Extending an application with security code using intermediate level obfuscation technique

Dmitriy Dunaev¹, László Lengyel²

¹Department of Automation and Applied Informatics
Budapest University of Technology and Economics (BME), Hungary, Budapest
²Department of Automation and Applied Informatics,
Budapest University of Technology and Economics (BME), Hungary, Budapest

ABSTRACT

Software piracy in general is the illegal copying, distribution, or use of software. The problem of mislicensing, unauthorized reproduction, or misuse of software has to be solved to successfully counteract software piracy. Such copy-protection and misuse-protection mechanism as a dongle is often used with expensive packages and vertical market software. Dongles mostly appear as two-interface security tokens with transient data flow that does not interfere with other dongle functions and a pull communication that reads security data from the dongle. Without the dongle, the software may run only in a restricted mode, or not at all. The potential weakness of dongles lies in the implementation of communication between a dongle and a copy-controlled software, e.g. if a dongle-check code is localized and a protection algorithm is revealed, the reverse engineer can modify the application to bypass the dongle check. The presented paper offers a solution to the problem of localizing a dongle-check code by offering a technique of adding a security code to an already existing application. The focus is set on intermediate level obfuscation that is used to protect software from reverse engineering and analysis.

Keywords: obfuscation, software protection, dongle, software piracy.

1. INTRODUCTION

In the general approach, code obfuscation is a set of program transformations that make program code and/or program execution difficult to analyze [1]. Obfuscation hinders manual inspection of program internals and as a result protects against reverse engineering. It protects both storage and usage of keys, and it 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 [2].

The process of obfuscation can be defined and therefore approached in different ways. We consider obfuscation as a one-way process of original code transformation that results in adding some excessive functionality with the purpose of protecting software from unauthorized analysis and reverse engineering. This process is one-way, which means that there is no effective way to subsequently return to the original state [3].

Definition.

Let TR be a transformation process $PR_1 \Rightarrow TR \Rightarrow PR_2$, by which a $PR_2$ program is obtained from a $PR_1$. We say that the process $TR$ is obfuscating process if the following requirements are met:

- Program $PR_2$, being obtained from $PR_1$, is significantly different from $PR_1$.
- The program analysis, study of operation principles and reverse engineering of $PR_2$ is significantly more difficult and time consuming than in case of $PR_1$.
- At any transformation of $PR_1$, the resulting $PR_2$ instance will be different.
- There is no effective way to transform $PR_2$ back to the original $PR_1$.

Since the resulting code obtained after entangling transformations is always different, the obfuscating techniques can be used for prompt identification of copyright infringers, i.e. the buyers of legal software that are engaged in illegal distribution of purchased software copies. To utilize this idea, it is enough to calculate the checksum of every obfuscated program copy, and register it together with customer data in the relevant storage (database). Hereinafter, in case of illegal software distribution, it is enough to calculate the checksum of one illegal copy and compare it with information in a storage in order to identify the copyright infringer.

In other cases, when one needs to withstand not only illegal software distribution, but also illegal software usage, the aforementioned technique is not enough, and the software protection dongles are used. A dongle is a piece of hardware that being attached to a computer allows secured software to run. The dongle does not contain the application in its entirety, but rather is an electronic key that “unlocks” the program. It can be successfully used as an anti-piracy measure, since making a copy of the hardware is much more difficult than copying the program itself.

The rest of this paper is organized as follows: in Section 2, we discuss the methodology and describe an intermediate
level obfuscation method. We point out its merits compared to a source code obfuscation and justify our selection. In Section 3, we present the worked out results by introducing a method of adding an external security code to an existing application. We show that intermediate level obfuscation technique can successfully solve the problem of adding security code to an existing routine, so that the secure routine can resist to code disclosure attempts. The focus is set on mixing the instructions of the original routine and the security code so that the latter cannot be localized within the routine. Finally, we draw the conclusions and outline the further work.

2. METHODOLOGY

If we consider a software application, it can be represented at three levels (Figure 1):
- source code;
- some intermediate representation;
- machine code.

Source code obfuscation means taking the application source code and obscuring it, so prying eyes cannot view it in 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. In addition, software protection models on source code level would not withstand attacks that combine static and dynamic analysis techniques [4].

![Figure 1. Levels of obfuscation](image)

Intermediate level obfuscation deals with a target platform independent intermediate code. Such code is usually a description of 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 execute 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.

An advantage with the 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 – 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.

However, 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 (routines) that are often shared among different software. We may want to include such third-party standalone assembly to the developed software, and the included assembly actively interacts with the main program. In this case, the intermediate level obfuscation techniques are preferable, since:
- source code may turn out to be not available for all components of the software;
- obfuscating source code of available components only, one cannot secure the code of included assembly, which can be in fact not provided (e.g. proprietary source code).

