Exploiting Number Theory for Dynamic Software Watermarking

Abstract

Software watermarking is a protection technique which aims at combating software piracy, thus defending intellectual property, by embedding stegosignatures or watermarks into a program. In the presence of an illegal copy of the program, the owner can reliably claim her rights by extracting the watermark from the program itself. In this paper, we introduce a new software watermarking technique which can be used even with compiled programs. The proposed technique is dynamic: the watermark can be recovered during the execution of the program and it is related to a specific path of execution that is triggered by a given input. We illustrate the technique by means of a watermarking protocol for C/C++/C# source code which tackles many different challenges in the task of encoding the watermark, embedding it into a source code and extracting it from a compiled program. We show the validity of our approach by proving its robustness against common attacks.

Keywords: Watermarking, Software protection, Steganography, Rights protection, Software analysis, Obfuscation

1 Introduction

Piracy is a major issue in software industry. It is estimated that unlicensed software is 37% of all software installed on personal computers [**?**]. Frontier Economics estimated that the global value of pirated software in 2022 will be 42$ - 95$ billion [**?**]. It is clear that, in order to protect intellectual property and avoid all the risks related to pirated software (e.g. economic losses and security threats), protection techniques are crucial for software developers. Among these techniques obfuscation plays an important role: by obstructing code comprehension, the attackers are less likely to steal or tamper code they do not understand [**?**]. Obfuscation alone, however, is not enough to protect intellectual property and it is widely used alongside software watermarking, usually in order to "blend" the structure of the watermark and the program [**?**].

Software watermarking consists in the embedding of indelible stegosignatures

or watermarks (usually an identifier representing the owner) in a cover program. By means of this identifier, software developers are able to prove the ownership of a program, even if it has been pirated [**? ? ? ?**].
Following the taxonomy proposed by Collberg and Thomborson [**?**], we can distinguish between *static* and *dynamic* watermarking. In *static* watermarking [**? ?**], the watermark is inserted in the source code of the program, either in the data (e.g. in a constant, string, image) or in the code section (e.g. in the control structure). As the name suggests, a static watermark can be extracted from the program without the need of executing the latter. Static watermarks are easy to embed and extract and can be easily attacked by means of obfuscators and even code optimizers [**?**].
In *dynamic* watermarking, instead, watermarks are stored in the execution state and they are generated during program execution typically on a special input. In [**?**], three types of dynamic watermarking are listed:

- *Easter egg watermarking*: the watermark is hidden in a location of the source code that is executed under very unusual circumstances;
- *Dynamic data structure watermarking*: the watermark is stored in the program data and it will be computed only when the program is fed with a specific input. The watermark can be recovered by means of an *extractor*, which is able to observe the program data during any execution state;
- *Dynamic execution trace watermarking*: the watermark is stored in the program execution trace and its computation depends on the input given to the program, the operations performed or the addresses used.

Generally, dynamic watermarking techniques are more robust to sophisticated attacks that aim at destroying the watermark with respect to static watermarking techniques [**?**]. Indeed, static watermarking can be compromised by syntactic code transformations (like code obfuscation or compilation) that alter the code and therefore the inserted watermark. Destroying or compromising a dynamic watermark is more difficult, since the watermark is embedded in the semantic of the program and it is more difficult for an attacker to design a program transformation that tampers with the dynamic watermark while preserving the semantics of the original program.

In this paper we introduce a new dynamic watermarking technique that leverage some fundamental notions of number theory in order to watermark source code and compiled programs. In section 2 we introduce the notions of software watermarking and watermarking robustness and we define the threat model for our proposal. In section 3 we give a theoretical overview of the intuition behind our dynamic watermarking scheme, by discussing some concepts related to number theory and by providing simple examples of the math procedures involved in all the steps of our watermarking algorithms. In section 4 and in section 5 we describe in detail the technical challenges tackled, the solutions to these challenges and the improvements that we adopted in order to enhance the initial theoretical scheme by making it usable in realistic scenarios

under different circumstances (different source languages, compiled programs, obfuscation and code optimization). In section 6 we discuss the effectiveness of the proposed dynamic watermarking scheme by evaluating its robustness to the menaces discussed in the threat model 2.1 and in section 7 we assess the validity of the approach by means of an experimental evaluation on real, widely used, programs. Finally, in section 9, we draw our conclusions and we discuss some future work.

2 Software watermarking model

Following the definition given in [**?**], given a set of programs P and a set of stegosignatures Ψ, we can define the two main components of a watermarking system: the *embedder* and the *extractor*. The embedder is the software component responsible for adding the watermark in the software. It is $Emb : P \times \Psi \mapsto P$.
The watermark extractor is the tool which is able to obtain, given a key, the stegosignature from a watermarked program: $Extr : P \times Key \mapsto \Psi$.

The proposed technique hides the watermark in numerical variables in the source code. The right value of the watermark is computed during the execution of the program and it is related to a specific path of execution that is triggered by a given input.

