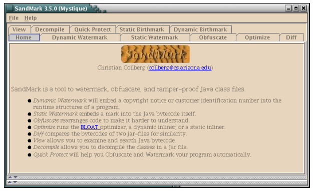


Software Watermarking

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

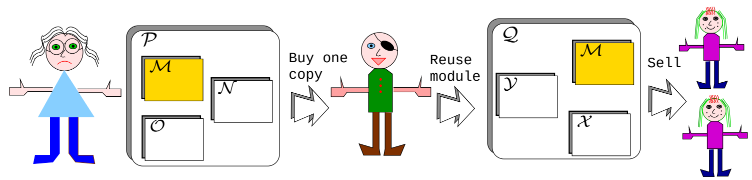
Software Watermarks & Fingerprints

- Embed a unique identifier in a program to trace software pirates.

- Watermarking
  1. discourages theft,
  2. allows us to prove theft.
- Fingerprinting
  3. allows us to trace violators.

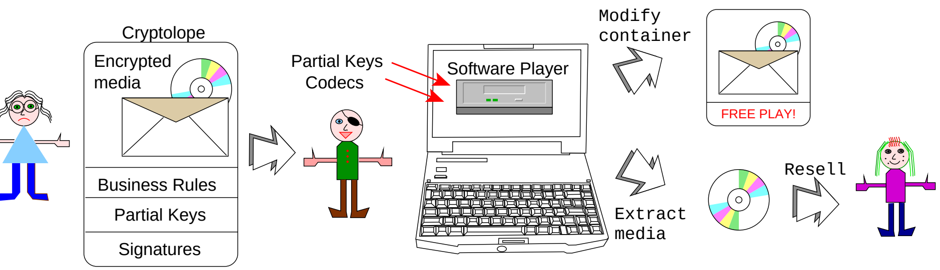
- We want to develop algorithms that produce marks that are hard to destroy, are stealthy, have a high bit-rate, and have little performance overhead.

Malicious Reverse Engineering



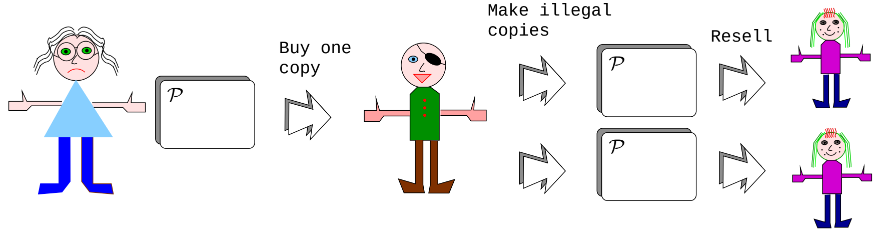
- Alice and Bob are competing software developers.
- Bob reverse engineers Alice's program and includes parts of it in his own code.
- Easier with Java bytecode, .NET, ANDF...
- ⇒ Alice obfuscates her code.

Tampering



- Alice is a media publisher. She packages her media into a cryptolope.
- Bob tampers with the software player to extract the decrypted media.
- InterTrust, Intel, IBM, Xerox, Microsoft,. . . .
- $\Rightarrow$ Alice obfuscates, watermarks, tamper-proofs the player.

Software Piracy



- Alice is a software developer.
- Bob buys one copy of Alice’s application and sells copies to third parties.
- $\Rightarrow$ Alice watermarks/fingerprints her program.

Software Protection Overview

Software Watermarking Overview

Static Software Watermarking Algorithms

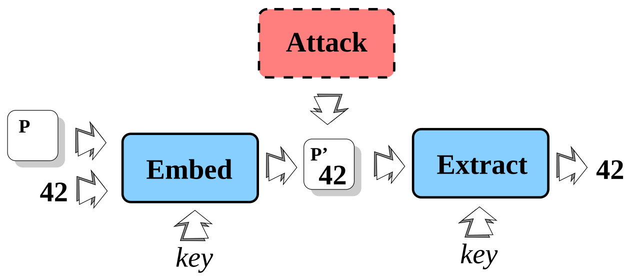
Attacks on Software Watermarks

Dynamic Software Watermarking

The SANDMARK tool

Conclusion

Software Watermarking



Embed an integer W in program P such that

- W is resilient against automated attacks
- W is stealthy
- W is large (high bitrate)
- the overhead (space and time) is low

Naive Approaches

```
String watermark = "Copyright␣2004...";
```

- High bitrate, little overhead, unstealthy.

```
switch (E) {
    case 1: {···}
    case 5: {···}
    case 9: {···}
}
```

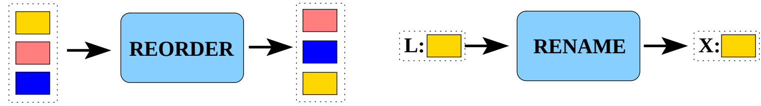
⇒

```
switch (E) {
    case 5: {···}
    case 1: {···}
    case 9: {···}
}
```

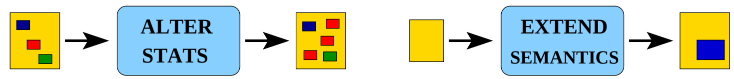
- Low bitrate, no overhead, stealthy, easy to destroy.

