Formal Considerations and a Practical Approach to Intermediate-Level Obfuscation

DMITRIY DUNAEV, LÁSZLÓ LENGYEL
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
H-1117, Budapest, Magyar tudósok krt. 2.
HUNGARY
dunaev@aut.bme.hu, lengyel@aut.bme.hu

Abstract: - The essence of obfuscation is to entangle the code and eliminate the majority of logical links in it. The offered theoretical apparatus allows describing obfuscated routines by concatenation of original and fake operational logics. This approach considers not only instructions or routines themselves, but the actions (results) they produce, what makes obfuscation a process of adding excessive functionality. The mathematical apparatus presented in the paper, discusses introductory terms, definitions, operations, and formulates a proposition about NP-completeness of a special deobfuscation problem. We formulate the problem statement and prove that the significance of operational logic in the obfuscated routine is an NP-complete problem. We point out the applicability limits of this proposition, and offer a practical approach that can noticeably reduce the probability of having a deobfuscator running in polynomial time. This paper also offers recommendations for constructing obfuscating transformations and points out a practical approach to creation of intermediate-level obfuscating algorithm.

Key-Words: - Obfuscation, operational logic, global context, NP-completeness, intermediate representation.

1 Introduction

Modern society is characterized by intensive development of computer software and, as a consequence, by rapid development of software piracy. As a resistance to computer piracy, the technologies of software protection from analysis, and unauthorized modification are being used. These technologies are also used in digital intellectual rights management [1], watermarking [2], cryptography [3], and for hiding malicious code [4].

Currently more and more software is distributed over the Internet. Once distributed to a client machine, the software owner actually loses all control of the (client) application. Consequently, adequate security is required in this complex environment. First, software may contain secrets that must be protected. To solve this issue there exist a number of encryption and authentication algorithms [5], but these require that secret keys have to be protected somehow. Second, the application logic and implemented algorithms must be protected from analysis and reverse engineering. Third, during execution of critical code or when confidential data is accessed, both code and data must be protected from malicious intents, such as dynamic analysis and tampering. All aforementioned problems must be faced to guarantee data confidentiality and secure program execution.

The obfuscating techniques relate to methods and apparatus for increasing the structural and logical complexity of the software by inserting, removing or rearranging identifiable structures of information from the software in such a way as to reinforce the difficulty of the process of reverse engineering [6]. Such techniques can be used to protect both storage and usage of keys, and can be applied in re-encryption functionality [17]. Obfuscation can hide certain properties such as a software fingerprint or a watermark, or even the location of a bug in case of an obfuscated patch. However, code obfuscation itself does not protect from code lifting or software piracy. It merely strengthens built-in protection mechanisms, e.g. against tampering or piracy.

The introduction of a non-black-box simulation technique by Boaz Barak [11, 15] has been a major landmark in obfuscation. In the last years, Barak's techniques were subsequently extended, e.g. by solutions based on semi-honest oblivious transfer that do not rely on collision-resistant hashing [21], or by new applications of obfuscation for network coding techniques, such as fountain code [22].

Most of obfuscation methods are based on compiler technologies, or require the presence of a source code of the obfuscated program. Others operate at intermediate level or at machine code on the target platform [23].

The rest of this paper is organized as follows. Section 2 introduces three levels of obfuscation and justifies the choice of intermediate-level. Section 3 presents introductory terms, definitions and operations, furthermore, discusses the key concept of operational logic. In Section 4, we formulate a proposition about **NP**-completeness of a special deobfuscation problem and prove it. Section 5 discusses a set of recommendations for constructing obfuscating transformations, and we show that effective use of these recommendations can significantly complicate the process of automatic deobfuscation of routines. Finally, in Section 6 we draw the conclusions and outline the further work.

2 Levels of Obfuscation

If we consider an application, it can be represented at three levels: (a) high-level source code; (b) some intermediate representation; and (c) low-level machine code.

We define a high-level code as a programming language with high level of abstraction from the particular computer instruction set. Similarly, a low-level code is a programming language that provides no (or very little) abstraction from the particular computer's instruction set. Intermediate representation corresponds to a target-independent intermediate code. An example is a three-address code (often abbreviated as TAC or 3AC), which instruction set is sufficient for translation of assembly code into intermediate representation. It is important that intermediate code will not execute in a real processor, it is only an internal representation of a program.

