Robust Object Watermarking: Application to Code

Julien P. Stern[1,2], Gaël Hachez[1], François Koeune[1], and Jean-Jacques Quisquater[1]

[1] UCL Crypto Group,
Batiment Maxwell, Place du Levant, 3
B-1348 Louvain-la-Neuve, Belgique
{stern,hachez,fkoeune,jjq}@dice.ucl.ac.be

[2] Laboratoire de Recherche en Informatique,
Université de Paris-Sud,
Batiment 490, F-91405 Orsay Cedex, France
stern@lri.fr

Abstract. In this paper, we focus on a step of the watermarking process whose importance has been disregarded so far. In this perspective, we introduce the *vector extraction paradigm* which is the transformation between digital data and an abstract vector representation of these data. As an application, we propose a new, robust technique in order to insert watermarks in executable code.

1 Introduction

The tremendous growth of the Internet probably marks the beginning of a new era in communications. Any digital document can be duplicated and distributed in a matter of minutes by means of a simple mouse click. Because of the very low cost of this kind of distribution and the availability of high quality printing and audio devices, digital distribution seems to be an attractive direction for the near future. However, no valuable document is currently being sold that way, as the cost of duplication and the quality of an illegally redistributed document is just the same as the original.

As it is impossible to prevent the copy of a digital document, a new direction is being explored, which consists in dissuading an user to redistribute his legitimate copy, by permanently embedding some piece of information in the data so as to enable tracing. This leads to a very large amount of research in the domain of watermarking and techniques to robustly embed information in many different types of media have been pursued.

While many solutions exist when the data to be marked is an image, a video or an audio record, several types of data still seem to resist unobtrusive and robust information embedding. In particular, text documents, source code and machine code create difficulties, notably because their structure allows a very low level of modifications.

In this paper, we show that, as far as machine code is concerned, we can take advantage of the difficulty to modify data to create a very robust way of embedding information. We also argue that, paradoxically, a lower level of tolerance versus modifications yields more robust watermarking schemes. In other words, we believe that the possibility of performing unobtrusive modifications are helping an attacker to defeat a marking scheme much more than it helps a legitimate user to mark a data.

We also introduce a new paradigm, the vector extraction paradigm (VEP), which is an important step between the raw digital data and the object that will actually be marked. The VEP allows to apply generic schemes to any type of data and to define classes of attacks against which a scheme will resist against.

Our technique differs from previous approaches by the representation we take of machine code. We do not consider it as a linear succession of instructions, nor do we use its execution flow, but rather we view the code as a whole statistical object. More precisely, the entity that we are actually going to mark is the function representing the frequencies of groups of consecutive assembly instructions in the code. With this approach, we strongly limit the range of modifications that can be applied to the marked object.

Outline: In section 2, we discuss related work. Section 3 introduces the *vector extraction paradigm* and section 4 presents our solution. Finally, implementation issues are discussed in section 5.

2 Related Work

There has been a very large amount of work in the domain of watermarking. We can however distinguish two main models.

The first one represents the data to be marked as a vector and the mark as a word on an alphabet. Each "letter" of this word is embedded in each component of the data vector. In the simplest case, the data and the mark are binary strings, and each bit of the data is flipped depending on the value of the corresponding bit in the mark. This model allows a nice modeling of the watermarking process and leads to the most sophisticated watermarking protocols [1, 15, 13, 12, 14]. Its main drawback, however, is that it is mostly a theoretical model. In order to be secure in practice, the size of the marks often would have to be larger than the data itself!

The second model takes its roots in the spread spectrum technique, and was first introduced in [4]. This model, which is harder to manipulate as a building block for complicated protocols, allows for practical and robust insertion of marks. The security of such an insertion techniques has been proven in [9] under an assumption restricting the range of alterations performed on the marked data. Interestingly, [7] recently showed an attack which matches the security bound obtained in [9], namely, for a data represented as a vector of size n, $O(\sqrt{n/\log n})$ different marked copies are sufficient to delete the mark, but $\Omega(\sqrt{n/\log n})$ are not.

On the side of these generic techniques, a lot of work has been performed in order to embed marks in *specific* data. We refer the reader to [5] for a summary of such techniques when the data is an image. A more recent and less data specific survey can be found in [11].

There have been very few published results for software marking. [8] simply writes a specific mark in a specific place in the file, [6] reorders blocks of instructions to form a signature. In [2] a good summary of existing techniques is given, and a method to mark code using the topology of dynamic heap data structure is presented.

While these techniques take advantage of the structure of source or executable code, they are, in spirit, following the first model above. Hence, except possibly in the case of tremendously large data, we believe that a sufficient level of security will not be obtained.

