Copy protection through software watermarking and obfuscation

GERGELY EBERHARDT, ZOLTÁN NAGY
SEARCH-LAB Ltd., {gergely.eberhardt, zoltan.nagy}@search-lab.hu
ERNŐ JEGES, ZOLTÁN HORNÁK
BME, Department of Measurement and Information Systems, SEARCH Laboratory
{jeges, hornak}@mit.bme.hu

Keywords: software copy protection, software watermarking, obfuscation, reverse engineering, trusted OS, mobile software

Enforcement of copyright laws in the field of software products is primarily managed in legal way by the software developer companies, as the available technological solutions are not strong enough to prevent illegal distribution and use of software. Almost all copy protection techniques were cracked within some weeks after their market launch. The lack of technical copyright enforcement solutions is responsible for the failure of some recently appeared business models, partially also for the recent dotcom crash. The situation is particularly dangerous in case of the growing market of mobile software products. In this paper we aim at proposing a scheme which combines obfuscating and software-watermarking techniques in order to provide a solution which is purely technical, but strong enough to overcome the problems concerning software copy protection. The solution we propose is focusing primarily on mobile software products, where we can rely on the hardware based integrity protection of the operating system. **(In: 2006/5, pp.51–57.)**

1. Introduction

According to the statistics of the Business Software Alliance (BSA), the global financial losses due to software piracy were about 30 billion USD in 2004 [3]. Nearly half of the used software is illegal in the European Union. Technical and legal actions could not change substantially this situation. It is a common belief that there is not much to be done against the piracy of software in the PC world. However in the world of mobile phones, where the integrity of the operating system can be trusted, the emerging market of mobile software products has still the opportunity to evolve in such way that the losses due to illegal software distribution could be avoided, or at least moderated.

In our belief the availability of a strong copy protection scheme is the most important prerequisite for further expansion of the mobile phone software market. This is why our research targeted embedded systems used in mobile phones. This way slightly different assumptions can be made than on usually discussed PCs.

A *trusted OS* is an essential ground for achieving a strong copy protection, since it is obvious that any software-based protection can be circumvented if the OS can be tampered. Solutions, when developers do not rely on the OS at all, are also possible. However, with these schemes are based on *security by obscurity:* they simply hide the parts of the code that checks the integrity and validity of the software, assuming that the time needed to reveal and remove this checking is long enough not to remarkably affect the revenues. Although the time needed for reverse-engineering and circumventing the protection of the software can be lengthened with this approach, experience proved that sooner or later all protections based on security by obscurity are cracked.

As opposed to this we propose a scheme, which merges public key infrastructure (PKI) with obfuscation and software watermarking techniques, assuming a trusted and tamperproof OS resulting in a protection supporting freely distributable, but also copy protected software.

2. Theoretical background

Two main categories of software copy protection mechanisms exist: the autonomous systems and those, which use external collaboration [10].

The protections of autonomous systems are integrated into the software itself, so the security depends only on the used software techniques. These techniques include integrity protection, software obfuscation, checksumming, encryption, preventing the running in debug mode and other methods making the task of the cracker harder [8].

The most common solutions are based on the program checking itself. As these checks are part of the program, one can reveal them by reverse engineering and can bypass the protection by modifying the code [10]. These techniques are neither theoretically nor – based on our experience – practically secure enough [2].

The other category of protection mechanisms use external collaboration; more specifically the program uses a tamperproof processor, an operating system or other secure hardware or software solutions. This support can be either on-line or off-line [10]. In case of on-line collaboration some of the checking functions are executed on other computers than the attacker could access. As opposed to this, the off-line collaboration does not require an on-line connection, but only a se-

cure hardware or software item. The secure hardware is usually a smart card, while the secure software is the part of a trusted OS, like in our case targeting mobile phone software.

To link the authorized user to his or her instance of the software, usually the services of a public key infrastructure [9] are used. For a copy protected software a license containing the information about the user, about the product issuer or distributor, and about the product itself (e.g. a hash of the particular software instance) should be attached to the product, so that the OS could check the authorization. The integrity of this license is protected by a digital signature, and the OS should not run copy protected software without the appropriate license.

However, to support multiple use cases and business models, and to allow comfortable software development, the OS should be capable of running both copy protected and unprotected software, which is one of the biggest challenges in developing a successful copy protection scheme. This means that in a manipulated piece of software or when the license is removed the OS should also detect that the code was initially protected.