Source code obfuscation means taking the application source code and obscuring it, so prying eyes cannot view its native format. Actually, source code level obfuscation is less secure than intermediate or executable level techniques. This is primarily because code obfuscators cannot take advantage of implementation details that are not permitted by language compilers. Thus, such obfuscators are restricted by the given programming language and by the given compiler. Consequently, most high-level obfuscation techniques such as logical obfuscation, data obfuscation and lexical obfuscation can be applied only at the presence of a source code.

Intermediate code is usually a description of high-level statements with some simpler instructions that accurately represent the operations of source code statements. Since intermediate code uses simpler constructs than a high-level code, it is easier to determine the data- and control flow. This fact is of high importance for obfuscation algorithms.

Another advantage with intermediate-level obfuscation is the possibility of creating a target-independent infrastructure. It means that for each platform that needs to be supported we only have to write the machine code – intermediate code and intermediate code – machine code translators, the obfuscator is already written for the intermediate code which does not change. If we need to port our obfuscator to another platform, we only need to write a new translator for the new processor.

Sometimes application source code is not available; in these cases, post-compilation obfuscation is the only possibility. A good example is third-party critical assemblies that are often shared among different software. We may want to include to our software such third-party standalone assembly that actively interacts with the main program. In this case, the intermediate-level obfuscation techniques are preferable, since:

1) Source code is not available for all components of the software.
2) Obfuscating a source code of available components only, one cannot secure a source code of included assembly, which can be proprietary and inaccessible.
3) On source code level, there is no way to obfuscate the logic of interaction between routine and main program that can easily be analyzed by a reverse engineer. Software protection models on source code level would not withstand attacks that combine static and dynamic analysis techniques [7].

After having analyzed the existing methods of protecting software [8] we have pointed out a number of drawbacks of such methods such as unacceptable execution slowdown of the protected code, failures due to the usage of undocumented hardware and/or software features, relying on source code or debug information, and relatively high probability of creating automatic tool [9] for deactivation of protection.

During the research, we concluded that to solve the problems we need to define the process of obfuscation as adding additional (redundant) entities to the program that would complicate the understanding of obfuscated code [10].

Consequently, we are to develop a mathematical apparatus and define such formal conditions by

which the deobfuscation problem is **NP**-complete. In our research, we are aimed at creating a complete method of intermediate-level obfuscation, which would operate in accordance with the worked out practical recommendations presented in this paper. The theoretical background is **NP**-completeness of deobfuscation problem, what proves the absence of deobfuscating algorithms of polynomial complexity.

3 Routines and Operational Logic

Let us examine a closed system (Fig. 1), where *O(M)* represents an obfuscated routine, and A represents a not obfuscated routine interacting with *O(M)*. Let vector *X* of length *p* denote input data and vector *Y* of length *q* denote output data respectively.

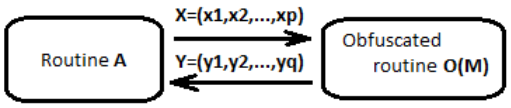


Fig.1 Example of program that contains an obfuscated routine

By the operational logic of the routine, we mean the logical descriptor, or in other words the logic, implemented by the routine. Consequently, for operational logic of routines it is true that:

$$\forall p_1, p_2 \in \Pi\ (f_{p1} = f_{p2} \Rightarrow (p_1 \in \pi \Leftrightarrow p_2 \in \pi)) \qquad (1)$$

Here $\pi \subseteq \Pi$ is an operational logic of routine; Π denotes a set of routines, execution of which is terminated at some final result; p_1, p_2 are the routines from set Π; f_{p1}, f_{p2} are functions computed by routines p_1, p_2 respectively.

We use the operator "*" to denote compound routines as a sequence of operational logics following each other.

Definition (Operation "*"). Operation "*" is a left-handed operation that denotes concatenation of operational logics.

