Cryptographic Program Watermarking

Julien Braine
2015 internship with Ron Steinfeld

June 1, 2015 - August 21, 2015

Abstract

Authorship protection, for images or programs for example, is becoming more and more necessary in the digital world we live in. Providing authorship protection consists in hiding a mark in the product that should be hard to remove and yet sufficient to prove authorship.

In practice, the methods to watermark programs use non-cryptographic heuristics to change the implementation [IN10]. However, the work of Barak et al. [BGI+01] showed that assuming a cryptographic technique called indistinguishability obfuscation, such methods are insecure.

Recently, with a proposed construction for indistinguishability obfuscation [GGH+13], cryptographic approaches that slightly alter program functionality have been studied to watermark programs [CHV15, NW15]. With this technique, we can watermark a set of cryptographic functions called puncturable pseudo random functions [CHV15, NW15].

In this internship report, I show that we can only watermark cryptographic functions. Intuitively, this is because non cryptographic functions follow patterns that allow slight program functionality changes to be detected and removed. This result questions the usefulness of watermarking as non cryptographic functions can not be watermarked and we have not yet found an application for cryptographic function watermarking that can not be done at least as efficiently by other means.

Contents

1 Introduction

The digital world around us has made copying, modifying and distributing content very easy. Under such circumstances, authorship protection is necessary as a potential thief can just copy the author's content and redistribute it as theirs. For most content such as software, images, music and films, authorship protection consists in hiding a mark in the product that should be hard to remove and yet sufficient to prove authorship.

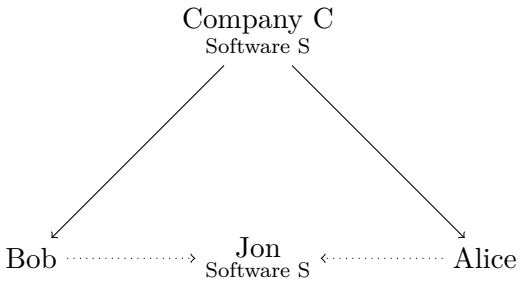


Figure 1: Goal of Watermarking

To illustrate what we want to achieve, imagine the scenario in figure 1. A company C sells a software S to Bob and Alice. One day C suspects Jon, who has never bought S from C to have a copy of S. The goal of authorship protection is to prove that Jon indeed has a copy of S and that he did not make that software himself. Additionally, C might want to know who of Bob and Alice gave S to Jon.

There are three basic requirements for watermarking. The first is that a marked object should be similar to the initial object. The second is that there should be no way of removing the mark without destroying the initial object. The last is that the presence of a mark must be deliberate so that its presence can prove ownership.

Figure 2: Image watermarking example [PMA11]

Work on watermarking products such as software, images, films and music for example is very common [PMA11, IN10], but usually use heuristics and do not have a security framework to work in. Furthermore, in 2001, Barak et al. [BGI+01] proposed a definition of program obfuscation called indistinguishability obfuscation (IO) that would provide a general attack on such methods for program watermarking by removing implementation details, while suggesting a cryptographic approach to watermarking. The recent discovery of a potential construction for IO [GGH+13] has lead to very recent work on cryptographic program watermarking [CHV15, NW15] which has given a construction to watermark a set of cryptographic functions called punctured pseudo random functions (PPRF), in a cryptographic security framework.

However, although watermarking cryptographic functions has applications and might lead to ideas for general program watermarking, it does not fulfill the original objective to protect software authorship as a software's primary function is rarely cryptographic. In this internship report, I studied the current methods for cryptographic program watermarking on cryptographic functions and its applications, and then focused on extending the scheme to non cryptographic functions.

2 Preliminaries

2.1 Notations

Program notations: We will call a program both Turing machines and circuits. To say that a program P has access to an oracle P', we will write $P^{P'}$. In this report, A is always

probabilistic polynomial time Turing machine that acts as an adversary (or distinguisher) and C will always be a circuit. We will call $I = \{0,1\}^i$ the input set of C and for simplicity, we will assume that the output set O is always $\{0,1\}$ (but all results work for any output set). We will write $C \simeq_\epsilon C'$ if circuits C and C' differ on less than $\epsilon(i)$ fraction of inputs and $C \equiv C'$ if C and C' agree on all inputs. Unlike in complexity theory, the execution time of an algorithm will not be compared with the length of its input, it will always be compared to λ, the security parameter. Furthermore, we will be assuming that all circuits discussed in this article have polynomial size with respect to λ and that i, the size of the input is of the same order than λ.

Function and distribution notations: A function $f : \lambda \to f(\lambda) \in [0,1]$ is negligible if and only if $\exists \alpha > 0, \exists N, \forall \lambda > N, f(\lambda) \leq 2^{-\alpha\lambda}$. We will write $f(\lambda) \simeq_\lambda 0$ to say that $f : \lambda \to f(\lambda)$ is negligible. We will write $f(\lambda) \simeq_\lambda 1$ to say that $1 - f(\lambda) \simeq_\lambda 0$. We will write $f(\lambda) >_\lambda 0$ if $\exists p$ polynomial such that $f(\lambda) > \frac{1}{p(\lambda)}$ and $f(\lambda) <_\lambda 1$ if $1 - f(\lambda) > 0$. We say that two distributions D_1 and D_2 are indistinguishable if $\forall A, |Pr(A(D_1) = 1) - Pr(A(D_2) = 1) \simeq_\lambda 0$. We will call $U(S)$ the uniform distribution for set S.

General notations: For two strings s and s', $s||s'$ means the concatenation of s and s'.