Watermarking Transformations

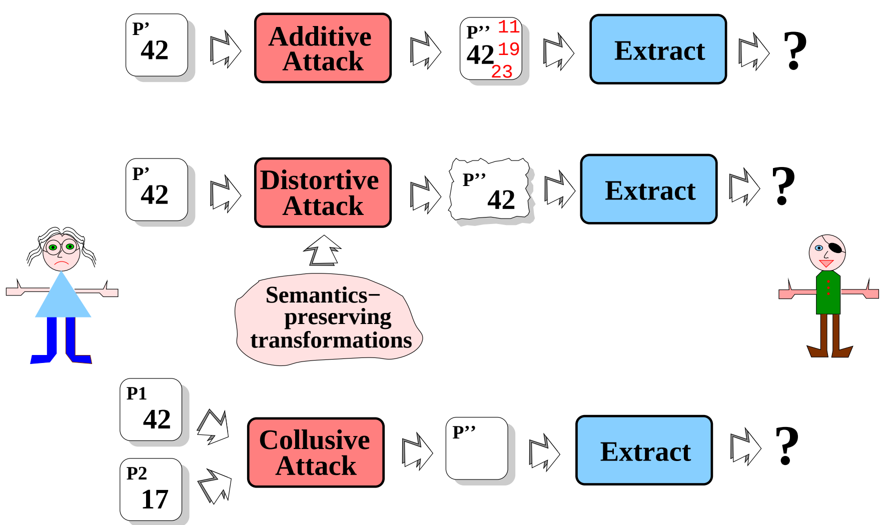
- Naive approaches typically use reordering (of statements, basic blocks, ...) or renaming (of registers, methods, ...):



- More powerful approaches extend program semantics or alter program statistics:



Attacks on Software Watermarks



Software Protection Overview

Software Watermarking Overview

Static Software Watermarking Algorithms

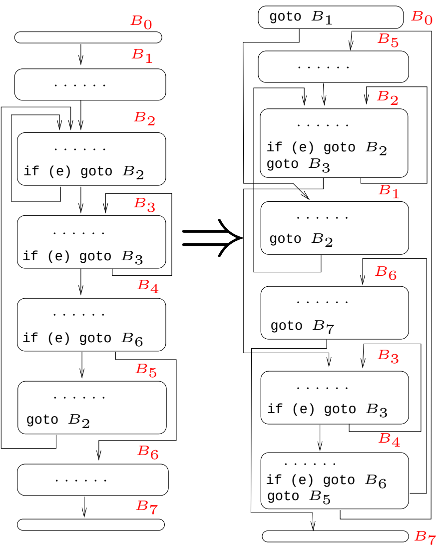
Attacks on Software Watermarks

Dynamic Software Watermarking

The SANDMARK tool

Conclusion

— Davidson & Myhrvold

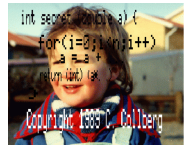


- The watermark is encoded in the basic block sequence $\langle B_5, B_2, B_1, B_6, B_3, B_4 \rangle$.

US Patent 5,559,884, 1996, Microsoft

— Moskowitz & Cooperman

```
class Main {
  const Picture C =
```

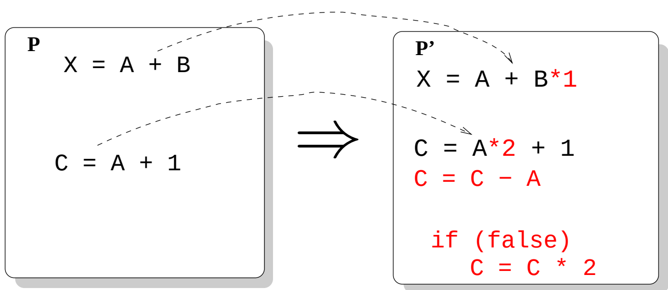


```
       ...
  Code R = Decode(C);
  Execute(R);
}
```

- A watermarked media object is embedded in the program's static data segment.
- "Essential" parts of the program are steganographically encoded into the media.
- If the watermarked image is attacked, the embedded code will crash.

US Patent 5,745,569, Jan 1996.

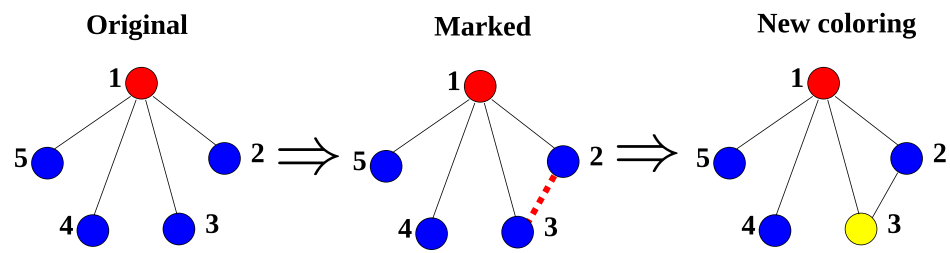
— Stern et al.



- Embed mark by adjusting frequency of instruction patterns:
  1. Replace instruction groups by semantic equivalents.
  2. Insert redundant instruction groups.

3rd International Information Hiding Workshop, 1999.

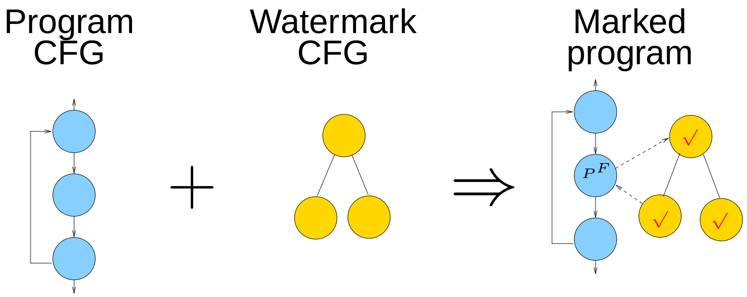
— Qu-Potkonjak



- Embed the mark by adding constraints (extra edges) to the register interference graph.
- Easy to attack by random register re-numbering.

