

Fig. 2. The C code listing for the LCM algorithm

The code in Fig. 2 was compiled to an executable. Having the executable, we suppose that we have no other knowledge about the original source code except what can be obtained by decompiling the executable.

```
FUNCTION
        Name: sub411B00
        Inputs: v0, v1
        Outputs: v7
        Locals: v2, v3, v4, v5, v6
    v4 := v0
    v3 := v1
LABEL1:
    v5 := v4 % v3
    v2 := v5
    v4 := v3
    v3 := v2
    if v2 > 0 goto LABEL1
    v6 := v1 * v0
    v7 := v6 / v4
    return v7

FUNCTION
        Name: sub411420
        Inputs:
        Outputs: v0
        Locals: v1, v2, v3, v4, v5
    param v2
    call scanf_int 1
    retrieve v2
    param v1
    call scanf_int 1
    retrieve v1
    param v2
    param v1
```

```
call sub_411B00 2
retrieve v4
v0 := v4
param v0
call printf_int 1
return v0
```

Fig. 3. The TAC code listing for LCM algorithm after decompilation

The platform-specific module of IL Obfuscator disassembles the executable, analyzes it and translates the assembler instructions to three-address code. The latter is passed to a platform-independent module. That is, the output of a platform-dependent module serves as the input of a platform-independent three-address code obfuscator. After obfuscating transformations are done, the process is repeated in reverse direction: TAC to assembly instructions, assembly instructions to executable.

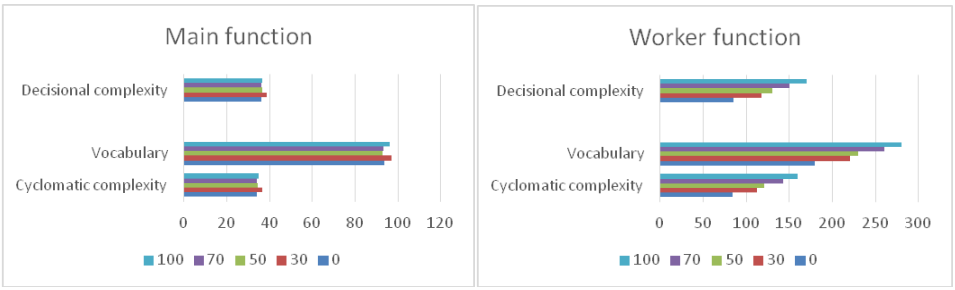

Fig. 4. Fine-tuning the Conditional meshing probability, [%]

In the listing on Fig. 3, the decompiled *sub_411B00* function corresponds to the original worker function (LCM), and *sub_411420* corresponds to the original *main* respectively. To obtain highly durable obfuscation results and face the possible time/cost limitations, one should select appropriate obfuscation parameters that will guide the transformation process. We are to show the impact of such parameters to obfuscation results on the routine in Fig. 3. This routine contains two functions: a very simple *main*, and a more complicated *LCM*, which we shall call a worker function.

One of the parameters to select is the *Conditional meshing probability*. It can vary from 0 to 100%. The parameter specifies the probability of substituting a conditional jump for a series of consequent conditions equivalent to the original one but using different constants to counteract the signature search. The drawback, however, is increase of executable file size and execution slowdown to a very slight degree.

Our research shows that the probability of conditional meshing has a direct impact on metrics related to execution flow and data. However, the ratio of this impact depends on the “density” of conditional jumps in the code that has been obfuscated. Fig. 4 shows that since the main function does not contain conditional jumps, its complexity is not influenced by the examined parameter. On the other hand, the worker function has

conditional jump instructions, and we observe a high increase in its complexity after obfuscation.