2.1 Threat model

The concept of *robustness* in software watermarking is a relative measure, there is no watermarking scheme that can be defined secure against any attack. As discussed in [**?**], however, we can think of a robust watermarking scheme as a scheme that guarantees the watermark to be successfully extracted after undergoing moderate distortive attacks or a scheme where the destruction or the removal of the watermark implies a significant lack of functionality or performance of the watermarked program. A watermarking scheme can be considered robust even by considering the time and the cost of breaking it. If the time required to remove the protection guaranteed by the watermark is longer than the time when the information protected has a significant value then the scheme is considered robust. Similarly, if the cost of breaking the scheme exceeds the benefits of the cracking process, then the watermarking scheme is considered robust as well.

Our threat model is based on various types of attacks and code transformations that can affect the robustness of our watermarking scheme. First of all we must consider that the scheme proposed is used even with compiled programs (we discuss the embedding phase for C/C++/C# source codes, but the scheme can be easily extended for other compiled languages). Of course compilation can be considered as a transformation applied to a program and it is quite difficult to predict its results (due to various compilers involved,

optimizations and so on). The first of our goals is to deal with compiled code and guarantee that the watermark hidden in a program can be extracted even after the compilation phase. Alongside compilation, in a similar manner to code optimization, also code obfuscation can interfere with the effectiveness of a watermarking scheme. The obfuscation transformation may disrupt the watermark hidden in the program, hence a robust watermarking scheme must be able to resist to those transformations.

With respect to actual attacks, we can classify them in three main categories, according to [**?** **?**]:

- *Subtractive attacks.* The attacker tries to remove the watermark without affecting the value of the program containing it.
- *Distortion attacks.* The attacker, since probably is not able to remove the watermark, tries to modify it in order to avoid that the real owner can prove her ownership.
- *Additive attacks.* The attacker tries to add a new watermark or replace the existing one, in a way that should be difficult to prove which watermark is inserted first.

3 Signature encoding and embedding

In this section we are going to discuss our technique for dynamic software watermarking. Our approach involves the manipulation of the source code of the original program by embedding additional variables (or reusing existing ones, if possible), in order to hide the stegosignature. These variables are needed in order to make use of a fundamental theorem of number theory: the Chinese Remainder Theorem.

Theorem 1 (*Chinese Remainder Theorem*).
Let $n_1, n_2, \ldots, n_k$ be integers greater than 1 and let $N = \prod_{i=1}^{k} n_i$.
Given $a_1, a_2, \ldots, a_k$ integers such that $\forall i \in \{1, 2, \ldots, k\}$ $0 \leq a_i < n_i$, if the n_i are *pairwise coprime*, then there exist one and only x, $0 \leq x < N$, such that the remainder of the Euclidean division of x by n_i is a_i $\forall i \in \{1, 2, \ldots, k\}$.

In literature there exist some watermarking techniques that leverage the Chinese Remainder Theorem, most notably [**?**] and [**?**], however our technique differs greatly from them. More in detail, as opposed to [**?**] we do not need to introduce custom functions in the original source code. The code embedding the watermark is blended in the original source code and does not introduce a significative overhead in computation time, since it is based on the simple update of integer variables. Differently from [**?**], our algorithm is dynamic, thus benefits from the advantages of dynamic approaches discussed in 1.

3.1 Signature Encoding based on the Chinese Reminder Theorem

The first step of our watermarking scheme involves the creation of the stegosignature. It should be an integer that will represent the solution x of the system of congruences solved leveraging the Chinese Remainder Theorem. It is worth noticing that using an integer stegosignature is not a limitation, since it is straightforward to encode an arbitrary string as a numeric value. In order to set up the congruences we also need a set of positive integers $\{n_1, n_2, \ldots, n_k\}$ pairwise coprime. In order to facilitate this operation, we select them as prime numbers.

The choice of the prime numbers is, however, not casual, since the maximal number, representing the stegosignature, that we can encode is $\max \stackrel{\text{def}}{=} \Pi_{i=1}^{k} n_i$. Namely we are limited by the product of the prime numbers. This is not a big issue, since we can arbitrarily choose prime numbers of any size.

The encoding of the signature $S < \max$ is performed by means of the following steps:

- We compute the reminders b_i of the division of S by n_i:

$$\begin{aligned} S \mod n_1 &= b_1 \\ S \mod n_2 &= b_2 \\ &\cdots \\ S \mod n_k &= b_k \end{aligned}$$

We can observe that the values $b_1, b_2, \ldots, b_k$ can be used to encode the signature S, since, knowing their values, along with secret values $n_1, n_2, \ldots, n_k$ is enough in order to reconstruct S. This procedure is described in detail by the following steps:

- Compute the products R_i of the $k-1$ prime numbers where each R_i is computed as the product of all the prime numbers in N except n_i

$$\begin{aligned} R_1 &= n_2 * n_3 * \cdots * n_k \\ R_2 &= n_1 * n_3 * \cdots * n_k \\ &\cdots \\ R_k &= n_1 * n_2 * \cdots * n_{k-1} \end{aligned}$$

- Compute a possible value of x_i as solution of the equation $R_i x_i = 1 \mod n_i$,

$$\begin{aligned} x_1 \text{ solution of } \quad R_1 x_1 = 1 \mod n_1 \\ x_2 \text{ solution of } \quad R_2 x_2 = 1 \mod n_2 \\ \cdots \end{aligned}$$

$$x_k \text{ solution of } \quad R_k x_k = 1 \mod n_k$$

to compute x_i we recall that in the equation $ax = b \mod m$ we can replace a with the reminder of a when divided by m and we get an equivalent expression.