3rd International Information Hiding Workshop, 1999.

— Venkatesan et al.



- Bogus branches tie the watermark CFG to the program.
- Basic blocks are marked so the watermark graph can be found.

4th International Information Hiding Workshop, 2001.

— Cousot and Cousot

```
int n = ...;
int a = 0, b = 1;
for(int i = 1; i < n; i++){
   int c = a+b;
   a = b;
   b = c;
}
```

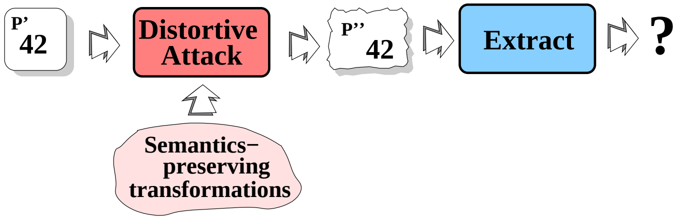
$\Rightarrow$

```
int n = ...
int a = 0, b = 1;
int d = 1, e = 35538, f = 1, g = −111353;
e = d ∗ e; d = e + 11445; g = f ∗ g;
f = g − 47305;
for(int i = 1; i < n; i++){
   int c = a+b; e = d ∗ 658; f = f ∗ 4;
   a = b;
   f = g + 1566; e = e + 971;
   g = g ∗ f; e = e ∗ d;
   b = c;
   d = e + 4623; f = g + 21494;
}
```

- Embed the mark in the result of a static analysis problem.
- Algorithm introduces many "weird" constants. Unstealthy, since 92% of all literal integers are 2^n, $2^n + 1$, $2^n - 1$.

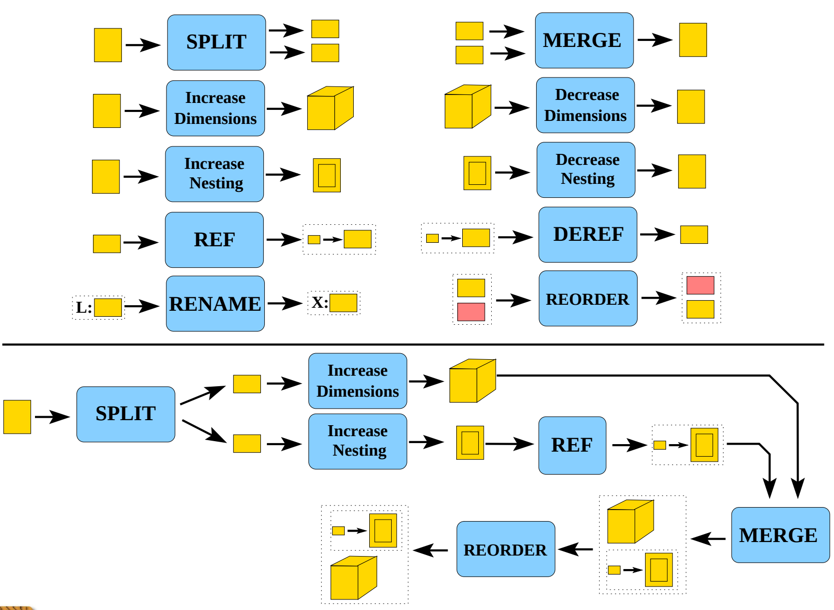
ACM Principles of Programming Languages, POPL'04

Semantics-Preserving Attacks



- Code optimization, decompile-recompile, translation, code obfuscation,. . . .
- Our SANDMARK tool relies on combining sequences of simple obfuscating and optimizing transformations.

Obfuscating Transformations



Original Code

```
public class C {
    static int gcd(int x, int y) {
        int t;
        while (true) {
            boolean b = x % y == 0;
            if (b) return y;
            t = x % y; x = y; y = t;
        }
    }
    public static void main(String[] a){
        System.out.print("Answer:␣");
        System.out.println(gcd(100,10));
    }
}
```


Boolean Splitting Obfuscation

```
public class C {
 static int gcd(int i, int j) {
  int t8, t7 , k;
  for (;;) {
    if (i%j==0) { t8=1;t7=0; }
    else        { t8=0;t7=0; }
    if ( (t7^t8)!=0 )
      return j;
    else {
      k=i%j; i=j; j=k;
    }
  }}
 public static void main(String[] Z1) {
  System.out.print("Answer:␣");
  System.out.println(gcd(100, 10)); }
}
```


Bogus Branch Obfuscation

```
public class C {
  static int gcd(int i, int j) {
    int t9, t8, q7, q6, q4, q3;
    q7=9;
    for (;;) {
      if (i%j==0) {t9=1;t8=0;} else {t9=0;t8=0;}
      q4=t8; q6=t9;
      if ((q4^q6)!=0)
        return j;
      else {
        if ( (((q7+q7*q7)%2!=0)?0:1)!=1 ) return 0;
        q3=i%j; i=j; j=q3;
      }
    } }
  public static void main(String[] Z1) {
    System.out.print("Answer:␣");
    System.out.println(gcd(100, 10)); }
}
```

