An Intermediate Level Obfuscation Method

Dmitriy Dunaev, László Lengyel

Budapest University of Technology and Economics
Department of Automation and Applied Informatics
Magyar tudósok krt. 2, H-1117 Budapest, 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; that is, to transform the code so that it becomes complex enough for analysis and unauthorized modification. The developed theoretical apparatus allows us to describe an entangled program using concatenation of operational logics of the routines. Consequently, this approach considers not only the instructions or routines themselves, but the actions, or results, they produce. This allows us to consider obfuscation as the process of adding excessive functionality to the program. This paper is unique in presenting an obfuscation method at intermediate code level that is based on the theory of optimizing transformations. The focus is set on generation of fake intermediate level code, suppression of constants, and meshing of control flow transitions.

Keywords: obfuscation; software protection; entangling transformations; fake context; intermediate code

1 Introduction

The general purpose of obfuscating techniques is to prevent, or at least hamper, interpretation, decoding, analysis, or reverse engineering of software. We may further state that more particularly, although not exclusively, the obfuscating techniques relate to methods and apparatus for increasing the structural and logical complexity of software. All that is done by inserting, removing, or rearranging identifiable structures of information from the software in such a way as to exacerbate the difficulty of the process of decompilation or reverse engineering [1].

The introduction of a non-black-box simulation technique by Boaz Barak [2] has been a major landmark in obfuscation. It has been proven that universal obfuscator does not exist [2, 8], since there exists a class of programs for which the virtual black box property is not feasible. According to [2], 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’*.

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 [3], or by new applications of obfuscation for network coding techniques, such as fountain code that is a rateless erasure code [4].

There are many different practical approaches to obfuscation, which are described and summarized in [5]. Most of them are based on compiler technologies, and some methods require the presence of a source code of the obfuscated program [6]. Others operate at intermediate level or at machine code on the target platform [7]. Usually, one of three directions is followed: source code obfuscation, which is achieved through source code transformations; intermediate level obfuscation through transformations on some precompiled code; or machine level obfuscation through binary rewriting.

Intermediate level obfuscation deals with 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. It is important that this code will not be executed in a real processor, it is only an internal representation of a program. Since intermediate code uses simpler constructs than the high-level language, it is much easier to determine the data and control flow. In addition, this is very important for obfuscation algorithms.

An advantage of intermediate level obfuscation is that we can create a target-independent infrastructure. It 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 intermediate level obfuscator 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 the paper is organized as follows. In Section 2, we discuss the related work by pointing out negative and positive results in the state-of-the-art, and justifying the concepts of our research. In Section 3, we present the intermediate level obfuscation method. The focus is set on dynamic calculation of constants, generation of fake instructions, meshing of control flow transitions, and basic blocks partitioning. Finally, we draw conclusions and outline further work.

2 Background

Negative results

The essence of obfuscation is to entangle the code and eliminate the majority of logical links in it; that is, 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 [2,8]. A later paper of Goldwasser and Kalai [9] shows the impossibility and improbability of obfuscating more natural functionalities.

Positive results

The classes of functions for which obfuscation was ruled out in [2] and [9] are somewhat complex. Quite another issue is the fact that obfuscation can be performed for simpler circuits [10]. We see that in spite of negative results for general-purpose obfuscation, there are positive results for simple functionalities, such as point functions. Canetti [11] shows that under a very strong Diffie-Hellman assumption, point functions can be obfuscated. Further works of Wee [12], Dodis and Smith [13] relax the assumptions required for obfuscation and consider other related functionalities.

Our work

In our approach, we do not restrict ourselves to point functions and do not assume simpler circuits. 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 [14] that by restricting ourselves to automatic generation of additional fake operations, we cannot guarantee the absence of effectively optimized algorithm, which could restore the original sequence and deobfuscate the routine. However, the problem can be solved if we neglect Barak’s functionality principle; that is, let the functionality of obfuscated routine *O(M)* be different from the functionality of the original routine *M*. The solution lies in introduction of 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 problem of mixing contexts has been discussed in [17].

We have proven that the problem of determining the significance of operational logic in such obfuscated routine is NP-complete [14]. We have also worked out a general approach to intermediate level obfuscation method and presented a bird-eye view of an obfuscation algorithm, pointing out general problems and proposing solutions [15].

