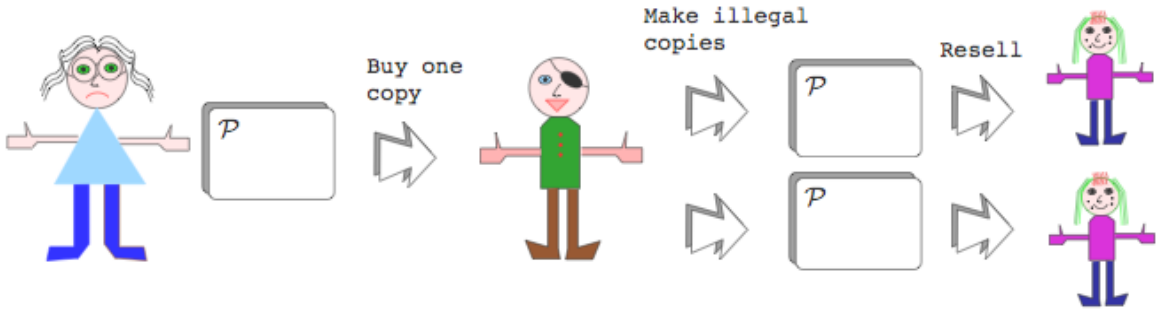
SW Watermarking

SW Piracy



- Alice is a software developer
- Bob buys one copy of Alice's application and sells copies to third parties
- Alice watermarks or fingerprints her program in order to protect the intellectual property
- Watermark: uniquely identifies the owner
- Fingerprints: uniquely identifies the owner and the purchaser

Watermark Evaluation

- Credibility: it should be easy to encode authorship minimizing the probability of coincidence with other messages
- Data-rate: maximize the length of messages that can be embedded
- Perceptual invisibility (stealth): the watermark should exhibit the same properties as the code around it
- Part protection: the watermark should be distributed uniformly throughout the code protecting all of its parts

Watermark Evaluation

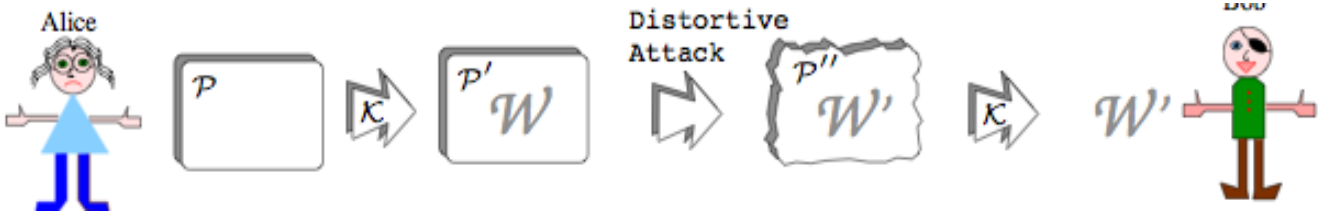
* Resilience: withstanding a variety of attacks
  * Subtractive Attack: the adversary tires to remove all or part of the watermark

Watermark Evaluation

* Resilience: withstanding a variety of attacks
  * Additive Attack: the adversary adds a new watermark

Watermark Evaluation

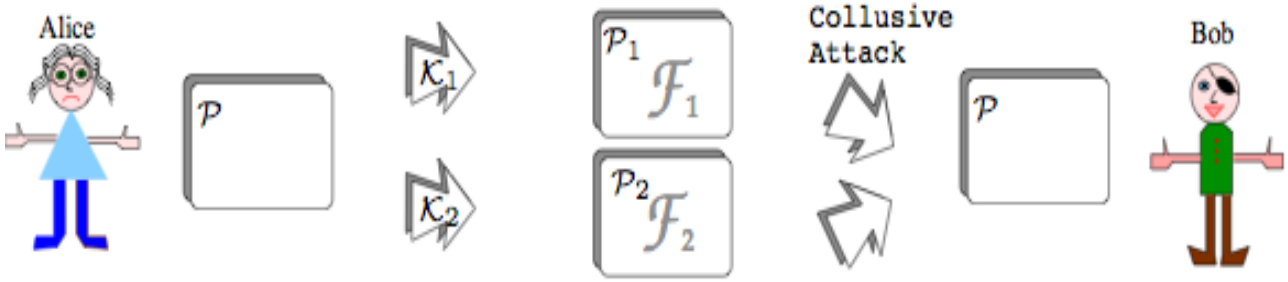
* Resilience: withstanding a variety of attacks
  * Distortive Attack: the adversary applies semantics-preserving transformations to destroy the watermark



Watermark Evaluation

* Resilience: withstanding a variety of attacks

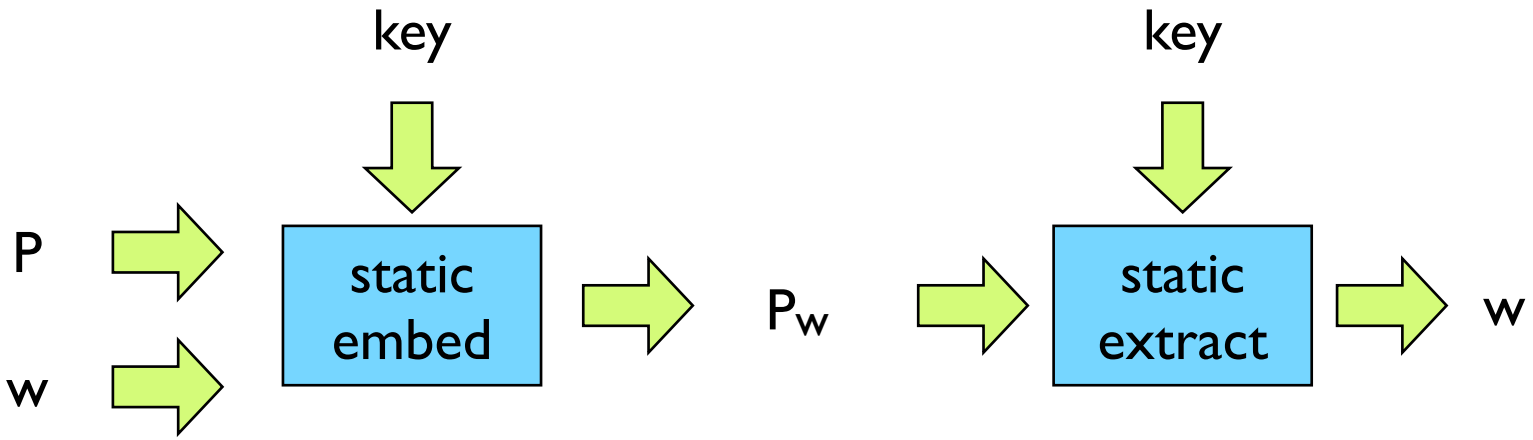
  * Collusive Attack: the adversary compares two different watermarked (fingerprinted) copies of the SW to identify the message



Why Software Watermarking

- ✓ Discourages illegal copying and redistribution.
- ✓ A copyright notice can be used to provide proof of ownership.
- ✓ A fingerprint can be used to trace the source of the illegal redistribution.
- ✓ Does not prevent illegal copying and redistribution.

Static SW Watermarking



embed(P, w, key) = P_w

extract(P_w, key) = w

recognize(P_w, key, w) = [0.0, 1.0]

Embedding preserves program semantics

Static SW Watermarking

* Watermark stored in the application executable itself. For example it can be stored directly in Data or Code of native executable or class file

* Static Data Watermarks

✓Embedded in the initialized data string:

```
> strings /bin/netscape | grep -i copyright

Copyright © 1998 Netscape Communication Corporation

Copyright © 1996, 1997 VeriSign, Inc.
```

easy to insert and extract but very sensible to distortive and subtractive attacks

Static SW Watermarking

* Static Code Watermarks

✓ Media watermarks are commonly embedded in redundant bits, bits that we cannot detect due to the imperfection of our human perceptions.

✓ Static code fingerprints can be constructed in a similar way, since object code also contains redundant information.