2.2 Definition of IO [BGI+01]

A polynomial time Turing machine (PPT) iO is an indistinguishability obfuscator for a set $\mathbb{C}$ if and only if

1. iO keeps for functionality: $\forall C \in \mathbb{C}, iO(C) \equiv C$.
2. iO hides the implementation: $\forall C \in \mathbb{C}, C' \in \mathbb{C}, (|C| = |C'| \text{ and } C \equiv C') \Rightarrow iO(C)$ and $iO(C')$ indistinguishable

Consequence: Intuitively, this means that iO hides implementation details as once iO is applied, we can not distinguish two different implementations of the same function. One way of seeing obfuscated programs, although not entirely correct, is as a mathematical function. This means we can almost see a program as the string of its concatenated outputs and thus creates a strong link with image watermarking.

Construction: A way to create an inefficient indistinguishability obfuscator is to enumerate all circuits and take the first circuit with same functionality. A complex efficient candidate construction using puzzles was proposed [GGH+13], but this will not be the focus of this report.

Remark: We will not be checking the sizes of circuits before applying iO as we can always use padding to satisfy this constraint. Also, we will be assuming that the set $\mathbb{C}$ on which iO works always contains all our circuits.

2.3 Definition of watermarking [CHV15, NW15]

A watermarking scheme $W = (Setup, Mark, Verify)$, where $Setup$ initializes the scheme, $Mark$ watermarks a circuit and $Verify$ checks if a circuit is marked, for a class $\mathbb{C}$ is ϵ-secure if and only if :

1. **Functionality preserving**: $Pr(Mark(C,k) \simeq_{\simeq 0} C | k = Setup(), C \hookleftarrow U(\mathbb{C})) \simeq_\lambda 1$.

   This guarantees that the marked program only slightly (negligibly compared to the number of inputs) alters functionality.

2. **Correctness**: $Pr(Verify(Mark(C,k),k) = 1 | k = Setup(), C \hookleftarrow U(\mathbb{C})) \simeq_\lambda 1$.

   This guarantees that $Mark$ returns a marked program.

3. **Unremovability**: $\forall A, Pr(A(C') \simeq_\epsilon C$ and $Verify(A(C'),k) = 0 | C' = Mark(C,k), k = Setup(), C \hookleftarrow U(\mathbb{C})) \simeq_\lambda 0$.

   This guarantees that removing the mark implies changing more than ϵ portion of the inputs of the original circuit, thus not preserving the functionality.

4. **Meaningfulness**: $\forall A, Pr(Verify(C,k) = 1 | C \hookleftarrow U(Circuits), k = Setup()) \simeq_\lambda 0$.

   This guarantees that the mark is deliberate: only a negligible number of circuits are marked.

Remark: We are forced to give a set $\mathbb{C}$ for the watermarking scheme as we can not say the unremovability property is true for all C : we would have a trivial adversary, the one that outputs the constant circuit C. This makes the definition behave strangely in some cases, but I have not found a better one, even after reading papers about formalizing human ignorance [Rog06]. Furthermore, as properties are satisfied for a uniform $C \in \mathbb{C}$, one can see $\mathbb{C}$ as the set in which we want to hide the circuits we watermark and $\lambda = log(|\mathbb{C}|)$ is the security parameter.

2.4 Definition of PPRF [BW13]

A set $\mathbb{C} = \{C_k\}$ is a punctured pseudo random function set if and only if

1. $\mathbb{C}$ is a pseudo random function :
   $\forall A, |Pr(A^{C_k}() = 1 | C_k \hookleftarrow U(\mathbb{C})) - Pr(A^C() = 1 | C \hookleftarrow U(O^I))| \simeq 0$.

2. $\mathbb{C}$ is puncturable : $\exists Punc$ such that , $\forall x^*$

   (a) $\forall x \neq x^*, Pr(Punc(x^*,k)(x) = C_k(x) | C_k \hookleftarrow U(\mathbb{C})) = 1$
   This means that $Punc(x^*,k)$ allows us to evaluate C_k on all input but x^*.

(b) $\forall A, |Pr(A(Punc(x^*,k), C_k(x^*)) = 1 | C_k \hookleftarrow U(\mathbb{C})) - Pr(A(Punc(x^*,k), U(O)) = 1 | C_k \hookleftarrow U(\mathbb{C}))| \simeq 0$
This means that $Punc(x^*,k)(x^*)$ appears random.

Remark: The puncturing property allows us to say that $C(x^*)$ is completely independent from all other values $C(x)$. That it to say that there is no specific properties that link the output values of circuits in $\mathbb{C}$.
Construction: The only known construction is to use the Goldreich-Goldwasser-Micali construction [GGM86] which works as follows. Take a pseudo random generator f (PRG) that takes a uniform random string and outputs a string indistinguishable from uniform with twice the length of the input string. For each k in O, let C_k be the circuit that creates recursively a random string of size $|I||k|$ (by applying $log(|I|)$ times f) and cuts that uniform string into $|I|$ parts and gives each part to one input. Let us show that $\mathbb{C} = \{C_k\}$ is a PPRF set.