String Encoding Obfuscation

```
public class C {
  static int gcd(int i, int j) {
     // As before
  }
  public static void main(String[] a){
    System.out.print(
       Obfuscator.DecodeString(    // Rename this!
          "\u00AB\u00CD\u00AB\u00CD"+
          "\uFF84\u2A16\u5D68\u2AA0"+
          "\u388E\u91CF\u5326\u5604"));
    System.out.println(gcd(100, 10)); }
}
```


Primitive Promotion Obfuscation

```
public class C {
 static Integer get0(Integer i,Integer j){
  Integer K, L, M, N; int t9, t8; K=new Integer(9);
  for (;;) {
   if (i.intValue()%j.intValue()==0){t9=1;t8=0;}else{t9=0;t8=0;}
   M=new Integer(t8);L=new Integer(t9);
   if ((M.intValue()^L.intValue())!=0)
     return new Integer(j.intValue());
   else {
    if ((((K.intValue()+K.intValue()*K.intValue())%2!=0)?0:1)!=1)
     return new Integer(0);
    N=new Integer(i.intValue()%j.intValue());
    i=new Integer(j.intValue());  j=new Integer(N.intValue());
 }}}
 public static void main(String[] Z1) {
    System.out.print(Obfuscator.get0(
       (String)new Object[] {"String␣as␣before"}[0]));
    System.out.println(get0((Integer)new Object[] {
       new Integer(100),new Integer(10)}[0],
       (Integer)new Object[] {
       new Integer(100),new Integer(10)}[1]).intValue());
}}
```

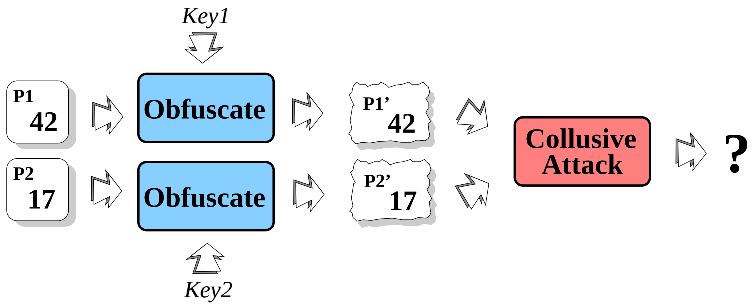

Method Signature Obfuscation

```
public class C {
 static Object get0(Object[] I) {
  Integer K, L, M, N; int t9, t8; K=new Integer(9);
  for (;;) {
   if (((Integer)I[0]).intValue()%((Integer)I[1]).intValue()==0)
      {t9=1; t8=0;} else {t9=0; t8=0;}
   M=new Integer(t8);L=new Integer(t9);
   if ((M.intValue()^L.intValue())!=0)
     return new Integer(((Integer)I[1]).intValue());
   else {
     if ((((K.intValue()+K.intValue()*K.intValue())%2!=0)?0:1)!=1)
      return new Integer(0);
     N=new Integer(((Integer)I[0]).intValue() %
                 ((Integer)I[1]).intValue());
     I[0]=new Integer(((Integer)I[1]).intValue());
     I[1]=new Integer(N.intValue());
 }}}
 public static void main(String[] Z1) {
    System.out.print(
       (String)Obfuscator.get0(new Object[] {(String)new Object[]
          {"String␣as␣before" }[0] }));
    System.out.println(((Integer)get0(new Object[]
         {(Integer)new Object[] {new Integer(100),
          new Integer(10)}[0], (Integer)new Object[] {
            new Integer(100), new Integer(10) }[1]})).intValue());
}}
```


This is what we started out with. . .

```
public class C {
   static int gcd(int x, int y) {
      int t;
      while (true) {
         boolean b = x % y == 0;
         if (b) return y;
         t = x % y; x = y; y = t;
      }
   }
   public static void main(String[] a){
      System.out.print("Answer:␣");
      System.out.println(gcd(100,10));
   }
}
```

Collusion Protection by Obfuscation

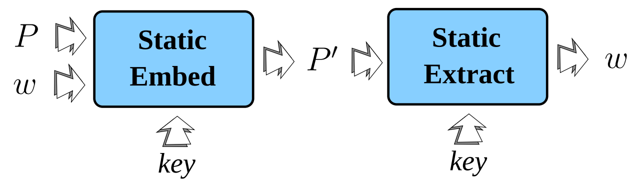


- Obfuscation can also be used to protect against collusive attacks.

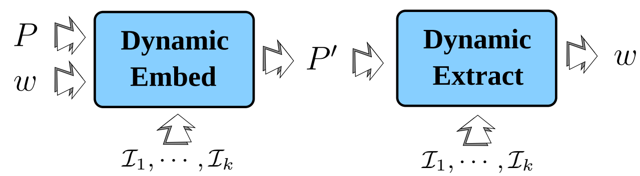
Collusion Protection by Obfuscation

```
public class C {
  static Object get0(Object[] I) {
    Integer K, J, M, N; int r, q, j; K=new Integer(9);
   j=2; j=60-(j+1); ++j; j=60-j;
    for (;;) {
      if (((Integer)I[0]).intValue()%((Integer)I[1]).intValue()==0)
          {r=1; q=0;} else {r=0; q=0;}
      M=new Integer(q); J=new Integer(r);
      if ((M.intValue()^J.intValue())!= 0)
        return new Integer(((Integer)I[1]).intValue());
      else {
        if ((((K.intValue()+K.intValue()*K.intValue())%2!=0)?0:1)!=1)
          return new Integer(0);
        N=new Integer(((Integer)I[0]).intValue()%
                  ((Integer)I[1]).intValue());
        I[0]=new Integer(((Integer)I[1]).intValue());
        I[1]=new Integer(N.intValue());
  }}}
  public static void main(String[] Z1) {
   int j=2; int i=2; i=80-(i+1); j=80-(j+1);
    System.out.print((String)Obfuscator.get0(new Object[] {
       (String)new Object[] { "String as before" }[0]}));
   ++i; i=80-i; ++j; j=80-j;
    System.out.println(((Integer)get0(
       new Object[] { (Integer)new Object[] {
                  new Integer(100), new Integer(10) }[0],
            (Integer)new Object[] {
                  new Integer(100), new Integer(10) }[1]
          })).intValue());
}}
```