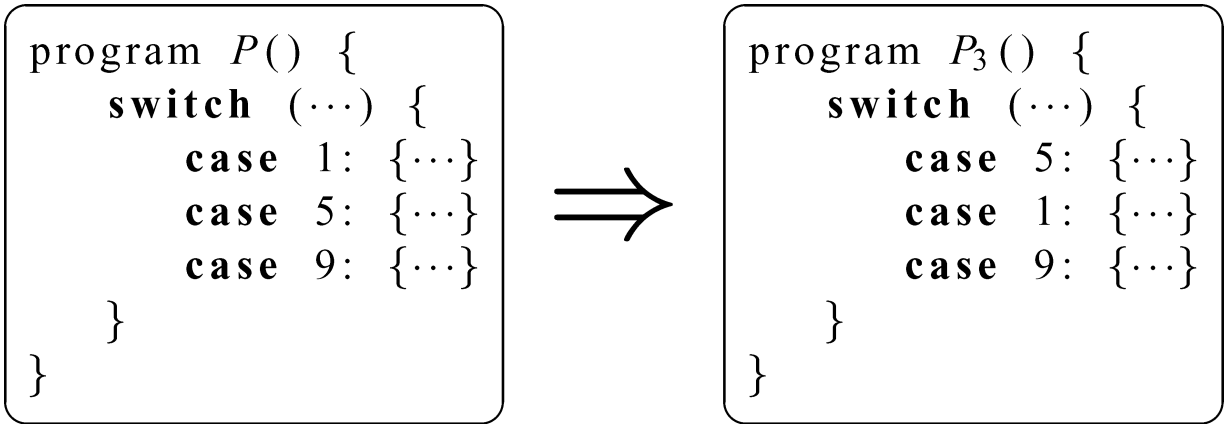
✓ Embedded in the text (code) section of the program, the signature is inserted in redundancy

How Statically Watermark SW

- Reorder code preserving functionality
- Whenever there are *m* language elements that can be arbitrary reordered, $log_2(m!) \sim O(m \log m)$ bits of information can be embedded

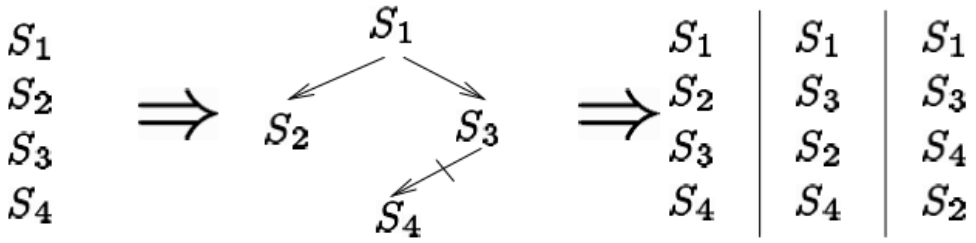
How Statically Watermark SW

- P_3 embeds the identifier 3 in the third permutation in lexicographic order of the values 1, 5, and 9
- <1,5,9> <1,9,5>**<5,1,9>**<5,9,1><9,1,5><9,5,1>
- The attacker can randomly permutate every switch statement in order to obliterate all marks



Reorder Statements

Reorder statements



Reorder switches

```
case e of {
  case 1 :  S1 |
  case 2 :  S2 |
  case 3 :  S3 |
}
```

⇒

```
case e of {
  case 2 :  S2 |
  case 3 :  S3 |
  case 1 :  S1 |
}
```

- m-cases $\Rightarrow \mathcal{O}(m \log m)$ watermarking bits.

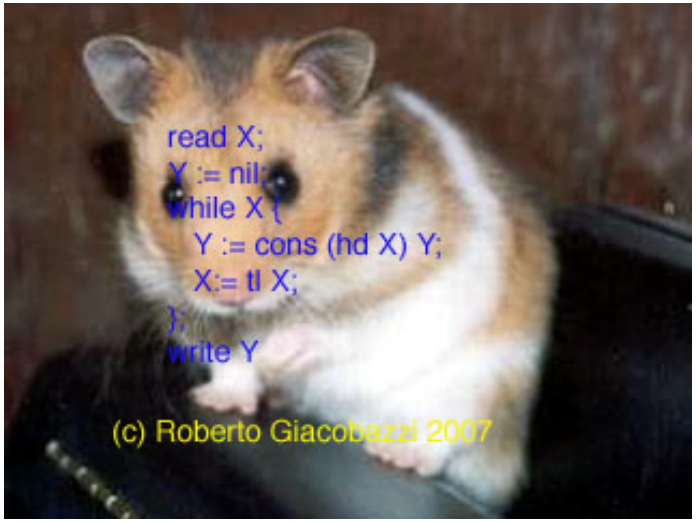
- Kirowski et.al.: Store watermarks in the register allocation.

Static Data Watermarks

- **US Patent: 5,745,569 Jan 1996**
- Method for stega-cipher protection of computer code
- Abstract: A method for protecting computer code copyrights by encoding the code into a data resource with a digital watermark. The digital watermark contains licensing information interwoven with essential code resources encoded into data resources. The result is that while an application program can be copied in an uninhibited manner, only the licensed user having the license code can access essential code resources to operate the program and any descendant copies bear the required license code.
- Inventors:
- **Moskowitz**; Scott A. (North Miami Beach, FL),
- Cooperman; Marc (Palo Alto, CA)Assignee:
- The Dice Company (Miami, FL)
- Appl. No.:08/587,943
- Filed:January 17, 1996

Moskowitz's SW Watermark

```
Class Main {
    const Pict P =
```



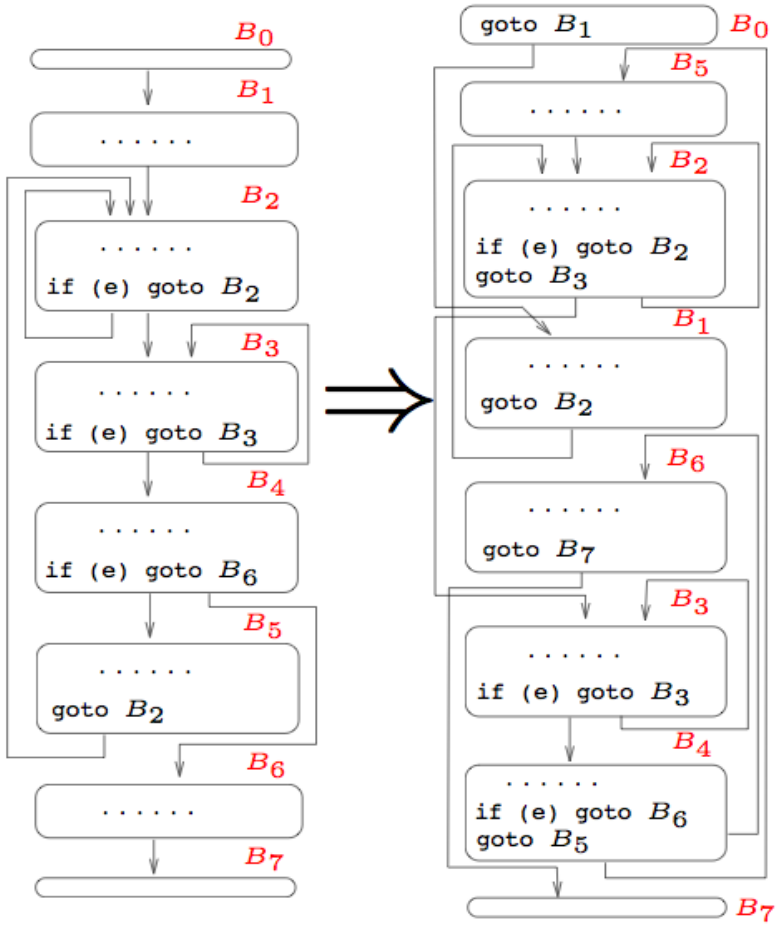

```
    ……..
    Code Q = Decode (P);
    Execute (Q);
}
```

- A watermarked media object is embedded in the program
- Essential parts of the code are steganographically encoded into static data (media)
- Type: similar to cryptography
- Attack: removing media objects = removing essential parts of the program
- The program fails to run
- The program cannot be disjoint from the copyright unless attacking media watermarks

Static SW Watermarking

- **US Patent: 5,559,884 June 1994**
- Method and system for generating and auditing a signature for a computer program.
- Abstract: A method and system for generating and auditing a signature for executable modules are provided. A signature is a mean that uniquely identifies an authorized copy of the executable module. **The signature of each authorized copy is encoded within the order of instructions of the executable module**. Each executable module is made up of multiple blocks of instructions. To place a signature within the executable module, a group of blocks having a flow of execution is selected from the executable module. The group of blocks is then reordered to form a signature for the executable module. To ensure that the reordered group of blocks has the same flow of execution, the blocks within the reordered group of blocks are modified to maintain the flow of execution. The reordered group of modified blocks replaces the unmodified group of blocks within the authorized copy of the executable module. The modified copy of the executable module executes in a manner that is functionally equivalent to the unmodified executable module. However, **the reordered blocks provide a signature that is unique to each authorized copy.** By inspecting the order of the group of blocks in a copy of the executable module, the signature can be determined, and thus it can be determined from which authorized copy the executable module derives.
- Inventors: Davidson; Robert I. (Bellevue, WA) and Myhrvold; Nathan (Bellevue, WA)
- Assignee: **Microsoft Corporation** (Redmond, WA)
- Appl. No.:08/268,967
- Filed: June 30, 1994

Microsoft SW Watermark



Encoding: $B_5B_2B_1B_6B_3B_4$

Easy to attack:

- Block reordering
- Code optimization (jump reduction)

