Watermarking Overview

- What is watermarking? The idea is simple:
  1. Start with a cover message of some kind. It can be an image, a sound-file, a video.
  2. Now, take a secret message and embed it inside the cover message. Usually, this secret message is some sort of copyright notice.

- In this example, the cover message is an image and the secret message is a the word “copyright”, a date, and my name.

- If someone were to steal this image and use it in a book or magazine, then I could take them to court for breach of copyright, and I would argue that this is indeed my image because it has my copyright notice in it.
• The fact that this copyright notice is visible does take away some of the value of the image, so, in most cases we want the watermark to be imperceptible. On the other hand, if the watermark is visible, then that will discourage someone from stealing the image in the first place.

• Fingerprinting is a variant of watermarking. When we watermark an image we store the same copyright notice in every copy. When we fingerprint an image we store a unique customer-identification-number in every image we sell. That way, if lots of illegal copies of the image should start to appear, then we may be able to trace the customer who bought the original copy, and then bring them to court.

• To build a good watermarking system these problems have to be solved:
  1. The watermark should be stealthy, i.e., it should be difficult for an adversary to locate.
  2. It should have a high data-rate, i.e., we should be able to store a large secret message in a small cover message.
  3. It should be resilient to attack, i.e. it should be hard to remove the watermark from the cover message.

• Scenario: Alice, has an object (an image, a program, whatever) that she wants to sell. Bob wants to steal this object to sell it on to a third party.

• First, Alice adds a watermark to her object. She uses a secret key to make sure that she is the only one who can extract the watermark.
- Then Bob steals a copy of Alice’s watermarked object. Bob sells a stolen copy to Charles who is Alice’s lawyer. Charles extracts the watermark using the secret key, and turns Bob over to the authorities.

- If Bob can find the location of $W$, he may try to crop it out of the object, without destroying too much of the object itself. We call this a subtractive attack.

- An even simpler idea is an additive attack, where Bob adds new watermarks to the object to make it hard for Charles to prove that Alice’s watermark is the original one.

- Or, Bob could launch a distortive attack, where he applies a sequence of transformations to the object. Ideally, he will add just the right level of distortion to the object so that the watermark will be useless, but the object itself still has some value to Bob.

- Alice can add tamperproofing to her object such that if Bob tries to remove the watermark the resulting object is completely useless.

- If Alice fingerprints her object she leaves herself open to collusive attacks. Bob can steal several copies of the object – each with its own fingerprint – and by comparing their differences he is able to extract the original object.
• Might some media watermarking techniques carry over to software watermarking?

• The most well-known image watermarking algorithm is called patchwork. The idea is to store the watermark in the brightness of a random sequence of pixels in the image.

  1. To embed the watermark we start by initializing a random number generator with our secret key.

  2. Using the random number generator we then pick a sequence of pairs of pixels. We then adjust the brightness of one of the pixels up by a small amount, and the brightness of the other pixel, down by the same amount.

Notice that the overall brightness of the image hasn’t changed.

1. To extract the watermark we visit the same pseudo-random sequence of pixels, this time summing up the difference of their brightness.

2. For a completely random sequence of pixels we’d expect this sum to be zero. If it’s not, we’ve detected a watermark.

This method is stealthy but the bit-rate is low, just one bit per image.

We expect \( S \approx 0 \) as \( n \) increases.

\( \delta \) would be 1–5 parts in 256 bits.

Each patch can contain more than one bit.
• Many media watermarking algorithms take advantage of the fact that our human sensory systems aren’t perfect.

• To watermark a sound-clip we can flip the least significant bits of each sample. This may introduce some noise, but if the sample is noisy to begin with we will not be able to detect it.

• Humans also are not very good at detecting short echoes. We can use this fact to encode a secret message by introducing short echoes in the sample. The length of these echoes would encode the watermark.

• Again, we can use a pseudo-random number generator to pick out the places where the watermarking bits are encoded.

We can encode watermarks in English text.