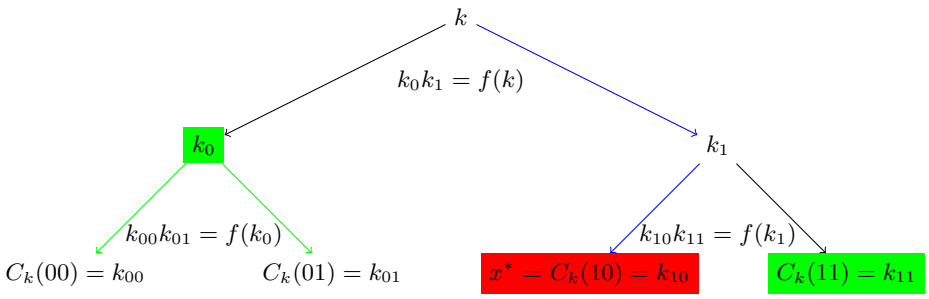


Figure 3: PPRF construction

- $\mathbb{C}$ is a PRF as f creates strings indistinguishable from uniform and therefore the first part and the second part of the generated string are each time independent and uniform.
- $\mathbb{C}$ is a puncturable : imagine we are puncturing at x^*. Then $Punc(x^*,k)$ is the program that uses k_0 and k_{11} instead of k in the example in figure 2.4, thus being able to evaluate any input but x^*, as k_0 allows to evaluate k_{00} and k_{01} and k_{11} allows to evaluate k_{11}. However, nothing allows to evaluate k_{10} given only k_0 and k_{11}.

3 Existing watermarking constructions

3.1 Basic construction [CHV15, NW15]

The idea is to have a special input x^* that when queried returns a special output y^* indicating that the program is marked. The goal would be that the adversary can not find that special input and thus can not remove the mark. By applying iO, we can hope that

an adversary can not look at the implementation of the program and see x^* in the code. Therefore, let us consider the following construction, for some distributions D and D'.

```
Setup()
  x* ↩ D(I)
  y* ↩ D'(O)
```

```
Mark(C) = return iO (
function :
  x* → y*
  x → C(x) when x ≠ x*
);
```

```
Verify(C) = return C(x*)==y*
```

It is easy to check that this construction has the functionality and correctness properties. However, we do not have meaningfulness as on average, half the circuits will be considered marked ($O = \{0, 1\}$), but this can be addressed by modifying several inputs and have the set of modified inputs match a pattern. Therefore, achieving meaningfulness is not an issue and for simplicity, we will be changing only one input for the moment. We must now check that our scheme has unremovability. But first, I have created a couple of attacks possible on this scheme, to give better intuition.

3.2 Attacks

3.2.1 The distribution attack

The simplest attack might be to change any point that has a high chance of being an x^*. Another is detecting something abnormal in the output, thus guessing that output to be y^*. This leads to the following remover.

```
UnMark(C) =
function :
  x  → ⊥ when D(x) big
| x → ⊥ when D'(C(x)) ≠ U(ℂ(x))
| x -> C(x) otherwise
);
```

Avoiding attacks on x^* directly seems pretty straightforward : we can just pick x^* uniformly from I. Avoiding attacks that detect abnormal output is already harder because we would need to pick y^* uniformly from $\mathbb{C}(x^*)$. But being able to pick uniformly from $\mathbb{C}(x^*)$ still seems a reasonable assumption.

3.2.2 The property attack

However, usually, the sets of circuits we try to watermark $\mathbb{C}$ is not completely random and has some properties. As a simple example, we might have $\forall C \in \mathbb{C}, \forall x, IsEven(x) \Rightarrow C(x) = C(x+1)$. We would then have the following remover.

```
UnMark(C) =
function :
  x → ⊥ when IsEven(x) and C(x+1) ≠ C(x)
| x → ⊥ when IsOdd(x) and C(x) ≠ C(x-1)
| x → C(x) otherwise
);
```

A more complex property attack could be if you knew $C \in \mathbb{C}$ solves a problem in NP. One could just check if $C(x)$ solves the problem. If it does, it probably is not modified. If it does not, it must be y^*. This kind of attack seems very hard to avoid as most software solve a specific problem and therefore have properties related to the problem... However, it has been proven that this scheme is unremovable for PPRFs [CHV15, NW15], because their puncturability implies the absence of specific properties.

3.3 Unremovability for PPRF [CHV15, NW15]

Theorem: Let $\mathbb{C}$ be a PPRF set. The scheme described in 3.1 is ϵ-unremovable with $\epsilon \simeq 0$ and D and D' being the uniform distributions [NW15].
Proof: Let $\mathbb{C}$ be a PPRF and the distributions D and D' used in setup are respectively $U(I)$ and $U(O) = U(\mathbb{C}(x^*))$. Let us show that our basic scheme is secure. We will prove this by contradiction and assume we have a remover R. Here are the steps of the proof.

1. From R, create a distinguisher for the distributions $(Mark(C, x^*, y^*), x^*)$ and $(Mark(C, x^*, y^*), U(O))$
   The intuition is that we can deduce x^* by looking where R changes the output values.

2. From a distinguisher for $(Mark(C, x^*, y^*), x^*)$ and $(Mark(C, x^*, y^*), U(O))$, create a distinguisher for the distributions $(Mark^{(1)}(C, x^*, y^*), x^*)$ and $(Mark^{(1)}(C, x^*, y^*), U(O))$, with $Mark^{(1)}$ defined by :

   ```
   Mark(1)(C) = return iO (
   function :
     x* → y*
     x → Punc_x*(C)(x) when x ≠ x*
   );
   ```

   The intuition is that $Mark^{(1)}$ and $Mark$ have the same functionality and are therefore indistinguishable thanks to IO. We then just need to use triangular inequality to get the result.

3. From a distinguisher for $(Mark^{(1)}(C, x^*, y^*), x^*)$ and $(Mark^{(1)}(C, x^*, y^*), U(O))$, create a distinguisher for the distributions $(Mark^{(2)}(C, x^*, y^*), x^*)$ and $(Mark^{(2)}(C, x^*, y^*), U(O))$, with $Mark^{(2)}$ defined by :

```
Mark^(2)(C) = return iO (
function :
  x* → C(x*)
  x → Punc_{x*}(C)(x) when x ≠ x*
);
```

The intuition is that if $Mark^{(2)}(C, x^*, y^*)$ and $Mark^{(1)}(C, x^*, y^*)$ were distinguishable, one could create a PPRF adversary : when given y from $U(\mathbb{C})(x^*)$ or $U(O)$, one could tell from which distribution it came by creating $Mark^{(1)}(C, x^*, y)$ and applying the distinguisher on it.