Microsoft SW Watermark

- Performance overhead of 0-11% for three standard high-performance commuting benchmarks
- Negligible slowdown for a set of Java benchmarks
- Data rate: If you have **m** items to reorder you can encode

$$\log_2(m!) \approx \log_2(\sqrt{2\pi m}(m/e)^m) = \mathcal{O}(m \log m)$$

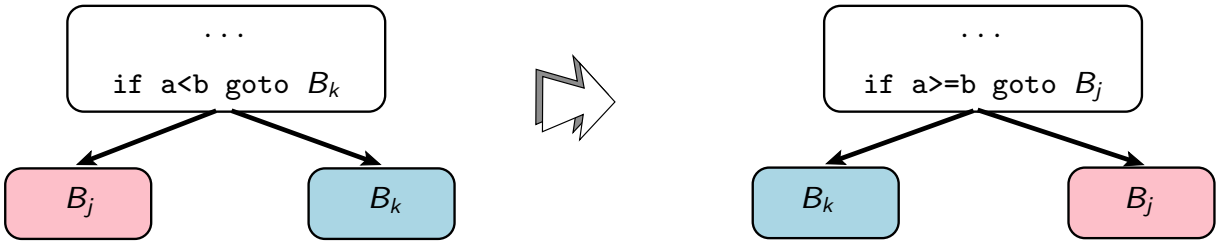
  watermarking bits
- Not very stealthy since the program control flow looks different from usual control flows generated by compilers

Watermarking in CFG

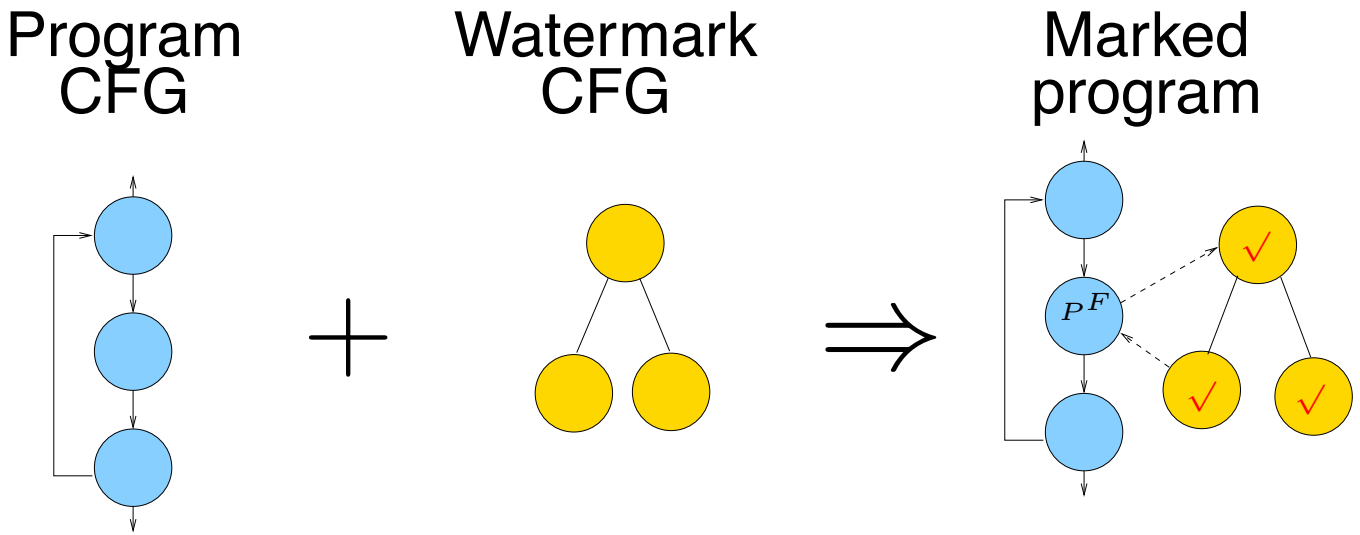
- Basic Idea:
  - ✓Embed the watermark in the CFG of a function
  - ✓Tie the CFG tightly to the rest of the program
- Issue
  - ✓Encode a number in a CFG
  - ✓Find a watermark in a CFG
  - ✓Attach the CFG embedding the watermark to the CFG of the program that we want to mark

Watermarking in CFG

- Generate a stealthy watermark CFG
  - basic blocks have out degree of one or tow
  - the graph is *reducible*
  - the graph is *shallow* (real code is not deeply nested)
  - the graph is *small* (real function are not big)
  - the mark encoding in the graph is resilient to *edge-flips*

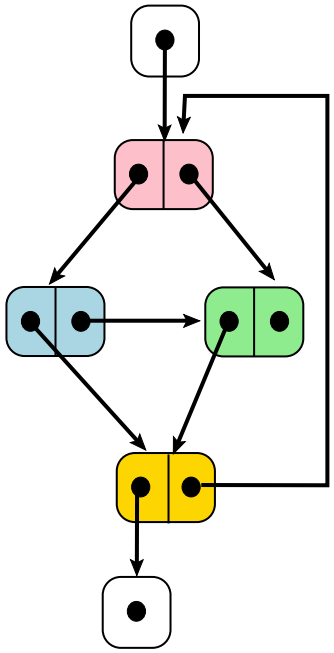


Watermarking in CFG



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

Watermarking in CFG



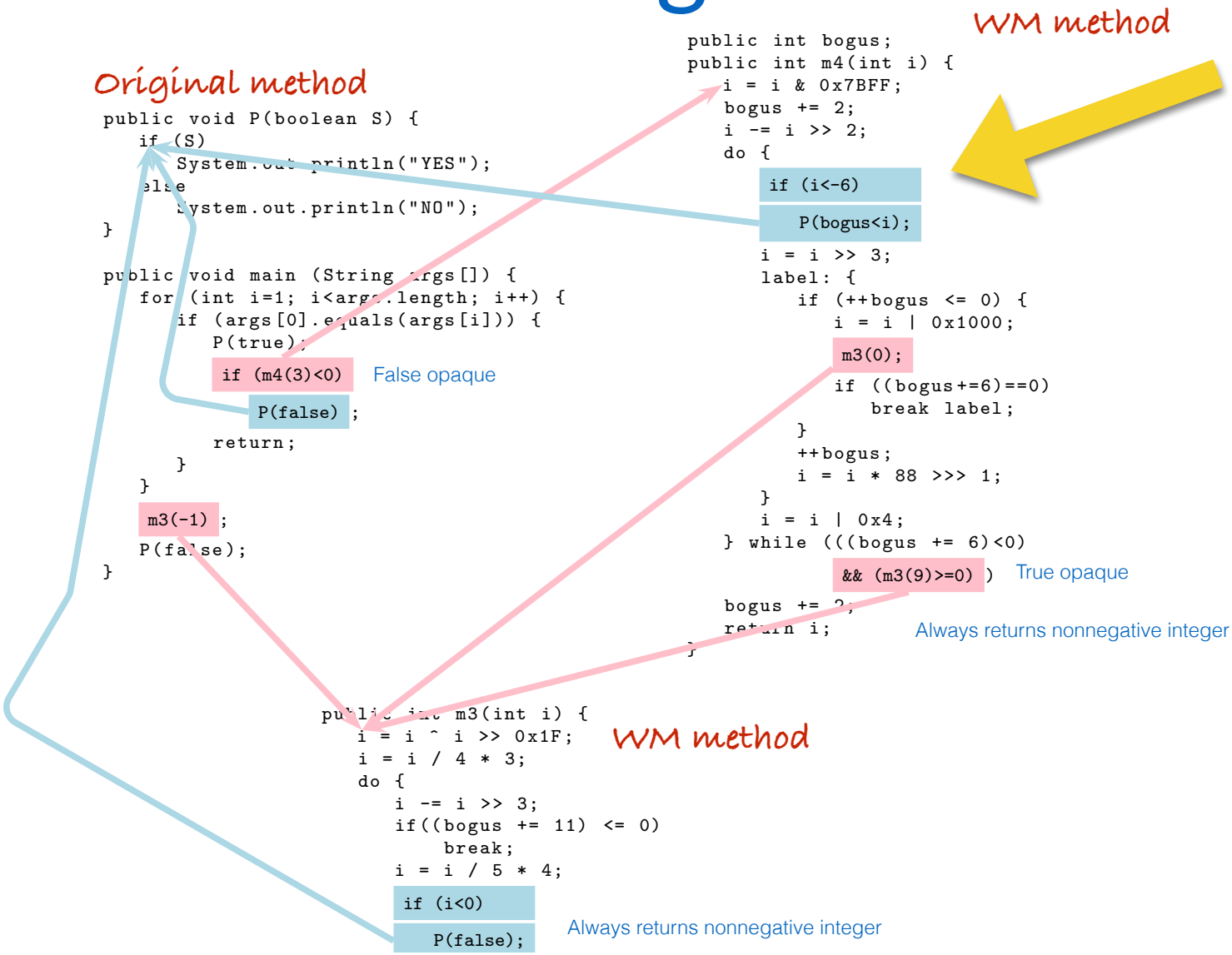
```
public static int bogus;
public static int m4(int i) {
    i = i & 0x7BFF;
    bogus+=2; i-=i>>2;
    do {
       i = i >> 3;
       label:  {
          if (++bogus <= 0) {
             i = i | 0x1000;
             if ((bogus += 6) == 0)
                break label;
             }
          ++bogus;
          i = i * 88 >>> 1;
          }
       i = i | 0x4;
    } while((bogus += 6)<0);
    bogus+=2; return i;
}
```