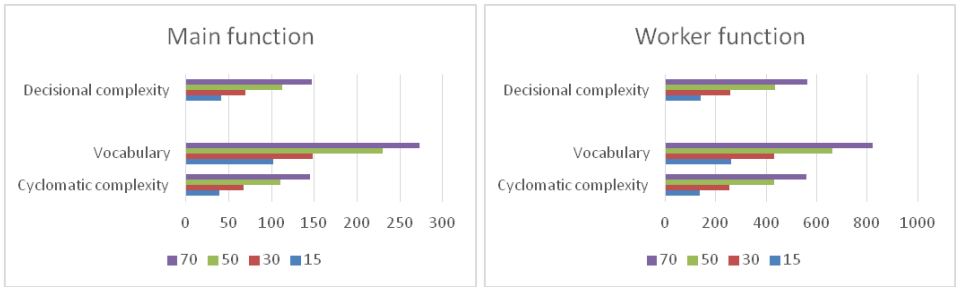

Fig. 5. Fine-tuning the Fake conditional jump probability, [%]

During evaluation, we apply 100% value to this parameter, since it highly increases the complexity and has no other drawbacks except for the slight increase in executable size.

One more parameter to be discussed would be the *Fake conditional jumps probability*. It can vary from 15 to 70%. It specifies the probability of creating a special conditional jump from an empty *NoOperation* instruction. Such special conditional jumps verify whether valid fake input parameters were provided. They are also used to expand the control flow graph and ensure its irreducibility; therefore, the lower limit is set to 15%.

Our research and experiments proved that the increasing probability of fake conditional jumps results in increasing complexity of original functions, since there are no preconditions as for conditional meshing. As we see in Fig. 5, the ratio of this impact does not depend directly on function type; a complexity increase of 4-5 times has been observed during other test cases as well. However, during fine-tuning we observed a sensible delay in obfuscation time. Let us examine the impact of *Fake conditional jumps probability* to obfuscation time.

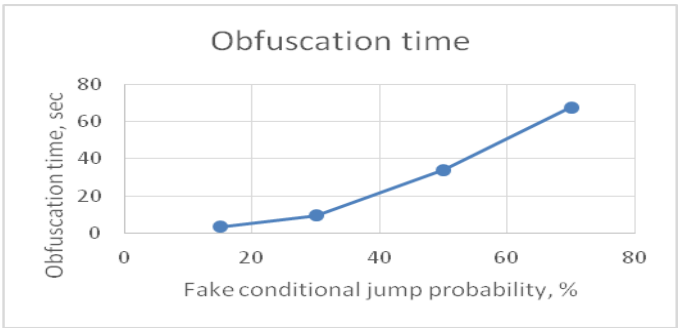

Fig. 6. Obfuscation time as a function of fake conditional jump probability

The increasing obfuscation time can be unacceptable for some use cases. We observed 30-40 times increase in obfuscation time at 70% compared to 15%. For other test cases, the delay reached 50 times.

During evaluation, we applied 30% value to this parameter. By that, we tried to find a golden middle between obfuscation time and the desired complexity. However, for cases with specific needs, other parameter values from the 15…70% scale can be applied. The higher the parameter value is, the higher the resulting complexity is. However, we have to face an increase in obfuscation time.

Another parameter to be discussed is the *Double meshing*, which specifies whether conditional meshing should be applied to fake conditional jumps. This is a binary parameter, which allows for two possible values: *true* (double meshing is applied) and *false* (double meshing is not applied).

Comparing Fig. 7 and Fig. 4, we conclude that applying double meshing increases the complexity metrics considerably in those functions that originally had conditional jumps in the source code.