• In a formatted hard-copy document we can encode a mark in the word or line spacing.

• In an soft-copy document we can encode the mark in white-space: One space means 0, two spaces means 1.

• If every line has to be justified we can’t encode a bit between every pair of words. We can use “Manchester-encoding”: 01 = 1, 10 = 0, 11 = null, 00 = null.

• We can also encode a mark in the syntactic structure of an English text. Here we’ve applied a syntactic transformation to make a “Cleft sentence.”

• Finally, we can store information in the choice of words.

attacks on media watermarks

we trade-off between

1. stealth (we want imperceptible marks),
2. hit-rate (we want to embed much data),
3. resilience (we want the mark to withstand attacks).

attacks: compression, scaling, cropping, blurring, rotation...
There are three things we want from our watermarking algorithm:
1. Marks should be hard to find (they should be stealthy);
2. The bit-rate should to be high (we want large watermarks);
3. Marks should be resilient (they should withstand as many kinds of attacks as possible).

Many simple image transforms will obliterate a watermark. Lossy compression will remove imperceptible bits. If watermarks are stored in the least significant bits of the media then we can randomly flip all such bits.

There will always be a trade-off between stealth, bit-rate, and resilience. You can increase resilience by including the watermark more than once, but then the bit-rate goes down.

Next, we will look at a number of software watermarking techniques that have been proposed in the past. At the same time we’ll look at the kinds of attacks that can be launched against these techniques.

First we’ll look at static watermarks. Static means the marks are stored permanently in the code or data sections of the program executable.

Static code watermarks are stored in the section of the executable that contains instructions. In Unix this is called the text segment, in Java it’s the method table.

Static data watermarks are stored in other sections of the executable, for example in the string section, or symbol table section, etc.
Static Data Watermarks — DICE Method

class Main {
    static Picture C = ...
    Code R = Decode(C);
    Execute(R);
}

- Data watermarks are very common since they are easy to construct and recognize.
- Easiest is of course to store the copyright notice into a character string. An alternative is to embed it into an image (or audio-clip) using a media watermarking algorithm. We then store this image in the static data section of the program.
- Bob can attack the media object with some distortive transformation, or break up the image in small enough pieces that the complete watermark cannot be found in any individual piece.
- Obfuscating transformations that can be applied to static data. For example, Bob can attack a watermark stored in a string by converting the string into a program that produces that string.

You can tamperproof against some types of attacks. An essential part of the program can be encoded into the image itself. Any standard attack against media watermarks (blurring, compression, etc) will also destroy an important part of the executable code. This technique doesn’t work well with Bondage-and-Dicipline languages like Java where it’s hard to generate and execute code on-the-fly, and impossible to do it stealthily.

Static Code Watermarks

- $n$-cases $\Rightarrow C(m \log m)$ watermarking bits.
- Knowski et al.: Store watermarks in the register allocation.
- We've seen that many media watermarking algorithms store marks in redundant bits. These are bits that we cannot detect because of limits of our visual or auditory systems.
- The same is actually true of code. Executable code contains redundant bits which can be used to store watermarks.
- For example, we can reorder the branches of a case-statement. This allows us to encode $O(m \log m)$ watermarking bits.
- And, we can even reorder statements, provided we make sure that dependencies are not violated.
- Kirowski, Qu, Potkonjak: Store the watermark in the register allocation.

Microsoft has patented a similar method where a software serial number is encoded in the basic block sequence of a program's control flow graphs.

The problem with all these code-based methods is that they are really easy to attack. The Microsoft method, for example, can be destroyed by randomizing the order of basic blocks. Or, better still, by adding your own watermark which then destroys the original one.
• Just optimizing the code is likely to destroy many code watermarks.

• Or, we can obfuscate the code. For example, we can destroy the flow-of-control by inserting bogus predicated branches which break up basic blocks.

• If obfuscation is too expensive, inlining and outlining, all kinds of loop transformations, and other optimizations will easily destroy static code watermarks.