Watermarking in CFG

- It is good if the *watermark functions can actually execute predictably and efficiently*
- This makes it easier to embed calls to them
- `m4` has been designed to always return a non-negative integer and it can be used as an opaque predicate
- Ensure that the *control flow cannot be optimized*
- `bogus` is a global variable so that the attacker has to perform inter-procedural analysis to determine that `m4` can be safely removed

```
public static int bogus;
public static int m4(int i) {
    i = i & 0x7BFF;
    bogus+=2; i-=i>>2;
    do {
        i = i >> 3;
        label: {
            if (++bogus <= 0) {
                i = i | 0x1000;
                if ((bogus += 6) == 0)
                    break label;
                }
            ++bogus;
            i = i * 88 >>> 1;
            }
        i = i | 0x4;
    } while((bogus += 6)<0);
    bogus+=2; return i;
}
```

Watermarking in CFG



A graph theoretic approach to software watermarking, Venkatesan et al IH 2001

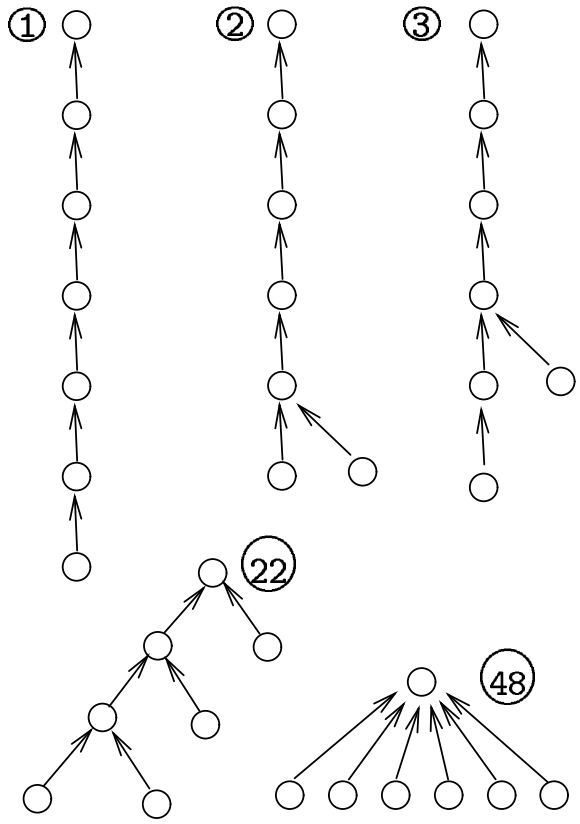
Watermark Extraction

- How do you find the watermark CFG among all the “real” CFGs?
- Idea:
  - ✓ Mark the basic blocks
  - ✓ a 0 for every cover program block, and a 1 for every watermarking block
- Recognition procedure
  - ✓ compute the mark value for each basic block in the program
  - ✓ assume that any function with more than t% blocks marked is a watermark function (increases resilience)
  - ✓ construct CFGs for the watermark functions
  - ✓ decode each one into an integer watermark
- The embedder can split the watermarking into pieces, for higher bit-rate

A graph theoretic approach to software watermarking, Venkatesan et al IH 2001

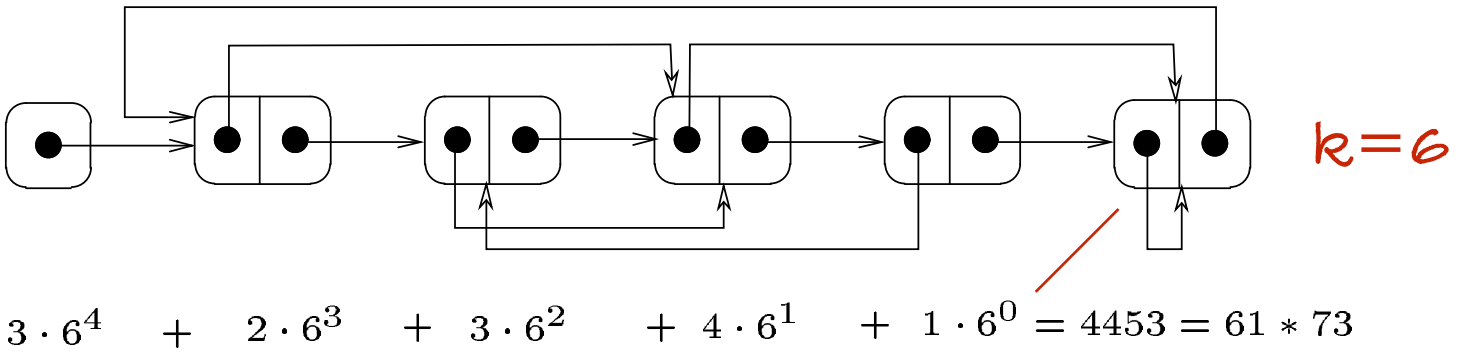
Graph embedding WM: oriented parent-pointer tree

Counting the members of a graph family. Here is an enumeration of the members of the oriented parent-pointer tree family with 7 nodes, ordered in "largest sub-tree first". Encoding and coding polynomial



- n is represented by the *index* of the the graph G in some enumeration.
- We must, efficiently, be able to
  1. given n, generate the n:th graph,
  2. given G, find Gs index n.
- Oriented parent-pointer trees $\Rightarrow$
  1. 655 nodes $\Rightarrow$ 1024-bit integer,
  2. bit-rate: 1.56 bits per word.

Graph embedding WM: radix graph

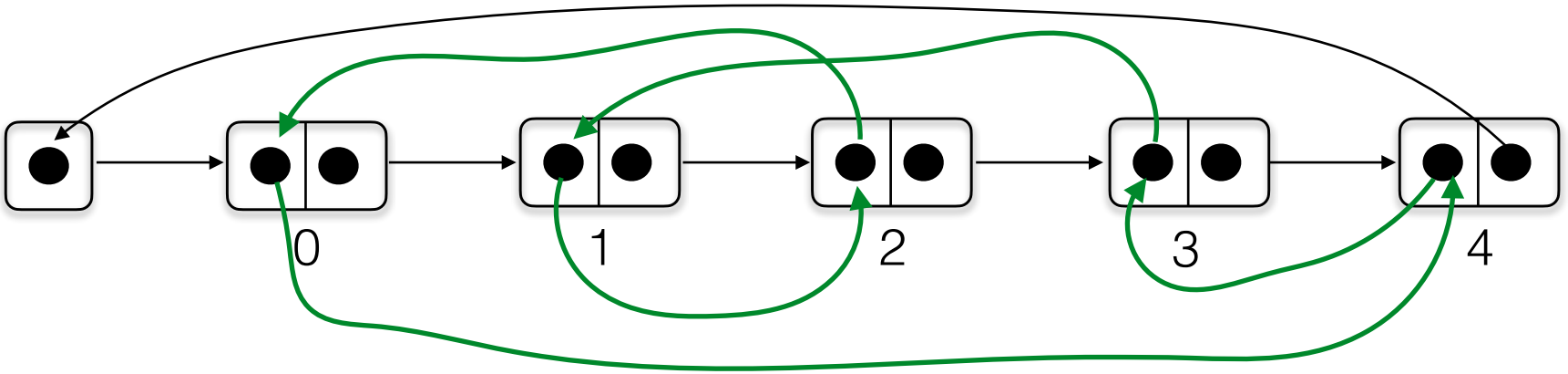


- G is a circular linked list.
- An extra pointer field encodes a base-k digit:

null-pointer	$\Rightarrow$	0
self-pointer	$\Rightarrow$	1
next node pointer	$\Rightarrow$	$2 \cdots$

- A 255-node list hides 2040 bits $\Rightarrow$ bit-rate is 4 bits per word.