3 Contribution

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 [16]. However, there remain a number of problems, especially with input data analysis.

The main problem is a proper selection (and isolation) of different kinds of data sets. It is obvious that e.g. constant values, abstract memory regions, and dead variables must be detected and separately processed by the obfuscating algorithm.

For successful input data analysis at intermediate level, we need the following information about the routine to be obfuscated:

1) three-address code representation of the routine;

2) information about abstract memory cells accessed by instructions of three-address instruction code;

3) information about arguments passed to the routine (estimated values, number of parameters, etc.)

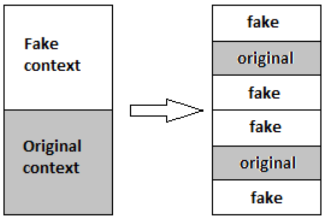

Figure 1
Proper mixing of contexts

We believe that the restriction of input data to just two types – (1) that can be moved to a fake context, and (2) that cannot be moved there – is very limited. Our task is to ensure the non-optimizability of the obfuscated routine, and for that a more thorough input data analysis is required. Otherwise, we will not be able to ensure the proper context mixing (Fig. 1) and optimization resistance. Below we introduce the input data analysis process that includes the following steps:

1) Separation of a set of “dead” local variables (set *D*). That means isolation of such variables that after a certain point are no longer used in the routine. For each point of the routine, we can associate a set *D* that reasonably can change its composition from point to point.

2) Separation of a set of simple type variables (set *C*). By the simple types, we define the following (C programming language) base data types: *char*, *short*, *long*, *longlong*. Some of the variables from the set *C* will be moved to additional fake context for better mixing original and fake contexts (Fig. 1).

3) Separation of a set of complex type variables (*CO*). Some of these variables will also be moved to additional fake context.

4) Separation of abstract memory regions, which constitute indivisible memory elements that have already been used in the subroutine as a unit and will not be used as a unit any more (set *CA*).

   Let us consider the following example code:

   ```
   %eax := &struct_1
   param eax
   call some_func
   //
   // hereinafter only tags of struct_1 are being used, but
   not the structure as a unit
   //
   ```

   In this example, *some_func* function gets a pointer to *struct_1* structure as a parameter. *some_func* works with the tags of *struct_1* (e.g. modifies the values). Later on, the routine performs some actions depending on the values of structure tags. It means that after having called *some_func* the structure is no longer used as a unit, but its tags are used separately, so in fact they correspond to ordinary variables. Such tags (elements) constitute the *CA* set. Consequently, the *CA* set should be determined and assigned for each point of the routine. Like for *CO*, elements of *CA* can also be moved to a fake local context.

5) Separation of abstract memory regions that constitute indivisible memory elements, but that are used separately within one or more basic blocks (set *CB*). These variables can be moved to the fake local context only within the boundaries of the basic block in which they are used. Subsequently, the values of these variables are to be restored.

6) Separation of abstract memory regions with known values (the set *V*). The set *V* can be left empty if there are no such variables in the routine.

The main point of the input data analysis is to ensure separation of different types of data for obfuscating method. The phases of obfuscation are the following:

1) Dynamic calculation of constants and some certainly known values.
2) Selection of a number of fake instructions per original.
3) Generation of fake instructions.
4) Generation of dead code.
5) Meshing of control flow transitions.
6) Partitioning of basic blocks.

Let us consider the above steps in details.

3.1 Dynamic Calculation of Constants and Known Values

A reverse engineer may use signature search to make certain conclusions about the functional logic of the routine being analyzed. This step is necessary as an obstacle to a signature search.

If a routine contains constants that are specific for implementation of some cryptographic algorithm [18], then its analysis becomes much easier, at least by the usage of a signature search. However, if there are memory regions, which content is exactly known in some fixed points of the routine code, one can increase the complexity metrics of a routine by introducing dynamic calculation of such values (to escape from constants). Thus, reverse engineering becomes more complicated.