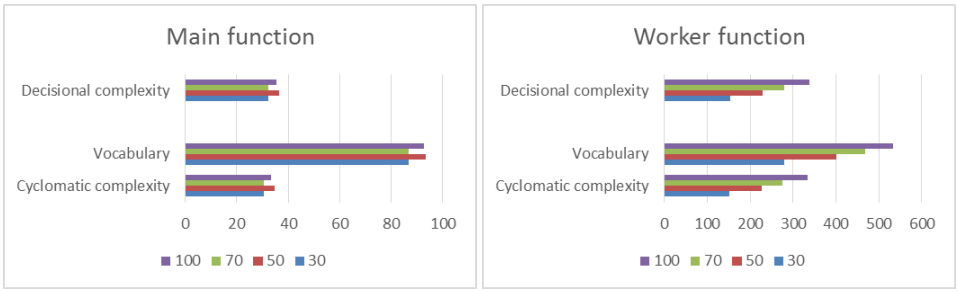

Fig. 7. Fine-tuning the Conditional meshing probability while Double meshing is applied, [%]

Moreover, 30% for *Conditional meshing probability* with *Double meshing=true* gives us almost the same complexity as 100% for *Conditional meshing probability* with *Double meshing=false*. We can conclude that *Double meshing* parameter has a strong impact on the overall complexity, and therefore has to be applied when strong obfuscation result is needed.

Double meshing should not be confused with *Multiple obfuscation*. The latter enables applying the selected obfuscating transformations for multiple times. Multiple obfuscation is intended to increase the deobfuscation resistance and by that to increase the complexity of the obfuscated code. At the same time, we can leave out some entangling transformation if needed for specific cases.

However, Fig. 8 shows that the increase in obfuscation time is what we pay for the high complexity. Since the obfuscation takes place only once, for the majority of use cases this drawback is not relevant. Therefore, during evaluation we select *Double meshing* parameter to be *true*.

Applying multiple obfuscation would result in excessive need for resources. At the same time, our measurements showed that the complexity would also increase. Therefore, we can recommend using multiple obfuscation only when there is a need of very high

complexity and strong deobfuscation resistance. However, since in order to obtain clear first-order results, we do not use multiple obfuscating transformations during evaluation.

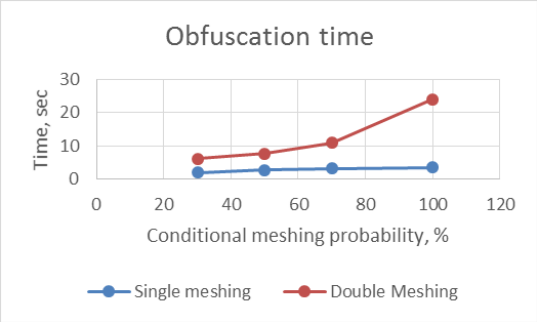

Fig. 8. Obfuscation time vs. Conditional meshing probability with and without Double meshing, [%]

The other parameters of IL Obfuscator have been examined and fine-tuned in the same manner as those shown above. The resulting values are grouped into Table 1.

Table 1. Parameters used during evaluation.

Parameter name	Unit description	Selected value
Fake local variables	% to original	100
Fake input parameters	positive integer	3
Unconditional meshing	probability value	100
Conditional meshing	probability value	30
Double meshing	binary	true
Fake conditional jumps	probability value	30
Minimum number of instructions per basic block	positive integer	10
Multiple obfuscation	binary	false

Having selected the parameters, the obfuscation process is composed of the following steps: creation of fake input parameters, creation of fake local variables, constants coverage, unconditional meshing, conditional meshing, fake conditional jumps generation, and fake instructions generation.

3.4. *Evaluation of obfuscating transformations*

To evaluate IL Obfuscator in a more realistic setup, where users are interested in obfuscating different kinds of functions and routines, we created a number of test cases. These test cases incorporate various types of programming techniques that are used in conventional programming:

- arithmetic operations (addition, subtraction, etc.);
- relational operations (less, greater or equal, etc.);

- logical operations (*and*, *or*, *not*, etc.);
- repetition structures, including pre- and postcondition loops (*for*, *do… while*, etc.);
- decision structures (*if…else*, *switch*/*case*);
- external functions (*scanf*, *abs*, etc.).