Graph embedding WM: permutation graph



p = <4,2,0,1,3>

Watermark by Register Allocation: QP

- Firstly introduced by Kirowski et al. 1998
- Originally proposed by G. Qu and M. Potkonjak
- Idea: Constraint-based SW watermarking

  Embed the watermark in the register allocation of the program

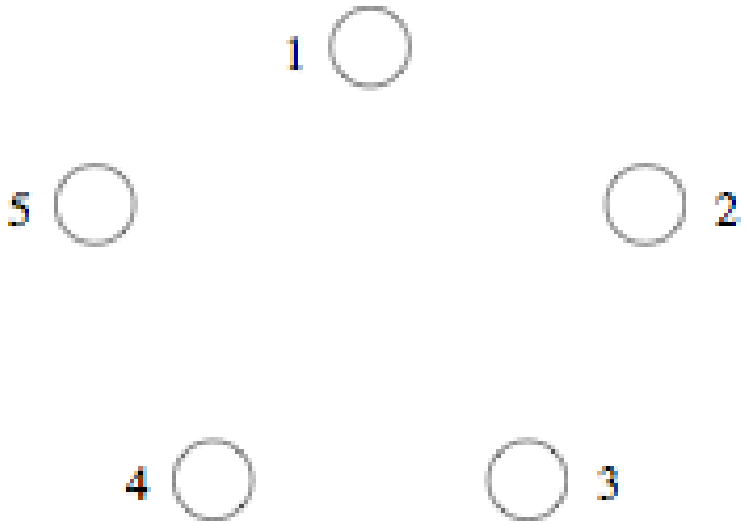
- Tools:
  - Interference graphs that model relations among variables
  - Graph coloring problem (NP hard)

QP Algorithm: Interference Graph

- Interference graph: Models the relationship between the variables in a program. Each variable is represented by a vertex.
- If two variables have overlapping live ranges then the vertices are joined by an edge.
- A variable is *live* at some point if it holds a value that may be needed in the future, or equivalently if its value may be read before the next time the variable is written to.
- Color the graph such that (color = register)
  - minimize the number of registers required
  - two live variables do not share a register

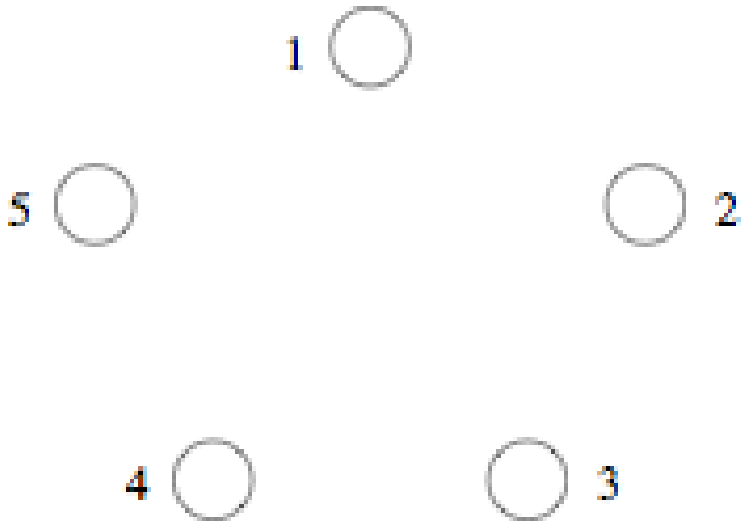
QP Algorithm: Interference Graph

```
Live variable analysis

V1 := 2 * 2

V2 := 2 * 3

V3 := 2 * v2

V4 := v1 + v2

V5 := 3 * v3
```



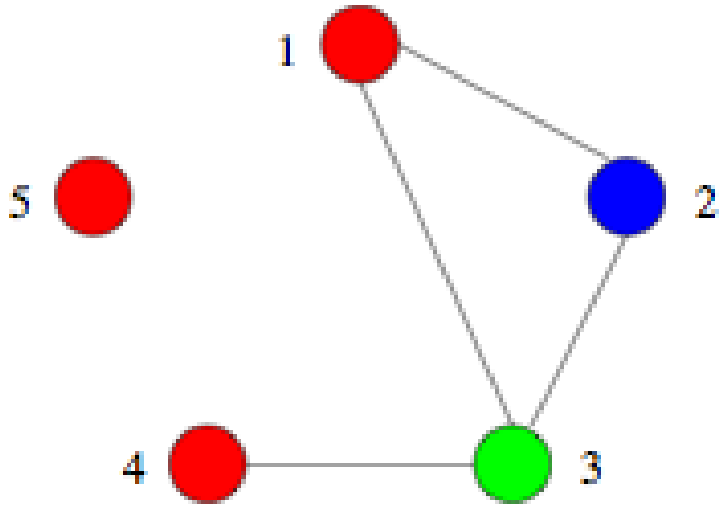
QP Algorithm: Interference Graph

```
Live variable analysis

V1 := 2 * 2

V2 := 2 * 3

V3 := 2 * v2

V4 := v1 + v2

V5 := 3 * v3
```



QP Algorithm: Interference Graph

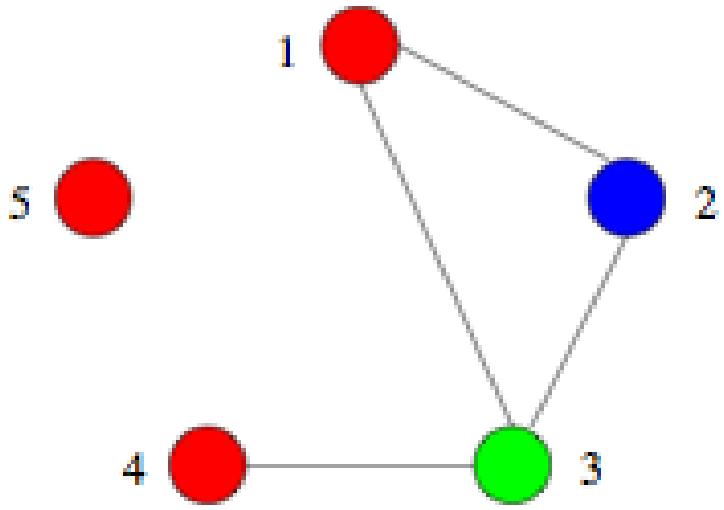
```
Live variable analysis

V1 := 2 * 2

V2 := 2 * 3

V3 := 2 * v2

V4 := v1 + v2

V5 := 3 * v3
```



QP Algorithm: Interference Graph

```
V1 := 2 * 2
V2 := 2 * 3
V3 := 2 * v2
V4 := v1 + v2
V5 := 3 * v3
```

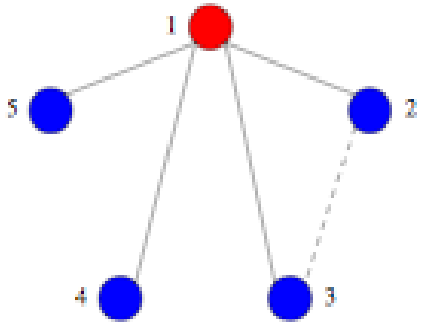
```
mult R1, 2, 2
mult R2, 2, 3
mult R3, 2, R2
add R1, R1, R2
mult R1, 3, R3
```

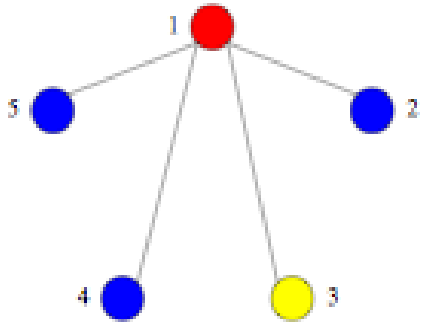


Hiding signatures in graph coloring solutions. Qu and Potkojak IH 1999

QP Algorithm

- Watermark: Add edges between chosen vertices based on the value of the mark to be inserted
- Connected vertices cannot be assigned to the same register!





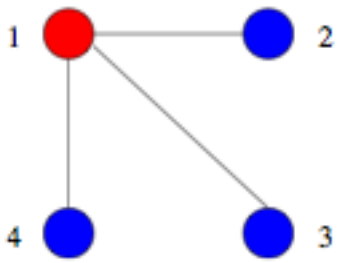
Hiding signatures in graph coloring solutions. Qu and Potkojak IH 1999

QP Algorithm

* Idea: select triples of vertex such that:
  - ✓ They are isolated and their connection will not affect other vertices in the graph
  - ✓ Given an n-colorable graph G = (V,E):

    $\{v_1,v_2,v_3\}$ is a *colored triple* if:

    - ✓ $v_1,v_2,v_3 \in V$
    - ✓ (v_1,v_2), (v_1,v_3), $(v_2,v_3) \notin E$
    - ✓ v_1,v_2,v_3 have the same color



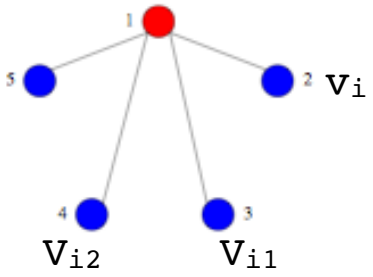
QP Algorithm

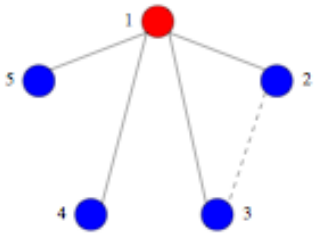
- Watermark is spread across all classes (good defense against subtractive attack)
  - Watermark is embedded as many times as possible
  - Message is converted to binary using the ASCII value of chars.
- The watermarking message to be embedded is converted in binary $\mathbf{m = m_0 \ldots m_k}$
- The algorithm assumes that the vertices of the interference graph are numbered
- The QP algorithm embeds each bit $\mathbf{m_j}$ of the watermark as follows