Static vs. Dynamic Watermarking

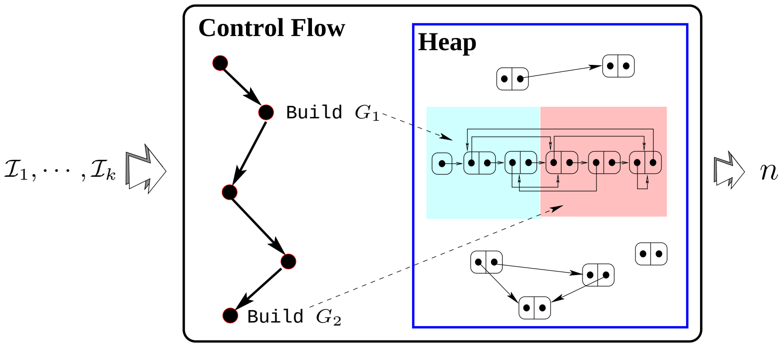


- Static algorithms are vulnerable to semantics-preserving code transformations.



- Dynamic algorithms extract the mark from the state of the program when run on a secret key input sequence.

— Collberg-Thomborson



- The watermark is embedded in the topology of a dynamic graph structure, built at runtime but only for the special input sequence $\mathcal{I}_1, \cdots, \mathcal{I}_k$.
- Why? Shape-analysis is hard.

ACM Principles of Programming Languages, POPL'99

CT — Example

```
public class Simple {
   static void P(String i) {
      System.out.println("Hello␣" + i);
   }
   public static void main(String args[]) {
      P(args[0]);
   }
}
```

⇓

```
class Watermark extends java.lang.Object {
   public Watermark edge1,edge2;
}
```

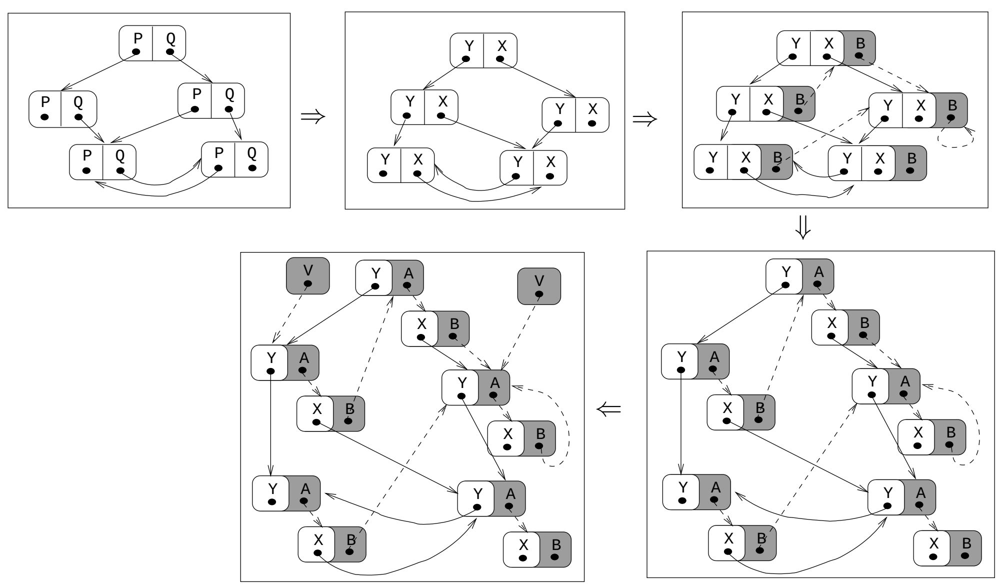
⇓

CT — Example...

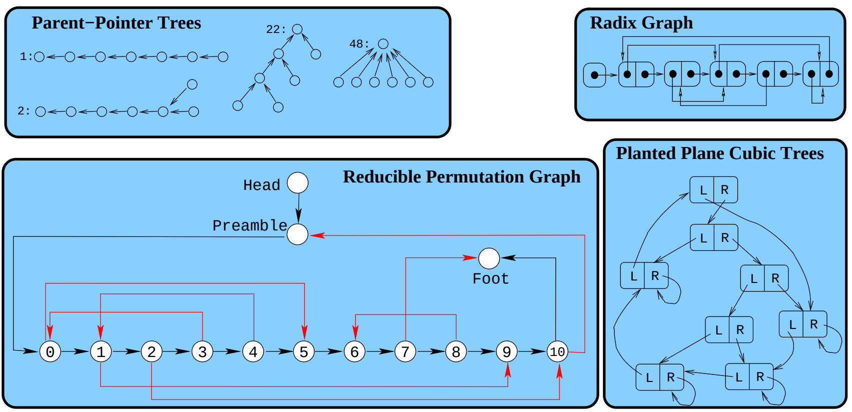
```
public class Simple_W {
   static void P(String i, Watermark n2) {
      if (i.equals("World")) {
         Watermark n1 = new Watermark();
         Watermark n4 = new Watermark();
         n4.edge1 = n1; n1.edge1 = n2;
         Watermark n3 = (n2 != null)?n2.edge1:new Watermark();
         n3.edge1 = n1;
      }
      System.out.println("Hello␣" + i);}

   public static void main(String args[]) {
      Watermark n3 = new Watermark();
      Watermark n2 = new Watermark();
      n2.edge1 = n3; n2.edge2 = n3;
      P(args[0], n2);
}
```