It is important to mention that in order to exclude the possibility of routine optimization and to make it optimizer-resistant, constants should not be fixed in code, but instead they are to be calculated during runtime. To provide parameters for runtime calculation, we use global (with respect to the routine) initializing values, which are a special type of fake arguments intended to conceal the constants. The main goal here is to complicate the signature search in the entangled code. For even greater complexity, not only the routine constants are to be concealed, but a number of other well-known constants, e.g. the magic initialization constants of MD5 and SHA-1 [18]: *0x98badcfe, 0x10325476*, etc.

Calculation algorithms can vary from a trivial addition to a sophisticated cryptographic algorithm. The choice of a particular algorithm depends on the desired execution speed (i.e. acceptable execution slowdown) of the obfuscated routine. Furthermore, it should be noted that using very complex and slow algorithms is not a good choice, since the growth of constants calculation complexity is not directly proportional to the growth of deobfuscation resistance (i.e. resistance to reverse-engineering).

3.2 Selection of Fake Instructions per Original

The fake per original (FPO) number is a global user-defined parameter. The obfuscation algorithm generates a FPO number for each original instruction; this number indicates the desired approximate number of fake instructions per single original. The greater the numerical value of this parameter, the more difficult it is to deobfuscate the routine and to determine its original operational logic.

To provide higher deobfuscation resistance, any fake variable must not be disposed until it has been used at least once. Therefore, original instructions must interact with fake context, as well as fake instructions must interact with original context. 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. So we get the formal description:

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

$$M_{R\_ORIG} \cap M_{R\_FAKE} \neq \Theta \tag{1}$$

However, with increase of FPO, execution slowdown is increased as well. It is important to mention that a number of fake instructions is not necessarily exactly equal to FPO, so that FPO serves as some approximate value. Sometimes, the number of fake instructions can exceed FPO by several instructions in order to comply with the aforesaid.

3.3 Generation of Fake Instructions

When a fake instruction is generated, the obfuscating algorithm takes one of the following actions:

1) Write the instruction to any free abstract cell from the set *D*.

   The term “free” in this context means that the cell is not filled with data. It should be noted that a fake abstract memory cell can have 4 states: *FILLED*, *FREE*, *USED*, and *NOT_INITIALIZED*. If a cell is in a *FILLED* state, then we can write a new value to this cell only if its old (original) value is used for the new value calculation. If a cell is in *FREE* state, then we can write a new value to it without any restrictions. Ia a cell is in *USED* state, then it contains some value, which is necessary to perform some further operation. For example, such cell can contain a value used for dynamic calculation of some constants. The value written to a *FREE* cell can be read from any other abstract cell. If we read from a fake abstract memory cell, which is in *FILLED* state, then after having read the value we procced to changing the state of such cell to *FREE*. The cell with *NOT_INITIALIZED* state cannot be read. However, we can write some new value to such cell only if a new value is not computed as a result of some operation with the previous (unknown) value of this cell.

2) Write the instruction to any abstract cell from sets *C'*, *CA'* or *CB'*.

   The *C'*, *CA'* and *CB'* sets are such memory cells, the content of which has been moved to a fake local context. Consequently, *C'*, *CA'* and *CB'* are subsets of *C*, *CA* and *CB*, respectively. Writing the generated instruction to these abstract cells should be done as described in Point 1.

3) Write the instruction by performing arithmetical operations. The sets are chosen as described in Point 2, but arithmetic operations are performed with original values of the cells.

4) Write the instruction to the fake global context. The sets and operations are chosen as described in Points 1-3, but the instruction is interacting with the fake global context.

5) Generate control flow transitions (jumps) to dead code.

   Such jumps should look "plausible" and should not differ from the original control flow transitions (CFTs). This means that conditions, at which the transitions take place, must seem feasible. It definitely requires the usage of the fake global context.

3.4 Generation of Dead Code

Dead code is a piece of code that is never executed. The task of an obfuscating algorithm is to ensure that neither a reverse-engineer (analyst) nor an automatic deobfuscation tool can prove that the specific piece of code is never executed (dead). It is obvious that such code should not differ a lot from the original instruction set or data. Its main purpose is to increase the complexity metrics of obfuscated code. Moreover, by injecting dead code we can counteract the alias analysis, analysis of values of the abstract memory cells, and consequently the determination of individual variables, structures, arrays, etc.

