Copy protection through software watermarking and obfuscation

Gergely Eberhardt, Zoltán Nagy [1,2,3]

SEARCH-LAB LTD.
Budapest, Hungary

Ernő Jeges, Zoltán Hornák [4,5]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary

Abstract

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 technique was cracked within some weeks after market launch. The lack of technical copyright enforcement solutions was responsible for the falling of some recently appeared business models, thus partially for the recent dot-com crash, and is particularly dangerous for the growing market of mobile software products. In this paper we aim to propose a scheme which combines obfuscating and software-watermarking techniques in order to provide a purely technical, but strong enough solution to software copy protection, focusing primarily on mobile software products, where we can rely on the hardware based integrity protection of the operating system.

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

[1] The project is being realized by the financial support of the Economic Competitiveness Operative Programme (GVOP) of the Hungarian Government
[2] Email: gergely.eberhardt@search-lab.hu
[3] Email: zoltan.nagy@search-lab.hu
[4] Email: jeges@mit.bme.hu
[5] Email: hornak@mit.bme.hu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

According to the statistics of the Business Software Alliance (BSA) the global financial losses due to software piracy are about $30 billion in 2004. Considering the European Union, nearly the half of the used software is illegal [6]. Technical and legal actions could not change substantially this bad situation in the recent decay. It is a common belief that there is not much we can do against piracy of software in the PC world. However in case of mobile software which runs in an embedded environment, where the integrity and functionality of the operating system can be trusted, the emerging market of mobile software products has still the opportunity to evolve in way where losses due to illegal software distribution can be avoided, or at least greatly moderated.

As the availability of a strong copy protection scheme is being more and more a prerequisite for further expansion of the mobile phone software market, our research targeted embedded systems used in mobile phones, for which slightly different assumptions must be made than on usually discussed PCs.

The *trusted OS* is an essential ground for achieving strong copy protection since it is obvious that any software based protection can be circumvented if the OS can be tampered and one can solve that a copy protected software receives simulated responses to function calls. Solutions to protect the spreading of illegal copies of software, when developers do not rely on the OS at all, are also possible, however these schemes must be based on security by obscurity: they simply hide the parts of the code that check 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 their revenues. Although the time needed to reverse-engineer the software and circumvent the protection can be lengthened with this approach, experiences proved that sooner or later all protections based on security by obscurity have been cracked. That's why our approach was to design such a copy protection scheme, where – based on the assumption that the integrity of the OS can be ensured – it can be proved that the complexity of cracking is as hard as breaking an encryption. That is the solution is based on a hard mathematical problem.

To support multiple use cases and business models, it is also an important aspect that the OS should be capable of running both copy protected and unprotected software. This means that even in a manipulated piece of software the OS should detect that the code was initially protected so in our model the sign of copy protection should be contained in the software itself, and the OS should do the checking of authorized use regarding to this embedded information.

To link the authorized user to his or her instance of the software, the public key infrastructure ([8]) can be used. If the software is detected to be copy protected, a software license should be attached to it, containing the information about the user, about the product issuer or distributor, and about the product itself (in form of a hash of the particular software instance) in

order that the OS could check the authorization. The integrity of this license is protected by a digital signature, and the OS should not run a copy protected software without an appropriate license.

In our proposed solution we use software watermarking to hide the sign of copy protection and apply obfuscation to effectively make it hard – by means of complexity – to remove the watermark. As opposed to usual protection models, which use watermarking to trace a content to identify its origin, we only use it to hide the sign that the code is copy-protected. In spite watermarking techniques proved to be weak in other fields of copy protection, namely in audio and video files, in case of software watermarking the situation is substantially different. In case of software there is no such requirement that the user should not notice the existence of the watermark, the instructions of the code can be arbitrary obfuscated as long as the user-perceptible output remains the same. This different possibility gives us the chance to implement much stronger watermarks to software than to audio/video files. We can differentiate two types of *software watermarking* techniques: static and dynamic. In case of static watermarks the information is injected into the application executable file itself. The watermark is typically inside the initialized data, the text, the symbol or the code sections of the executable [1], [5], [7], [9], [11], [12]. Dynamic watermarks, which we are primarily focusing on, are stored in the program's execution state, rather than in the program code itself, which mean that some run-time behavior of the executable signals the presence of a watermark [3], [4], [10].