The number of functions is not limited, however for testing and demonstration purposes we used only two functions – the *main* function and some *worker* function; the latter does the calculations and returns the result to main. Utilizing this approach, we show the capability of IL Obfuscator to obfuscate function calls on data level.

Each test case aims at testing different repetition structures (pre/post-conditioned loops, counters); relational, arithmetic, and logical operators; embedded and multiple function calls, etc. For that, we have written a number of test algorithms: Fibonacci sequence, Units Conversion, Least Common Multiple, Greatest Common Divisor, Geometric Sequence, Factorial, etc.

We have worked out a method of evaluating platform-independent obfuscating transformations, which is based on Halstead submetrics, complexity and potency metrics.

First, we have calculated the Halstead submetrics for the original routines. Then, after having applied obfuscating transformations, we calculated the submetrics again. The ratio of how many times the submetrics have increased is grouped into Table 2.

Table 2. Complexity increase ratio [obfuscated/original].

Halstead submetric name	Fibonacci	Least common multiple	Greatest common divisor
Program vocabulary	26.6	21	32.5
Program length	495	458	708
Volume	1133	932	1574
Difficulty	27	24	31
Effort	29505	22835	48963

We see that indicators such as program vocabulary and difficulty have increased by 20-30 times. Halstead effort, which is the quantitative measurement of the mental effort required to develop or maintain a program, has increased very significantly by 20000-50000 times. Our tests have never resulted in increase of effort less than four orders of magnitude.

The greater values for Greatest common divisor algorithm result from high initial complexity of the original code, e.g. presence of loops and conditional jumps. The more manifold the original routine is, the more possibilities there are for obfuscating transformations to increase the code complexity. However, we should not forget that the fake code should be somewhat similar to the original one in order not to be deobfuscated automatically, and this fact limits our choices for a very simple code.

Trying to qualitatively measure potency as a whole becomes difficult since the analysis would be based on human cognitive ability. Therefore, the potency metrics

provide quantitative results of how much obscurity is added to the program that prevents human beings from understanding it.

Table 3 presents the ratio of potency metrics. It contains the results, applied to three test cases. The numbers show that the applied transformations aimed at adding the obscurity, resulted in high increase of overall code complexity. There is no huge difference between potency ratios for the three test cases.

Table 3. Potency metrics increase ratio [obfuscated/original].

Metric applied	Fibonacci	Least common multiple	Greatest common divisor
Cyclomatic complexity	108	115	178
Data flow complexity	325	237	374
Oviedo's complexity	253	210	325
Decisional complexity	110	117	179
Data complexity	3.5	2.7	3.7

According to Collberg et al. [21], obfuscating transformation is potent if a measure of the extent how transformations change the complexity of obfuscated code is positive. According to our measurements provided in Tables 2 and 3, the IL Obfuscator can be named **potent**, and we can classify its potency as **high**.

The resilience, however, is different from potency. It is a measure of how strong the program can resist an attack against a deobfuscator. Such attack can be defined as an attempt to transform the code back to its original structure by an automatic tool. Considering that the cognitive ability of a computer program is far inferior to that of humans, the resilience of obfuscating transformation has a large-scale impact on the overall quality of obfuscation.

According to [21], the resilience of obfuscating transformation can be obtained as a combination of programmer effort and deobfuscator effort.

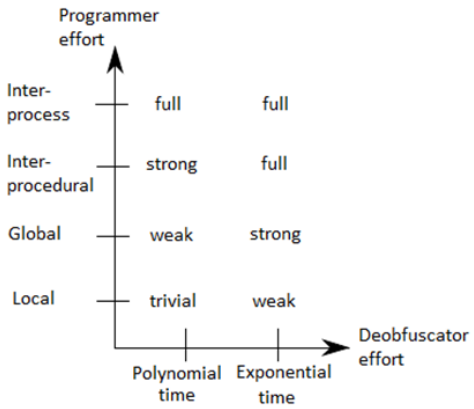

Fig. 9. Qualitative measure of resilience [21]

Programmer effort, the work required to automate the deobfuscation of a transformation, is measured as a function of the scope of obfuscating transformations. The deobfuscator effort is a qualitative measure of the execution time and space required by an automatic deobfuscator to effectively reduce the potency of obfuscating transformations.