As opposed to usual protection models used in multimedia files using watermarking to trace a content in order to identify its origin, software use watermarking to make possible the indication on the code that it is copy-protected. As the instructions of the code can be obfuscated arbitrarily as long as the user-perceptible output remains the same, this enables us to implement more efficacious watermarks to software than to audio/video files.

We can differentiate between two types of *software watermarking* techniques: static and dynamic. In case of static watermarks the information is injected into the application's executable file itself. The watermark is typically inside the initialized *data, code or text* sections of the executable [1,7,8,11,13,14]. As opposed to this, the dynamic watermarks are stored in the program's execution state, and not in the program code. This means that the presence of a watermark is indicated by some run-time behavior of the executable [5,6,12].

To prevent the easy removal of watermarks we can use *software obfuscation,* which is a collection of several different code transformations with the common goal to make the reverse engineering more difficult both in case of automatic tools and for the human understanding of the code [4,15]. The most important aim of these transformations is to make code transformations meant to remove the watermark hard to accomplish.

3. Requirements and quality goals

In this section we define the requirements and assumptions towards our copy protection scheme.

3.1. Trusted operating system – integrity & confidentiality

For a trusted OS we assume that the *integrity* of the undergoing processes is ensured (e.g. protected memory areas cannot be changed). However the *confidentiality* of the information flowing through the OS does not have to be guaranteed. This implies that although the method of watermark detection is kept in secret, it is hard to remove the watermark from the protected application even if the watermark generation and detection algorithms are well known to the public, thus avoiding security by obscurity.

The OS should also be capable of checking the digital signature of the applications and support PKI with certificate chains and revocation.

3.2. Same observable behavior

In our case the *observer* is the customer, longing to use the protected program. From his or her point of view the transformed program should be functionally the same as the original one. For example it should have the same windows, the same files and the same connections to the outside world, and all these should behave in the same way. However the speed, the memory usage and the inner states of the program during the execution, including the program code could be modified slightly or even to a greater extent.

3.3. Harder to reverse engineer

On one hand the obfuscation should make the *automatic* decompilation of the code very difficult, while on the other hand the code should be made incomprehensible for *human understanding* to the greatest extent possible [8]. Our goal is to make the reverse engineering difficult. This means that the reverse engineering and thus removing the protection should be extremely time-consuming in order not to be worth the effort and time spent on doing it. To state it in other way, *difficult* and *to make harder* here means that the decompilation of the protected program should include the solving of such a complex problem, whose complexity can be deduced to a problem accepted to be hard in cryptography, so typically being equivalent with as if one should have to break a cryptographic algorithm.

3.4. Harder to remove the watermark

This quality indicator is strongly connected to the one described in the previous paragraph. A robust obfuscation method not only makes the reverse engineering harder, but also prevents the easy removal of the watermark.

As in case of the trusted OS we assume only the integrity, but not the confidentiality: we have to hypothesize that the watermark detection method cannot be kept in secret. Thus, the removal of the watermark has to be hard enough also when the detection method is public, or when everybody can detect the watermark. The watermark has to be embedded in the original code deeply enough so that the watermark should not be removed without the full understanding of the *whole*

code. The watermark generation on asymmetric encryption can be used for example where the private key is kept in secret, but the algorithm itself is considered to be public.

3.5. Scalability of the transformations costs

Because every transformation has a cost in terms of speed and memory usage, it is important that the protection is effective from this point of view as well. To fulfill the different requirements and limitations on transformation costs for the different areas of the code, the accomplished transformations should be *scaleable* in terms of speed and memory usage of the resulting transformed applications.

4. The proposed copy protection scheme

To summarize, our scheme is constructed from the above mentioned building blocks, taking into account the listed requirements. The integrity of the software is ensured through a digitally signed license, and if the license or its digital signature is invalid, the OS prevents the application from running. However, if there is no license attached to the application, the OS can start it, but should continuously check for the presence of watermarks in it, which designates that it is a protected piece of software, so should have a license file attached. The removal of watermark is hard because of different obfuscation methods utilized, so the attacker cannot change or break the protection to make the application run without the license.

The license checking algorithm being the most important part of our scheme is shown on the *Figure 1.*

(1) Check if a digitally signed license is attached to the application.