- The signature S can be computed as follows:

$$S = (\Sigma_{i=1}^{k} b_i * R_i * x_i) \mod \max$$

By briefly summarizing the signature is encoded by means of the following values:

- Secrets: $n_1, n_2, \ldots, n_k$ prime numbers
- Embedded in the code: $b_1, b_2, \ldots, b_n$

3.2 Signature Embedding and Extraction

In order to perform the embedding and the extraction of the watermark from the program, some private information is needed. More in detail, we need the following information:

- *The signature* S;
- *An Extraction key*: made of the pairs $\{(n_1, W_1), (n_2, W_2), \ldots, (n_k, W_k)\}$ where $\{n_1, n_2, \ldots, n_k\}$ are prime numbers and $\{W_1, W_2, \ldots, W_k\}$ are the program variables whose values (at certain program points and for certain inputs) encode the secret signature in the program
- *Set of inputs* I that trigger the computation of the secret signature, namely of the "right" values of variables $\{W_1, W_2, \ldots, W_k\}$

The *embedder* is a program that on INPUTS $P, S, (n_i, W_i), I$ returns the marked program P^w, and where:

- P is the original program
- S is the signature that we want to embed in P
- $\{n_1, n_2, \ldots, n_k\}$ are the prime numbers used to encode the signature S according to the Chinese Reminder Theorem (see Sec.3.1 for details)
- $\{W_1, W_2, \ldots, W_k\}$ are the variables that we insert in the program P and that contain the information needed to reconstruct the signature S when combined with $\{n_1, n_2, \ldots, n_k\}$
- I is the set of inputs that stimulate the computation of $\{W_1, W_2, \ldots, W_k\}$ that allows the correct reconstruction of signature S

The embedder algorithm consists in the following main phases, depicted in figure 1:

1. Encoding of the signature S in the values $\{b_1 \ldots b_k\}$ by leveraging the Chinese reminder theorem: given a set $\{n_1, n_2, \ldots, n_k\}$ of prime numbers the signature S can be encoded in terms of the values $\{b_1, b_2, \ldots, b_k\}$ such that $b_i = S \mod n_1$

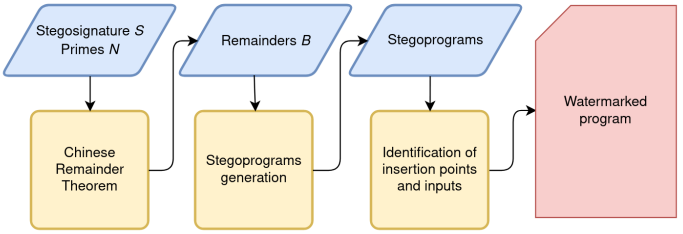

Fig. 1 *Embedding phase*: given a stegosignature S and the primes N we compute the remainders B which are then used to generate the stegoprograms used for embedding (in carefully chosen source code locations), the values needed in order to reconstruct the watermark, given a specific input

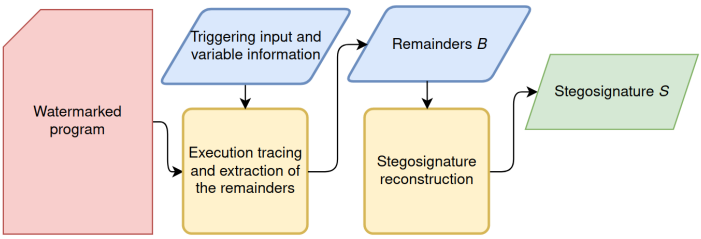

Fig. 2 *Extraction phase*: given a watermarked program, the input triggering the execution trace and the information about the variables to track, we extract the values of the remainders B which are then used to reconstruct the stegosignature.