Due to the usage of fake global context (*Fake input parameters = 3*), the obfuscating transformations applied in IL Obfuscator are inter-procedural, that is they affect information flow between procedures (functions). The deobfuscator effort can be characterized by exponential time, as shown in [22]. Therefore, the resilience of IL Obfuscator is **full**.

The cost of obfuscation refers to how much computational overhead is added to an obfuscated routine. Unlike potency and resilience, cost presents negative impact on the overall quality of obfuscating transformations.

Cost is usually measured by comparing resources (time, space, memory, etc.) needed to execute the obfuscated program with respect to that for the original one. In other words, the obfuscation cost is the time/space penalty, which obfuscating transformations incur on the routine.

In order to evaluate the cost of IL Obfuscator, we conducted multiple tests regarding execution time, memory usage, and executable size. The respective results are shown in Table 4.

Table 4. Increase ratio of cost indicators [obfuscated/original].

Indicator name	Fibonacci	Least common multiple	Greatest common divisor
Execution time	5%	4%	6%
Memory usage	23%	11%	14%
Executable size	19 times	22 times	29 times

Collberg et al. [21] offers the qualitative measure of obfuscator cost based on the extra execution time/space requirements of obfuscated routine *Obf(P)* in comparison to original routine *P*. The obfuscating transformations are:

- dear, if executing *Obf(P)* requires exponentially more resources than *P*;
- costly, if executing *Obf(P)* requires $O(np)$, $p>1$ more resources than *P*;
- cheap, if executing *Obf(P)* requires $O(n)$ more resources than *P*;
- free, if executing *Obf(P)* requires $O(1)$ more resources than *P*.

Since executing the obfuscated routine requires linearly more resources than for the original one, we can classify the overhead added by IL Obfuscator as **cheap**.

4. Conclusion

In the paper, we have presented an Intermediate Level Obfuscator, focusing on its parameterization and evaluation. We have shown that our obfuscator implements the defensive mechanisms inside the software, and can be named highly potent, fully resilient, and cheap. Owing to high potency and resilience, it can successfully secure the

sensitive software components. Due to its low cost, it further allows users to directly interact with the obfuscated systems without additional firewalls and/or gateways.

The great advantage of the IL Obfuscator is that it can be applied to partitioned routines. Even if there is no possibility to add a fake global context to the original routine as a whole, it can always be done with respect to the partitioned routines with nesting level greater than zero. Another advantage is the ability to obfuscate already obfuscated programs, or to obfuscate the selected routines of a program. By that, we obtain a multistage obfuscation technique.

The IL Obfuscator can be named universal, since it is independent on the type of information that is conveyed between the communicating entities.

The IL Obfuscator can be successfully used to protect software from reverse engineering. The algorithm based on intermediate code is completely automatic and can therefore be used as part of a software protection utility. The main advantage of this method compared to its counterparts is its platform independence. Doing obfuscation at intermediate level allows us to use the same software module at different hardware platforms, therefore IL Obfuscator can be a good choice in providing security for multiplatform systems.

Since data exchange between intermediate level component and platform-specific modules of IL Obfuscator is XML-based, it is possible to make use of a client-server technology. The code transformation engine can be deployed on a server, while platform-specific modules can run on client machines.

Future research of authors will include combination of intermediate level and machine code level obfuscation techniques (e.g. polymorph code generators), which would further raise the barriers to someone trying to decompile or steal one's code.

Acknowledgments

This work was partially supported by the TÁMOP-4.2.1.D-15/1/KONV-2015-0008 project. This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] Sallai G. "Future Internet visions and research clusters", *Acta Polytechnica Hungarica,* vol. 11, no. 7, pp. 5-24, 2014.

