Digital Watermarks for C/C++ Programs

Andrey Fionov
Siberian State University
of Telecommunications and Information Science
Novosibirsk, Russia
a.fionov@ieee.org

Abstract

The results of the first stage of the project devoted to development of watermarking system for C/C++ programs are presented. Watermarks are embedded by making slight equivalent variations of program's source codes. It is supposed that such watermarks will be robust with respect to attacks on binary executable files when the adversary has no access to their source codes. The rate of embedding for typical Symbian applications is experimentally obtained. Remaining problems and directions of future research are discussed.

I. Introduction

Digital watermarking is a branch of steganography which aims at hiding data (watermarks) in digital media or files in such a way that it is hard to remove or tamper the hidden mark. In contrast to the classical problem of steganography, i.e. the problem of concealing the fact of data transmission, it is usually not required to conceal the presence of watermarks since in majority of applications, it is known in advance whether a file is watermarked or not.

One application of digital watermarks is protection of authorship. Suppose you have created a software distribution package and embedded the watermark "FRUCT" in it. After some time you find a company selling this package as its own, may be, with changed name and meta data in the inner files. But if the watermark is robust, i.e. it withstands removing or tampering, you will be able to recover the word "FRUCT" and prove your authorship in the court. It is highly improbable that the watermark "FRUCT" had arisen in files of the package by chance.

Another, more challenging application is unauthorized copying prevention. The idea is to supply each copy of distributed data with a unique watermark which would allow to trace back a user who makes illegal copies. The problem here arises when two or more users create a coalition. They can compare their copies of data and locate all differences. It is assumed that the knowledge of these locations may help them to alter the watermark and thus to produce a copy which cannot be traced back to either of them. To prevent this possibility, the so-called fingerprinting codes are used as watermarks. Special construction and redundancy of such codes guarantee that no coalition of the size smaller than some predefined value can create an untraceable watermark. The challenge to watermarking scheme consists in providing enough room for long fingerprinting codes.

As we speak about watermarking in programs, it should be noted that embedding data in program files has its own specific, for example, it cannot be done by direct modification of some bits of executable files because any such modification, with high probability, will destroy the program. Embedding must be done in a way that ensures proper program functioning. A problem of software watermarking was stated and considered in a number of works [1], [2], [3], [4], [5], [6], [7]. There are two basic approaches. The first one suggests the usage of special additional functions that are added to the body of a program [2], [4], [5]. The code of additional functions contains the desired watermark, e.g., in the form of constants used in various computations. These added functions must not look like a dead code (a code which is never executed) because otherwise they may be easily removed by dead code elimination algorithms. So they are tightly coupled with the program and are invoked during program execution. The advantage of this approach consists in virtually unlimited length of watermark. However, the drawback is a potential degradation of program performance. Another drawback is vulnerability to attacks that find some known patterns of watermarking function constructions and the ways of invocation.

The second approach does not rely on adding explicit codes but rather employs some subtle redundancies in program files that allow to embed a watermark [1], [3], [6], [7]. A similar approach is used in [8], [9], [10] to hide data directly in executable files. The common feature of these methods is finding some equivalent alternatives in executable file construction and embedding data through the selection of a particular alternative. The suggested alternatives include instruction type selection, instruction scheduling, code layout, register allocation, variables reordering and permutation of addresses in the import tables. Note that some ideas may be implemented in a ready executable file while others are efficient only at compile time and, therefore, require a specially designed compiler. We will briefly illustrate the essence of each method. The examples throughout the paper will be given in C/C++ and ARM processors assembly languages. ARM processors are dominant in the mobile and embedded electronics market and constitute the hardware platform for mobile devices of Nokia, Sony Ericsson and others.