CT — Semantics-Preserving Attacks



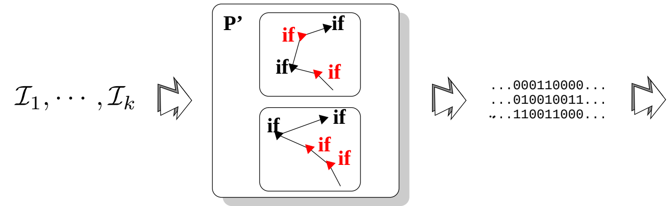
CT — Error-Correcting Graphs



- Current work: Define classes of graphs with efficient embedding and error-correcting properties.

Collberg et al., Workshop on Graphs in Computer Science 2003.

EXTEND SEMANTICS — Path-Based Watermarking



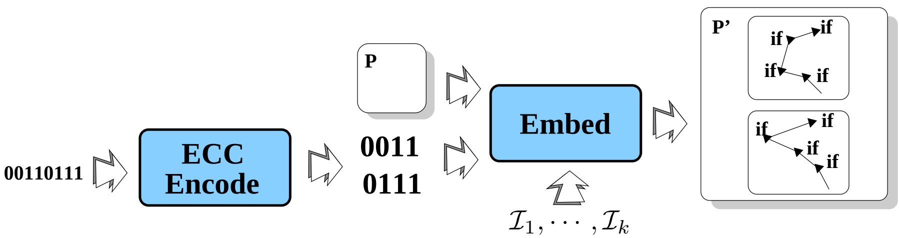
- The branches executed for the secret input generate a stream of 0s and 1s from which the watermark is extracted.
- An attacker can easily insert new branches:

  Java ⇒ Use an Error Correcting Code

  x86 ⇒ Tamper-proof the branches

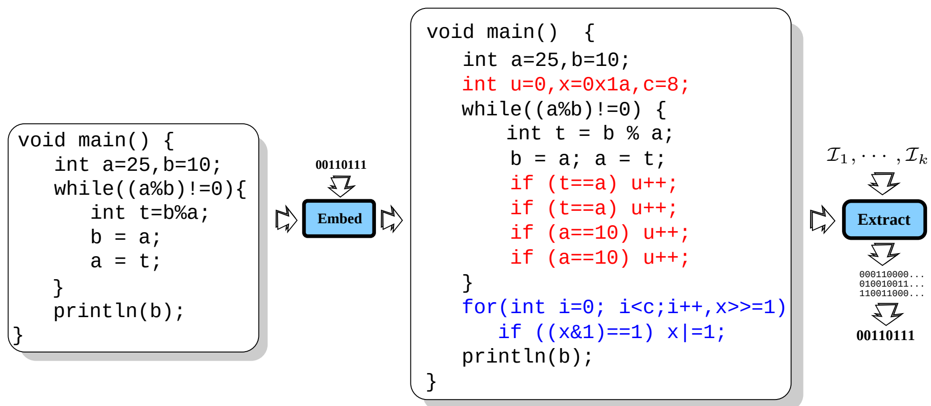
Collberg et al., ACM PLDI'04

PBW — Embedding



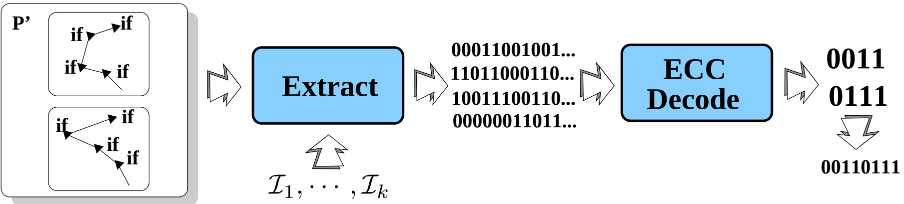
- The watermark is split into a large number of redundant pieces using an error correcting code.
- Each piece is individually embedded in the program.
- We want to be able to lose some pieces and still recover the watermark.

PBW — Code Generation



- Several different types of code can be generated to increase stealth.
- Ensure to protect against simple branch-flips!

PBW — Extraction



- The program is run with the secret input.
- Branches are monitored and a bitstream extracted.
- Using the error correcting code, the watermark pieces are extracted from the bitstream and recombined into the watermark.

PBW — ECC Encode

$p_1 = 2, p_2 = 3, p_3 = 5$

$$W = 17 \Rightarrow \begin{array}{lll} W \equiv 5 \bmod p_1p_2 & & 5 = 5 \\ W \equiv 7 \bmod p_1p_3 & \Rightarrow & p_1p_2 + 7 = 13 \\ W \equiv 2 \bmod p_2p_3 & & p_1p_2 + p_1p_3 + 2 = 18 \end{array} \Rightarrow \begin{array}{l} \overbrace{11\cdots01}^{64} \\ \overbrace{01\cdots11}^{64} \\ \overbrace{10\cdots00}^{64} \end{array}$$

- Choose $p_1, \dots, p_k$ pairwise relatively prime, split watermark into $\frac{k(k-1)}{2}$ pieces of the form $W \equiv r \bmod p_ip_k$.
- Use an enumeration scheme to turn these into integers, run through a block-cipher, embed into program.
- The Generalized Chinese Remainder Theorem allows W to be reconstructed from $\lceil \frac{k}{2} \rceil$ pieces.