The defined operation has the following properties:

1) Non-commutative. $\pi_1 * \pi_2 \neq \pi_2 * \pi_1$
2) There exists an identity element. The identity element on the set of operational logics Π is a special operational logic for which $\pi * e = e * \pi = \pi$.
3) Associative. $(\pi_1 * \pi_2) * \pi_3 = \pi_1 * (\pi_2 * \pi_3)$.

Let us assume that operational logic can be described by a function $f_i(X,V_{i-1},Y_{i-1},V_i,Y_i)$, where X is a vector of input values, Y is a vector of output values, and V contains intermediate (transitional) values. The vector indices denote the iterations and are used to separate the input and output parameters. Here and below, the $F(X,Y)$ function denotes such a routine that takes X vector as input and returns Y vector as output. The expanded $F(X,Y)$ function:

$$F(X,Y) = f_1(X,V_0,Y_0,V_1,Y_1) * f_2(X,V_1,Y_1,V_2,Y_2) * \ldots * f_n(X,V_{n-1},Y_{n-1},V_n,Y_n) \qquad (2)$$

Consequently V_{i-1}, Y_{i-1} being input vectors of function f_i are also output vectors of function f_{i-1}; similarly V_i, Y_i are output vectors of function f_{i-1} and input vectors of function f_{i+1}. We can state that there are just vectors V and Y, the elements of which change between functions.

The operation "*" in (2) denotes concatenation of operational logics, which are represented as functions. Let us specify the properties 1-3 with respect to function f_i.

1) Operation "*" is non-commutative.
2) Function $f_i(X,V_{i-1},Y_{i-1},V_i,Y_i)$ is identity function (e) if the following system holds true:
$$\begin{cases} V_i = V_{i-1} \\ Y_i = Y_{i-1} \end{cases}$$
3) Operation "*" is associative.

If for each function f_i there existed a unique inverse function f'_i such that $f_i * f'_i = f'_i * f_i = e$, then a family of f functions were a group with operation "*". However, we suppose that not all f_i-s have an inverse f'_i. Hence, we need to consider in details the invertibility in general and inverse functions in particular.

Definition (Invertible function). Function f is called invertible if there exists an f' such that $f * f' = f' * f = e$.

Following the above definition we conclude that an inverse function f' must be able to restore the input vectors of function f by having the X and the output vectors of function f. It is evident that for $f_i(X,V_{i-1},Y_{i-1},V_i,Y_i)$ function to be invertible it is necessary that the values of elements in output vectors V_i, Y_i must depend on values of corresponding elements in input vectors V_{i-1},Y_{i-1}. Still, this requirement is not sufficient. Dependence must be such that the inverse computations can be performed in polynomial time.

Having defined the apparatus, let us discuss the process of deobfuscation. Let us consider the following equation:

$$\pi_{orig} = \pi_1 * \pi_2 * \ldots * \pi_n \quad (3)$$

Here π_{orig} stands for some operational logic of routine *M* before obfuscation; this operational logic is split to π_i – single (elementary) operational logics, $i = [1...n]$.

Definition (Elementary operational logic). Elementary operational logic is an operational logic, which corresponds to a single intermediate-level instruction.

After applying (obfuscating) transformations, we get:

$$\pi_{orig} = \pi_1 * v_1 * \pi_2 * v_2 * \ldots * \pi_n * v_n \quad (4)$$

that is

$$\pi_{orig} = \pi_0 * v_0 * \pi_1 * v_1 * \pi_2 * v_2 * \ldots * \pi_n * v_n \quad (5)$$

where $v_0, v_1, v_2, \ldots, v_n$ are additional (entangling) operational logics added during obfuscation, and $\pi_0 = e$.

Note that if $v_0, v_1, v_2, \ldots, v_n$ in (5) are equal to *e*, then the equation (5) can be reduced to the system of equations (6).

$$\begin{cases} \pi_0 = v_0 = e \\ \pi_1 = \pi_1 * e \\ \pi_2 = \pi_2 * e \\ \quad \ldots \\ \pi_n = \pi_n * e \end{cases} \quad (6)$$

It is obvious that having system of equations (6) the components of (5) can be analyzed separately, one-by-one, and that not only simplifies the analysis, but also increases the probability of creating an optimization algorithm in complexity class **P**.