(2) If there is a license, and if both the license (e.g. its hash signature) and the digital signature are valid, then the application can be run without accomplishing any further checking for watermarks.

(3) If the license or its digital signature are not valid, the application should stop immediately, and the OS should make the appropriate steps (for example by logging or even reporting the event), as in this case the copy protection is violated.

(4) If there is no license attached to the application, it could either be freely distributable software but also a copy protected but manipulated program, so it should be allowed to start.

(5) Parallel to this the OS should start the continuous search for watermarks.

(6) If a watermark is found in the application, its running should be stopped immediately, and again the appropriate steps should be made, because the presence of the watermark unambiguously signals that a license should be attached to the application, without which it is an illegal copy.

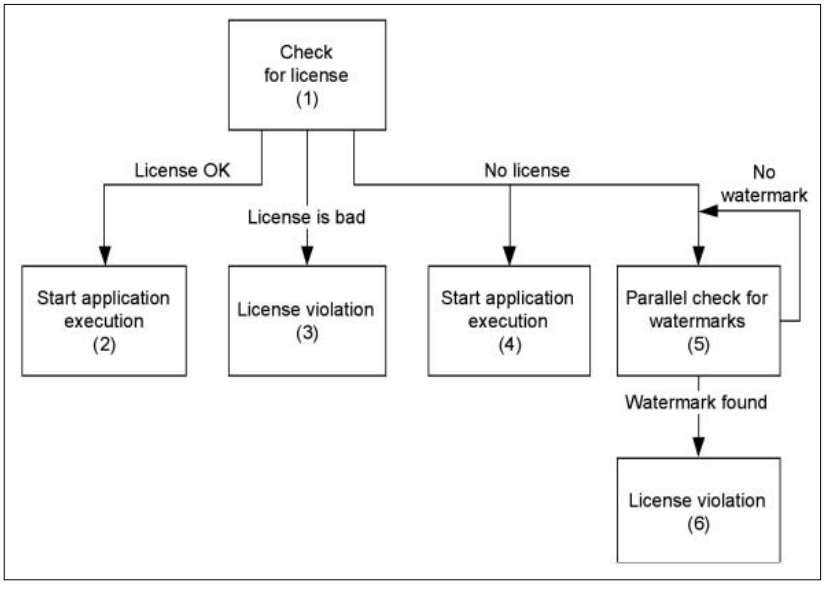
So, in our scheme we use software watermarks to indicate that the program is copy protected, thus the inserted watermark should have the following properties:

- it has to be able to store only a one bit information, indicating that the program is copy protected,
- it should not use a secret method to hide the watermark, as the attacker can reveal and understand the watermark detection algorithm,
- all watermark removal should not be possible via automated or manual means even in cases when the attacker might know the detection algorithm.

To fulfill these requirements we have chosen to use dynamic watermarks, because in this way the binary form of the watermark is formed during the program execution.

To detect dynamic watermarks the program should run for a while. This implies that the program state should be checked continuously during the execution. There is no time limit, after which it is assumed that there is no watermark in the program. The watermark detection procedure could slow down the execution of the application, so in the proposed copy protection checking algorithm the watermark detection is done only in case the license file is not present. So it will be the developers' essential interest to provide licenses to their released products in order to exploit the capabilities of the device to the best extent possible.

Figure 1. The license checking algorithm



The connection between the program and the OS during the watermark checking process is illustrated in the *Figure 2.*

Our dynamic watermark only stores one-bit information in form of a special number derived from a random value appended with its value transformed with a function f.