PBW — ECC Decode

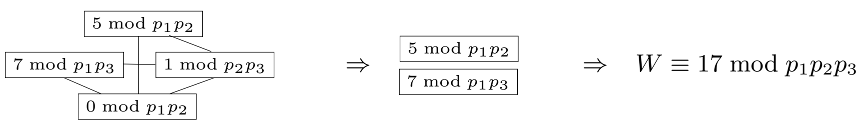
- Slide a 64-bit window across the bitstream. Throw out those that don't meet randomness criteria.

  ...1011100110100000000000110100010010...

- Reconstruct $W \equiv r \bmod p_ip_k$ by inverting enumeration scheme.

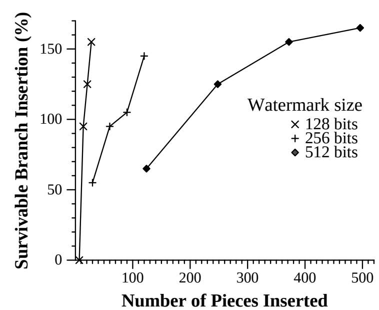
$$\begin{array}{l} 11\cdots01 \\ 01\cdots11 \\ 10\cdots00 \\ 10\cdots11 \end{array} \Rightarrow \begin{array}{l} 5 \\ 13 \\ 17 \\ 0 \end{array} \Rightarrow \begin{array}{l} W \equiv 5 \bmod p_1p_2 \\ W \equiv 7 \bmod p_1p_3 \\ W \equiv 1 \bmod p_2p_3 \\ W \equiv 0 \bmod p_1p_2 \end{array}$$

- Build a graph of statements inconsistent wrt to GCRT. Compute "most consistent" subgraph.



PBW — Adding Branches Attack

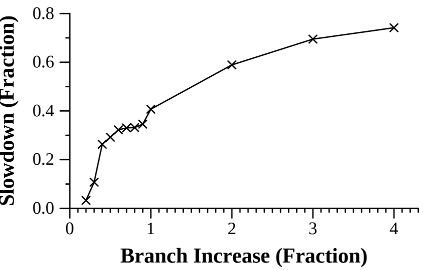
- Attack model: the attacker randomly adds bogus conditional branches to the program.
- The more pieces we add, the more pieces an attacker has to destroy in order to destroy the watermark



- With a 256-bit mark and 100 pieces, the attacker needs to double the number of branch instructions in the program in order to destroy the mark.

PBW — Adding Branches Attack

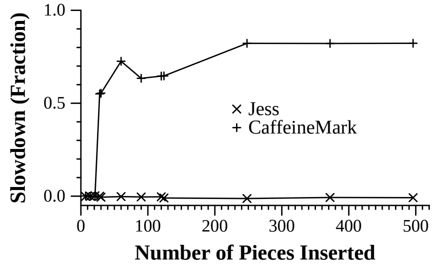
- How much does CaffeineMark slow down versus wrt the number of branches the attacker added?



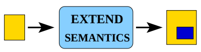
- By doubling the number of branches, the attacker slows down the program by about 40%.

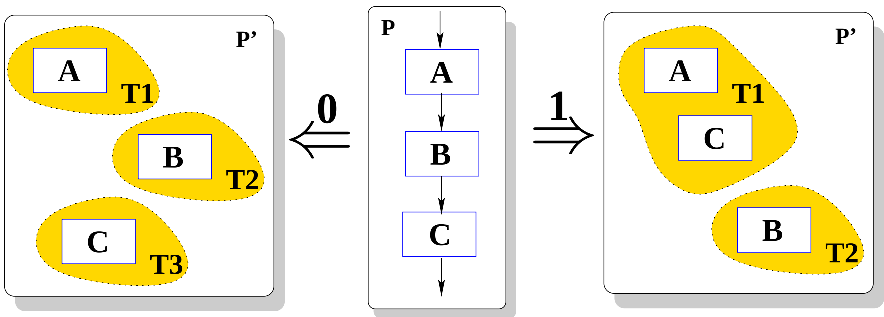
PBW — Time Overhead

- How does the program slow down as the number of watermark pieces is increased?
- The more pieces we insert, the more pieces the attacker needs to destroy.



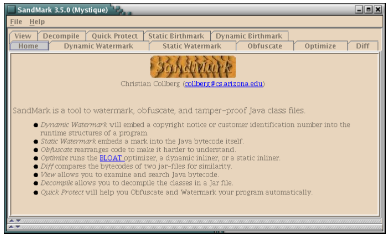
- For Jess we avoid the hotspots, so slowdown is negligible.
- For CaffeineMark we can't avoid the hotspots, so slow-down is $> 50\%$.