Nevertheless, the irreducibility of (5) to (6) can be achieved. This would require that routines with operational logic π_i and v_i should deal with different elements of the output vectors, and the routine with operational logic v_{k-1} should restore the essential elements of input vectors before the routine with operational logic v_k starts to work with these elements. However, this approach cannot be considered as highly durable on intuitive grounds. Another solution to ensure the irreducibility of (5) to (6) is the usage of homomorphic encryption algorithms, or computations on encrypted data:

$$m_1 \; op \; m_2 \Leftrightarrow E(m_1) \; op' \; E(m_2) \quad (7)$$

That is, a specific operation $m_1 \; op \; m_2$ on two initial data bijectively corresponds to a different operation $E(m_1) \; op' \; E(m_2)$ on the encrypted data.

Based upon the above manipulations we can formulate the following proposition.

Proposition

Restricting ourselves to automatic generation of routines with operational logics $v_0, \ldots, v_n$, we cannot guarantee the absence of effectively optimized algorithm, which can restore the original sequence (3).

Proof

B.Barak has shown that obfuscation in general is impossible, since there exists a class of functions for which virtual black-box property does not occur. According to [11], program obfuscation is an efficient transformation *O* of a program *P* into an equivalent program *P'* such that *P'* is far less understandable than *P* (i.e. P' protects any secrets that may be built into and used by P). A virtual black box property states that any information that can be extracted from the text of *P'* can be also extracted from the input-output behavior of *P'* [11].

Although even if obfuscated routine does not belong to a class of non-obfuscable functions, the automatically generated obfuscation algorithm is very likely to be reduced to the system of equations (6). It should be emphasized that the reducibility to (6) does not mean that their analysis would be trivial, since the operational logics $\pi_0, \pi_1, \ldots, \pi_n$ can be implemented with relatively high complexity metrics. However, static or semi-static analysis of obfuscated code can still be used to restore the original operational logic of the routine.

Subsequently, one can create automatic or semi-automatic tools to perform a full or partial optimization (deobfuscation). The Barak's virtual black box property in this case does not occur and we cannot guarantee the absence of effectively optimized algorithm that can restore the original operational logics.

Suppose now that

$$\pi = \pi_1 * v_1 * \pi_2 * v_2 * \ldots * \pi_n * v_n \neq \pi_{orig} \quad (8)$$

That is the operational logic of the obfuscated routine *O(M)* is different from operational logic of the original routine *M*. We can achieve this e.g. by introducing a global (with respect to *O(M)*) 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, from its part, may be shared across routines and does not expire right after a routine execution. Global context 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 [19].

In such a way, if operational logics $\pi_0,\ldots,\pi_n$ interact with false context, then without having analyzed the calling routine *A* (Fig.1), the separation of original and fake data becomes an intricate problem and the irreducibility of (5) to (6) can be ensured. Consequently, a reverse engineer will have to apply deobfuscation and optimization algorithms to both routines *A* and *O(M)*, and therefore it would require more resources. □

4 NP-completeness of Special Deobfuscation Problem

We define obfuscation as the process of adding additional (redundant) entities to the program and by that modifying the original routine so that it would complicate the understanding of program code. Following this definition, we can formulate a proposition about **NP**-completeness of deobfuscation problem for the current case.

4.1 Proposition

The problem of determining the significance of the operational logic $\pi_i(v_i)$ in equation (8) is **NP**-complete.

4.2 Definitions and statements

An operational logic is called significant if its presence affects the result of routine operation.

An expression is satisfiable if there is some assignment of truth-values to the variables that makes the entire expression true.

A decision problem is in **NP** if it can be solved by a non-deterministic algorithm in polynomial time [12]. An instance of the Boolean satisfiability problem is a Boolean expression that combines Boolean variables using Boolean operators.

4.3. Proof

First let us prove that a problem of determining the significance of operational logic can be reduced to the Boolean satisfiability problem (SAT). In complexity theory, the SAT is a decision problem, which instance is a Boolean expression written using only *and*, *or*, *not*, variables, and parentheses. The question is: given the expression, is there some assignment of "true" and "false" values to the variables that will make the entire expression true? A formula of propositional logic is said to be satisfiable if logical values can be assigned to its variables in a way that makes the formula "true". It has been proved by a Cook–Levin theorem the Boolean satisfiability problem is **NP**-complete [13, 14].