4. From a distinguisher for $(Mark^{(2)}(C, x^*, y^*), x^*)$ and $(Mark^{(2)}(C, x^*, y^*), U(O))$, create a distinguisher for the distributions $(Mark^{(3)}(C, x^*, y^*), x^*)$ and $(Mark^{(3)}(C, x^*, y^*), U(O))$, with $Mark^{(3)}$ defined by :

```
Mark^(3)(C) = return iO (C);
```

Again, $Mark^{(2)}$ and $Mark^{(3)}$ have the same functionality and are therefore indistinguishable thanks to IO

5. Show that a distinguisher for the distributions $(Mark^{(3)}(C, x^*, y^*), x^*)$ and $(Mark^{(3)}(C, x^*, y^*), U(O))$ is absurd.

   The intuition is that $Mark^{(3)}(C, x^*, y^*)$ does not depend on x^*, therefore distinguishing $(Mark^{(3)}(C, x^*, y^*), x^*)$ and $(Mark^{(3)}(C, x^*, y^*), U(O))$ is equivalent to distinguishing x^* from $U(I)$ which is absurd as $x^* \hookleftarrow U(I)$.

3.4 Additional goals

We now have a basic watermarking scheme for PPRFs, but we might want to add additional security to it, or add properties such as finding the person who leaked the program. From now on, x^* will be a general notation for a special input where the value is altered or suspected to be altered.

3.4.1 Message embedding [NW15]

The idea is to allow more than just the binary values "the program is marked" or "the program is unmarked", and add information to the mark. Information could be to whom the program was given for example, that would allow to trace the content as explained

in figure 1. In our definition, we now replace $Verify$ by $Extract$ that must return the message with which the program was marked or $\perp$ when the program is unmarked.

First, let us assume we want to add a message m of length n. The idea is to change y^* to $y^* \oplus m$. $y^* \oplus m$ still has same distribution as y^*, so this would not alter unremovability. $Extract$ now returns $C(x^*) \oplus y^*$. As y^* might only be one bit if $O = \{0, 1\}$, we need to use several x^* and y^* and have m written over the set of the queried x^*. This only alters functionality at a negligible number of points as $|m|$ is assumed polynomial in λ.

The only problem with this construction is that we break meaningfulness as all programs are marked. To avoid this problem, we can add a string of zeros at the beginning of m. This way, we know that if the message extracted does not start by this string of zeros, then the program is not watermarked.

3.4.2 Protection against chosen watermark attacks [CHV15, NW15]

We now assume that the adversary has $Mark(P)$ and P for some program P (for example one your old employees now works for a rival company). He is now presented C' and wishes to unmark it. To do so, he could use the following remover.

```
UnMark(C) =
function :
  x → ⊥ when P(x) ≠ Mark(P)(x)
| x → C(x) otherwise
);
```

To avoid this attack from working, the idea is to make x^* depend on C. For example, now instead of changing the value of C at point x^*, we change it at $C(x^*)$ (we might cut or use several x^* to have the same length as the input). $Verify$ will now check that $C(C(x^*)) = y^*$. Of course, we might have an issue if $C(x^*)$ is also changed, but this happens with negligible probability.

3.4.3 Protection against partial functionality change [NW15]

We now assume that the adversary does not mind having the program not work on some big portion of inputs. Let us construct an ϵ-unremovable scheme, for $\epsilon < 1$ from our $\simeq 0$-unremovable scheme. The attack we need to avoid is the adversary changing randomly half the inputs for example and thus removing the mark with probability $\frac{1}{2}$. The idea to avoid such an attack and make our scheme $\epsilon < 1$-unremovable, is to use n marks and requiring only one of the marks to be in the circuit for the circuit to be considered marked. Of course, n is a polynomial number in λ to keep functionality. The adversary's probability of removing will then become $\frac{1}{2^n}$ which is negligible.

3.4.4 Public verification [CHV15, NW15]

Assume I am sold a program by a company C and I know another company has a very similar program. I would like to check that C is indeed the owner of the program they are selling me. To do so, we would need a scheme with public $Verify$, that is to say that everyone has access to the $Verify$ function instead of just the selling company. However, an adversary can then run $Verify$, see where it queries C and deduce where x^* is.

To avoid this, we change the functionality on an exponential number of x^* and at each call to $Verify$, we only query a subset of the x^* chosen at random. This way, if the adversary wants to remove the mark, he must change a non negligible portion of the x^* which will be of exponential size. However, the adversary must run in polynomial time, so he can not change that many values without having a short description of those values, and the scheme stays secure.

More precisely, we duplicate information on all inputs of the form $PRF(r)||r$ for some PRF kept secret. Notice that this is still a negligible portion of the inputs, so we keep functionality. $Verify$ now chooses an r at random and queries at $PRF(r)||r$. Of course, $Verify$ will use IO, otherwise, the adversary could find the PRF used and remove all inputs of the form $PRF(r)||r$.