The instruction selection method consists in finding different code sequences that perform the same computation. For example, the statement `d = d + 1` could be executed by adding 1 to or by subtracting -1 from `d`. The statement `d = d * 2` could be performed by adding `d` to `d` or by left shifting `d`. The alternatives can be encoded by 0 and 1. The selection of one of them is determined by the embedded bit of data. Note that the ability to implement the alternatives in an efficient way depends on type of processor, and is limited in case of RISC such as ARM. In ARM, doubling of a variable can be done by

`add rd, rd, #0` or `mov rd, rd, asl #1`,

where both variants are equivalent in byte length and execution time, whereas subtraction of immediate negative number is not possible.

The instruction scheduling method finds blocks with independent processor instructions. All permutations of independent instructions are ordered lexicographically and considered equivalent. The data then are embedded as the number of particular permutation. For example, in the block

```
mov r3, r4
add r0, r4, #1
```

two operations are independent and the possible alternative is

```
add r0, r4, #1
mov r3, r4
```

which may encode one bit of hidden information.

The code layout method is based on the observation that any machine code can be represented as a set of blocks with sequentially executed instructions separated by some kind of “go to” (jumping) instructions. These blocks of code can be placed in memory (in the content of executable file) in different order. Again, the data are embedded as the number of particular permutation of the code blocks.

The register allocation method deals with assigning different processor registers to a variable. If in the previous example variable `d` can be placed in a register and in its scope the compiler has some free registers, say `r3` and `r4`, an extra bit may be embedded by assigning either `d = r3` or `d = r4`. For variables that cannot be placed in registers, different memory locations may be used. Enumeration of all possible combinations of register or memory allocations enables one to hide data.

We also can take advantage of dynamic linking technology, which prevails in modern programming. Various permutations of entries in import/export tables and supporting structures can be used to hide data.

All the methods considered may be combined together to increase the embedding rate. The problem, however, exists that data embedding is not always robust. First of all, if embedding is made directly in a binary executable, the same program that embeds data can be used to erase or modify the watermark. That is why in our project we study embedding techniques that are applied to source codes. In this paper we present the current results, problems and directions of future work.

II. Implementing Embedding Methods for C/C++ Sources

We consider a stand-alone program that is used to embed a watermark in the source files of a C/C++ project and run just before the compiler. What methods out of those described in Introduction are appropriate in such a situation? First, it is clear that it is hard to influence the ways of construction import/export tables by the compiler. Second, it is mainly compiler's deed to select instructions for translating source codes. Other three methods seem plausible. Instruction scheduling is converted into statement ordering, we also, to some extent, can control register and memory allocations by the order of variable declarations. Changing the code layout is also possible but is the most complicated. Moreover, this method is protected by the patent [1]. In the first version of our watermarking program we confined ourselves to statement ordering and local variables ordering.

Consider an example of our implementation. Let there be given the following function:

```
int f (int x)
  {
  int a, b;
  a = x + 1;
  b = x - 1;
  return a + 2 * b;
  }
```

Computational part of the function is compiled with GCC into

```
ldr r3, [fp, #-16]
add r3, r3, #1
str r3, [fp, #-20]
ldr r3, [fp, #-16]
sub r3, r3, #1
str r3, [fp, #-24]
ldr r3, [fp, #-24]
mov r2, r3, asl #1
ldr r3, [fp, #-20]
add r3, r2, r3
mov r0, r3
```

where the memory allocation for variables corresponds to the order of their declaration:

```
x = [fp, #-16], a = [fp, #-20], b = [fp, #-24].
```

According to our approach we can embed two bits of watermark in this function by enumeration of variable declarations and the order of independent statements:

```
0 →  int a, b;   1 →  int b, a;

0 →  a = x + 1;  1 →  b = x - 1;
     b = x - 1;       a = x + 1;
```

So the initial function, in which variable declarations and statements are ordered lexicographically, corresponds to the watermark 00.