In general, alias analysis determines whether separate memory references point to the same area of memory. This allows the compiler to determine what variables in the program will be affected by the given statement. Thus, we can significantly reduce the effect of optimizing transformations with respect to obfuscated routine through compliance with the conservativity principle of preliminary code analyzing algorithms. For example, we can counteract the alias analysis by using the following techniques:

1) Initialization of abstract cells (including those used in original code) with pointers to other abstract cells.

2) Creation of loop structures in a dead code, in which previously initialized (see point 1) abstract cells are used.

In other aspects, the algorithms used in the dead code should be similar to those used in the original executable code.

3.5 Meshing of Control Flow Transition Blocks

A branch is a piece of code in a computer program which is conditionally executed depending on how the flow of control is altered at the branching point. Explicit branches in high-level programming languages (e.g. C/C++) usually take the form of various conditional statements that encapsulate the branches of code that should be executed (or not) upon some condition; low-level instructions that define corresponding branches of code are called jump instructions. A three-address code has support for both conditional and unconditional jumps, which are essentially *goto* statements.

In general, jump instructions have unconditional and conditional forms where the latter may be fulfilled or not, depending on some conditions. The truthiness of this condition is typically evaluated and temporarily stored by some previous instruction, but not necessarily the one immediately before. Usually, this temporary information is stored in a flag register.

3.5.1 Unconditional Jumps

Fig. 2 shows an example of a meshed CFT block that represents an unconditional jump. *Blocks 1, 2, 3, 4, 5,* and *6* are basic blocks. Considering the memory map, *Block 3* directly follows *Block 4*, while other blocks are contained separately. On this figure, full straight lines denote unconditional jumps in direction of the arrow. Dashed lines denote true jumps, i.e. the CFTs that actually take place in the original routine. Dotted lines denote transitions that do not take place in the routine.

On Fig. 2, we see three different types of jumps denoted by straight, dashed, and dotted lines, respectively. However, since CFT conditions are dynamically calculated using a global context, for a reverse engineer the transitions are equally likely, and therefore the reverse engineer cannot distinguish them. Moreover, *Blocks 2, 5* and *6* may contain either dead code, or a piece of code that is actually executed. It should be noted in particular that *Blocks 3* and *4* form a single memory region, so *Block 3* can be totally fake. In this case, it does not matter whether the control flow is transferred to *Block 3* or to *Block 4*.

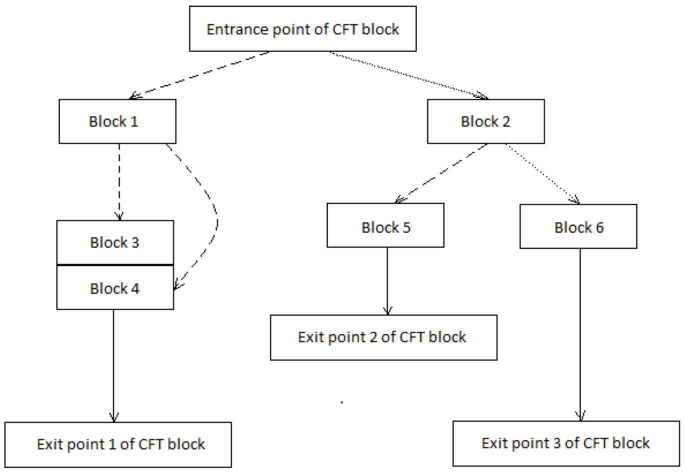
Entrance point of CFT block
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Exit point 2 of CFT block
Exit point 1 of CFT block
Exit point 3 of CFT block

Figure 2
Meshed unconditional jump

3.5.2 Conditional Jumps

As we know, there are six logical conditions (comparisons): greater, less, equal, not equal, greater or equal, less or equal. We can make comparisons between variables, while the value of one of them is fixed. For example, *if (a > 3) goto L*. We will call this kind of comparison a *constant comparison*.

Comparisons can also be made between variables, while the values of both of them are unknown. For example, *if (a > b) goto L*. We will call this kind of comparison a *variable comparison*.

In computer programs, both types of logical comparisons are widely used, and if reverse engineered, can contain sensitive information for better understanding of a logical and functional structure of a program. Consequently, logical comparisons must be taken into account during obfuscation process.