• Tamperproofing code watermarks seems to be hard. This is particularly true for Java, since Java programs can’t (stealthily) inspect their own code.

• Static watermarks are simple, have decent bit-rates, they’re stealthy, but they’re not resilient against attack.

Slide 10C–11

- Dynamic watermarks are stored in a program’s execution state, rather than in the program code itself. This makes (some of) them easier to tamperproof against obfuscating transformations.

- A dynamically watermarked program is run with a special input sequence which makes it enter a state which then represents the watermark.

- **Easter Egg watermark** the special input sequence (the watermark key) triggers some unexpected behavior. Typically, some special image is displayed.

- **Data Structure watermark** a special internal variable holds the watermark.

- **Execution Trace watermark** the watermark is encoded in the instructions that are executed for this special input.

Slide 10D–1

Slide 10D–2
Easter Egg Watermarks

- The watermark performs an action that is immediately perceptible.
- Extraction is trivial.
- Effects must not be too subtle.
- Easter Egg watermarks perform some action that is immediately perceptible by the user. The nice thing about this technique is that watermark extraction becomes is trivial.
- Easter Eggs are simple to locate in the (decompiled) code. This is particularly true in languages like Java where to draw something you have to use standard library functions. There are even several web-site repositories of such watermarks.
- If we make the watermark really subtle it will be difficult to argue that it really is a watermark and not the consequence of bugs or random programmer choices. If we make the watermark really obvious, it will be easy to find in the code.
- If we can find the code that triggers the watermark we can probably also remove it.

Data Structure Watermarks keep the effects of the watermark internal to the program.

- You run the watermarked program with the special input, and that makes a particular program variable hold a particular value, and that is your watermark.
- The main problem is recognizing the watermark. We’re going to have to look inside the running program to extract this watermark value from some program variable.
- We will need a special program, a recognizer, that examines the running program and extracts the watermark. This is both the strength and the weakness of this method.
1. It’s an advantage because this recognizer is not shipped with the application. So there is no way for Bob to locate the watermarking code by just looking at the decompiled code.

2. It’s a disadvantage because we’re going to have to write this watermark recognizer, and that might prove to be difficult.

- Unfortunately, data structure watermarks can also be attacked by obfuscation. For example, a boolean variable can be split into two integer variables.
• We would like is a watermarking method that would allow us to say:
  – This method is definitely resilient against attacks A, B, and C.
  – Or: Any attack against this method will result in the de-watermarked program being too slow or large.
• Data structure watermarks seem to have the greatest survival chances. Our watermark is stored in the topology of a heap-allocated graph; not in node- or edge-data of the graph, since any simple data can easily be obfuscated.
• OO programs have many linked data structures, so the graph-building code will be stealthy. Locating this code automatically will be hard, since pointers are hard to analyse.
• The watermark graphs can be tamperproofed.

**Slide 10E–2**

1. In order to be able to prove that our watermark is actually a watermark introduced on purpose, and not just a fluke, we have to base it on some unique mathematical property. We use the assumption that factoring large integers is hard. Alices elects two large primes $P$ and $Q$, and computes their $P \times Q = n$.

2. She embeds $n$ in the topology of a graph. This graph is her watermark $W$.

3. She converts this watermark graph into a program which builds the graph.

4. This program is embedded into the original program $O$, such that when $O_0$ is run with $I$ as input, $W$ is built. Also, a recognizer program $R$ is constructed, which is able to identify $W$ on the heap, and extract $n$ from it.

5. Alice adds tamperproofing, to prevent Bob from transforming the graph to such an extent that $R$ cannot identify it.

6. The application (including the watermark, tamperproofing code, and recognizer) is obfuscated to prevent a attacks by pattern-matching.

7. The recognizer is removed from the application. $O_3$ is the version of Alice's program that is distributed.

8. Charles links in the recognizer program $R$ with $O_3$.

9. The application is run with $I$ as input, and the recognizer $R$ produces $n$. Since Charles is the only one who can factor $n$, he can prove the legal origin of Alice's program.