[2] Popa M. "Techniques of program code obfuscation for secure software". *Journal of Mobile, Embedded and Distributed Systems,* vol. 3(4), 2011.

[3] Papadopoulos P., Papadogiannakis A., Polychronakis M., Zarras A., Holz T., and Markatos E. "k-subscription: privacy-preserving microblogging browsing through obfuscation", *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[4] Brakerski Z., Rothblum G.N. "Virtual black-box obfuscation for all circuits via generic graded encoding." In *Proceedings of 11th Theory of Cryptography Conference,* TCC 2014, San Diego, CA, USA, February 24-26, pp. 1-25, 2014.

[5] Canetti R., Lin H., Tessaro S., and Vaikuntanathan V. "Obfuscation of probabilistic circuits and applications". *LNCS*, vol. 9015, pp. 468-497, 2015.

[6] Ismail N.A., O'Brien E.A. "Enabling Multimodal Interaction in Web-Based Personal Digital Photo Browsing." In *Proceedings of Int. Conf. on Computer and Communication Engineering*, 2008.

[7] Dunaev D., Lengyel L. "Formal considerations and a practical approach to intermediate-level obfuscation". *WSEAS Transactions on Information Science and Applications*, Volume 11, pp. 32-41, 2014.

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

[9] Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vadhan S., and Yang K. "On the (im)possibility of obfuscating programs." In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology, CRYPTO'01*, (London, UK), pp. 1-18, Springer-Verlag, 2001.

[10] Goldwasser S., Kalai Y.T. "On the impossibility of obfuscation with auxiliary input." In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, pp. 553-562, 2005.

[11] Dunaev D., Lengyel L. "Overview of an obfuscation algorithm." In *Proceedings of the International Conference on Computer Science and Information Technologies*, CSIT'2012, (Lvov, Ukraine), pp. 36-38, 2012.

[12] Dunaev D., Lengyel L. "A method of machine code translation to intermediate representation." In *Proceedings of the 4th IEEE International Conference on Cognitive Infococommunications*, CogInfoCom'2013, (Budapest, Hungary), pp. 785-790, 2013.

[13] Grune D., Langendoen K.G., Jacobs C.J., Bal H.E. "Modern compiler design." *Worldwide series in computer science*, Chichester, New York, Weinheim: J. Wiley and sons, 2001.

[14] Watson A.H., McCabe Th.J. "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric." *NIST Special Publication* 500-235, 1996.

[15] Halstead M.H. "Elements of Software Science", Amsterdam: Elsevier North-Holland, 1977.

[16] Al Qutaish R. E., and Abran A. "An analysis of the design and definitions of Halstead's metrics", In *Proceedings of the 15th International Workshop on Software Measurement*: IWSM'2005, (Montreal, Canada), pp. 337-352, September 2005.

[17] Elshoff J.L. "An analysis of some commercial PL/1 programs." In *IEEE Transactions on Software Engineering*, 1976.

[18] Oviedo E.I. "Control flow, data flow and program complexity." In *Proceedings of IEEE Annual International Computers, Software & Applications Conference*, Chicago, IL, pp. 146-152, Nov. 1980.

[19] McClure C.L. "A model for program complexity analysis." In *Proceedings of the 3rd International Conference on Software Engineering*, pp. 149-157, 1978.

[20] Chapin N. "A Measure of software complexity." In *Proceedings of National Computer Conference*, pp. 995-1002, 1979.

[21] Collberg C., Thomborson C., and Low D. "A taxonomy of obfuscating transformations." Technical Report 148, Department of Computer Science, University of Auckland. 1997.

[22] Dunaev D., Lengyel L. "Complexity of a special deobfuscation problem." In *19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems*, ECBS'2012, (Novi Sad, Serbia), pp. 1-4, 2012.