QP Algorithm: Embedding

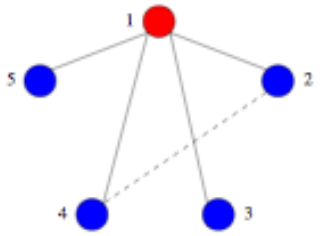
for each vertex $v_i \in V$ which is not already in a triple
    if possible find the nearest two vertices v_{i_1} and v_{i_2}
    such that
        v_{i_1} and v_{i_2} are the same color as v_i,
        and v_{i_1} and v_{i_2} are not already in triple.
    if $m_i = 0$
        add edge (v_i, v_{i_1})
    else
        add edge (v_i, v_{i_2})
end for

$i_2 > i_1 > i$





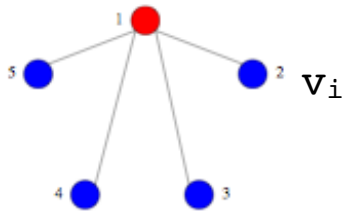
embed 0



embed 1

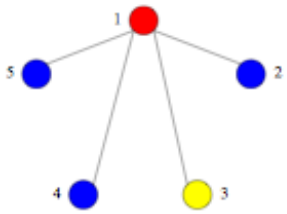
QP Algorithm: Recognition

for each vertex $v_i \in V$ which is not already in a triple
    if possible find the nearest two
        vertices v_{i_1} and v_{i_2} such that
        v_{i_1} and v_{i_2} are the same color as v_i,
        and v_{i_1} and v_{i_2} are not already in triple.
    if v'_i and v'_{i_1} are different colors
        found a 0
        add edge (v_i, v_{i_1})
    else
        found a 1
        add edge (v_i, v_{i_2})
end for

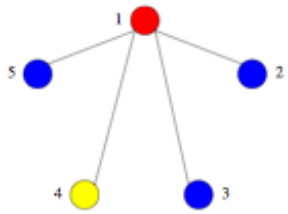


v_i

V_{i2} V_{i1}



found 0



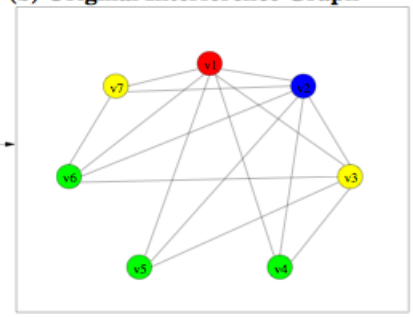
found 1

QP Algorithm: Example

(a) Original Bytecode

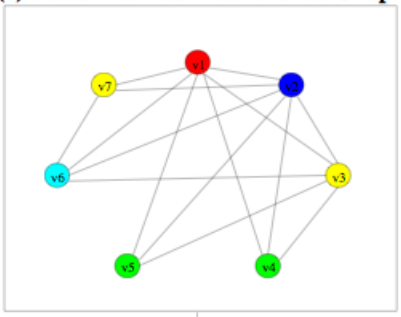
METHOD: fast_memcmp:([B[BI)Z
0 : iconst_0
1 : istore 3
3 : iconst_0
4 : istore_3
5 : iconst_1
6 : istore 3
8 : iload_2
9 : iconst_1
10 : isub
11 : istore_2
12 : iload_2
13 : iconst_0
14 : if_icmplt –> 21
17 : iconst_0
18 : goto –> 22
21 : iconst_1
22 : ifne –> 33
25 : iload 3
27 : invokestatic
30 : goto –> 34
33 : iconst_1
34 : ifne –> 60
37 : aload_0
38 : iload_2
39 : baload
40 : aload_1
41 : iload_2
42 : baload
43 : if_icmpeq –> 50
46 : iconst_0
47 : goto –> 51
50 : iconst_1
51 : istore 3
53 : iload_2
54 : iconst_1
55 : isub
56 : istore_2
57 : goto –> 12
60 : iload 3
62 : ireturn

(b) Original Interference Graph



Embed Watermark

(c) Watermarked Interference Graph



(d) Register Assignment Table

variable	register number
v1	0
v2	1
v3	2
v4	3
v5	3
v6	4
v7	2

(e) Watermarked Bytecode

METHOD: fast_memcmp:([B[BI)Z
0 : iconst_0
1 : istore 3
3 : iconst_0
4 : istore_3
5 : iconst_1
6 : istore 4
8 : iload_2
9 : iconst_1
10 : isub
11 : istore_2
12 : iload_2
13 : iconst_0
14 : if_icmplt –> 21
17 : iconst_0
18 : goto –> 22
21 : iconst_1
22 : ifne –> 33
25 : iload 4
27 : invokestatic
30 : goto –> 34
33 : iconst_1
34 : ifne –> 60
37 : aload_0
38 : iload_2
39 : baload
40 : aload_1
41 : iload_2
42 : baload
43 : if_icmpeq –> 50
46 : iconst_0
47 : goto –> 51
50 : iconst_1
51 : istore 4
53 : iload_2
54 : iconst_1
55 : isub
56 : istore_2
57 : goto –> 12
60 : iload 4
62 : ireturn

Hiding signatures in graph coloring solutions. Qu and Potkojak IH 1999

QP Algorithm: Evaluation

- Very low data-rate (a bit for every 3 variables that are not together alive)
- Easy to be attacked (e.g. by register re-allocation)
  - ✓ It is sufficient a register permutation!
- It is quite stealthy (hard to reverse the embedding)
  - ✓ The code looks like the original complied Bytecode
- Highly credible: very low probability of finding coincidence messages in watermarked code

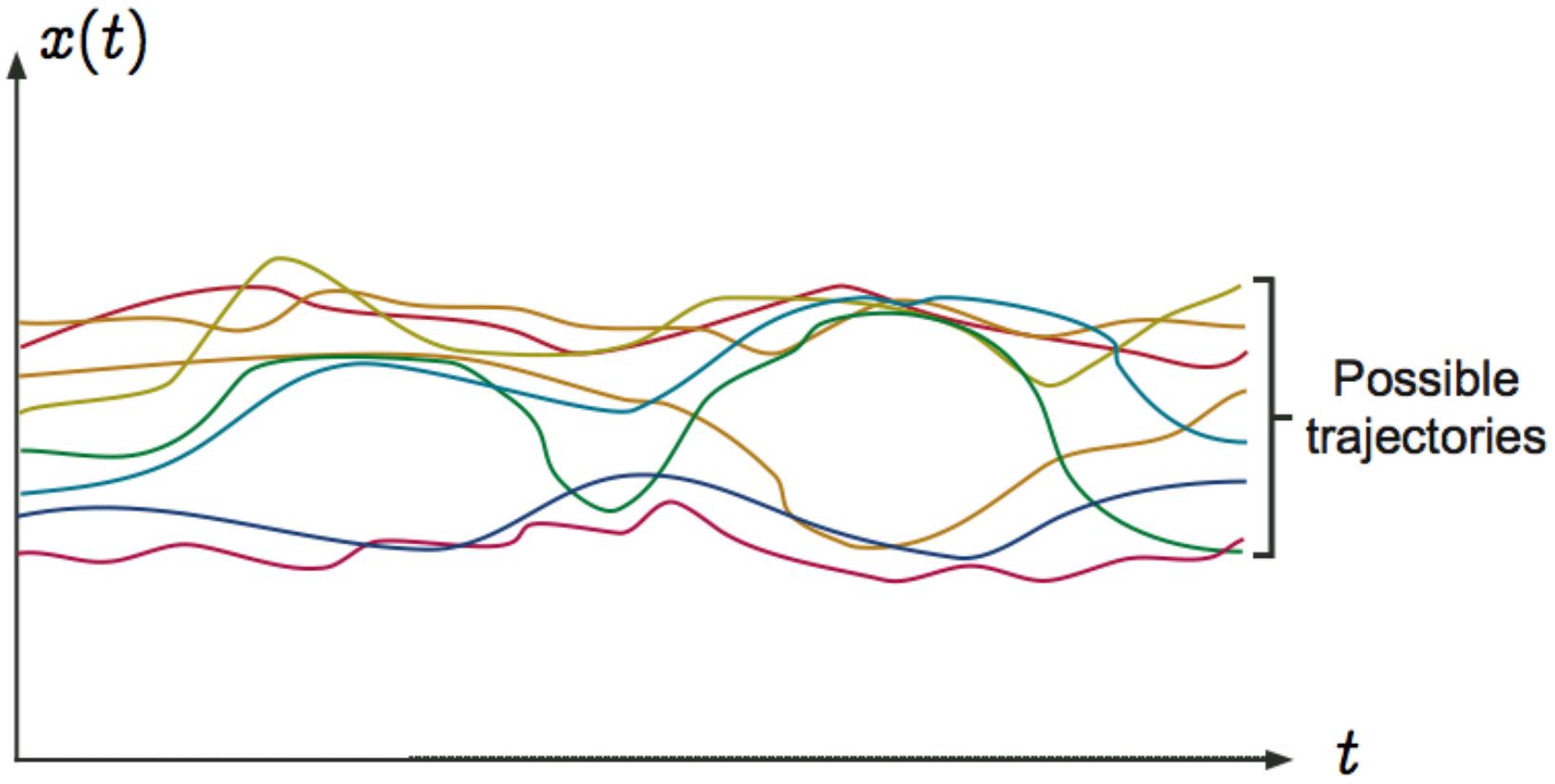
Limitations of Static Watermark

- Static SW Watermarking is weak for Distortive attacks:
  - Just optimize code: this would destroy many watermarks
  - Obfuscate code: destroy the control-flow by inserting branches and new blocks
  - Inlining and outlining, loop transformations may destroy watermarks
  - Decompilation + recompilation may destroy watermarks

What to do?