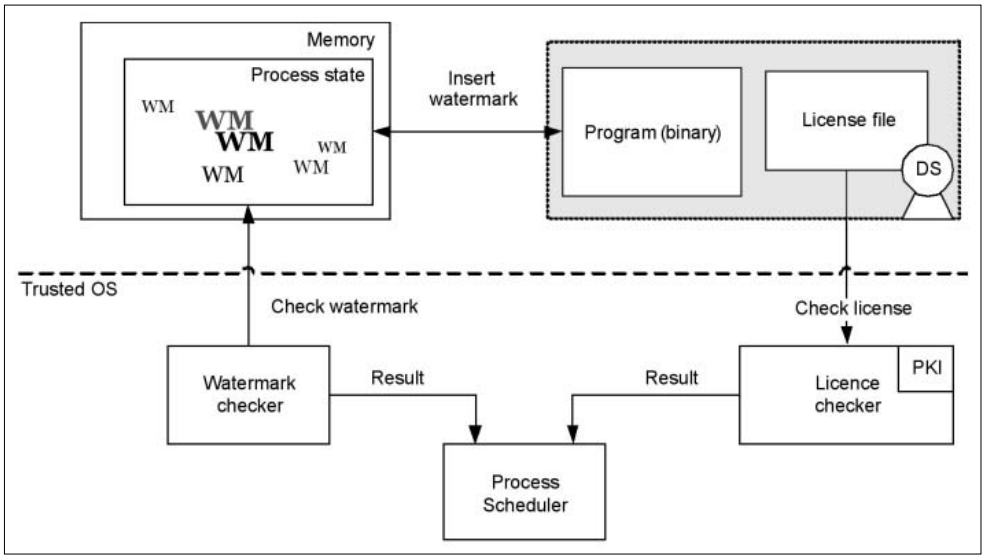


Figure 2.
Connection between the program and the OS during the watermark checking process

The transformation accomplished with this function can be a digital signature, a hash value, CRC or others for example even a simple XOR operation with a constant value; its role is simply to keep the probability of appearing of such a value pair in a non-protected application low, so if such a value appears in an application frequently enough, it can designate the presence of the watermark.

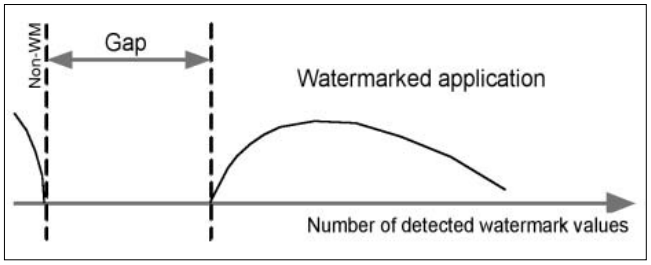
So the watermark can be defined as follows:

$$WM = \left(RND;\ f(RND)\right)$$

These different *WM* values should be hidden in the application as frequently and for as long as possible, which means that these values should appear in the state of the program frequently enough to allow their detection and statistical evaluation of these detections. To achieve this goal we can for example pick the parameters of different data obfuscation transformations in a way that one or more original values of a variable (typically a loop control variable) are transformed into watermarks values, which are then stored in program state in the transformed domain.

The gap between the prevalence of such watermark values in non-watermarked and watermarked applications, as shown in the *Figure 3*, can allow easy detection of the watermarks by statistical means.

Figure 3.
The gap between non-watermarked and watermarked applications regarding the prevalence of watermark values



As the attacker cannot know the exact value of the watermark due to its randomness, he or she can only detect, when it is formed in the program's state resulting from the appropriate inputs. Therefore the attacker has to execute all branches of the program with all possible input values to be sure that the watermark is fully removed.

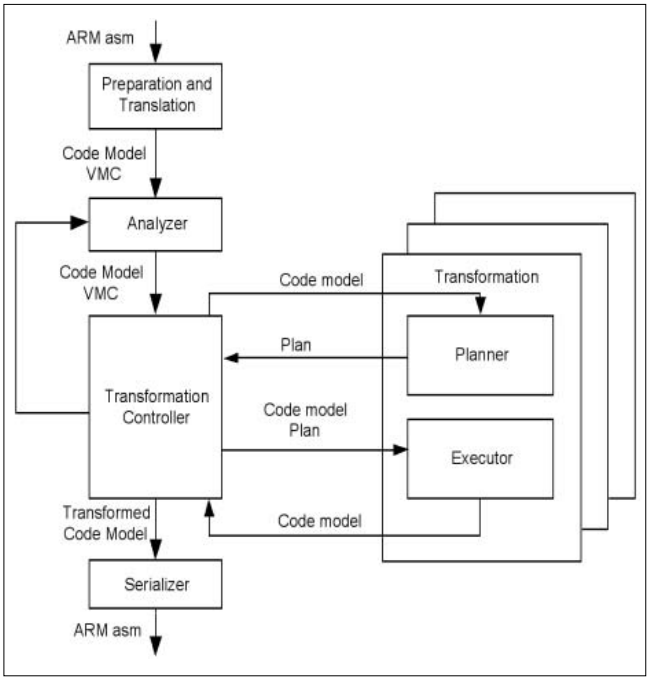
5. The architecture of the system

Figure 4 below illustrates the main parts of the system and their connections. The obfuscation and the software watermarking transformations take place integrated into the usual C/C++ compilation process, which usually starts with a preprocessing step.