To protect the watermark from easy removal we use *software obfuscation*, which is a collection of many different code transformations with a common goal to make the reverse engineering harder both for automatic tools and from the aspect of human understandability of the code [2], [13]. The most important aim of these transformations in our case is to make hard automated reobfuscation of the code, which could remove the watermark. The level and complexity of obfuscation then ensures very hard understandability for a human.

Our scheme is constructed by combination of these techniques (PKI, obfuscation and software watermarking), taking into account the assumptions of a trusted and tamperproof OS and the need for both free distributable and copy protected software.

To put the above described building blocks of our scheme in a nutshell, the integrity of software is ensured through a digitally signed license, presence of which is mandatory if the product is copy protected. Whether software is copy protected or not is signaled to the trusted OS by a dynamic watermark, which is embedded in the software and made irremovable by obfuscation methods. For all used obfuscating and watermarking transformations iff should be formally proven that the transformed and the original codes are identical, which means that their observable behavior is the same for any valid input (which implies that for invalid input the behavior can be different as described later

on).

Because the integrity of the operating system can be ensured in mobile devices, we can assume that the watermark detection and license checking functions in the OS cannot be tampered, thus the device can reliably detect the watermark and can check the license rights utilizing strong cryptography through PKI and digital signature based solutions. As the majority of embedded systems in mobile phones use ARM processors, the first implementation of our scheme shell do the obfuscation and watermarking transformations on assembly level of this processor.

2 Requirements and Quality Goals

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

Trusted Operating System – Integrity and publicity

The integrity of the undergoing processes in a trusted OS are ensured (e.g. protected memory areas cannot be changed), however the confidentiality of the information flowing through the OS is not necessarily guaranteed. This implies that although the method of watermark detection could not be kept in secret, it should be hard to remove the watermark from the protected application even if the watermark generation and detection algorithm are well known to public.

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

Same observable behavior

The observer in our case is the customer, who wants 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 same windows, same files and same connections to the outside world, and all these must behave in the same way. Only the speed, the memory usage and the inner states of the program during the execution, including the program code can be modified slightly or even heavily. However, this must not affect the user experience of the program. (The protected code should fit into the available memory space and the overall speed-decrease should not be embarrassing.)

Harder to reverse engineer

This goal can have two different meanings, both of which are interesting for us: on the one hand the obfuscation should make it very difficult to *automatically* decompile the code while on the other hand the code should be made incomprehensible for *human understanding* as much as possible [7]. We want to protect the target program from both reverse engineering methods.

So, our goal is to make the reverse engineering hard to accomplish, which means that a human (cracker) could understand the program - even using both automatic tools and human intelligence - spending so much time in doing it, that it is simply not worth to remove the protection. So in this case the word *hard* means that the decompilation of the protected program is an NP-complete problem (in worst case the cracker can always re-implement the full application, so theoretically and practically we cannot go beyond that limit).

Hard to remove the watermark

This quality goal is strongly connected to the previous one. Robust obfuscation method not only makes the reverse engineering a hard task, but also makes the watermark hard to remove.

The applied methods should not rely on security-by-obscurity type secrets, as in the planned protection scheme we assume only the integrity, but not the confidentiality of the OS, so we have to assume that the watermark detection method cannot be kept in secret. Thus the removal of the watermark has to be hard task also in case the detection method is publicly known, and moreover everybody can detect the watermark oneself. Naturally for the watermark generation asymmetric encryption can be used, where the private key is kept in secret, but the algorithm itself is considered to be public. For this reason the watermark has to be embedded in the original code so deeply that the watermark could not be removed without the full understanding of the whole code.

The cost of the transformations should be low

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

To give the control to the user to specify these requirements the source code, which should be syntactically and semantically correct, can also contain additional *directives* about the expected obfuscation level, which can mark program fragments that should not be obfuscated, due to implementing some speed critical functions; or as opposite, fragments that need the highest available level of obfuscation.

3 The Proposed Copy Protection Scheme

In our scheme the OS's integrity is protected, thus the user cannot change the OS or its undergoing processes in any way. This is mandatory because it will be the responsibility of the OS to search for watermark in the programs, and decide about allowing a program to run or not.