- *Dynamic* SW Watermarking
  - Embed the watermark into the concrete semantics of the program
  - The concrete semantics is (a mathematical model of) the set of possible executions in all possible execution environments
- It is resilient to:
  - Subtractive attacks: Hard to find!
  - Distortive attacks: semantics-preserving transformations preserve the watermark
  - Collusive attacks: Hard to compare the semantics of two programs on any possible input

Program Semantics



Rice Theorem vs SW Watermarking

- A property π is *extensional* if for any P, Q $\in$ Java:

$$\pi(P) \ \& \ [P] = [Q] \Rightarrow \ \pi(Q)$$

- An extensional property π is decidable iff

$$\pi = \varnothing \ \text{ or } \ \pi = \text{Java}$$

- The property ω is:

The program contains a dynamic watermark

ω is extensional and $\pi \neq \varnothing$ and $\pi \neq$ Java!!!

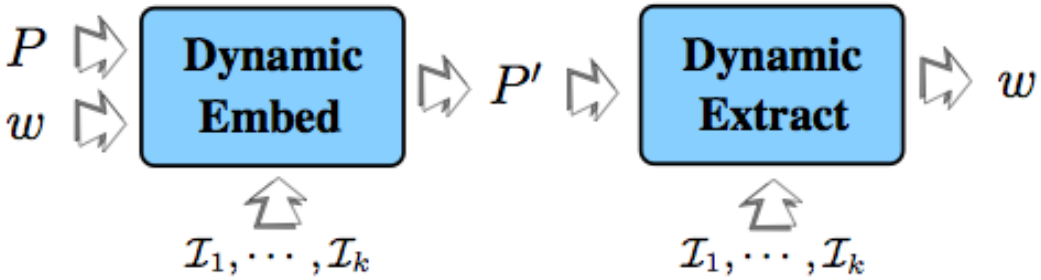
- The property ξ is:

The program contains a static watermark

- ξ is not extensional $\Rightarrow$ dynamic watermarks are usually more resilient!

Static vs Dynamic Watermarking

- Static: Vulnerable to semantics-preserving code transformations
- Dynamic: extract the watermark from the state of the program



Dynamic SW Watermarking

- Idea:
  - ✓ Get a hard SW-analysis problem (EXP or PSPACE)
  - ✓ i.e. an extensive property π such that $\pi(P)$ is hard!
  - ✓ Embed the watermark in the semantics of P such that to get the watermark you need to analyze π

Aliasing: a good candidate!

- 1-level of aliasing is easy (P) [Banning'79]
- ≥ 2-level of aliasing is NP [Horowitz'97]
- …with dynamic memory allocation is undecidable!

- Shapes:
- Shape analysis predicts the structure of dynamically allocated structures (heap, stack, etc…)
  - Most tools predict heap construction but not destructive updates!

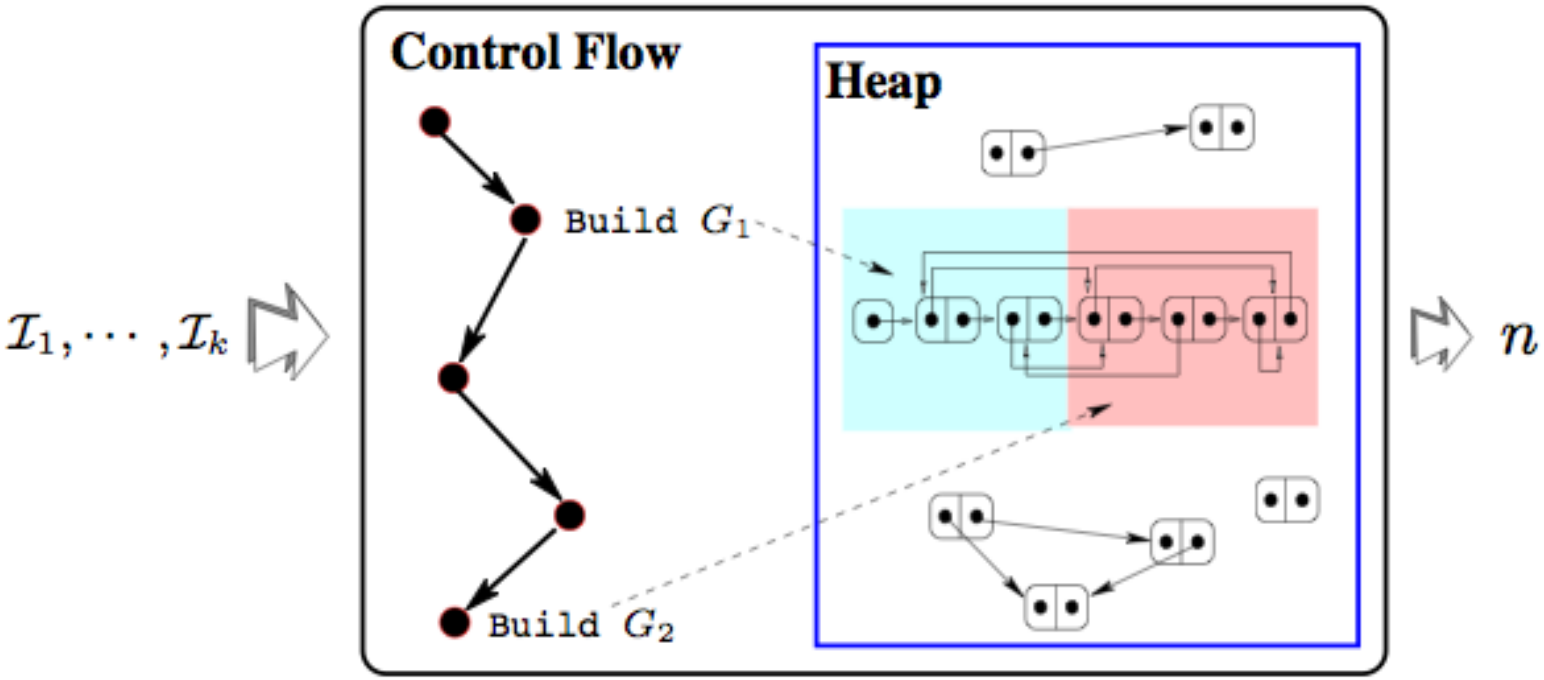
Three Dynamic Methods

- Dynamic Data-structure watermark
  - ✓ Embed the watermark in the dynamic data structures used by the program: heap, stack…
- Dynamic Graph watermark
  - ✓ Embed the watermark in the topology of a graph generated under some inputs
- Dynamic Execution watermark
  - ✓ Embed the watermark within the trace structure

The (mixed) CT method

- Collberg-Thomboroson dynamic data-structure watermark
- Shape-analysis is hard:
  - ✓ need to *approximate* conservatively the possible shapes that heap-allocated structures in a program can take!
- The watermark is embedded in the topology of a dynamic data structure (heap), built at run time only under some special inputs

The CT dynamic watermarking



Dynamic graph-based software fingerprint. Collberg et al. TOPLAS 2007

An example of CT

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

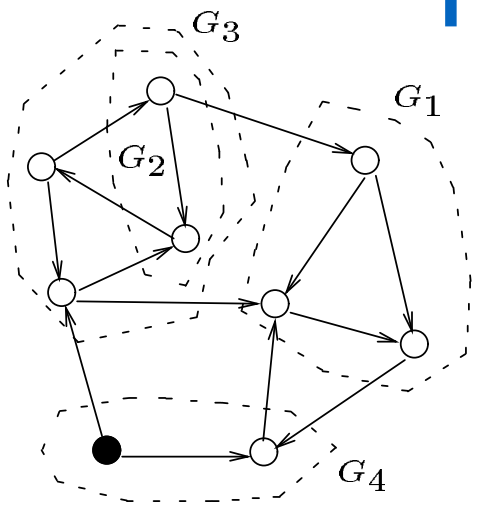
```
class Watermark extends java.lang.Object {
    public Watermark edge1,edge2;
}
```

An example of CT

On the special input World the program builds the WM graph on the heap

```
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);
}
```

Graph construction & recognition




```
if (input = I_1) G_1 = ···;
if (input = I_2) G_2 = ···;
if (input = I_3) G_3 = G_2 ⊕ G_3;
    ......
if (input = I_k) G = G_1 ⊕ G_3 ⊕ ···;
```

- For each input $\mathcal{I}_i$, one graph component is built.
- After input $\mathcal{I}_k$ the entire graph has been assembled.
- To find the root of G we need only consider nodes created during the processing of $\mathcal{I}_k$.