2. Development of the *stegoprograms* $\{S_1, S_2, \ldots, S_k\}$: each stegoprogram S_i is a code fragment that when correctly stimulated computes the value b_i for variable W_i.
3. Identification of the suitable insertion points in P for the stegoprograms $\{S_1, S_2, \ldots, S_k\}$
4. Identification of the triggering inputs for the computation of the values $\{b_1, b_2, \ldots, b_k\}$ in the stego-programs $\{S_1, S_2, \ldots, S_k\}$
5. Embedding of $\{S_1, S_2, \ldots, S_k\}$ in P

The extractor (depicted in figure 2) is a program that takes as input the marked program P^w and the key given by:

- the triggering inputs for each stegomark S_i
- other possible keys for the extraction of the value b_i from the stegomark S_i (this depends on the choice of the encoding of b_i in the stegoprogram)
- the pairs $\{(n_1, W_1), (n_2, W_2), \ldots, (n_k, W_k)\}$ that associate the variables computing the values $\{b_1, b_2, \ldots, b_k\}$ to the prime numbers $\{n_1, n_2, \ldots, n_k\}$. This association is necessary in order to correctly reconstruct S according to the Chinese Reminder Theorem

The extractor algorithm consists in the following main phases:

- Monitoring of the execution of the watermarked program P^w on the triggering inputs and extraction of the values $\{b_1, b_2, \ldots, b_k\}$
- Reconstruction of the signature S from $\{b_1, b_2, \ldots, b_k\}$ and secret prime numbers $\{n_1, n_2, \ldots, n_k\}$

Example 1 Consider the prime numbers: $n_1 = 2, n_2 = 3, n_3 = 5$ and $n_4 = 7$; we have that $\max = 2 * 3 * 5 * 7 = 210$, so we can embed a signature $S = 199 < 210$. We start by computing the reminders b_i

$$199 \mod 2 = 1 \text{ thus } b_1 = 1 \; ; \; 199 \mod 3 = 1 \text{ thus } b_2 = 1$$
$$199 \mod 5 = 4 \text{ thus } b_3 = 4 \; ; \; 199 \mod 7 = 3 \text{ thus } b_4 = 3$$

We now compute the products R_i:

$$R_1 = \frac{\prod_{i=0}^{k} n_i}{n_1} = 105 \; ; \; R_2 = \frac{\prod_{i=0}^{k} n_i}{n_2} = 70;$$
$$R_3 = \frac{\prod_{i=0}^{k} n_i}{n_3} = 42 \; ; \; R_4 = \frac{\prod_{i=0}^{k} n_i}{n_4} = 30$$

We compute the solutions x_i of the equations $R_i x_i = 1 \; mod \; n_i$. A possible solution is

$$x_1 : 105x_1 = 1 \mod 2 \implies x_1 = 3$$
$$x_2 : 70x_2 = 1 \mod 3 \implies x_2 = 4$$
$$x_3 : 42x_3 = 1 \mod 5 \implies x_3 = 3$$
$$x_4 : 30x_4 = 1 \mod 7 \implies x_4 = 4$$

thus we have $x_1 = 3$, $x_2 = 4$, $x_3 = 3$ and $x_4 = 4$. And we can reconstruct S as follows:

$$\begin{aligned} S &= (b_1 * R_1 * x_1 + b_2 * R_2 * x_2 + b_3 * R_3 * x_3 + b_4 * R_4 * x_4) \mod 210 \\ &= (1 * 105 * 3 + 1 * 70 * 4 + 4 * 42 * 3 + 3 * 30 * 4) \mod 210 \\ &= (315 + 280 + 504 + 360) \mod 210 \\ &= 1459 \mod 210 = 199 \end{aligned}$$

4 Embedding Algorithm

We have to embed in the values $B = \{b_1, b_2, \ldots, b_n\}$ in the program. A possible way to embed these numbers is to define variables $W = \{W_1, W_2, \ldots, W_k\}$ such that for every $i \in [1, k]$ it holds that $W_i \mod n_i = b_i$. In this way the numbers $n_1, n_2, \ldots, n_k$ are secrets needed for the extractions of the values $b_1, b_2, \ldots, b_n$ from the marked program and for the reconstruction of the value of the signature S.

4.1 Identification of the Insertion Points

Since our watermarking scheme embeds the values $b_1, b_2, \ldots, b_n$ in the source code, in order to extract the mark it is essential to ensure the execution of the

code-fragments (stego-programs) computing the values $b_1, b_2, \ldots, b_n$. Hence, given a certain input, we need to determine which code will be executed in the original program, in order to extract the program points where we could insert the stegoprograms computing the b_i.

It is easy to notice that a new problem arises: *given an input, how can we identify what source code lines in the original source code will be executed?*

This is not a straightforward task to solve, since we are dealing with source code written in C, C++ or C#, so we have to deal with compiled programs whose relationship with the original source code is more complex than a simple translation. We recall that, of course, compilation is not a bidirectional process and given an executable is usually not possible to determine the correspondence between a compiled instruction and the source code line from which it is generated. So how do we determine the insertion points? The approach that we use for C/C++ is slightly different from the one used for C#. This is because while C and C++ programs are compiled into executable files (ELF, PE...), C# source code is converted, by the .NET framework, into an intermediate language, called *CIL* (Common Intermediate Language), that only at runtime is converted by the Common Language Runtime into machine code as typically happens with just-in-time compilers. In the following two subsections we will describe the different approaches that we leverage in order to identify the insertion points for our watermarking code. In both C/C++ and C#, we make use of debugging data formats so that we can have more information about the original source code, starting from a compiled program. It is worth noticing that the debugging information is needed only during the embedding phase, in order to identify the insertion points. The extraction algorithm does not need any debugging information, so the program can be compiled and distributed without debugging symbols.