Remark: A full construction with message embedding, chosen watermark attacks, protection against partial functionality change and public verification is available in [NW15]

3.4.5 Collusion resistance

Let us go back to figure 1 with Bob and Alice having a marked program and wanting to give an unmarked program to Jon. In our construction, we only assumed that Bob or Alice would try to give an unmarked program to Jon alone. But what if they were to collude and together tried to construct an unmarked program for Jon. We say that a watermarking scheme is collusion resistant if any set of users given a marked program can not create together an unmarked program. Although collusion resistance does not seem to have been studied in the previous articles, I believe collusion resistance to be one of the most important additional goals.

In the case where we do not use messages, there is no problem as both Bob and Alice have the same marked program, therefore they do not learn anything more from being together.

However, in the case of message embedding, Bob and Alice might analyze their differences to find and, change or remove, the message. This might not enable them to remove the mark as both Bob's and Alice's messages start with a string of zeros indicating that the program is marked and therefore there are no differences in the beginning of the message making the beginning unremovable. However, they might be able to remove the message content or even frame someone else, so that they are not accused of giving the program to Jon.

First, we can prove that Bob and Alice can only find x^* where they do not have the same $C(x^*)$. Intuitively, this is because only the differences Bob and Alice have can help them find the x^*. Now let us assume we have n users that have legal copies of the program (each marked with a user id).

Now let us say $n-1$ users collude to try and frame the last user, that is to say give to Jon a program marked with the last user's id. To prevent that, we need to have at least an x^* where all the $n-1$ users have the same values and the last one a different, so that they can not change the value specific to the last user. We must have that for each group of $n-1$ users, so the message length is at least in the number of users. It is easy to show that if the message for each user is of length n and has a 1 in position user id and 0 everywhere else, stops anyone from framing anyone else [BS96]. However, they can still remove the user id by erasing all the 1s.

What would be great is to be able to find for sure at least one of the colluders. This has been achieved in $O(n^4)$ [BS96] and we know that all algorithms are $\Omega(n^2)$ [BS96].

4 Limits of watermarking

4.1 Application

We have found only one application to PPRF watermarking: traitor tracing. Imagine a TV company broadcasting an encrypted TV show as in figure 4.1. Each subscriber receives a PPRF to decrypt the signal marked with the id of the subscriber. Now, if Jon wishes to decrypt that signal, he will receive a program marked with the id of the person who gave it to him, allowing to find the "traitor" who gave the program to Jon.

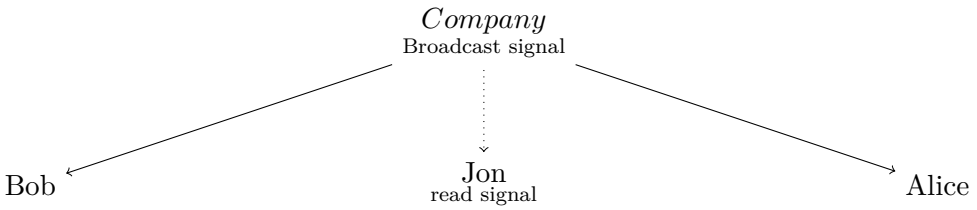


Figure 4: Traitor Tracing Illustration

However, collusion resistance has strong complexity with our scheme and another scheme which has $Poly(log(n), \lambda)$ message length has been found using IO [BZ14]. This encourages us to try and watermark more than PPRFs.

4.2 Watermarkable sets

Given the different attacks I found, the limitation to PPRFs seems unjustified when all we needed is resistance to property attacks and a uniform picking in $\mathbb{C}(x^*)$. Property

attacks convinced us of the need for puncturability, this is all the more the case that not all PRFs are watermarkable [CHV15]. However, there is no reason why puncturability should always be on random functions. We now change the definition of puncturable in 2.4 so that instead of $Punc(x^*, C)(x^*)$ being indistinguishable from $U(O)$, it is now indistinguishable from $U(\mathbb{C}(x^*))$.

We can now prove that any puncturable (in the new sense) set is watermarkable. The proof is exactly the same and only the question of existence and utility of puncturable non PPRF functions remains. Existence is easy to prove as the following set is puncturable and is not a PPRF.

Let $\{f_k\}$ and $\{g_k\}$ be a PPRF. Let $\mathbb{C} = \{C_k, C_k(x) = f_k(x)||1\}$ is watermarkable. However, we have found no use for such sets as they seem very tied to cryptography and do not offer much more than PPRFs. Considering the property attack, this gave me the intuition that we can not watermark sets of programs that are not cryptographic functions.

4.3 Definition of cryptographic functions

All cryptography is based on the existence of one way functions. A one way function is a function for which it is easy to compute $f(x)$, but given $f(x)$, it is hard to find an element in the pre-image. The idea for defining non-cryptographic functions is to have a particular function that is not one way.

Let n be of polynomial size. Let $f : (S = \{x_1, ..., x_n\}, C) \rightarrow ((C(x_1), ..., C(x_n)), x_1, ..., x_n)$. $\mathbb{C}$ is non-cryptographic if and only if $\exists A, \forall S \subset I, |S| = n, \forall C \in \mathbb{C}, Pr(f(A(f(S, C))) = f(S, C)) > 0$

4.4 Impossibility proof for non cryptographic functions

Theorem: If $\mathbb{C}$ is a non cryptographic set, there exists no ϵ-secure watermarking scheme for $\epsilon > 0$.

Proof: Let $\mathbb{C}$ be a non-cryptographic set, $S \subset I, |S| = n$, $F^{-1}(C(S), S)$ be the pre-image set of $(C(S), S)$ by f and $f^{-1} : (C(S), S) \rightarrow (S, C)$ one of the functions that calculate an element in the pre-image of f.

The idea behind the proof is to create from f^{-1} a program called a learner that with oracle access to a circuit in $\mathbb{C}$ can recover the original circuit. This allows us to remove the mark as we get access to the original circuit.