The main difference of our approach is that we will extract a representation of machine code which allows the use of spread spectrum techniques, thus yielding more robust schemes.

3 The Vector Extraction Paradigm

A typical application of spread spectrum watermarking techniques (such as [4]) represents the data to be marked as a vector and modifies each of the vector components by a very small random (but known) amount, the *mark*. The presence of this mark is then tested by correlation. The security of such a technique relies on the assumption that in the space of the vectors which represent the data, a large degradation of the usability of the data implies a large distance between the corresponding vectors in the vector space, and conversely.

However, we believe there is one step missing: a piece of data is *not* a vector. A piece of data can be represented by many different vectors, in many different spaces. So, before actually applying marking techniques, there is a need to extract a vector from the data. This extraction step, which is implicit in most currently published works, is very important, as it will define which types of attacks the scheme will resist. We now present a robustness framework for data marking:

1. Start with a piece of data and provide a (possibly non-invertible) transformation which gives a vector from the data. Define a distance d on the vector space (We call this first step the *extraction step*);
2. List possible attacks that can modify the data without altering its usability;
3. Show that, after these attacks, the distance of the modified vector to the original one is small.

If the above can be shown, then we can apply techniques such as [4] or any other spread-spectrum techniques and apply the security framework of [9, 7]. Of course, all published schemes are already using (sometimes implicitly) a vector extraction step. What we want to underline here is that this step is *fundamental*: choosing one VEP or another will result in a scheme being robust against different classes of attacks.

If we list the classical requirements on watermarks, we can separate those that will depend on the VEP from those that will only depend on the spread-spectrum technique in use. This distinction between VEP and vector marking allows us to benefit from the large attention the second problem has already received. The imperceptibility and the robustness of the watermark will depend on the VEP (as stated in the previous paragraph). But other properties such as watermark recovery with or without the original data, watermark extraction or verification of presence for a given watermark will depend on the chosen spread-spectrum technique. For example, we know from spread spectrum theory that insertion – by a malicious adversary or not – of a second watermark will not induce a sufficient modification of the first one, which will with very high probability remain visible; as this property is inherent to spread spectrum technique and does not depend on our VEP, we can rely on this theory.

Note that step 2 is very subjective because "usability" of data usually cannot be properly defined. However, consider the following setting: the data to be marked is an image, the extraction step yields a vector whose coefficients are the DCT coefficients of the image, the distance chosen is the Euclidean one. If we perform on the image a flip around its vertical central axis, we will obtain a new image, whose quality is comparable to the original one. Furthermore, this operation obviously does not remove the mark. However, the detection algorithm will fail to find the mark, because the distance between the new and the old vector will be very large.[1]

This means that marking the DCT coefficients of an image using spread spectrum techniques will not resist against this specific attack. On the other hand, it will resist against JPEG compression very well, because even a high compression will result in a small variation of the distance in the vector space.

We refer the reader to [10] for many attacks on different types of data which all work by finding a transformation which heavily modifies the structure of the extracted vector.

Now, consider the case of executable code. The margin to modify such data is low, because the *correctness* of the output of the code has to be preserved. Hence, an (hopefully) exhaustive list of attacks can be given. If we can exhibit a vector extraction step, a distance on the corresponding vector space and show that all the attacks in the list result in a small variation in distance, we will have a robust method of marking.

It is interesting to note that images were the first target of choice for watermarking, certainly because of the ease to unobtrusively modify an image. We argue that this ease turns out to be a disadvantage. As a matter of fact, it will be extremely difficult to find a vector representation and a distance that will resist against the very broad range of modifications available. Hence, we conjecture that robust image marking will be almost infeasible, at least using spread spectrum techniques.

---

[1] Of course, if we flip the image again before testing, the watermark will be detected, but this requires this specific operation to be done.

4 Our Solution

4.1 Specificity of Codemarking

Imperceptibility of the watermark is one important requirement, but its meaning will vary with the data to be watermarked. It will for example be very different for an image (invisibility) than for an audio record (inaudibleness).

As far as code is concerned, imperceptibility means the correctness of the program must be preserved. By correctness, we mean that the observable behaviour of the code is not modified. More formally, we must have this property (based on the definition of code obfuscation in [3]).

Let C be a code and and C' the watermarked C code. C and C' must have the same observable behaviour. That is, with a given input:

1. If C fails to terminate or terminate with an error, then C' must fails to terminate or terminate with an error.
2. Otherwise, C' must terminate and produce exactly the same output as C.