**Slide 10E–4**

**Slide 10E–5**
- The main problem with this technique is, of course, the recognition part. There may be a whole slew of structures on the heap, and we will have to be able to find ours. This is no different from the situation in media watermarking, where embedding a mark is easy, but extracting it is hard.

**Slide 10E-6**

- How do we embed a number into the topology of a graph?
- The first method is called **Radix-k**. The graph is simply a circular linked list with an extra pointer field which encodes a base-\( k \) digit. A null-pointer encodes a 0, a self-pointer a 1, a pointer to the next node encodes a 2, etc.
- The bit-rate is very good; we get 4 hidden bits per word.

**Slide 10E-7**

- \( G \) is a circular linked list.
- An extra pointer field encodes a base-\( k \) digit:
- null-pointer \( \iff 0 \)
- self-pointer \( \iff 1 \)
- next node pointer \( \iff 2 \)
- A 255-node list hides 2040 bits \( \Rightarrow \) bit-rate is 4 bits per word.

**Slide 10E-8**

- \( n \) is represented by the index of the graph \( G \) in some enumeration.
- We must, efficiently, be able to generate the \( n \)th graph,
- 1. Given \( n \), find \( G \) index \( n \).
- 2. Oriented parent-pointer trees \( \Rightarrow \)

**Slide 10E-9**

- 1. 655 nodes \( \Rightarrow \) 1024-bit integer.
- 2. bit-rate: 1.56 bits per word.
• Or, to embed a number \( n \) in a graph we simply pick the \( n \)-th graph in some enumeration of all graphs of a particular class.

• This means that we must be able to
  1. generate the \( n \)-th graph in this particular enumeration,
  2. and, conversely, given a graph we must find its index in this particular enumeration.

Of course, this number will be large, so these procedures must be efficient. This, unfortunately, rules out many classes of graphs, because in general graph isomorphism is an intractable problem.

• But for some simple classes of graphs, there exist polynomial time algorithms for finding the index of a graph. Simplest of all, is oriented “parent-pointer” trees.

Slide 10E–10

Slide 10E–11

• To find the watermark graph we would need to examine all reachable heap objects. This, of course, would be intractable. It turns out we can do much better than that.

• First, remember that we have a special input sequence \( \mathcal{I}_1 \) to \( \mathcal{I}_k \) during which this graph has to be built. For stealthiness reasons we build a small part of the graph for each input. When we get to the end of the input sequence, we expect the entire graph to have been built.

• The crucial observation is that to recognize the watermark graph, all we have to do is examine those heap-objects which are allocated during processing of input \( \mathcal{I}_k \). There will only be a small number of such nodes, and we can exhaustively look for the particular node which belongs to our graph.

Slide 10E–12

Slide 10E–13
• None of the obfuscations that we discussed earlier will have any effect on the graph watermark. Bob can optimize the code as much as he wants, he can insert bogus branches, he can even add a level of interpretation.

• There are, however, a small number of other obfuscation techniques he can employ.

  1. He can rename and reorder all the fields of every dynamically allocated object in the entire program. He has to do this uniformly over the whole program because he cannot know which nodes encodes our watermark. He, of course, has to make the same changes to the code which manipulates these objects.

  2. He can add extra pointers to the nodes of linked structures. This will make it hard for the recognizer to identify the real graph within a lot of extra bogus pointer fields. Again, this will have to be done uniformly over the entire program. Bob, of course, has to be careful so that the obfuscated program doesn’t allocate way too much memory as a result.

  3. He can add levels of indirection, for example by splitting nodes into several linked parts. Again, this will cost extra dynamic memory. It could easily double the memory requirements of the program.

  4. Finally, Bob could add extra bogus nodes pointing into our graph, preventing us from finding the root.

---

Slide 10E–14

Slide 10E–15

---

Slide 10E–16

Slide 10E–17

• This is an example of a particularly nasty attack.