A three-address code instruction *if (a > 3) goto L* can be represented as shown on the listing below:

```
if (a<0) goto L1;
; Garbage code
if (a<2) goto L2;
; Garbage code
if (a>6) goto L3;
; Garbage code
if (a<4) goto L4;
; Garbage code
if (a>=4) goto L5
```

Labels *L1*, *L2*, and *L4* denote the code that will be executed if the original condition $a > 3$ is not satisfied. Labels *L3*, and *L5* denote the code that will be executed if the condition is satisfied.

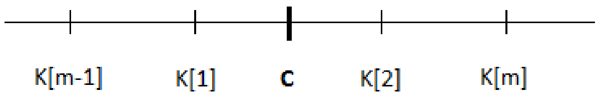

Figure 3
Number scale with generated constants

The proposed method of obfuscating constant comparisons is described below. At first, integers are selected on a number scale in such a way that among these integers there must be numbers greater and less than the original constant *C*. The distance between any adjacent selected integers is equal to two. Fig. 3 represents a number scale with original constant *C* and generated constants *K[i]*, where *i=[1..m]*.

<u>Propositions:</u>

$$a > C \equiv a \geq C+1 \tag{2}$$

$$a < C \equiv a \leq C+1 \tag{3}$$

$$a \geq C \equiv a > C-1 \tag{4}$$

$$a \leq C \equiv a < C+1 \tag{5}$$

$$a > C+i \Rightarrow a > C \tag{6}$$

$$a < C-i \Rightarrow a < C \tag{7}$$

$$a = C+i \Rightarrow a \geq C+i \tag{8}$$

$$a = C-i \Rightarrow a \leq C-i \tag{9}$$

$$a = C \equiv (a > C-1)\,\&\,(a < C+1) \tag{10}$$

$$a \neq C \equiv (a \geq C+1)\,\&\,(a \leq C-1) \tag{11}$$

For better understanding of meshing method, let us consider the logical comparison $a > C$ as an example. Here follows an algorithm for meshing this logical comparison:

1) Choose *m* and generate *K[i]*, where *i=[1..m]*.
2) Select *i*.
3) Select one operation out of five: *a>K[i], a<K[i], a=K[i], a≥K[i], a≤K[i]*.
4) For selected operation do the following:
    a) If *a>K[i]* or *a≥K[i]* was selected:

For *K[i]>C* we should generate a CFT to code that should have been executed when original condition *a>C* is true, and then label *i* as “used”, so that it will not be used at the next iteration. Otherwise we can have an ambiguous situation when *a* can be at the same time greater than, less than, or equal to *C*. In case of ambiguous situation, we should mark *i* as usable with all operations except for *a>K[i]* and *a≥K[i]*. In this case garbage code is to be generated, and further condition testing is requited. For the nearest right-hand adjacent constant *K[2]*, the following condition is to be generated: *a≥K[2]*.

b) if *a<K[i]* or *a≤K[i]* was selected:

If *K[i]<C*, then we should generate a CFT to the piece of code that should have been executed when original condition *a>C* is false, and then label *i* as “used”. If *K[i]>C*, then we have an ambiguous situation.

5) Iterate steps 2-4 in a loop until all *i*-s are labeled as “used”.

For *a≥C* the steps are similar to those described above with the only difference: if conditions *a>K[i]* (*a≥K[i]*) for *K[i]>C*, and *a<K[i]* (*a≤K[i]*) for *K[i]<C* are not satisfied, then consequently *a=C* and hence the condition *a≥C* is satisfied.

Similarly we can write this algorithm for *a<C* and *a≤C*.

Proposition.

A set of conditions, generated by the above meshing algorithm, coincides with the original condition.

Proof.

Let us prove this proposition with respect to condition *a>C*. The proof for other conditions is similar.