Note that [2] also proposed a definition for the correctness preservation of the code. This definition is based on semantic preservation but this is more restrictive than our definition.

4.2 Vector Extraction Step

The main difference of our scheme compared to previous work is in the vector extraction step that we apply to the code prior to marking. Our choice was motivated by the fact that the modifications which can be applied to a code are mainly local ones and functions reordering. Hence, we looked for a vector extraction step that would not be structurally changed by blocks reordering and that would be only slightly modified by local modifications.

Vector Extraction Let n be a security parameter. We define a set $\mathcal{S}$ of n ordered groups of machine language instructions. For each group of instructions i in $\mathcal{S}$, we compute the frequency c_i of this group in the code (e.g. the number of occurrences of this group divided by the size of the code), and we form the vector $c = (c_1, \ldots, c_n)$. This vector, that we call the *extracted vector*, is the entity we are going to mark. The distance we define on the vector space is the Euclidean norm.

4.3 The Scheme

For clarity of discussion, we briefly detail the rest of the algorithm, which is a simple application of [4].

Initialisation We set a detection threshold δ $(0 < \delta < 1)$.

Watermark Insertion

1. We apply the vector extraction step to obtain a vector c (of length n).
2. We choose a n coordinates vector $w = (w_1, \ldots, w_n)$ whose coefficients are randomly distributed following a normal law with standard deviation α.
3. We modify the code in such a way that the new extracted vector $\tilde{c}$ is $c + w$. (This step will be detailed in section 4.4).

Watermark Testing

1. We apply the extraction step to obtain a vector d.
2. We compute a similarity measure Q between $d - \tilde{c}$ and w, e.g.:

$$Q = \sum_{i=1}^{n} \frac{(d_i - \tilde{c_i})w_i}{\sqrt{(d_i - \tilde{c_i})^2}} \quad .$$

3. If Q is higher than δ then the algorithm outputs "marked" else it outputs "unmarked".

4.4 Details

We now consider precisely the problem of modifying the frequencies of groups of instructions in the code to obtain a given frequency.

Recall that we are starting with an extracted vector $c = (c_1, \ldots, c_n)$, with the c_i representing the number of occurrences of the group of instruction i in the code. What we want is to modify the code in such a way that the new extracted vector is $\tilde{c} = c + w$ for some vector w.

In order to do so, we will randomly perform a small modification to the code, (while preserving its correctness) and compute the new frequency vector. If the new frequency vector is closer to $\tilde{c}$ than the previous one, we accept the modification and start again, else we simply refuse the modification. Our technique is actually a very simple case of probabilistic stabilising algorithms.

The modifications that we will apply, depend on the specificity of the assembly language we are working with. A study of these modifications in the case of the `x86` language is given in section 5.

4.5 Analysis of Attacks

We now informally analyze the attacks which can be applied to a program. Because of the inflexibility of machine code, we can summarise most (if not all) the attacks that can be applied, and thus argue that our solution is really robust.

We believe that there are five different attacks that can be mounted which could modify the frequencies we are working with:

Local modifications This attack modifies the code in a similar way as we did in order to insert the mark.

Code reordering This attack modifies the code by reordering large independent pieces of codes. This attack defeats many schemes which uses the structure of the code. In our case however, it will only slightly modify the frequencies (of groups larger than one instruction), and will be considered as a local modification.

Code addition The attacker can add a (possibly) large piece of code to the initial program. This piece of code should either not be executed or do nothing (to preserve the functionality of the initial program).

Code decompiling The code can be decompiled and recompiled again.

Code compression The attacker can also modify the code by using code compression tools (e.g. ASPack), which decrease code size by compressing code and adding a decompression routine at the beginning.

The "code decompiling" attack is the most serious threat. If a code can effectively be totally decompiled and recompiled, it is likely that the new frequency vector can be very far from the original one (and thus that the mark can be removed). However, code decompiling is a very difficult, time consuming operation, especially for large programs, and it should be stressed that if only minor parts of the code are decompiled and recompiled, the frequencies will not vary a lot. Hence, we consider that the effort which would have to be invested in such an attack is too important and that it is certainly a better choice for an attacker to rewrite the code from scratch.

The "code addition" attack is also an attack which could result in a large variation of the extracted vector. But once again, in practice, the size of the additional code will have to be very large in order to influence the frequencies of the instructions, and even larger (that is, *much* larger than the program itself) in order to make sure this frequency change actually obfuscates the watermark. This is due to the fact that the attacker does not know *which* deviation α was added as watermark: random frequency modifications will therefore most probably reduce the signal at some places, but also amplify it at others, leaving the mark present. To make watermark disappear, the attacker has therefore to add a very big amount of code, so that frequencies of the marked part become negligible. Thus, we consider that this attack degrades the quality of a program in a noticeable way. Additionally, this attack will be relatively easy to detect as we will very likely have a large part of the code with very unnatural frequencies.