 — Nagra-Thomborson



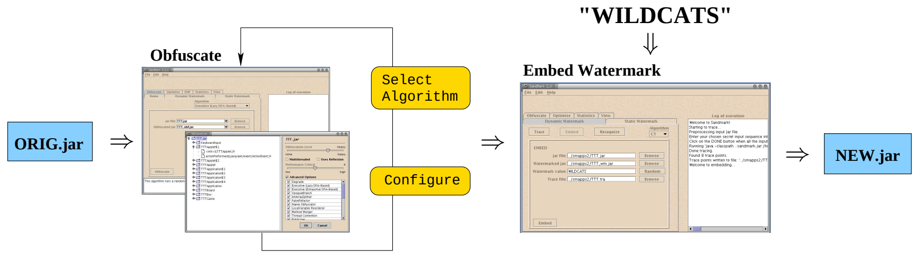
- Embed mark in which threads execute which basic blocks.
- Can have huge performance degradation.
- Why? Parallelism-analysis is hard.

ANDMARK — A Software Protection Tool



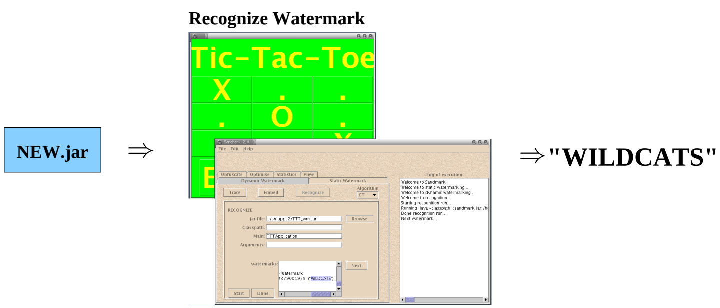
- 33 obfuscation algorithms
- 16 watermarking algorithms
- 6 birthmarking algorithms
- 6 bytecode diff algorithms
- bytecode visualization tools
- 6 software complexity metrics
- large toolbox (static analysis, graphs,. . .)

A Session with SANDMARK



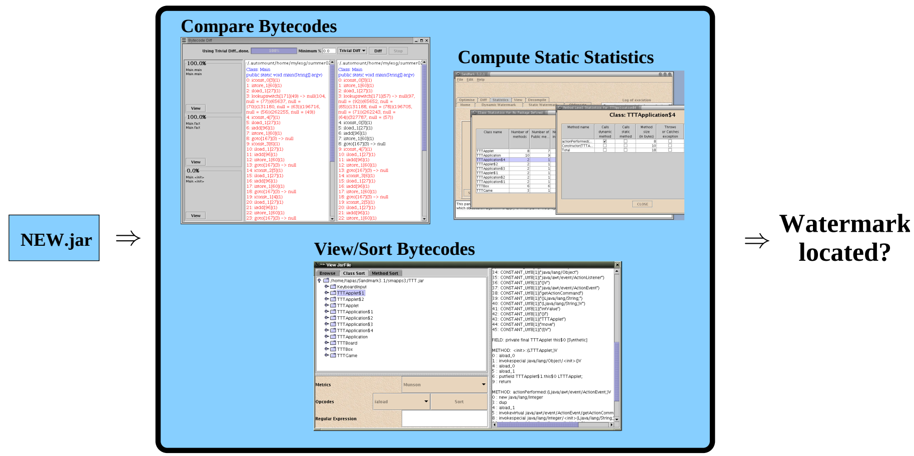
- We obfuscate to protect against reverse engineering and collusive de-watermarking attacks.

A Session with SANDMARK



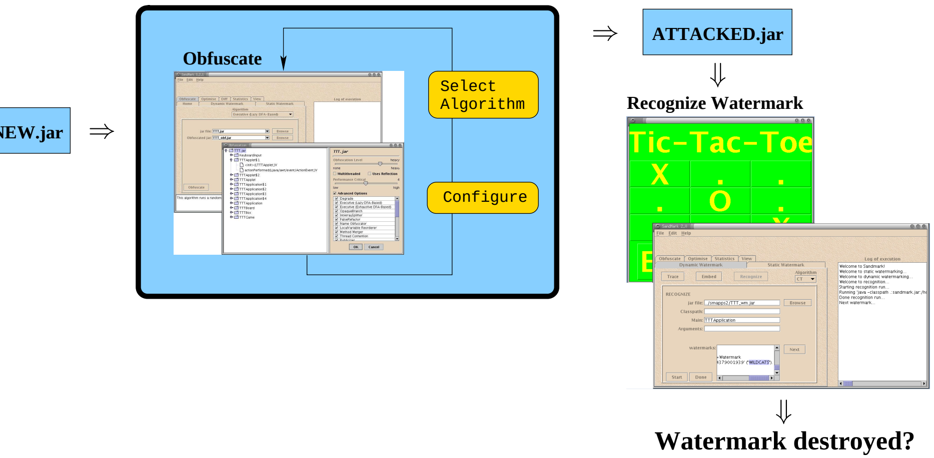
- We extract the watermark to prove ownership.

A Session with SANDMARK



- To simulate a manual attack we examine the obfuscated/watermarked program using various static analysis tools.

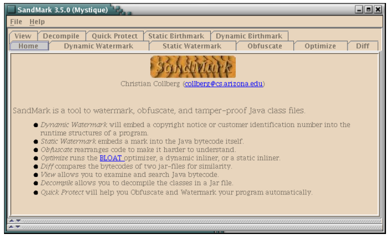
A Session with SANDMARK



- To simulate an automatic attack we use SANDMARK's obfuscators ("*SoftStir*") to attack the watermark.

Conclusion

- Many interesting problems left to work on!
  - Formal models of attack and stealth.
  - Combining error correction and tamper-proofing.
  - Watermarking other languages.



- Download from `sandmark.cs.arizona.edu`.