In order to test the significance of a single operational logic (i.e., to check its effect on the output of the program), it is necessary to exclude this logic from the sequence (8) and verify the execution results at all possible input sets.

That is, in fact, checking a Boolean formula

$$\cup_i(X_i \cdot \neg Y_i) \qquad (9)$$

Here *X* is the output data obtained before the exclusion of a verified operational logic and *Y* is the output data obtained after the exclusion. If the result of (9) is not zero, the verified operational logic is significant. Obviously, the problem of determining the significance of the operational logic is reduced to the Boolean satisfiability problem and, therefore, lies in the class **NP**. □

4.4 Limits of applicability and conclusions

It is essential to note that the proposition about **NP**-completeness of deobfuscation is valid only in the case if there is no essential difference between original and fake routines. For instance, if original instructions use floating-point types and fake (additional) instructions work only with integer numbers, then the separation of such instructions can be done automatically in a polynomial time.

The above proposition is also restricted by the fact that the calling routine *A* (Fig.1) is not available and cannot be analyzed. But what if a reverse engineer gets access to *A*? It turns out that in this case the proposed approach loses only a part of its durability. A reverse engineer would still need to prove that there is only *A* routine that calls *O(M)*, what it is not always possible. However, in this case

the deobfuscation can still be carried out, but it would require much more effort, in contrast to the case where the Barak's functionality condition [7] is followed.

5 Practical Approach: Construction of Obfuscating Transformations

We are aware of the fact that the universal obfuscator does not exist [11, 15]. Boaz Barak has proven that there exists a class of programs for which the virtual black box property is not feasible. However, even if the obfuscated program does not belong to the Barak's class of non-obfuscable programs, then there is still a risk (non-zero probability) that obfuscating algorithm results in system (6).

It follows that for effective intermediate-level obfuscation we must add global (with respect to an obfuscated routine) fake context. In order to provide high resistance to different deobfuscation methods, transformations should be applied according to some recommendations. These recommendations and techniques are offered based upon the analysis of compiler theory and code optimization techniques.

5.1 Masking the control flow graph of the routine

The first thing to be done by any optimization algorithm is the construction of a control flow graph (CFG). The formal definition of CFG is the following:

G=(V, E, start, stop) is a control flow graph ⇔

1) *(V; E)* – directed graph
2) $start \in G.V, stop \in G.V$
3) $|in(start)|=|out(stop)|=$ *empty set*
4) $\forall v \in G.V\ start \rightarrow^* v \rightarrow^* stop$

In a CFG, each node represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow (Fig. 2).

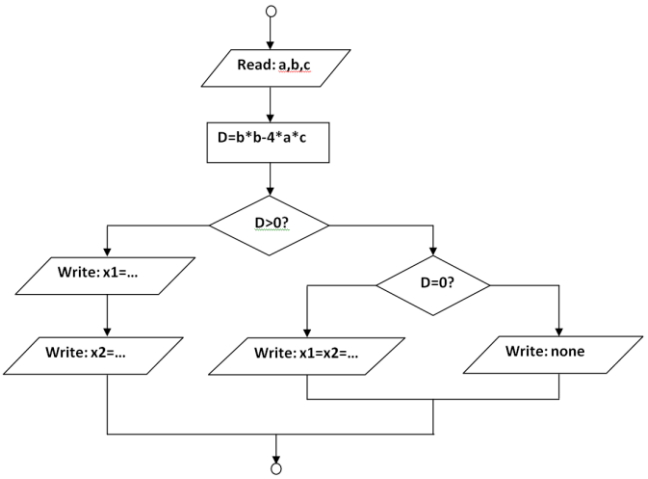


Fig.2 CFG of a program that calculates roots of a quadratic equation

Having constructed a CFG, a reverse engineer can use various data-flow analysis methods. Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in program being analyzed. The CFG can be successfully used to determine those parts of a program to which a particular value assigned to a variable might propagate.