4.1.1 C/C++

In order to produce debugging information we compile our source code with `gcc` using the `-g` option, which uses operating system's native format (stabs, COFF, XCOFF or DWARF). Once we have the debugging information in our executable we are ready to run it and trace all the executed instructions. We implemented two different approaches to get the executed trace at the source code level: the first one is based on Dynamic Binary Instrumentation, the second is based on debugging.

Dynamic Binary Instrumentation based approach

We leverage Intel Pin [**?**], a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures, to create a tracer *pintool*. The tool is able to trace the execution of a program and to log the addresses of the executed instructions that lie on the memory area of the program itself (thus is able to distinguish between imported functions from

external libraries and functions defined in the scope of the program). Then we translate the addresses into file names and line numbers. Given an address in the executable or an offset in a relocatable object, we are able, by using the debugging information, to figure out the source code line associated with it.

Debugging based approach

In this approach we use `GDB` to step over the lines of our executable. This task is performed in a fully automated manner by leveraging the opportunity to load extensions in `GDB`. In this way we are able to get the information about the source code line executed during the debugging process, and logging the information of interest in external files thanks to the `GDB` Python interface.

4.1.2 C#

We compile and execute C# programs with the Mono framework. In order to produce debugging information we use the `-debug` option with the `csc` compiler. This option, unlike `GDB`'s `-g`, does not add debugging information in the binary, but generates a new symbol file, called *PDB* which is used by the debugger to map a source code line to an executable location so that it can set a breakpoint. In our case, we use the `PDB` to get information about the executed source code lines. We use a debugging based approach and we leverage the Mono Soft Debugger Client (*SDB*). SDB is a cooperative debugger, part of the Mono VM. The word "cooperative" means that SDB is built into the Mono runtime, unlike regular debuggers (e.g. `GDB`, `LLDB` etc), which act as controllers for separate processes. The application communicate with the Mono runtime and request debugging operations to be performed on the target process. However, unfortunately `SDB` doesn't provide neither a complete set of APIs nor a simple way to extend its capabilities by loading extensions. To overcome this limitation and to fully automate the process of extracting the executed source code lines we wrote a controller script which interacts autonomously with `SDB` and is able to step over the lines executed and log them.

4.2 Generation of the code for encoding each b_i

As already discussed in section 3.1 we select a set of $N = \{n_1, n_2, \ldots, n_k\}$ prime numbers and a signature S, represented as a number s.t. $S < \Pi_{i=1}^{k} n_i$. Both the set N and the integer S are secrets and are given as input to our code generation algorithm. Following the Chinese Reminder Theorem, we compute the division between S and the values in the set N and we obtain a set of reminders b_i that we have to embed in the original source code.

Since the plain insertion of these values in the source code may raise suspicion we hide the reminders b_i in stegoprograms. The stegoprogram creation is based on the declaration and the initialization of a set of new variables $W = \{W_i, W_2, \ldots, W_k\}$ and on some code which performs computations on them. The goal is that the stegoprogram S_i for b_i is a program that at a specific point of the execution trace on a specific input ensures that the variable

W_i assumes the value b_i. For the sake of simplicity let's focus on the insertion of a single variable W_i, but of course the same considerations are applied to the other variables. We recall that, thanks to the techniques discussed in 4.1 we are able to know how many iterations will be performed for every cycle in the chosen execution trace. This goal is achieved thanks to the knowledge of the executed lines of code and their order of execution. Given this knowledge we are able to reconstruct the entire execution trace, including the information about conditional statements and loops. It is important to notice that we do not necessarily need loops with a constant amount of iterations. Even if we deal with a variable as upper limit for the cycle, we are able to know how many iterations are performed by analyzing the execution trace. Let us consider one of these cycles assuming that it performs I iterations in total. We choose two values: k and r with $k < I$ and r random value with $r * k <= b_i$. We initialize, before the cycle, the variable W_i as $b_i - (r * k)$. Inside the cycle we will update the variable W_i by increasing it of r. This means that after k iterations of the cycle, the value of W_i will be equal to its initial value $b_i - (r * k)$ plus $r * k$ which of course is equal to b_i.