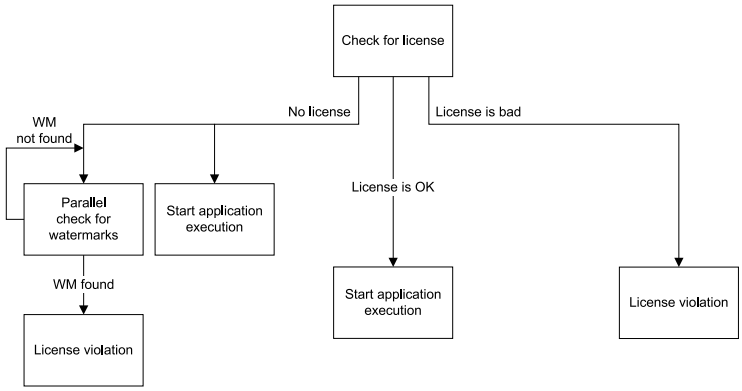

Fig. 1. License checking algorithm

The license checking algorithm being the most important part of our scheme is the following (see Figure 1):

(i) Check for digitally signed license file for the program.

(ii) If there was a license and the signature was correct, then run the program without checking for a watermark. If the license does not fit the program, or the digital signature is incorrect the execution is immediately stopped, or necessary steps (e.g. reporting or logging) can be taken, as the copy protection, or at least the program's integrity is violated.

(iii) If a license is not attached to the program, it can either be an unprotected code or a manipulated protected code, so the OS should start the search for a watermark in it. If the watermark is found, then the execution of the program should be stopped, or again the necessary steps should be taken, as the presence of the watermark indicates, that this is a copy protected program, so there must have been a valid license attached to it, and without it, it is a pirated copy. If there is neither a license file, nor watermark is found, the program is not considered to be copy protected, and can be run or copied freely.

The generation of the watermark is done in the following way:

In our solution the watermark is derived from a random value and a CRC is calculated for this random value. So the watermark can be expressed with an f function:

$f(RND, CRC) = w$.

This w value should be hidden in the application as much times as possible using dynamic or static watermark techniques.

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

The signature check of a license file is a slow process so it can be started immediately when the program starts or even before – when it is installed.

In case of static watermarks, the watermark information is embedded in the structure of the program, and can be detected by analyzing the binary ap-