Consequently, we are to apply CFG masking techniques. The simplest include adding unreachable, dead, and redundant code. Unreachable code is part of the source code that can never be executed because there exists no control flow path to the code from the rest of the program (Fig. 3, func1). Unreachable code is sometimes mixed up with dead code, although dead code mainly refers to code that is executed but has no effect on the output. Therefore, we define dead code as such piece of source code which is executed but whose result is never used in any other computation (Fig. 3, func2). While the result of a dead computation may never be used, the dead code may raise exceptions or affect some global (with respect to routine) state. Redundant code is source code or compiled code in a computer program that has any form of redundancy, e.g. recomputing a value that has previously been calculated and is still available (Fig. 3, func3).

```
double func1 (int a)
{
  double x=sqrt(5);
  if (x>10)
  {
    … // unreachable code
  }
  return x;
   … // unreachable code
}
```

```
int func2 (int a, int b)
{
  int c=a+b; //dead code
  return a*b;
}

int func3 (int a)
{
  int b=a*2; //redundant
             //code
  return a*2;
}
```

Fig.3 Example of unreachable (*func1*), dead (*func2*) and redundant (*func3*) code.

Other well-known techniques for CFG masking include function inlining/outlining, opaque predicates, eliminating library calls, function cloning, loop unrolling, and direct graph transformations. Branch instructions play a very important role in programming since they determine the sequence of program execution, execution of conditional statements and loops. The above-listed techniques can be used for general obfuscation, but we offer two additional methods: replacement of branch instructions with their equivalents in which the transition address is calculated dynamically, and replacement of branch instructions by exception generation mechanisms (e.g. Structured Exception Handling) [16].

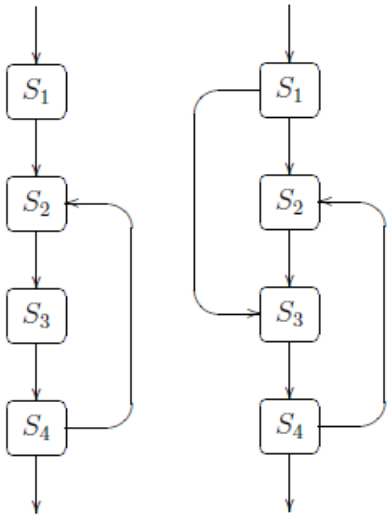
5.2 Transformation of reducible CFG to irreducible

A control flow graph *(V;E)* is reducible if and only if it can be partitioned into two sets of edges *EF* and *EB (E = EB∪EF)* such that *(V;EF)* is acyclic, and for every edge in *EB*, its head dominates its tail; that is, *EB* is a set of back edges.

Informally, we may say that a graph is reducible if a repeated application of the following two actions yields a graph with only one node:

1) replacing self loop by a single node;
2) replacing sequence of nodes such that all the incoming edges are to the first node and all the outgoing edges are to the last node.

The main property of reducible CFGs is that there are no jumps to a loop body from outside the loop. Consequently, the only possible entrance point to the loop is its header. Fig.4 gives an example of (a) reducible and (b) irreducible CFG.



(a) (b)

Fig.4 Reducible (a) and irreducible (b) CFG.

The analysis of a reducible control flow graph is much simpler than of irreducible one. Moreover, a number of optimization algorithms can be applied only with respect to a reducible graph. Node-splitting is a technique that can be used to convert any control flow graph to a reducible one. However, as has been observed for various node-splitting algorithms, there can be an exponential blowup in the size of the graph. It has been proven in [20] that exponential blowup is unavoidable. Therefore the necessity of graph transformation from reducible to irreducible can greatly complicate the optimization and deobfuscation algorithms.

5.3 Original instructions must interact with fake context, as well as fake instructions must interact with original context.

Let us discuss the problem of mixing of the original and fake (additional) contexts. This issue is important since if original instructions interact only with the original context, and fake instructions interact only with the fake context, it would be relatively easy to separate the first from the latter. Therefore, we need to ensure that fake instructions could interact with original context, and original instructions – with fake one. Let us denote a set of all memory regions used by original instructions by M_{ORIG}, and a set of all memory regions used by fake instructions by M_{FAKE}. The above-stated informal criterion can be expressed more formally: $M_{ORIG} \cap M_{FAKE} \neq \Theta$, where Θ denotes an empty set.