Example

Let us consider the embedding of the signature in a simple C# function used to compute the n-*th* Catalan number:

```
using System;

class GFG {
    static uint catalanDP(uint n)
    {
        uint[] catalan = new uint[n + 2];
        catalan[0] = catalan[1] = 1;

        for (uint i = 2; i <= n; i++) {
            catalan[i] = 0;
            for (uint j = 0; j < i; j++)
                catalan[i]
                    += catalan[j] * catalan[i - j - 1];
            }
        return catalan[n];
    }
}
```

the signature S is supposed to be 25 and $N = \{11, 13\}$ therefore $B = \{3, 12\}$. If we compute the 100-th catalan number, the first `for` loop will be performed 98 times. We choose, for example, $k = 6$, $r_1 = -3$, $r_2 = 2$, hence our variables W will be initialized as:

$$W_1 = b_1 - r_1 * k = 3 - (-3) * 6 = -15$$

$$W_2 = b_2 - r_2 * k = 12 - 2 * 6 = 0$$

The initialization and update code for the variables W is then inserted in the original source code:

```
using System;

class GFG {
    static uint catalanDP(uint n)
    {
        uint[] catalan = new uint[n + 2];
        int w1 = -15, w2 = 0;
        catalan[0] = catalan[1] = 1;

        for (uint i = 2; i <= n; i++) {
            w1 += -3;
            catalan[i] = 0;
            w2 += 2;
            for (uint j = 0; j < i; j++)
                catalan[i]
                    += catalan[j] * catalan[i - j - 1];
            }
        return catalan[n];
    }
}
```

It is easy to notice that at the k-th iteration every w_i variable will assume the value of the corresponding b_i.

4.3 Further improvements

The name of the variables W should be chosen carefully in order to blend them in the original source code thus making it difficult to identify the added code. The user can choose the name of the introduced variables or specify a variable which is dead or unused.

There are a large numbers of ways to update the value of the W variables in the cycle. The techniques to use can be chosen in accordance to the context. By developing several different techniques and choosing randomly among them, the identification of the watermarking code will be more difficult.

It is important to notice that, in order to deceive the attackers, the variables W can be used also in the source code not executed in the chosen trace.

Since compilers may apply code optimization routines, such as dead code elimination, that is the removal of the code which does not affect the program results, the variables W, which are updated but do not affect any result, may be removed by the program. In order to avoid the elimination, we force the use of those variables by adding data dependencies and making use of opaque

predicates. In this way we guarantee that, even if compiler optimizations are in use, the variables used to embed the algorithm will resist to the compilation phase.

5 Extraction Algorithm

The extraction algorithm is pretty straightforward. By knowing the variables used and the value k we use the techniques discussed in 4.1 (binary instrumentation, automated debugging) in order to trace the value of the variables W, thus obtaining, after exactly k updates, the values of the b variables. Thanks to the Chinese Reminder Theorem, we can then reconstruct S, as discussed in 3.1. We underline that both the insertion and the extraction phases are completely automated, thus not requiring any human analysis except for the run of the program with the correct inputs.

5.1 Dealing with compiled programs

It is important to notice that, during compilation and especially when dealing with stripped binaries, some information such as the original name of the variables is lost. This could be a problem for our watermarking scheme, since our extraction algorithm obtains the values needed to reconstruct the watermark by observing the content of specific variables $W = \{W_1, W_2, \ldots, W_k\}$. In order to overcome this limitation, we need a way to identify the variables W even if their original names are lost during compilation. Our solution to this problem is to perform a preliminary parsing of the original source code in order to identify all the initial values of the variables in the code. We then choose, as random initial values for our W variables, random integers that are not used elsewhere as initial values for any variable, in accordance to the procedure described in 4.2. As an enhancement, we generate initial values that are as much as possible similar to the ones already used by the program. This is easily done by collecting the initialization values during the analysis of the trace of the program described in 4.1 and choosing values close to the existing ones. In this way, the information that we need to store and to provide as input to the extraction algorithm is not the set of variable *names* W, but the set of their initial *values* as this allows us to identify, during the execution of the program, what are the variables to observe in order to find the correct values needed to reconstruct the watermark.

6 Threat analysis

As discussed in the section 2.1, the notion of robustness, when associated to a watermarking system, is a relative measure and no watermarking scheme can be considered 100% secure. In this section we discuss the robustness of our watermarking scheme when it undergoes the threats introduced in 2.1.

Obfuscation and code optimization

In sections 4.3 and 5.1 we talked about several enhancements in our theoretical algorithm that allowed us to deal with different types of code transformation. Both obfuscation and code optimization may affect the name and the presence of the set of variables W used in order to embed the watermark in the program. However, the presence of the variables can be guaranteed by means of ad-hoc data dependencies and opaque predicates. The changes that may occur to the name of the variables is also a problem that we solved, since we can infer which are variables of the set W by simply looking at their initial value. We can even state that, in our case, obfuscation and code optimization could be considered an advantage and not a threat, since they make the reverse engineering process harder to an attacker.

Subtractive attacks

In order to perform an effective subtractive attack, the attacker needs to know the meaning of the variables used in the program and should be able to remove them without incurring in the risk of corrupting the correct functionality of the program. This operation is quite hard to perform because of the data dependencies used in order to ensure the robustness of the watermarking scheme against code optimization, and moreover, it relies on the perfect knowledge of the program by the attacker, a scenario that could be considered unrealistic, because if the attacker had complete knowledge of the code, then he could easily re-implement it, thus removing any code protection mechanism.