• ① shows our original watermark graph.

• In ② Bob has renamed and reordered node pointer fields.

• In ③ each node has received a bogus pointer field B and bogus edges have been added.

• In ④ each node has been split in two by adding a bogus pointer field A.

• Finally, in ⑤ bogus nodes have been allocated which point into the graph, obscuring which node is the root.
- Graph watermarks can be tamperproofed.
- We’d like graphs that are inherently immune to attack. The parent-pointer trees we saw earlier are resilient to renaming and reordering attacks since each node only has one pointer.
- The previous slide shows another type of graph which is resilient to node-splitting attacks:
  1. Each node of our original watermark graph is expanded into a 4-cycle.
  2. Bob splits some nodes. They way the graph is constructed, these bogus nodes must fall on a cycle.
  3. So, during recognition, we shrink the biconnected components of the underlying (undirected) graph. The result is a graph isomorphic to our original watermark graph.

- Some languages (like Java, Icon, and Modula-3) support reflection. This means that it is possible to examine the run-time types of objects. This gives us another possibility for tamperproofing.
- To prevent Bob from adding bogus pointers, we can examine the structure of the graph nodes at run-time. The problem with this approach is, of course, that it is not very stealthy.
- To prevent reordering and renaming attacks we can access watermark pointers through reflection. For example, rather than doing `O.car=`V`, we let car be represented by the first relevant pointer in the node 0. Again, this is not very stealthy.
A planted plane cubic tree on 8 nodes.
- Bit-rate is 0.5 bits per word.
- Planarity check:
  - For each internal node \( x \), the left-most child of \( x \)'s right subtree is \( L \)-linked to the right-most child of \( x \)'s left subtree.

- We have assumed that all attacks preserve the semantics of the watermarked program. This is reasonable, since if Bob has no idea where the watermark is, his best shot is to apply transformations indiscriminately all over the place.

- Of course, if Bob has some idea where the watermark graph is being built, he can do better than that, for example by inserting extra nodes or edges. Bob has to be careful not to destroy the program itself along with the watermark.

- Planted plane cubic trees is a class of graphs, that allows us to perform some consistency checks to make sure that it hasn't been tampered with. Occasionally, we will test the planarity of the graph to make sure that Bob hasn't added any bogus edges. This can be done cheaply.

---

University of Arizona

CS 620
Language-based Approaches to System and Software Security

A Mathematical Model

\( F \)

Copyright © 1999 C. Collberg

—

A Model of Software Watermarking

**Definition 1 (Software Watermark)** Let \( \mathcal{W} \) be a set of mathematical structures, and \( p \) a predicate such that

\[ \mathcal{W} \ni x \Rightarrow \exists \mathcal{W}_x \text{ s.t. } p(\mathcal{W}_x) \text{ holds} \]

for any \( x \in \mathcal{W} \). We choose \( p \) and \( \mathcal{W} \) such that the probability of \( p(x) \) for a random \( x \notin \mathcal{W} \) is small.
**Software Watermarking: The Recognizer**

**Definition 4 (Watermark Recognizer)** \( \mathcal{R}_T(P_w, S(P_w, I)) \) is a recognizer of \( w \in W \) in \( P_w \in \mathcal{P} \) with input \( I \) wrt a set of transformations \( T \subset \mathcal{T} \), if,

\[ \forall t \in T : p(\mathcal{R}(t(P_w), S(P_w, I))) = p(w) \]

- \( \mathcal{R}_\emptyset(P_w, S(P_w, I)) \) is the trivial recognizer.
- \( \mathcal{R}_T(P_w, \emptyset) \) is a static recognizer.
- \( \mathcal{R}_T(\emptyset, S(P_w, I)) \) is a pure dynamic recognizer.
- \( \mathcal{R}_{\text{sem}}(P_w, S(P_w, I)) \) is a strong recognizer.