Based on the aforesaid, we can formulate a proposition.

5.3.1 Proposition

Any fake variable must not be disposed until it has been used at least once.

5.3.2 Proof

Let us examine the following assembly code for x86:

```
...
i) mov eax, [ebp + imm8]
...
//Plenty of code, but eax register is never used here
...
j) mov eax, ebx
...
```

Obviously, the assignment for *eax* is active between lines *i* and *j*, and is overwritten at line *j*. However, the *eax* register is not used between *i* and *j*. Thus, it would be clear for a reverse engineer that there is no need in assigning *eax* at position *i*, so that it might be a fake assignment instruction. □

Let us denote by M_{W_ORIG} and M_{W_FAKE} the sets of memory regions that original and fake instructions write to; M_{R_ORIG} and M_{R_FAKE} will stand for the sets of memory regions that original and fake instructions read from. The more rigorous formal description of the recommendation:

$M_{W\_ORIG} \cap M_{W\_FAKE} \neq \Theta$

$M_{R\_ORIG} \cap M_{R\_FAKE} \neq \Theta$

5.5. Global variable can be reassigned a new value only if its previous value is used as a parameter of an assignment instruction.

In practice, a compiler aims not to use global (with respect to routine) variables as temporary ones. Local context suits much better for that. It is evident that obfuscated code should behave the same way. In this aspect, recommendation 5.5 is an extension of proposition 5.3.1 with respect to global variables.

5.6. Dead code should not differ greatly from the actual executable code.

A family of instructions used in the actual executable code must match with the family of instructions used in the dead code. For example, if actually executed code uses only a standard subset of the general instruction set, then the usage of FPU instructions or other instructions, which are not specific for the environment, will lead to a simplification of dead code detection.

6 Conclusion

In this paper, we have discussed the approach to intermediate-level obfuscation and introduced the concept of operational logic. We have shown that restricting ourselves to automatic generation of additional fake operations, we cannot guarantee the absence of effectively optimized algorithm, which could restore the original sequence. However, the problem can be solved if we neglect the Barak's functionality principle, that is, let the operational logic of obfuscated routine *O(M)* be different from operational logic of original routine *M*. The solution lies in introduction of a global fake context.

We have proven that the problem of determining the significance of operational logic in such case is **NP**-complete. We believe that this approach can provide a considerably higher durability of obfuscated code to existing optimization algorithms.

We have discussed a set of recommendations to be followed for constructing obfuscating transformations. Presenting them, we pay attention to the fact that effective use of these recommendations can significantly complicate the process of automatic deobfuscation of routines, moreover, observing these recommendations, we can significantly reduce the probability of creating a deobfuscator running in polynomial time. We point out the fact that after having introduced the fake global context, it became more difficult to restore the original operational logic of subroutine without a detailed analysis of other routines that interact with it. Herewith, the static and semi-static analysis can also be impeded.

Based on the presented theoretical considerations and practical recommendations, we have worked out and implemented an obfuscation algorithm at intermediate code level that works with three-address code. Its architecture is based on a modularity principle allowing code obfuscation at different hardware platforms by using the same software module. To support such obfuscation, we have worked out methods of entanglement of branching instructions, methods of interaction between plugged obfuscating instructions and original program data, as well as methods for injection of additional external code to a program that would allow adding protecting code to an already existing program [18].

At present time, we are carrying out measurements upon the implemented algorithm. Our task is to show the advantages of our method and

compare the results with different existing obfuscation techniques.

Owing to intermediate level obfuscation, such situations as when one function can be called from both the obfuscated, and the non-obfuscated code, can be successfully handled [19]. We have worked out methods of translation from native code into an intermediate representation and back. However, we believe that such translation mechanisms can be significantly improved by combining intermediate-level obfuscation with machine-level techniques, which would further increase the security and optimization resistance. Usage of machine-level obfuscation mechanisms will ensure not only the integrity control of protected code, but will also provide higher resistance to deobfuscation. Another aspect to be considered is implementation of a polymorphic machine code generator that will provide better resistance to optimization algorithms based on signature search.

Acknowledgements

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

This work was partially supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR_12-1-2012-0441).