Code compression will certainly remove the watermark. We however insist on the fact that the modified object will no more be valid code, and that the only way to use this code is to uncompress and execute it. It is therefore possible to detect compression and undo it (as the compression algorithm must be coded somehow in the beginning of the file). Once uncompressed, the watermark will reappear unmodified. These two properties together make this situation somewhat different from "classical" watermark removal attacks, in which such a detection-recovery property does not exist. Note that this resistance is more general than the "image flip" example described in section 2: in this case, the watermark could be restored if detected, but the removal was not visible (it is generally not

possible to detect an image was flipped) and the modified object could perfectly be used as is. We therefore conclude our model resists this attack.

Finally, the "local modifications" attack are not a threat to our scheme. Intuitively, the spread spectrum techniques have this very nice property that if an attacker cannot perform "stronger" modifications than the original one, the mark will be recovered with very high probability (and we will of course try to insert marks with the largest possible norm). We refer the reader to the analysis of [9] for further details.

5 Implementation

Our first step was to find a set of groups of instructions which occur very frequently in an assembly program. The `x86` assembly language was our target for this first implementation. However, many other languages are interesting targets, for example the `JAVA` bytecode.

We started sampling on about one hundred various programs. It turned out that there were about 1000 different usable groups of instructions of size lower or equal than four.

Figure 1 gives a good intuition of what we are willing to mark. They represent the number of occurrences of all the possible combinations of up to 4 instructions, with those occurring less than 50 times removed. While it is discrete, this representation shows very well how a spread spectrum approach will work.

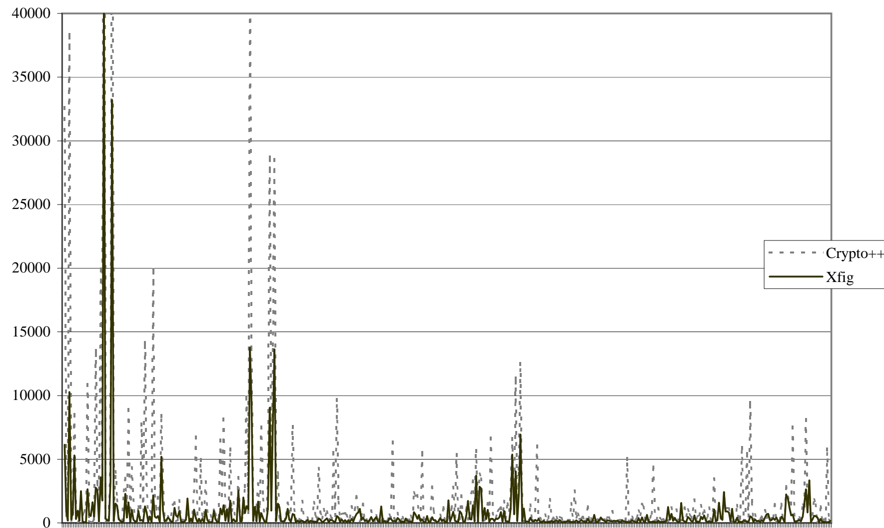

Fig. 1. Experimental distribution of the frequencies of different groups of instructions.

The second step consisted in building a "codebook" of equivalent groups of instructions. We now discuss the way to build this codebook, and give some examples.

The first remark is that every operation which can modify the frequency of our patterns and that does not modify the behaviour of the program is valid. Hence, most of the techniques used in previous low-level watermarking schemes can be used in our setting. However, we need to keep a minimal amount of control of the modifications, they must either be predictable or modify a small number of frequencies (so that we can test the result of the modification and accept it if it goes in the right direction). The simplest example would be to reorder blocks of instructions, which has already been studied in other marking schemes.

We can distinguish two kinds of code modifications:

local: This includes modifications restricted to a specific instruction block. (An instruction block is a block of instruction beginning after a jump and ending by a jump (of any kind) or the target of a jump).

widespread: This includes modifications performed on several blocks at the same time and implies the need to modify jumps and target of jumps.

Local Modifications The simplest local modification consists in modifying the order of two consecutive instructions which do not influence one another. We can, for example, swap the following pairs: `(mov/push)`, `(mov,pop)`, `(mov/xor)`, `(mov/and)`, `(mov,sub)`, `(not,xor)`, etc. Of course, we have to be careful. Because these modifications are data dependent, we have to verify that the registers modified in one instruction are not used in the other one. Also, flag modification must be taken in account. So, we decided, in our first attempt, to only use pairs with one instruction which does not modify any flag `(mov, push, pop, lea, not, in, out, rep, set, xchg, ...)`.