Formally, we say L is a learner if and only if $Pr(L^C() \simeq C | C \hookleftarrow U(\mathbb{C})) > 0$. Let us first prove that when we have a learner, we have a remover, under the assumption that the learner has high entropy on the variables it queries.

4.4.1 Learner to remover

If the learner never queries at x^*, then the learner will find the original circuit which is unmarked. Therefore we would have a remover. However, if the learner queries the oracle

at x^*, it might not learn the original circuit as the value has been modified and having a remover seems hard. Let us consider a few different cases :

1. The distribution of the x^* has very high entropy, that is to say close to uniform. The number of x^* is negligible compared to the number of inputs, therefore the probability that the learner queries at one of the x^* is negligible and we have a remover.

2. The distribution of the x^* has very low entropy: x^* is conveniently chosen exactly where the learner queries. We just change those positions and remove the mark, there is no need to use the learner.

3. One of the x^* has high entropy and another x^* is at a position the learner queries. We do not have a solution in the current context, the only viable option is to assume that the points the learner queries have high entropy, thus putting us back to the first setting.

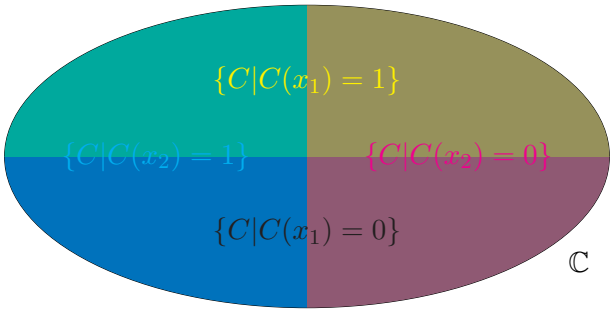
Let $Q(C)$ be the random variable describing the set of queries $L^C()$ makes and X^* the random variable describing the set of inputs changed by $Mark$
Assumptions: $Q(Mark(U(\mathbb{C})))$ is a polynomial size uniform set and $Verify(C)$ can be defined as $Verify^C()$.
Lemma: $Pr(L^{C'}() \simeq C | C' = Mark(C), C \hookleftarrow \mathbb{C}) > 0$.
Proof: $Pr(L^{C'}() \simeq C | C' = Mark(C), C \hookleftarrow \mathbb{C})$
$\geq Pr(L^{C'}() \simeq C$ and $Q(C') \cap X^* = \emptyset | C' = Mark(C), C \hookleftarrow \mathbb{C})$
$\geq Pr(L^C() \simeq C | C \hookleftarrow \mathbb{C}) Pr(Q(C') \cap X^* = \emptyset | C' = Mark(C), C \hookleftarrow \mathbb{C})$
$\geq Pr(L^C() \simeq C | C \hookleftarrow \mathbb{C})(1 - poly(\lambda) Pr(X \in X^* | X \in Q(C'), C' = Mark(C), C \hookleftarrow \mathbb{C}))$
$\geq Pr(L^C() \simeq C | C \hookleftarrow \mathbb{C})(1 - poly(\lambda) Pr(X \in X^*)$ as $Q(C')$ is uniform
$\geq \underbrace{Pr(L^C() \simeq C | C \hookleftarrow \mathbb{C})}_{>0} (1 - poly(\lambda) \underbrace{Pr(X \in X^*)}_{\simeq 0:\ Mark \text{ keeps functionality}})$
$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\simeq 1}$
$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{>0}$

The idea behind the proof is that when picking $|Q(Mark(U(\mathbb{C})))|$ random elements in I, we have very low probability for even one of them landing in X^* as $|X^*|$ is negligible because of functionality preservation. The proof can probably be extended to distributions close to uniform instead of exactly uniform, but that result will not be necessary to get the impossibility result.

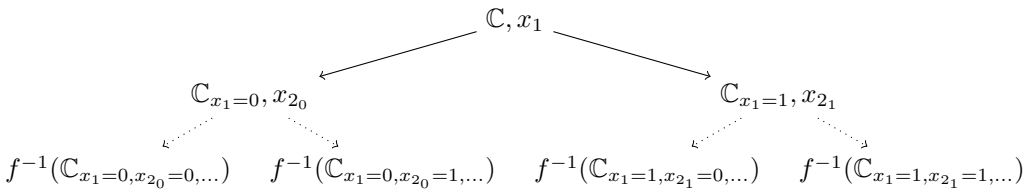
4.4.2 From non-cryptographic functions to learning

The basic idea

The idea is to query $(x_1, ..., x_n)$ in the circuit and return $f^{-1}((C(x_1), ..., C(x_n)), x_1, ...x_n)$. The hope is that $F^{-1}((C(x_1), ..., C(x_n)), x_1, ...x_n)$ has only one element, the circuit C. We will be assuming that $O = \{0, 1\}$ for simplicity.



The idea is to query x_1. The result divides the initial space into two parts : $F^{-1}(0, x_1) = \{C|C(x_1) = 0\}$ and $F^{-1}(1, x_1) = \{C|C(x_1) = 1\}$. Then we keep on querying on $x_2, ..., x_n$, splitting the space into 2^n parts. The hope is that for n polynomial, those parts become of size 1 so that we get the original circuit. To simplify notations, we write $\mathbb{C}_{x_1=y_1,...x_n=y_n}$ instead of $F^{-1}((y_1, ...y_n), x_1, ...x_n)$ and $f^{-1}(\mathbb{C}_{x_1=y_1,...x_n=y_n})$ is an element in $\mathbb{C}_{x_1=y_1,...x_n=y_n}$. The learning algorithm would then look like this.