Distortion attacks

Distortion attacks are usually considered easier to perform because the attacker does not need to know all the pieces of information that are used by the watermarking scheme, but a portion of it could be sufficient to corrupt the embedded stegosignature. In order to perform a successful distortion attack against our scheme the attacker needs to know the variables involved (at least some of them) and the correct trace of execution (identified by a specific input) where the watermark is computed. We make the same considerations about the attacker's knowledge that we discussed while talking about subtractive attacks. In order to make distortion attacks harder, it could be even possible to introduce other dummy variables, acting like variables in the set W, so that an attacker trying to modify only a subset of the variables embedding the watermark should incur in a reduction of the possibilities of guessing the right ones.

Additive attacks

From the literature [**?**], we know that, unfortunately, no watermarking scheme can be considered immune to additive attacks. However, several solutions have been proposed in order to making easier to understand, in the presence of different watermarks, which one is the authentic one. Most of the solution proposed rely on a third party where the watermark is registered along with a

timestamp [**? ?**] so that the original watermark can be identified by its earlier timestamp. Although we could leverage similar solutions, it is important to highlight that additive attacks are extremely difficult to perform against our watermarking scheme. This is due to the fact that the process of inserting a watermark into a binary file relies on the access to the original source code, and during the embedding phase, the program must be compiled with the debugging information, which is subsequently eliminated when the program is distributed. An adversary, of course, lacks both the original source code and the debugging information necessary to carry out an effective additive attack. In order to succeed in such an attack, the attacker must not only directly modify the compiled binary to include the variables necessary to recover the watermark, but also ensure that the added code will be executed, which poses considerable challenges (as outlined in section 4.1). This task is exceedingly challenging, and as with subtractive attacks, requires the attacker to possess an unrealistic comprehensive understanding of the code and its execution scenarios. It is important to notice that our extraction algorithm is capable of reconstructing the watermark accurately even in cases of additive attacks, where the attacker attempts to add a new watermark. This is possible because our algorithm takes both the triggering input and the variable information as input. In situations where the attacker seeks to replace the existing watermark, instead, the same considerations outlined for subtractive attacks apply.

7 Evaluation

In order to assess the effectiveness of our approach we tested the embedding and the extraction phases and measured the overhead, in terms of size and execution time, of the watermarked programs with respect to the original ones. We used source code of real world, widely adopted, utilities taken from the GNU `binutils` and `coreutils`[1] packages as well as algorithms from the Rosetta Code project[2]. We chose programs with loops in the source code. We compiled the set of programs using the GNU `gcc`[3] compiler, using different levels of optimization (by changing the value of the `-O` flag). We tested the robustness to obfuscation too, by compiling the programs after obfuscating them by means of the Tigress C Obfuscator[4]. In all of our tests we were able to embed and extract the watermark correctly. The code samples obtained from Rosetta Code were subjected to testing also on Windows operating system. Our approach is effective in multiple operating systems, because the embedding phase relies on debugging information that can be added to both ELF and PE files. To perform this analysis on a Windows OS, we utilized Mingw-w64 [5], which enabled us to easily use `gcc` and `gdb`. Additionally, we also

[1] https://www.gnu.org/software/binutils, https://www.gnu.org/software/coreutils/
[2] https://rosettacode.org/wiki/Rosetta_Code
[3] https://gcc.gnu.org/
[4] https://tigress.wtf/
[5] https://www.mingw-w64.org/

the instrumentation-based approach can be applied on Windows, as the latter is supported by Intel PIN (both using LLVM clang-cl and MSVC). This further highlights the versatility of our approach, and underscores its potential usefulness for a wide range of applications. The particular nature of our approach, based on the simple update of integer variables, does not introduce a significant overhead. We observed, indeed, that the differences in terms of execution time among different runs of the original program were often bigger than the difference between the execution time of the original program w.r.t. the watermarked one. This means that our watermarking scheme does not introduce a significant overhead in terms of execution time. The size of the watermarked program is almost the same of the original one too, and the difference is way smaller than the one introduced using various optimization levels. The embedding and extraction phases, due to their nature, are of course slower, but the embedding phase is performed only once and the extraction is performed only in the rare case where the watermark should be verified. It is evident that in the presence of constantly updated software, it is unnecessary to repeat the embedding procedure each time the software is compiled. In fact, the embedding of the watermark only needs to be performed once, prior to the distribution of the program. Furthermore, since our approach focuses on a granular level of functions, if a software update does not impact the watermarked function or its execution, there is no requirement to repeat the entire embedding phase. In such cases, the program can be compiled directly with the necessary compilation options for distribution purposes.