To embed the watermark 11 we rewrite the function in the form

```
int f (int x)
  {
  int b, a;
  b = x - 1;
  a = x + 1;
  return a + 2 * b;
  }
```

which is translated into

```
ldr r3, [fp, #-16]
sub r3, r3, #1
str r3, [fp, #-20]
ldr r3, [fp, #-16]
add r3, r3, #1
str r3, [fp, #-24]
ldr r3, [fp, #-20]
mov r2, r3, asl #1
ldr r3, [fp, #-24]
add r3, r2, r3
mov r0, r3
```

where

```
x = [fp, #-16], a = [fp, #-24], b = [fp, #-20].
```

Let us consider the robustness of the method. Suppose that we have two executable or object modules that contain our function with watermarks 00 and 11, respectively. By comparing two object files with `cmp -l` we find 4 differences (the first column shows the address, and the other two columns show different bytes of compared files):

79	203	103
91	103	203
97	30	24
105	24	30

(the same differences but at other addresses would be found in executables). Here 203 and 103 appear due to different codes of `add` and `sub` instructions, whereas 30 and 24 mean different memory allocations for variables `a` and `b`. Observe now that we cannot tamper this watermark, i.e. to change it to 01 or 10, by changing these known different bytes. Indeed, if we swap 203 and 103 in attempt to tamper the watermark, the function will be damaged because it will add 1 to `b` and subtract 1 from `a` which gives incorrect result. The same damage will ensue if we swap 30 and 24 (the result will be `b + 2*a`). We can conclude that it is not generally possible to tamper a watermark in binary executable file simply by changing the bytes where we find differences. These differences can only help to detect the location of a watermark. Of course it is possible only if software packages are distributed with different watermarks (fingerprints) and the malicious users create coalitions.

More serious attack may be implemented if we disassemble the executable. Though to disassemble a large real executable and analyze it is itself a difficult task, there are additional complications due to embedding in the source C/C++ level. Let us look at assembly codes for our function. First, we can see that there are no consecutive independent instructions that could be permuted to tamper the watermark. It is because, generally, independent C/C++ statements do not compile into single independent instructions but rather into independent groups of instructions. So we need an additional layer of disassembly for determining groups of instructions corresponding to higher level language statements.

As for variables allocations, it seems an easy task to mutually exchange the offsets (-20 and -24 in our example) to tamper some bits of a watermark. But sometimes (not in our example) an address of a variable is needed. In this case the offset will be used as an ordinary constant (without the sign in ARM) and more thorough analysis of the code may be required to tell this offset from some other constant.

Actually, to implement the attack we need a kind of decompilator rather than disassembler. But it is clear that the demand to make decompilation to tamper a watermark is the best level of security which can be afforded: if one has source codes, one can embed any watermark by using the same embedding program.

III. Embedding Performance

When we speak of embedding performance we are interested in the number of bits that can be embedded in a project files. According to the embedding methods used the performance is determined by two things: how many variables are declared in each function and how many groups of independent statements we can find. There are, however, practical restrictions that decrease the performance. The most noticeable among them are the following.

1) We permute only variables declared in the beginning of a block. But a usual practice of C++ programming is to declare variables as they are used. Although it may be possible to change memory allocations for those variables at the compiler level, we cannot change their position in the source code because it will violate the scope and visibility and may lead to errors.
2) We cannot permute seemingly independent statements if they contain any function call. This is because we cannot be sure that a function has no side effects. If it has, changing the position of the statement may lead to errors.
3) We cannot permute seemingly independent statements if they contain any indirection. It is hard to trace the content of pointers and references to guarantee independence of statements.

As experiments show, these limitations greatly decrease the performance.

To test the practical performance of embedding in real files, about 30 C/C++ Symbian projects with source codes available were picked up over the Internet. The performance results for some projects are shown in Table I. The rest of projects were completely unsuitable for

TABLE I
EMBEDDING PERFORMANCE FOR VARIOUS SYMBIAN PROJECTS

Project Name	The Number of C/C++ Files	The Size of C/C++ Files, bytes	The Length of Watermark, bits
HView_v1_13beta_source	10	58K	27
SymTorrent_1.30_source_2007	104	680K	40
DosBox	107	2066K	123
putty_src_1.5.1	227	3650K	1018
vim	395	3980K	207
OpenVideoHub	405	6853K	2205

the suggested method of watermarking, allowing to embed 0 to 10 bits only. The analysis of reasons of such low embedding performance show that, in addition to the restrictions indicated above, the widespread use of object-oriented technology does not go well with our method of embedding data. Almost any operation involves class component functions and it is hard to decide whether it is safe to change the order of operations due to possible side effects to the class parameters. Some new ideas are needed for object-oriented programs.