The inputs of the system are the C/C++ source files of the application to be protected. Different directives can be placed in the source code to control the obfuscation and watermark insertion process. These directives are collected during this preparation process, and the original source is passed to the *Compiler*, which will produce LST file and the debug information file.

The latter two form the basic sources of information about the code, but other files originating from the *Disassembler* (disassembly LST), *Linker* (map file) and the *Profiler* (profile information) can be also used for this purpose.

Figure 4.
The modules of the system and their connections



The collected information is merged and prepared to form a compiler independent representation called the *Code Model*, which is used to accomplish the needed transformations, while the code itself is translated into an abstract representation called *Virtual Machine Code (VMC)*, on which the transformations will be carried out.

After this preparation the system analyzes the gathered information, during which the *Analyzer* module performs control and data flow analysis and finalizes the *Code Model*, which then contains all necessary information to plan and accomplish any watermarking or obfuscating transformations.

The transformations are accomplished in several iterations. Upon every iteration the *Transformation Controller* creates a detailed plan about the transformation to be accomplished in the current step, along with their sequence. During the transformations the intermediate representation must remain in consistent state, which means that even after every iteration the code should be functionally equivalent with the original one.

To ensure efficiency and functional equivalency of the code, every transformation is done in two steps. The first step is responsible for making a transformation plan *(Planner)* and the other is responsible for the execution of this plan *(Executor)*. The Planner is responsible for finding proper and optimal parameters for a specific transformation in the current context, while the Executor ensures the functional equivalency. This way it is enough to prove the correctness of the Executor formally.

The transformation steps batched in the iterations are applied to the abstract representation of the code until the expected goals are reached, namely the hiding of the adequate number of watermarks and reaching the desired level of obfuscation. After the transformations are accomplished, the abstract representation of the code is translated (serialized) back to an assembly source code, which is ready to be compiled by an assembler. The result of the process is the compiled object code, which is on one hand obfuscated and on the other hand it contains the watermark.

6. Results

To evaluate our copy protection scheme we have implemented the full framework system based on the above introduced architecture along with a number of transformation methods. At the end of this article introducing the proposed scheme and the framework system architecture we illustrate the capabilities of such architecture with a simple example, the implementation of the *Hide Library Calls* obfuscation technique.

Most programs heavily use calls to the standard libraries and to the operating system, and since the semantics of the library functions are well known, such calls can provide useful clues to perform reverse engineering on an application. The *Hide Library Calls* technique can be used to dismiss this help from the code.

There are different methods to obfuscate the calls to such fixed functions. The basic idea behind all of these transformations is that the original API call is changed to an inner *wrapper* function of the application, which will call the real API function. The obfuscator can create an interface function to each API call, but the call of each API function can be integrated into a single function as well. In this latter case usually the value of an input parameter designates which API call should be called by the wrapper function.

In case of this obfuscation the *Planner* (see architecture in Figure 4 above) is simple, because it has to find the possible call references and has to choose a subset of them, which are to be hidden (even all can be chosen to be hidden). So the generated *Plan* contains the list of call places. The steps of the algorithm for hiding library calls once the plan is ready is as follows:

1. Creating a new function, which will be the wrapper function calling the actual API functions.

2. Assigning identifiers (values) to API calls to be hidden, and creating the instructions and the blocks of the wrapper function regarding to the calls and the identifiers.

3. Changing the original instructions calling API functions to point to the new wrapper function and passing the appropriate identifiers as additional arguments to it.

The following example shows an API call after the *Hide Library Calls* transformation is accomplished:

```
   ldr lr, LI11          @ Save return address
   mov ip, #1            @ Set ip to the called function ID
   ldr r5, LI12          @ Load address of global variable
   str ip, [r5, #0]      @ Save ID to the global variable
   b HideCalls_2         @ Call function
LI11:
   .align 0
   .word .L12            @ Address of the next function
LI12:
   .align 0
   .word .LD110          @ Address of the global variable
.L12:
```

7. Summary

In the above article we have presented our scheme, which combines cryptography, software watermarking and obfuscation in order to achieve a strong technical solution for software copy protection, targeting primarily the mobile software developers. Based on this scheme we have designed the architecture of a protection tool that can be integrated in a development environment to provide copy protection services.