The first question, is how should we chose the x_i. If we could, we would like to choose the next query x so that it splits the set at the current node into to sets of identical size, so that whatever the outcome, the size of the set at the next node is divided by 2. If that was possible, we would be making exactly λ queries as $|\mathbb{C}| = 2^\lambda$, but we can not make that assumption. Instead, let us assume for the moment that we have an oracle $Best$ that returns the best query x to make at the current node, that is to say that $Best(Cset)$ is the query that splits our current set $Cset$ into two sets with as similar a size as possible.

We still seem to run into problems: assume $\mathbb{C} = \{C_\alpha, \alpha \in I\}$ with C_α the circuit that returns 1 on input α and 0 otherwise. Even with an oracle such as $Best$ there is a path of exponential length in the tree, therefore our algorithm will not run in polynomial time on some inputs... This seems to completely destroy our attempt to learn using f^{-1}.

However, all circuits in $\mathbb{C}$ are similar, therefore, returning any of the circuits in $\mathbb{C}$ works. A generalization of this idea is possible as if all x do not split the current set into similar sizes, than that means that all circuits are similar.

Let us assume for a moment that we also have an oracle returning the size of $F^{-1}(C(S), S)$ for any S and C. We could then imagine the following algorithm for some predefined $p < 0.5$.

```
L^C() =
Cset = ℂ
x = Best(Cset)
while (|Cset_{x=0}| > p|Cset| and |Cset_{x=1}| > p|Cset|)
   Cset = |Cset_{x=C(x)}
   x = Best(Cset)
return f^{-1}(Cset)
```

Lemma: The runtime of such an algorithm is $O(\frac{\lambda}{p})$.
Proof: The set at the current node always decreases by at least a factor $1-p$ at each step. Therefore, after n steps, the current set has at most size $\frac{2^\lambda}{(1-p)^n}$ and the algorithm stops at worst when the current set has size 1. But, $2^\lambda(1-p)^n = 1 \Rightarrow n = -\frac{\lambda}{log(1-p)} = O(\frac{\lambda}{p})$

Assumption: Assume $f^{-1}(Cset)$ returns a uniform element in $Cset$.
Lemma: This program is a learner: $Pr(L^C() \simeq_\epsilon C) \geq (1 - 2\frac{p}{\epsilon})^2$
Proof: As long as we are not reaching a leaf of the tree, we do not lose any information, therefore, we keep correctness. Formally, assume by induction that our theorem is valid for the sets at nodes beneath the current one, we than have:
$Pr(L^C() \simeq_\epsilon C | \text{we are currently at node } Cset)$
$= Pr(C(x) = 0)Pr(L^C() \simeq_\epsilon C | C(x) = 0, \text{we are currently at node } Cset) + Pr(C(x) = 1)Pr(L^C() \simeq_\epsilon C | C(x) = 1, \text{we are currently at node } Cset) \geq (Pr(C(x) = 0) + Pr(C(x) = 1))(1 - 2\frac{p}{\epsilon})^2 = (1 - 2\frac{p}{\epsilon})^2$.

We now need to prove that our program learns correctly a the leaves. Assume we reach a leaf. That means that $\forall x, |Cset_{x=0}| < p|Cset|$ or $|Cset_{x=1}| < p|Cset|$. Intuitively, this means most circuits agree with $Perfect(x) = (|Cset_{x=0}| < |Cset_{x=1}|)$ and returning $Perfect$ would work. However, we can not return $Perfect$ as $Perfect$ might not be of polynomial size. Therefore, we also need to show that the random circuit returned is similar to $Perfect$.

$Pr(f^{-1}(Cset) \simeq_\epsilon C | \text{We are at leaf } Cset) \geq \frac{1}{|Cset|} \sum_{C \in Cset} Pr(f^{-1}(Cset) \simeq_\epsilon C)$
$\geq \frac{1}{|Cset|} \sum_{C \in Cset} Pr(f^{-1}(Cset) \simeq_{\epsilon/2} Perfect \text{ and } Perfect \simeq_{\epsilon/2} C)$
$\geq Pr(f^{-1}(Cset) \simeq_{\epsilon/2} Perfect) \frac{1}{|Cset|} |\{C | Perfect \simeq_{\epsilon/2} C\}|$
$\geq (\frac{1}{|Cset|} |\{C | Perfect \simeq_{\epsilon/2} C\}|)^2$ because $f^{-1}(Cset)$ is a uniform element in $Cset$
$\geq (1 - \frac{|\{C | Perfect \not\simeq_{\epsilon/2} C\}|}{|Cset|})^2$

Now we want to prove that most circuits are similar to $Perfect$. So let us count the number of circuits that are far from $Perfect$.
Let $W(Cset) = \{C \in Cset | Perfect \not\simeq_{\epsilon/2} C\}$
Let $Err(Cset) = \{(C, x) \in Cset \times I | Perfect(x) \neq C(x)\}$
We have $|Err(Cset)| \leq |I|p|Cset|$ because for each $x \in I$, there is at most p portion of the circuits that disagree with $Perfect$.
We also have $|Err(Cset)| \geq |W(Cset)||I|\epsilon/2$ because each circuit in $W(Cset)$ "uses" $|I|\epsilon/2$ elements of $Err(Cset)$.
Therefore, $|W(Cset)| \leq 2\frac{p}{\epsilon}|Cset|$ and $1 - \frac{|\{C|Perfect \not\simeq_{\epsilon/2} C\}|}{|Cset|} \geq 1 - 2\frac{p}{\epsilon}$. And we get our result : $Pr(f^{-1}(Cset) \simeq_\epsilon C | \text{We are at leaf } Cset) \geq (1 - 2\frac{p}{\epsilon})^2$.
Consequence: By choosing $p(\lambda) = \frac{\epsilon(i)}{poly(\lambda)}$, we have a learner that runs in polynomial time assuming:

- We have a uniform inverter : $f^{-1}(Cset) = U(F^{-1}(Cset))$.
- We have a $Best$ oracle that chooses our next query
- We have a $|Cset|$ oracle gives us the size of $Cset$

Now, we would like to remove those assumptions.