**Software Watermarking: Coding Efficiency**

**Definition 5 (Watermark Coding Efficiency)**

\( H(w) = \log_2 |W| \) is the entropy of \( w \), in bits, when \( w \) is drawn with uniform probability from \( W \).

Let \( |P|, P \in \mathcal{P} \) be the size (in words) of \( P \) as expressed in some encoding.

Let \( |S(P)| = \max_{I \in \text{dom}(P)} |S(P, I)| \) be the least upper bound on the size of \( P \).

An embedding of \( P_w \) of \( w \) in \( P \) has a high static data rate if

\[ \frac{H(w)}{|P_w| - |P|} \geq 1 \].

An embedding \( P_w \) of \( w \) in \( P \) has a high dynamic data rate if

\[ \frac{H(w)}{|S(P_w)| - |S(P)|} \geq 1 \].

**Software Watermarking: Programs**

**Definition 2 (Programs)** Let \( \mathcal{P} \) be the set of programs. \( P_w \) is an embedding of a watermark \( w \in W \) into \( P \in \mathcal{P} \).

Let \( \text{dom}(P) \) be the set of input sequences accepted by \( P \). Let \( \text{out}(P, I) \) be the output of \( P \) on input \( I \).

Let \( S(P, I) \) be the internal state of program \( P \) after having processed input \( I \). Let \( |S(P, I)| \) be the size of this state, in accessible words.

**A Model of Watermarking: Transformations**

**Definition 3 (Program Transformations)** Let \( \mathcal{T} \) be the set of transformations from programs to programs.

\( \mathcal{T}_{\text{sem}} \subset \mathcal{T} \) is the set of semantics preserving transformations:

\[ \mathcal{T}_{\text{sem}} = \{ t : \mathcal{T} \mid P \in \mathcal{P}, I \in \text{dom}(P), \text{dom}(P) = \text{dom}(t(P)), \text{out}(P, I) = \text{out}(t(P), I) \} \].

Similarly, \( \mathcal{T}_{\text{stat}} \subset \mathcal{T} \) is the set of state preserving transformations:

\[ \mathcal{T}_{\text{stat}} = \{ t : \mathcal{T} \mid P \in \mathcal{P}, I \in \text{dom}(P), S(P, I) = S(t(P), I) \} \].
An ideal watermark should have these properties. Alice can embed the watermark, a number \( n \), into a program \( P \) such that:

- a) regardless of what attack Bob launches on the watermarked program, Alice can still extract the watermark;
- b) the embedding is imperceptible;
- c) the watermarked program is as efficient as the original program;
- d) if Bob were able to destroy the watermark then the resulting program should be defective, or at least much to slow or large to be of any use to him;
- e) the watermark should be large.
- Static code watermarks can be pretty secure against subtractive attacks, such as ripping out an interesting module from the program. Obviously, they’ll have to be included multiple times, in all parts of the program. These watermarks are stealthy, but they can’t handle semantics-preserving attacks; they aren’t even secure against optimization.

- Easter Egg watermarks are secure against semantics-preserving attacks, but they can’t handle distortive attacks, and they are very unstealthy.

- The Dice watermark aren’t disturbed by optimization and many types of obfuscation. They can handle image transforms insofar as the media watermark they use can handle it. The tamperproofing they use (extracting a piece of code from the image and jumping to it) is pretty unstealthy.

- Execution trace watermarks are unstealthy, but don’t handle even optimization and certainly not distortive attacks.

- The graph watermarks, are very secure against a range of semantics-preserving transformations: you can optimize the program, you can insert bogus code, you can even add a layer of interpretation, and still, the watermark is intact. It’s also very stealthy in programs that use a lot of dynamic allocation. But, it doesn’t handle subtractive attacks very well.

- Clearly, there is a lot of room for both theoretical work and practical experimentation. We need to find out exactly what types of attack we should worry about most, we need to find out what kinds of tools an adversary have to their disposal, etc.