From theoretical considerations provided in [5] it follows that for effective intermediate-level obfuscation we must add global (with respect to an obfuscated routine) fake context. In addition, for providing high resistance to different deobfuscation methods, transformations should be applied according to recommendations discussed in [6].

It has been proved by Boaz Barack in his works [7-8] that universal obfuscator does not exist, since there exists a class of programs for which the virtual black box property is not feasible. Even if the obfuscated program does not belong to the Barack's class of programs, which cannot be obfuscated, then an entangling algorithm can reduce it to the system of equations described in [5] with some non-zero probability. However, such reducibility does not mean that the reverse engineering would be trivial, because our worked out intermediate level obfuscation algorithm is implemented with still high level of complexity.
3. RESULTS

Here we are to introduce a method of adding an external (security) code to an existing application. A good example of such code is a code responsible for communication between a dongle and a copy-controlled software. Such copy-protection mechanism as a dongle is often used with expensive packages and vertical market software, such as CAD/CAM software, MICROS Systems hospitality and special retail software, Digital Audio Workstation applications, and some translation memory packages.

The simplest implementation of such code might define a function to check for the dongle’s presence, and return “true” or “false” accordingly. The problem here is that the dongle requirement can be easily circumvented by modifying the software to always answer “true”. This can be done only if a reverse engineer detects the piece of code responsible for communication with a dongle within a program code. We are to show that intermediate level obfuscation technique can successfully solve the problem of adding security code to an existing routine, so that it can resist to code disclosure attempts. However, it should be noted that the effective external code injection must comply with a number of conditions:

- Execution time of routine subject to external code injection can increase significantly. Because of that fact, the execution time of such routine should not be critical for an application; in other words, the slowdown of the routine with injected dongle-check code should not affect greatly the application overall performance.
- It is desirable that the routine with injected dongle-check code should perform some non-trivial actions. For example, if some routine is implemented as (a part of) a computational algorithm, then with respect to previous point it can be used as an acceptor of a dongle-check code.

![CFG of the routine](image)

**Figure 2.** Injection of external code to the routine: moving basic block 1 to external routine and obfuscating the input data of basic block 1 and the combined dongle-check code.

Therefore, at initial point we have two routines: some routine of an application to be protected (the acceptor routine), which has been selected according to the aforementioned principles (Figure 2, left), and an external routine with some dongle-check security code (Figure 2, right). It should be emphasized that simple combination of two routines by moving the code of dongle-check routine to the original routine and mixing it does not work. The reason is that both routines almost surely contain multiple basic blocks with many transitions between them what results in non-trivial control flow graphs (CFGs). It follows that the problem of combining two routines by injecting the complete dongle-check code into original routine becomes a hard task and also time- and resource consuming.

Our proposition is to select one basic block - we call it the “soluble” block - from the original routine, and inject this basic block into an external routine as shown by grey arrow on Figure 2. In this case, all the operands of instructions of the “soluble” basic block become arguments, values of which have to be passed to the external routine. It is worth mentioning that instructions of the “soluble” block must not necessarily sequentially follow each other after being injected to the external routine, but instead they may be “scattered” throughout the dongle-check code. However, it is important to make sure that the injected instructions of “soluble” block are always executed only once and in a certain order. This can be achieved in two ways:

- by injecting instructions of the “soluble” block to such basic blocks of dongle-check code that are executed exactly once;
- by injecting original instructions to branches following conditional statements, but in this case the code duplication techniques must be used.

At any case, the original instructions cannot be injected to basic blocks within a loop, because here the principle of single execution would not hold true anymore.

Let us discuss in details the way of injecting the instructions of the “soluble” block into external routine. Figure 3 gives us an example of a CFG of some external routine (e.g. dongle-check code or any other security or copy-protect code).
The code of basic blocks 1, 6 is executed exactly once, whereas basic blocks 2, 3, 4, 5 form two branches and the actual direction of control flow depends on some condition. If the \{2, 3, 5\} branch is active, then basic blocks 2, 4 can be executed multiple times, since they are contained within a loop.

Taking into account the previously mentioned constraints, we can make the following propositions:

- The “soluble” block can be injected into basic blocks 1 and/or 6, since these basic blocks are always executed once when the control is transferred to external routine.
- The “soluble” block can be injected to basic blocks 3 or 5, however, the injected code must be duplicated in both branches (e.g. injected to 3 and duplicated into 5, and vice versa).
- There is no way to use basic blocks 2 or 4 to host any instructions of the “soluble” block, since they are contained within a loop body.

After having injected the “soluble” block to the external routine, we have to apply the intermediate level obfuscation algorithm described in [9], first with respect to the original routine, and then to the modified external routine. It is important that in the original routine, instead of the “dissolved” basic block the following instructions are called: parameters assignment, jump to external routine, context recovering.