A slightly more sophisticated modification consists in swapping small *groups* of instructions instead of single instructions. An example is:

```
mov eax,ebx
sub eax,edx
push eax
mov eax,ebx
add eax,ecx
push eax
xor eax,eax
```

which becomes

```
mov eax,ebx
add eax,ecx
push eax
mov eax,ebx
sub eax,edx
push eax
xor eax,eax
```

A even more sophisticated modification consists in replacing a group of instructions by an equivalent one. For example, we could replace:

```
mov eax,ebx
mov ebx,ecx
```

by

```
xch eax,ebx
mov ebx,ecx
```

Similarly,

```
ror eax,19
```

can be replaced by

```
rol eax,13
```

provided we make sure the overflow and carry flags will not be tested further in the code.

Widespread Modifications As already pointed out, the probably most natural widespread modification is the reordering of different blocks [6].

Another possibility is to modify the jump sequences: a typical sequence when a test is performed is:

```
cmp eax,ebx
jne _Label_A
jmp _Label_B
```

This can be replaced by:

```
cmp eax,ebx
je  _Label_B
jmp _Label_A
```

It should be remembered that any modification to the code, which preserves its correctness, and which modifies a small number of frequencies can be applied to enhance the efficiency of our scheme. Future research will include: increasing the number of modifications we allow ourself to perform on the code, as well as studying the possible modifications for other languages.

6 Conclusion

We introduced the vector extraction paradigm and proposed a concrete application of this paradigm to machine code. We argued that our new method is robust for marking machine code and showed, contrary to common beliefs, that is it easier to robustly mark data which stand minor modifications (such as code, where *correctness* has to be preserved) than data which stand large modifications (such as images, or source code).

References

1. D. Boneh and J. Shaw. Collusion-secure fingerprinting for digital data. In D. Coppersmith, editor, *Proc. CRYPTO 95*, number 963 in Lecture Notes in Computer Science, pages 452–465. Springer-Verlag, 1995.
2. C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL99)*, 1999.
3. C. Collberg, C.Thomborson, and D. Low. Breaking abstraction and unstructuring data structures. In *IEEE International Conference on Computer Languages (ICCL '98)*, 1998.
4. Ingemar J. Cox, Joe Kilian, Tom Leighton, and Talal Shamoon. A secure, robust watermark for multimedia. In Ross Anderson, editor, *Workshop on Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
5. Ingemar J. Cox and Matt L. Miller. A review of watermarking anf the importance of perceptual modeling. In *Proc. of Electronic Imagining '97*, 1997.
6. R. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program. U.S. Patent No. 5,559,884, 1996.
7. Funda Ergun, Joe Kilian, and Ravi Kumar. A note on the limits of collusion resistant watermarks. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 140–149. Springer-Verlag, 1999.
8. K. Holmes. Computer software protection. U.S. Patent No. 5,531,021, 1994.
9. J. Kilian, T. Leighton, L. R. Matheson, T. G. Shamoon, R. E. Tarjan, and F. Zane. Resistance of digital watermarks to collusive attacks. Technical Report TR-58598, Departement of computer science, Princeton University, 1998.
10. Fabien A. P. Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Attacks on copyright marking systems. In David Aucsmith, editor, *Second Workshop on Information Hiding*, number 1525 in Lecture Notes in Computer Science, pages 218–238. Springer-Verlag, 1998.
11. Fabien A. P. Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Information hiding. a survey. In *Proceedings of the IEEE, special issue on protection of multimedia content*, 1999. To appear.
12. B. Pfitzmann and M. Waidner. Annonymous fingerprinting. In Walter Fumy, editor, *Advances in Cryptology—EUROCRYPT 97*, number 1233 in Lecture Notes in Computer Science, pages 88–102. Springer-Verlag, 1997.
13. B. Pfitzmann and M. Waidner. Asymmetric fingerprinting for larger collusions. In *4th ACM Conference on Computer and Communications Security*, pages 151–160, 1997.
14. Birgit Pfitzmann and Ahmad-Reza Sadeghi. Coin-based anonymous fingerprinting. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 1999.
15. Birgit Pfitzmann and Matthias Schunter. Asymmetric fingerprinting (extended abstract). In Ueli Maurer, editor, *Advances in Cryptology—EUROCRYPT 96*, number 1070 in Lecture Notes in Computer Science, pages 84–95. Springer-Verlag, 1996.