For all *K[i]>C*, the algorithm generates *a>K[i]* (or *a≥K[i]*). If the conditions are satisfied, the control flow is transferred to in the same block as if the original condition *a>C* were satisfied. Since difference between two neighboring *K[i]*-s is equal to *2*, we get: |*K[1]-C*|=|*K[2]-C*|=*1*. Thus, it follows:

$$a > C \equiv \forall K[i] > C : (a \geq K[2] \,\&\, (a > K[i], i = [1..m]) \qquad (12)$$

For all *K[i]<C*, the algorithm generates *a<K[i]* (or *a≤K[i]*). If the conditions are satisfied, the control flow is transferred to in the same block as if the original condition *a>C* were not satisfied. Since |*K[1]-C*|=|*K[2]-C*|=*1*, we get:

$$\neg(a > C) \equiv \forall K[i] < C : (a \leq C \,\&\, (a < K[i], i = [1..m]) \qquad (13)$$

■

The proposed algorithm can be supplemented by a code duplication technology. In fact, if one creates multiple polymorphic duplicates of code that is executed if the condition *a>C* is true, and the control flow is transferred to different duplicates in generated branches, then obfuscated code significantly better resists automatic deobfuscation tools.

For hiding constants, as well as for increasing the complexity metrics of a routine, we can use a method based on the following identity:

$$a > C \equiv a * k > C * k \tag{14}$$

The * operation here is not a multiplication, but denotes any operation that satisfies (14). For example,

$$a > C \equiv a + k > C + k \tag{15}$$

It must be specially noted that this method should be used with great caution, because it can possibly lead to an overflow error, and consequently the identity (14) will not hold true anymore. In this case we can use another operation, such as the following:

$$a > C \equiv a - k > C - k \tag{16}$$

Herewith, the overflow error which arose in (15), will not arise in (16).

This method is suitable not only for constant comparison, but for variable comparison as well.

3.6 Partitioning of Basic Blocks

Basic blocks obtained at routine transformations are ranked by the following algorithm:

1) Determine the maximum number of functions that a basic block (BB) can be partitioned to: *MAX_NL-1*. Here *MAX_NL* stands for a maximum nesting level; this serves as an external parameter of the obfuscation algorithm. The greater the *MAX_NL* is, the more functions will be obtained from the BB.

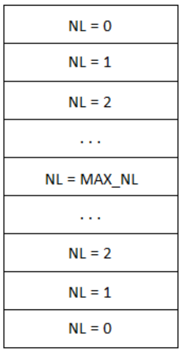

Figure 4
Assigning nesting levels to groups of instructions in a basic block

2) Nesting levels are assigned to groups of three-address code instructions within the BB as shown on Fig. 4.
3) Zero-nested instructions *NL=0* remain at their original positions. These instructions are supplemented with additional instructions that assign arguments (parameters) for the subsequent nested functions.
4) Instructions of the first nesting level *NL=1* are transferred to a separate function. The operands of these instructions are replaced with the appropriate function arguments (parameters). As in the previous step, the function instructions are supplemented with instructions that assign arguments (parameters) for the subsequent nested functions.
5) Step 4 is repeated for all nesting levels sequentially until *NL=MAX_NL*.

Summary and Conclusions

In comparison with other known algorithms of obfuscation (Table 1), the intermediale-level obfuscator presented in this paper looks very promising. Its low complexity results from easiness of analysis of three-address code and simplicity of implementation of entangling transformations on the intermediate level. Other areas of comparison include the following:

1) Portability is an indicator of transferability of an implemented algorithm from one machine to another.
2) Flexibility is an indicator of the possibility of usage of the algorithm in different development environment or programming language.
3) Scalability is an indicator of degree of controllability of obfuscation process by the user.

Table 1
Comparison of different obfuscation techniques

	Collberg	*Wang*	*TAC IL Obfuscator*
Portability	no	yes	yes
Flexibility	medium	medium	high
Scalability	high	low	high
Complexity	high	medium	low

Collberg's algorithm [19] cannot be named as portable, because it was designed directly for use with the Java Virtual Machine. Intermediate level obfuscation algorithm is most flexible, since it is most isolated from high level programming constructs. Wang's algorithm [20] shows low scalability, because it uses very specific constructs. However, Wang's algorithm has a very high resilience, but specific opaque constructs are not protected at all, which is a significant drawback. The problem we have faced with Collberg's algorithm was that a high number of parameters controlling the algorithm makes empirical testing almost impossible. The intermediate level obfuscator, however, has just several parameters, and consequently we can control its output.

It should be specially noted that the presented method preserves its optimization-resistance only if there is no essential difference between routines, implementing original and fake operational logic. For example, if the instructions of the original routines used floating-point type operands, and fake instructions only work with integer numbers, then in such case the separation of original and fake instructions can be done automatically in polynomial time.