The basic steps of the applied intermediate level obfuscation algorithm are the following:

- Input data parsing: routines are analyzed, disassembled and translated to specific intermediate representation (three-address code). At this stage, the control flow graph is created, and data dependencies are analyzed. The functions (procedures) are partitioned to basic blocks.
- Addition of fake local context: at this stage a fake local context is added, which will later be used for a garbage code generation and for ensuring polymorphism.
- Garbage code generation: the garbage (fake) code is generated so that it interacts with both local and global contexts. Here code polymorphism is provided by the replacement of some original instructions with a range of other instructions, which after having been executed present the same result. Both instruction sets should work with local and global contexts alike.
- Meshing of branch instructions: since branch instructions play a very important role in programming by determining the control flow and the sequence of program execution, both conditional and unconditional control transfer instructions are to be modified. For example, one single branch instruction can be replaced by several other instructions, and that definitely complicates the code analysis and reverse-engineering.
- CFG masking: different techniques are used, e.g. function inlining/outlining, opaque predicates, adding redundant code, eliminating library calls, function cloning, loop unrolling, and direct graph transformations. The above-listed techniques are also used for general obfuscation. In addition to these, we use another method: replacement of branch instructions with their equivalents in which the transition address is calculated dynamically.
- Splitting basic blocks to a set of functions: by that step, we obtain a sufficiently large number of functions, which are then mixed. As a result, consistently located pieces of code will be “scattered” by a sufficiently large distance from each other.
- Generation of machine instructions: for machine-code generation a polymorphic code generator should be used. That is done to prevent signature search as this is a common approach in reverse-engineering.

To improve the optimization resistance and degree of protection against reverse engineering, we propose the usage of multi-stage obfuscation with respect to the original routine. The described algorithm, excluding the step of splitting...
basic blocks to a set of functions, obfuscates the routine. After that, an already obfuscated “soluble” basic block is injected into external routine as described above, and the obfuscation algorithm is applied with respect to the external routine and the original routine together.

4. CONCLUSIONS AND FUTURE WORK

In the presented paper, we have discussed a method of adding an external (security) code to an application. The emphasis was set on ensuring the impossibility (or very high cost) of localizing and modifying the injected code. The best way of protecting software against reverse engineering and unauthorized modifications is using obfuscation.

The typical usage of the described technique is adding a dongle-check code into an already existing application without having to recompile the software. The dongle itself is usually provided by third party vendors and is aimed at protecting software from unauthorized usage or just protecting one’s code on the client’s end; our task is to ensure the security of a dongle-check code. We are aware of potential weaknesses in the implementation of communication between a dongle and a copy-controlled software, e.g. if a dongle-check code is localized and a protection algorithm is revealed, the reverse engineer can modify the application to bypass the dongle-check. That is why we can further state that intermediate level obfuscation is a good solution that can be used to thwart such reverse engineering attempts.

We have shown that a simple combination of a dongle-check code and a routine cannot be named safe; moreover, sometimes it becomes time- and resource consuming. Instead of that, one basic block is selected from the original routine (“soluble” block) and injected to a dongle-check code. In this case, all the operands of instructions of the injected basic block become arguments, values of which are passed to the external routine. In order to properly identify potential “soluble” basic blocks, which can be injected to external routines, a proper profiling of the acceptor routine should be carried out.

In the presented paper, we have generally described the intermediate level obfuscation algorithm. Its main advantage is that it can be applied to partitioned routines at no allowance and by that we can have a multistage obfuscation technique, where routines can be obfuscated many times. In general, such obfuscation technique can be used to protect software from unauthorized analysis and modification, and consequently to prevent reverse engineering. The algorithm based on this method is completely automatic and can therefore be used as a part of a software protection utility like copy-protection dongles. The main advantage of this technique compared to its counterparts is its platform independence. Doing obfuscation at intermediate level allows us using the same software module at different hardware platforms.

Future work 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 provided by obfuscation algorithm.

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


AUTHOR

Dmitriy Dunaev received his M.Sc. degree in Electrical Engineering from Budapest University of Technology and Economics (BME) in 2006. In 2010, he joined a postgraduate program in computer sciences at BME, and now he is pursuing his PhD degree doing research at the Department of Automation and Applied Informatics. He also works as a lecturer for B.Sc. students at the same University since 2010. He is a member of Applied Computer Science Group. His fields of interest are obfuscation, reverse engineering, and software security.

László Lengyel received his PhD degree in 2006. He is an Associate Professor at the Department of Automation and Applied Informatics of the Budapest University of Technology and Economics (BME). His research field is metamodeling, graph rewriting-based software model transformation, model-driven development, constraint validation, validated model transformation and aspect-oriented techniques.