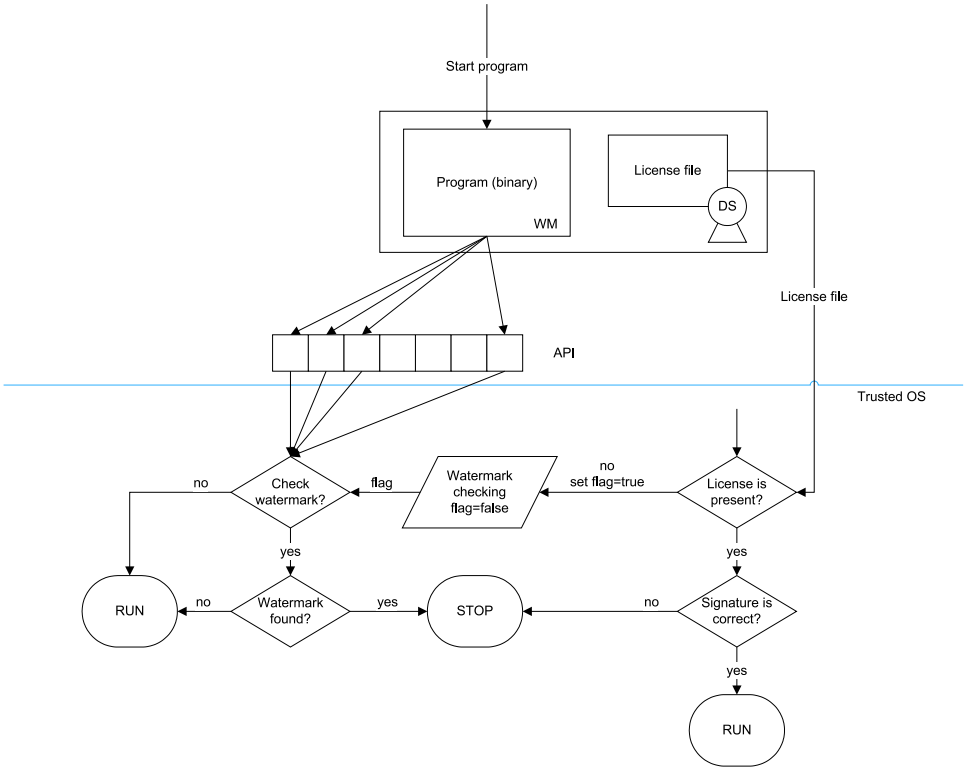

Fig. 2. Connection between the program and the OS during the watermark checking process

plication without executing it, even before the protected application is started.

In case of dynamic watermarks, the binary form of the watermark is formed during the program execution, thus the program should be run for a while for the watermark to be detected. This means that the watermark should be checked continuously during the program execution, and usually a time limit is given, after which it is assumed that there is no watermark in the program, but it is not necessary. Thus the watermark detection procedure takes time, so it could slow down the device, thus the execution of the application. All this implies that for dynamic watermarks the watermark detection is done only in case the license file is not present, every program without a signature will run slower (note that time spent on watermark detection can be freely defines, e.g. to 10% of the actual process's execution time). So it will be an essential interest of the developers to provide licenses for their products if they want to exploit all the capabilities of the hardware.

4 The Architecture of the System

Figure 3 illustrates the modules 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