8 Related Work

In recent years, there has been significant research activity in the field of software watermarking, with a focus on both static and dynamic approaches. While static watermarking is comparatively easier to implement, it has been shown to be inherently less resilient to common attacks than dynamic watermarking[**?** **?** **?**]. Many software watermarking algorithms rely on the specific structure of the binary, leveraging the location of the code or its organization in functions and basic blocks. Graph-based approaches [**?**], for instance, leverage graph theory problems like Graph Coloring [**?**] to insert watermarks by adding edges to a given variable allocation graph. Some researchers, such as Jiang et al. [**?**], have extended this approach through the use of public-key cryptography. However, these schemes are challenging because they rely on a NP-complete problem [**?**]. Moreover, graph-based approaches for software watermarking often suffer from the problem that the inserted code does not relate to the original software. In contrast, our proposed approach selects both the variables and their initialization values carefully to seamlessly blend with the original code, without introducing new branches or basic blocks that could be exploited to isolate the watermarking code. This allows for an effective watermarking scheme that is well-integrated with the original software and is resistant to attacks. Other watermarking techniques rely on reordering based

techniques, such as reordering of basic blocks [**?**], operands in mathematical equations [**?**], or function dependency-oriented sequencing [**?**]. Another interesting approach is SoftMark [**?**], which employs function relocation, where the order of functions implicitly encodes a hidden identifier. However, reordering-based approaches often suffer from limited data rate due to their dependence on given features, such as the number of functions or operands, which constrains the encoding bits. Our approach is not subject to such limitations as we do not depend on such components, allowing us to achieve a higher data rate for inserting our watermarking code. Another class of watermarking schemes is based on transformation techniques used for code obfuscation. For instance, XMark [**?**] applies control obfuscation based on Collatz conjecture to selected conditional constructs. This approach has been extended in [**?**] using Shamir's secret sharing scheme. Another widely used obfuscation technique in watermarking is opaque predicates. Arboit et al. [**?**] proposed two watermarking methods for Java programs by inserting fragments of a watermark encoded as constants into opaque predicates. Later, Myles and Collberg [**?**] assessed this approach through static and dynamic implementations within the SandMark framework. In [**?**], opaque predicate obfuscation is employed for injecting watermarks in smart contracts. However, subsequent research has offered efficient and effective techniques to identify invariant opaque predicates [**? ?**]. As a result, the use of opaque predicates has become a significant limitation since their nature as tautologies makes their removal safe. Abstract interpretation, a static analysis techniques mainly used for the verification of software, has been used for defining an abstract watermarking scheme and for formalizing a general theoretical framework for software watermarking [**? ? ?**]. Abstract watermarking aims to demonstrate that a program's abstract semantics satisfies an abstract specification, without concerning itself with irrelevant details about concrete semantics. The watermark can be hidden in a way that it can only be extracted by an abstract interpretation of the concrete semantics of the code [**?**]. There are other worth mentioning approaches. In [**?**] secret code is hidden by exploiting the variable-length instructions of the Intel x86-64 instruction. This approach has been later evaluated in [**?**]. In [**?**] the authors leverage Return Oriented Programming for steganography purposes. Our proposed watermarking scheme exhibits a multitude of advantages in comparison to many of the existing schemes in literature. Firstly, it is a dynamic scheme that has been shown to resist a variety of optimizations and obfuscating transformations in the evaluation section of our paper. Furthermore, the additional code introduced by our scheme integrates smoothly into the original source code without any alteration to the control flow graph or the introduction of additional basic blocks. Notably, the scheme does not rely on the use of opaque predicates, and the associated overhead in terms of program size and execution time is negligible. *

9 Conclusions and future work

In this paper, we propose a novel dynamic watermarking technique. The embedding of the the watermark is performed by the leverage of some number theory concepts and its extraction is related to specific paths of execution of the programs. The dynamic nature of the proposed scheme makes it more robust to several kind of attacks, as shown in our analysis. The implementation of the proposed scheme faced several intriguing technical challenges. The solutions proposed are easy to implement, suggesting that our watermarking scheme has a good practicability. As a future work we plan to extend the proposed scheme by searching different ways to embed the stegosignature without the need of debugging information. We are working also on finding ways to protect the variables involved in the watermark embedding and extraction, by means of data dependencies and obfuscation techniques.

Declarations

- Funding
  The authors did not receive support from any organization for the submitted work.
- Conflict of interest/Competing interests
  All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.
- Ethics approval
  - This material is the authors' own original work, which has not been previously published elsewhere in any form or language (partially or in full) and it is not an expansion of previous work.
  - The paper is not split up into several parts to increase the quantity of submissions.
  - The paper is not currently being considered for publication elsewhere, in any form or language.
  - The paper reflects the authors' own research and analysis in a truthful and complete manner. The results are honest and without fabrication, falsification or inappropriate data manipulation.
  - The paper properly credits the meaningful contributions of co-authors and co-researchers.
  - The results are appropriately placed in the context of prior and existing research.
  - All sources used are properly disclosed.
- Consent to participate
  Not applicable
- Consent for publication
  Not applicable

- Availability of data and materials
  The data and materials that support the findings of this study are available from the corresponding author, upon reasonable request.
- Code availability
  The code that support the findings of this study is available from the corresponding author, upon reasonable request.
- Authors' contributions
  All the authors contributed equally to the design and implementation of the research, to the analysis of the results and to the writing of the manuscript.