The architecture is robust and open in a sense that the module dealing with transformations – both water-

marking and obfuscation – is completely independent of the processor, the OS and the development environment, as it works on an abstract representation of the source code. This way, by replacing the preprocessing, translating and serializing modules, we can integrate our system into several environments.

As in the case of code transformations the formal proof of correctness of transformations is essential, all transformation are done in two steps: after planning the particular transformations in order to form a transformation sequence that fulfills our goals, the separate and much simpler transformation steps are executed so that their accomplished activity can be formally proven to be correct. This proof should be done for all transformations that can be executed within our framework.

Having the framework ready, the next step in our research is to broaden the set of such transformations to test different kinds of obfuscation, and to inject certain code fragments into our *Code Model* to implement dynamic watermarking. Our goal is on one hand to develop and test the efficiency of several control and data obfuscation methods, and on the other hand to hide dynamic watermark in the code and accomplish several measurements regarding the ability of an independent code (which will be the OS) to detect them.

Acknowledgements

The project is being realized by the financial support of the Economic Competitiveness Operative Programme (Gazdasági Versenyképesség Operatív Programja, GVOP 3.1.1/AKF) of the Hungarian Government.

References

[1] G. Arboit,
"A Method for Watermarking Java Programs via Opaque Predicates",
In: The 5th International Conference on Electronic Commerce Research (ICECR-5), 2002.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang,
"On the (im)possibility of obfuscating programs",
In: Proc. CRYPTO'01.
Lecture notes in computer science, Vol. 2139.
Springer, Berlin-Heidelberg-New York, 2001.
pp.1–18.

[3] First Annual BSA and IDC Global Software Privacy Study, Business Software Alliance and IDC Global Software, 2004.

[4] C. Collberg, C. Thomborson, D. Low,
"A Taxonomy of Obfuscating Transformations",
Technical Report 148, Dept. of Computer Science, The University of Auckland, 1997.

[5] C. Collberg, C. Thomborson,
"On the Limits of Software Watermarking",
Technical Report 164, Dept. of Computer Science, The University of Auckland, 1998.

[6] C. Collberg, C. Thomborson, G. M. Townsend,
"Dynamic Graph-Based Software Watermarking",
Technical Report TR04-08, 2004.

[7] R. Davidson, N. Myhrvold,
"Method and system for generating and auditing a signature for a computer program",
US Patent 5,559,884,
Microsoft Corporation, 1996.

[8] G. Hachez,
"A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards",
Ph.D. thesis, Universite Catholique de Louvain, 2003.

[9] International Telegraph and Telephone Consultative Committee (CCITT):
The Directory – Authentication Framework,
Recommendation X.509, 1988.

[10] A. Mana, J. Lopez, J. J. Ortega, E. Pimentel, J. M. Troya,
"A framework for secure execution of software",
International Journal of Information Security,
Volume 2, Issue 4, Springer, November 2004.
pp.99–112,

[11] A. Monden, H. Iida, K. Matsumoto,
"A Practical Method for Watermarking Java Programs",
The 24th Computer Software and Applications Conference (Compsac'00), Taipei, Taiwan, Oct. 2000.

[12] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, Y. Zhang,
"Experience with Software Watermarking",
In: Proc. of the 16th Annual Computer Security Applications Conference, ACSAC 2000,
pp.308–316.

[13] J. P. Stern, G. Hachez, F. Koeune, J.-J. Quisquater,
"Robust Object Watermarking: Application to Code",
In: A. Pfitzmann, editor, Information Hiding '99,
Vol. 1768 of Lectures Notes in Computer Science (LNCS), Dresden, Germany, 2000.
pp.368–378.

[14] R. Venkatesan, V. Vazirani, S. Sinha,
"Graph Theoretic Approach to Software Watermarking",
In: Proceedings of the 4th International Workshop on Information Hiding table of contents, 2001.
pp.157–168.

[15] G. Wroblewski,
"General Method of Program Code Obfuscation",
Ph.D. thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.