IV. Problems and Future Research

The main problem we did not address so far is code optimizations made by the compiler. Any modification of source code may suffer from optimization. The methods described in Section 2 work well only if optimization is switched off. Otherwise, two variants of the function considered, corresponding to watermarks 00 and 11, are compiled by GCC into

```
add r3, r0, #1
sub r0, r0, #1
add r3, r3, r0, asl #1
mov r0, r3
```

and

```
sub r3, r0, #1
add r0, r0, #1
add r0, r0, r3, asl #1
```

respectively. The difference here is only dictated by the order of statements, and does not depend on the order of variables declarations (GCC always allocates the first free register (`r3`) to the variable that is used first). A positive effect of optimization for watermarking is that reordering of the statements has led not only to the change in code layout but also to the change in code length. As a consequence, all next codes will be shifted in memory and due to cross-references between various parts of the program there will be many differences throughout the whole executable files effectively hiding the location of the watermarked function.

To alleviate the situation with optimization, source files that contain critical parts of application may be deleted from the list of files subject to watermarking. Then such critical files can be compiled with optimization.

A more challenging approach is to completely rewrite the compiler's code generator, converting it into a stego code generator. Actually, a conventional compiler chooses deterministically only one way of code generation among many equivalent (or almost equivalent) alternatives. A compiler with a stego code generator could select one of many alternatives based on the value of watermark to be embedded. Probably, it is the best way to implement watermarking.

References

[1] R.L. Davidson and N. Myhrvold, “Method and system for generating and auditing a signature for a computer program,” US Patent 5559884, Sept. 1996.

[2] C. Collberg and C. Thomborson, “Software watermarking: Models and dynamic embeddings,” in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 1999, pp. 311-324.

[3] J. Stern, G. Hachez, F. Koeune and J.J. Quisquater, “Robust object watermarking: Application to code,” in *Information Hiding, Lect. Notes in Comput. Sci.*, Berlin: Springer, vol 1768, 1999, pp.368-378.

[4] R. Venkatesan, V. Vazirani and S. Sinha, “A graph theoretic approach to software watermarking,” in *Information Hiding, Lect. Notes in Comput. Sci.*, Berlin: Springer, vol. 2137, 2001, pp. 157-168.

[5] C. Collberg, C. Thomborson and G. Townsend, “Dynamic graph-based software watermarking,” Technical report, Dept. of Computer Science, Univ. of Arizona, 2004.

[6] D. Curran, M.O. Cinneide, N. Hurley and G. Silvestre, “Dependency in software watermarking,” in *Information and Communication Technologies: from Theory to Applications*, 2004, pp. 569-570.

[7] T.R. Sahoo and C. Collberg, “Software watermarking in the frequency domain: Implementation, analysis, and attacks,” Technical report, Dept. of Computer Science, Univ. of Arizona, 2004.

[8] R. El-Khalil and A. Keromytis, “Hydan: hiding information in program binaries,” in *Int. Conference on Inform. and Communications Security*, Springer-Verlag Berlin Heidelberg, Lect. Notes in Comput. Sci., vol. 3269, 2004.

[9] B. Anckaert, B. De Sutter, D. Chanet and K. De Bosschere, “Steganography for executables and code transformation signatures,” in C. Park and S. Chee (Eds.): *ICISC 2004*, Springer-Verlag Berlin Heidelberg, Lect. Notes in Comput. Sci., vol. 3506, pp. 431-445, 2005.

[10] I. Nechta, B. Ryabko and A. Fionov “Stealthy steganographic methods for executable files,” in *XII Int. Symposium on Problems of Redundancy*, St.-Petersburg, May 26–30, pp. 191–195, 2009.