References:

[1] A. Sethi, *Digital Rights Management and Code Obfuscation*. University of Waterloo, Ontario, Canada, 2004.

[2] Y. Zeng, F. Liu, X. Luo, Ch. Yang, Robust Software Watermarking Scheme Based on Obfuscated Interpretation. In *Proceedings of International Conference on Multimedia Information Networking and Security*, Nanjing, PRC, November 2010.

[3] D. Hofheinz, J. Malone-Lee, M. Stam, Obfuscation for Cryptographic Purposes. In *Proceedings of the 4th conference on Theory of cryptography (TCC'07)*, Germany, Berlin, 2007.

[4] M. Christodorescu, S. Jha, Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of 12th USENIX Security Symposium*, August 2003, pp. 169–186.

[5] A. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[6] C. S. Collberg, C. Thomborson, Watermarking, Tamper-proofing, and Obfuscation – Tools for Software Protection. In *IEEE Transactions on Software Engineering*, vol. 28, August 2002, pp. 735–746.

[7] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere, Hybrid Static-Dynamic Attacks against Software Protection Mechanisms. In *Proceedings of the 5th ACM workshop on Digital rights management*, 2005, pp. 75-82.

[8] D. Dunaev, Obfuscation for Protecting Software from Analysis and Modification. In *Proceedings of the Automation and Applied Computer Science Workshop*, AACS'2011, Budapest, Hungary, June 2011, pp. 290-296.

[9] D. V. Sklyarov *The Art of Breaking and Protecting Information*. St. Petersburg, BHV-Petersburg Press, 2004.

[10] D. Dunaev, L. Lengyel, Actual Problems of Protecting Programs from Reverse Engineering and Modification. In *Proceedings of the 6th International Scientific and Technical Conference on Computer Science and Information Technologies*, CSIT'2011. Lvov, Ukraine, November 2011, pp.124-126.

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

[12] M. Sipser *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, MA, 2006. Sections 7.3–7.5, pp.264–293.

[13] S.Cook The Complexity of Theorem Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[14] L.Levin *Universal search problems* (Russian: *Универсальные задачи перебора*). Problems of Information Transmission, 1973, 9 (3): 265–266. Russian, translated into English by B.A.Trakhtenbrot A survey of Russian approaches to perebor (brute-force searches) algorithms. Annals of the History of Computing, 1984, 6 (4): 384–400.

[15] B. Barak *Non-black-box Techniques in Cryptography*. PhD thesis, Department of

Computer Science and Applied Mathematics, Weizmann Institute of Science, January 2004.

[16] Microsoft Corporation. *Structured Exception Handling*. MSDN Library. Retrieved: October 2013.

[17] S. Hohenberger, G. N. Rothblum, A. Shelat, V. Vaikuntanathan, Securely Obfuscating Re-encryption. *Theory of Cryptography, Lecture Notes in Computer Science*, Volume 4392, 2007, pp 233-252.

[18] D. Dunaev, L. Lengyel, Extending an Application with Security Code Using Intermediate Level Obfuscation Technique. *International Journal of Application or Innovation in Engineering & Management*, Volume 2, Issue 6, 2013, pp. 433-438.

[19] D. Dunaev, L. Lengyel, Aspects of Intermediate Level Obfuscation. In *Proceedings of IEEE 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems: ECBS-EERC'2013*, Budapest, Hungary, August 2013, pp. 138-143.

[20] L. Carter, J. Ferrante, C. Thomborson, Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Louisiana, USA, 2003, pp. 106-114.

[21] A.Hessler, T.Kakumaru, H.Perrey, D.Westhoff, Data obfuscation with network coding. *Computer Communications*, 2012, vol. 35(1), pp. 48-61.

[22] N. Bitansky, O. Paneth, From the impossibility of obfuscation to a new non-black-box simulation technique. In *Proceedings of IEEE 53rd Annual Symposium on Foundations of Computer Science (FOCS)*, 2012, pp. 223-232.

[23] H. Fang, Y. Wu, S. Wang, Y. Huang, Multi-stage binary code obfuscation using improved virtual machine. In ISC (X. Lai, J. Zhou, and H. Li, eds.), *Lecture Notes in Computer Science*, vol. 7001, 2011, pp. 168-181.