The great advantage of the presented intermediate level obfuscation method 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. For that, we can use parameters of nested functions to pass pointers to buffers containing a fake local context. The resulting nested routines can be "scattered" in different parts of the application in an arbitrary manner.

Neither Collberg nor Wang provide a method for multiple obfuscation; that is, after having been obfuscated once, the program cannot be obfuscated again. The presented algorithm allows to obfuscate already obfuscated programs, or to obfuscate the selected routines of a program. By that, we obtain a multistage obfuscation technique.

In general, the presented method can be used to protect software from unauthorized analysis and modification, and consequently to prevent its reverse engineering. The algorithm based on this method is completely automatic and can therefore be used as a 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.

This paper has set the ground for a new understanding of obfuscation. This research, furthermore, besides advancing academic research has major practical implications in software development, in counteracting software piracy, and in information protection. Intermediate level obfuscation raises the barriers to someone decompiling and stealing your code, and by that discourages casual attacks and makes one's intellectual property less likely to be stolen.

Future research of the authors will include, but will not be limited to, working out methods of translation from machine code into an intermediate representation and back. Such translation mechanisms must be implemented using machine-level obfuscation techniques, which would further increase the security of the program.

Acknowledgement

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] Popa, M. (2011) Techniques of Program Code Obfuscation for Secure Software. *Journal of Mobile, Embedded and Distributed Systems*, Vol. 3(4)

[2] Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vadhan S., and Yang K. (2001) 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

[3] Hessler A., Kakumaru T., Perrey H., Westhoff D. (2012) Data Obfuscation with Network Coding. *Computer Communications*, Vol. 35(1), pp. 48-61

[4] Bitansky N., Paneth O. (2012) 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), pp. 223-232

[5] Balakrishnan A., Schulze C. (2005) Code Obfuscation Literature Survey. CS701 Construction of Compilers, Computer Sciences Dept., University of Wisconsin

[6] Ceccato M., Di Penta M., Nagra J., Falcarin P., Ricca F., Torchiano M, Tonella P. (2009) The Effectiveness of Source Code Obfuscation: An Experimental Assessment. In Proceedings of the 17th International Conference on Program Comprehension (Vancouver, Canada), pp. 178-187

[7] Fang H., Wu Y., Wang S., Huang Y. (2011) 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, pp. 168-181, Springer Verlag

[8] Barak B. (2004) Non-Black-Box Techniques in Cryptography. PhD thesis, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science

[9] Goldwasser S., Kalai Y. T. (2005) 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

[10] Lynn B., Prabhakaran M., Sahai A. (2004) Positive Results and Techniques for Obfuscation. *Advances in Cryptology*. *Lecture Notes in Computer Science*, Vol. 3027, Springer Verlag, pp. 20-39

[11] Canetti R., Goldwasser S. (1997) Towards Realizing Random Oracles: Hash Functions that Hide All Partial Information. *Advances in Cryptology.*

Lecture Notes in Computer Science, Vol. 1294, Springer Verlag, pp. 455-469

[12] Wee H. (2005) On Obfuscating Point Functions. In Proceedings of the 37th Annual ACM Symposium on Theory of Computing, pp. 523-532

[13] Dodis Y., Smith A. (2005) Correcting Errors without Leaking Partial Information. In Proceedings of the 37th Annual ACM Symposium on Theory of Computing, pp. 654-663

[14] Dunaev D., Lengyel L. (2014) Formal Considerations and a Practical Approach to Intermediate-Level Obfuscation. *WSEAS Transactions on Information Science and Applications*, Volume 11, pp. 32-41

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

[16] Grune D., Langendoen K. G., Jacobs C. J., Bal H. E. (2001) Modern Compiler Design. *Worldwide Series in Computer Science*, Chichester, New York, Weinheim: J. Wiley and sons

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

[18] National Institute of Standards and Technology (2002) “Secure Hash Standard”, Fips 180-2, Federal Information Processing Standard, publication 180-2, tech. rep., Department of Commerce

[19] Collberg C., Thomborson C., Low D. (1997) A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland

[20] Wang C., Hill J., Knight J., Davidson J. (2000) Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12, 2000