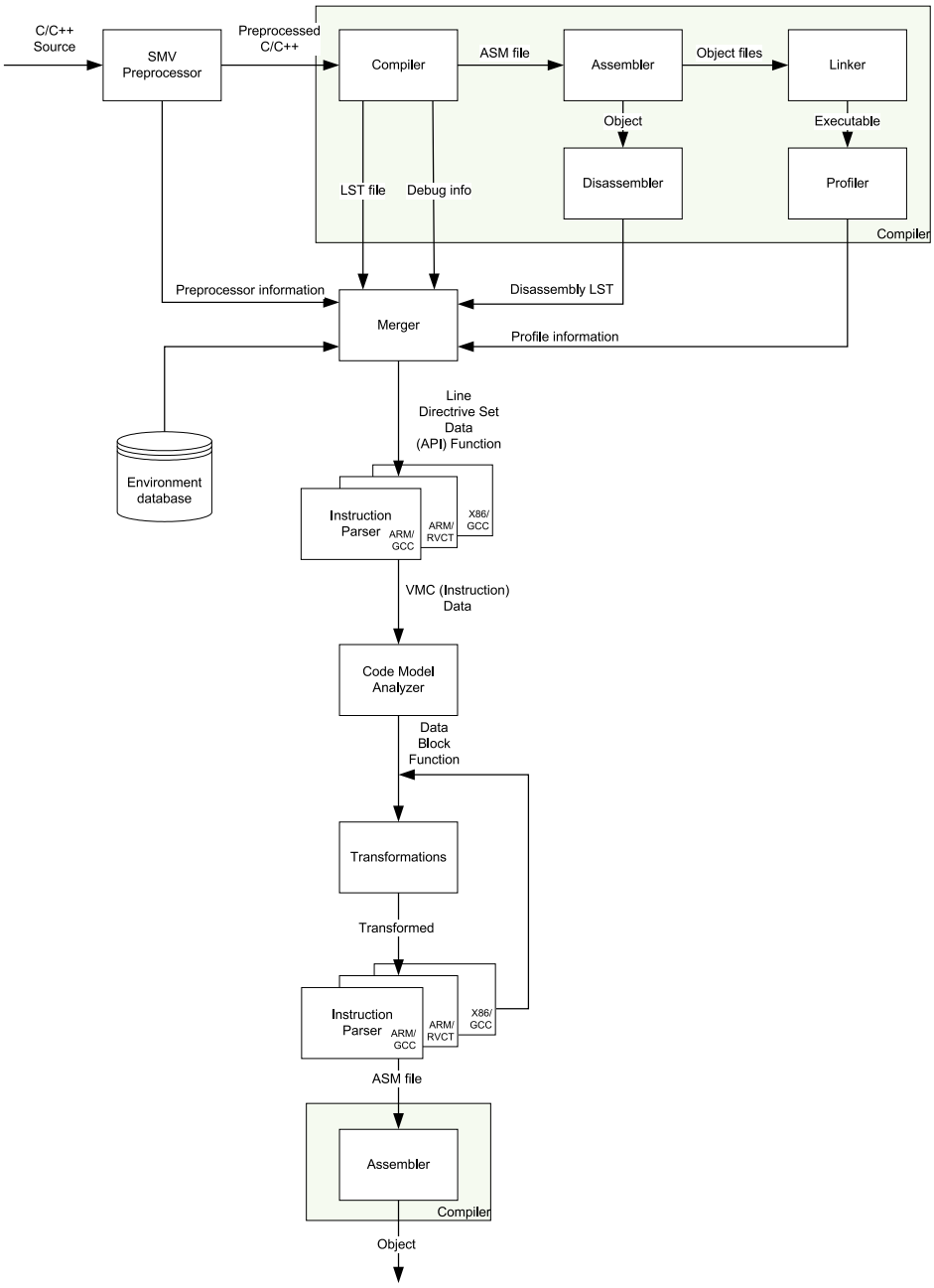

Fig. 3. The modules of the system and their connections

a preprocessing step. The input containing different directives of the system is formed of the C/C++ source files, specified by the user, which can influence and help the obfuscation and watermark insertion process. These directives are collected by the *Source Code Preprocessor*. The original source is passed to the *Compiler* which will produce the necessary files for the obfuscation and watermark injection. These files are the LST file and the debug information file, but other files can be used as well originating from the *Disassembler* (Disassembly LST), *Linker* (Map file) and the *Profiler* (Profile information).

The described information collected from different sources will be merged into a compiler independent description by the *Merger* module. The resulting merged information will be the essential input of our system to accomplish the needed transformations.

After this preparation the system analyzes the assembly language code extracted form the LST file. It detects and analyses the lines containing the assembly statements. During this analysis the labels, the instructions and their parameters will be detected, and the *Line* data structure will be constructed and filled in. Based on this prepared list of *Lines* and other data the *Code Model Analyzer* module performs control flow and data flow analysis and finalizes the intermediate representation of the assembly code, called the *Code Model*. After this step, the *Code Model* contains all information needed to plan and accomplish the watermarking and obfuscating transformations. These transformations are responsible both for control and data obfuscation and watermark injection.

Before the transformations are executed the *Transformation Controller* creates a detailed plan about the used transformations and their sequence. The transformations are responsible for accomplishing a transformation in a way, that the intermediate representation remains in consistent state, which means that after every transformation the code should be functionally equivalent with the original one.

To ensure efficiency and functional equivalency of the code, accomplishing every transformation is done in two steps. The first step is responsible for making a transformation plan and the other is responsible for the execution of this plan. The planner is responsible for finding proper and optimal parameters for a specific transformation in the current context, but the functional equivalency has to be ensured by the executor. In this way it is enough to formally prove that the executor is correct, which is much simpler than the planner.

After these steps the transformations are applied to the *Code Model* until the expected goal (the hiding of watermark and the intended level of obfuscation) is reached.

After the transformations are accomplished, an assembly source code has to be generated based on our (transformed) representation. This will be an assembly file ready to be compiled by an assembler. The result of the process will be the compiled object code, which is on the one hand obfuscated and on the other hand it contains the watermark.

5 Summary

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

As in 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 to form a transformation sequence to fulfill our goals, the separate and much simpler transformation steps are executed in way 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.

References

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

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

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

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

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

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

[7] Hachez, G., "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.

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

[9] Monden, A., H. Iida, and K. Matsumoto, *A Practical Method for Watermarking Java Programs*, The 24th Computer Software and Applications Conference (compsac2000), Taipei, Taiwan, Oct. 2000.

[10] Palsberg, J., S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, *Expreience with Software Watermarking* In Proceedings of the 16th Annual Computer Security Applications Conference, ACSAC '00, pages 308-316, 2000.

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

[12] Venkatesan, R., V. Vazirani, and S. Sinha, *A Graph Theoretic Approach to Software Watermarking*, In Proceedings of the 4th International Workshop on Information Hiding table of contents, pages 157-168, 2001.

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