Removing assumptions for learning

- **Uniform inverter:** Now, we want $f^{-1}(Cset) \simeq_{\epsilon/2} Perfect(x)$. We know that $|W(Cset)| \leq \frac{2p}{\epsilon}$. We would like to make $|W(Cset)| = 0$ with non negligible probability so that $\forall f^{-1}, Pr(f^{-1}(Cset) \simeq_{\epsilon/2}) > 0$.

  The idea behind this is instead of returning $f^{-1}(Cset)$, we return $f^{-1}(Cset_{u_1=Perfect(u_1),...,u_k=Perfect(u_k))}$ with u_i picked at uniformly in I. As we can not evaluate $Perfect(u_i)$ easily, we replace it by $C(u_i)$. This does not change much as $Pr(C(u_i) \neq Perfect(u_i)) \simeq 0$.

  Now, we know that for each u_i we add, we remove at least $\frac{2p}{\epsilon}$ portions of the circuits in $W(Cset)$ on average as circuits in $W(Cset)$ differ with $Perfect$ on at least $\frac{2p}{\epsilon}$ portion of the inputs. Therefore, that means that $|W(Cset_{u_1=Perfect(u_1),...,u_k=Perfect(u_k)})|$ is on average $|W(Cset)|(1 - \frac{2p}{\epsilon})^k$. Thus $k = O(log(|W|)\frac{\epsilon}{2p}) = O(\lambda\frac{\epsilon}{2p}) = O(\lambda poly(\lambda))$ to reduce $|W(Cset_{u_1=Perfect(u_1),...,u_k=Perfect(u_k)})|$ to zero.

- ***Best* oracle:** The idea is to estimate $Best(Cset)$ instead of having $Best(Cset)$. To do that, we pick $x_1, ..., x_k$ at random and we chose the best x within $x_1, ..., x_k$. This still requires the $|Cset|$ oracle. The probability that the best x within $x_1, ..., x_k$, with $k = poly(\lambda)$, does not split the current set into two sets with more than $p|Cset|$ elements when such an $x \in I$ exists is negligible (proof is done using the Tchebychev inequality). Therefore, estimating the best x instead of having the best x still works.

- **$|Cset|$ oracle:** The idea is that over-splitting does not change correctness. Therefore, in the previous scheme, instead of estimating the best x and splitting only for that x each time, we split for all the x's tried. This still takes polynomial time and does not change correctness: we just have some useless splittings. But this avoids the need to have $|Cset|$ as we do not need to know whether the splitting at x is useful we only need the probability that one of the splits is good to be high.

Now, let us look back to what our scheme looks like when we remove those assumptions.

$$L^C() = f^{-1}(Cset\underbrace{\underbrace{x_{1,1} = C(x_{1,1}), ..., x_{1,k} = C(x_{1,k})}_{\text{Estimates the first best } x}, ..., x_{O(\frac{\lambda}{p}),1} = C(x_{O(\frac{\lambda}{p}),1}), ..., x_{O(\frac{\lambda}{p}),k} = C(x_{O(\frac{\lambda}{p}),k}),}_{\text{We arrive at a leaf}}$$

$$\underbrace{u_1 = C(u_1), ..., u_n = C(u_n)}_{\text{Returns a circuit similar to Perfect}})$$

All these inputs are chosen at random, so the learner has the entropy required by the remover. The only assumption left is the fact that $Verify(C)$ can be written as $Verify^C()$. This is because an adversary can use iO to hide anything that is special to C's implementation, so only functionality can be checked.

Therefore, we have finished our proof and proven that non-cryptographic function sets can not be securely watermarked.

5 Conclusion and Perspectives

The need for cryptographic watermarking seems quite close as IO might become practical. Although we can watermark cryptographic functions called PPRFs [CHV15, NW15], and I have shown that we can watermark any puncturable function, the applications of watermarking these functions seem limited to traitor tracing that we can already do more efficiently [BZ14]. Furthermore, the existence of the property attack makes non-cryptographic functions not watermarkable, destroying our initial goal of general software watermarking.

This seems to end most research paths on cryptographic watermarking. However, there are still things to do: finding a definition for watermarking that fits our intuition better, finding other cryptographic applications to watermarking such as traitor tracing and studying the case of Turing machines instead of circuits.

References

[BGI+01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. 2001.

[BS96] Dan Boneh and James Shaw. Collusion-secure fingerprinting for digital data. 1996.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications, 2013.

[BZ14] Dan Boneh and Mark Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. 2014.

[CHV15] Aloni Cohen, Justin Holmgren, and Vinod Vaikuntanathan. Publicly verifiable software watermarking. 2015.

[GGH+13] Shelly Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Anant Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. 2013.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. 1986.

[IN10] A.N. Fionov I.V. Nechta. Digital watermarks for c/c++ programs. 2010.

[NW15] Ryo Nishimaki and Daniel Wichs. Watermarking cryptographic programs against arbitrary removal strategies. 2015.

[PMA11] Ante Poljicak, Lidija Mandic, and Darko Agic. Discrete fourier transform–based watermarking method with an optimal implementation radius. 2011.

[Rog06] Phillip Rogaway. Formalizing human ignorance. 2006.