Is this good?

- Stealth:
  - ✓ The code resembles the standard code found in many real pointer-rich applications
- Resilience
  - ✓ They are weak under “destructive updates”
  - ✓ The attacker may obfuscate the code to destroy the watermark by inserting destructive updates that do not modify the semantics ⇒ hard to do!
- Easy to build hard to detect

Quite good!!!

Experience with CT algorithm

- Implementation and experiments on a watermarking system (`JavaWiz`) for Java programs that embeds the watermarks in dynamic data structures:
  - ✓ Moderate increase in code size
  - ✓ Moderate increase in execution time
  - ✓ Moderate increase in heap-space usage
  - ✓ Watermarking resilient to variety of program transformations attacks
- Watermarking techniques benefit from the protection mechanism of randomization, obfuscation and tamper proofing (detailed discussion in the paper)

Experience with CT algorithm

program	description	test input
javac	a compiler for Java	the JavaCup source code
javadoc	a Java API documentation generator	the JavaCup source code
JavaCup	an LALR parser generator for Java	the CORBA grammar
JTB	JTB [16] is a frontend for The Java Compiler Compiler from Sun Microsystems	the Java 1.2 grammar
JavaWiz	the watermarking system reported in this paper	the JavaCup source code
compress	a java virtual machine spec benchmark	some tar files shipped with compress
BLOAT	BLOAT [9] is a Java bytecode optimization tool	the JavaCup source code

Table 2. Programs on which we have experimented

Experience with CT algorithm

program	code size		wm time	retr time	execution time		heap space usage	
	before	after			before	after	before	after
javac	192	201	18.8 s	7.1 min	79.4 s	82.5 s	6,415	6,453
javadoc	187	191	19.9 s	8.9 min	26.7 s	27.4 s	9,770	10,000
JavaCup	362	373	5.6 s	4.6 min	4.3 s	4.6 s	4,041	4,080
JTB	810	815	5.2 s	0.6 min	9.9 s	10.1 s	440	475
JavaWiz	582	591	4.3 s	2.2 min	4.7 s	4.9 s	2,012	2,045
compress	24	32	4.6 s	0.6 min	68.8 s	72.4 s	477	514
BLOAT	1,415	1,427	7.0 s	3.6 min	55.7 s	57.9 s	3,322	3,362

Table 3. Experimental Results

For the considered benchmarks:

- JavaWiz adds 4–12 kilobytes of code (less than 10 kilobytes of code on average)
- embedding a watermark is done in less than 20 seconds
- watermarking increases a program's execution time less than 7%, often much less
- the heap space requirement increases, although it should be noted that the increase depends on the size of the objects of the class which is chosen as node class
- watermark retrieval is done in about 1 minute per 1 megabytes of heap

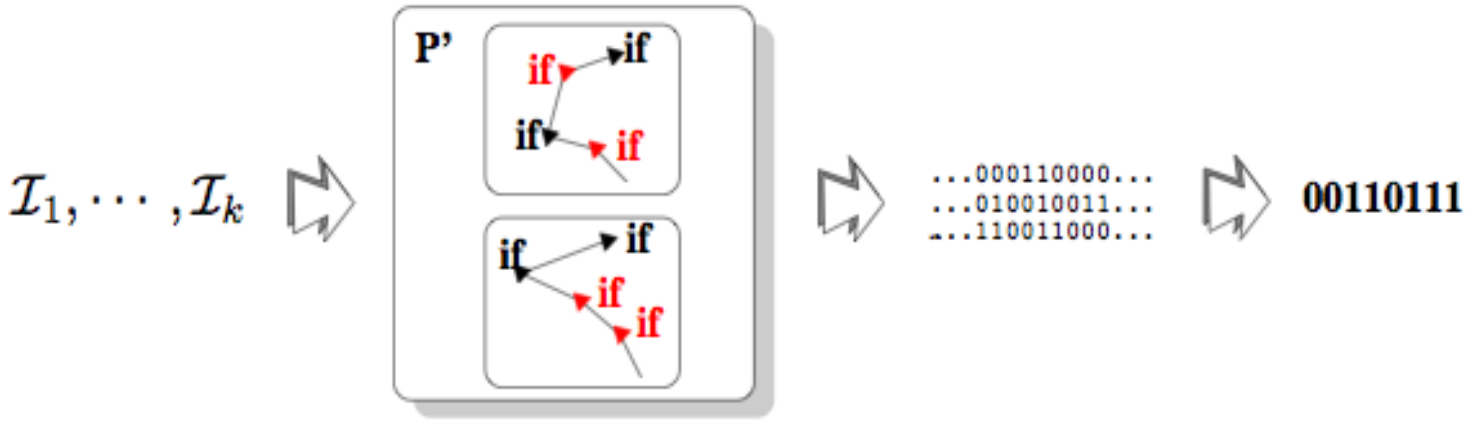
Experience with CT algorithm

- We have tried to attack the watermarked programs with the Java bytecode obfuscator **WingGuard** and the Java packaging tool **JAX**. We can view obfuscation and packaging as attacks because they are semantics-preserving program transformations. JAX is particularly interesting as an attack because it attempts to eliminate dead code.

- In all cases, we found that the watermark was intact after the attacks.

Path-Based Watermarking

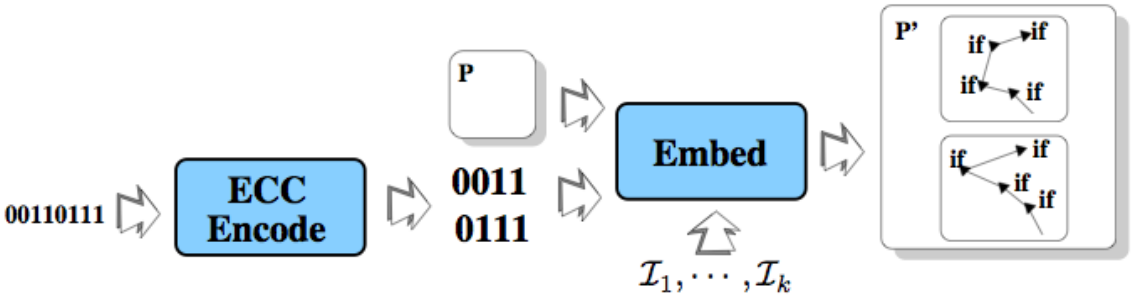
- Idea: the watermark is encoded in the branch structure of the program during execution on secret input
- Branches executed under secret input generate a stream of 0 (fall through) and 1 (branch) which exhibits the watermark
- ATTACK: The attacker can easily insert new branches!



Dynamic path-based software watermarking. Collberg et al. PLDI 2004

Path-Based Watermarking Resilience

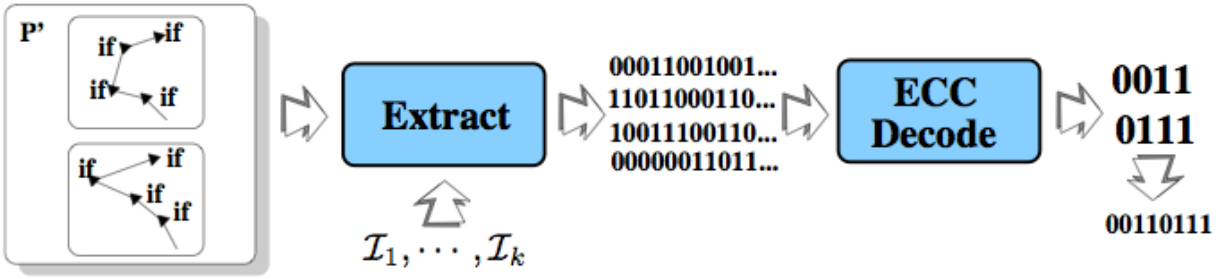
- To *increase stealth* of large watermarks embedding we split the mark in *multiple pieces which are spread over the program*
- To *incresce resilience* we make the pieces *redundant*, so that finding a subset of them will be enough to extract the watermark
- Embedding:
  - The watermark value is turned into a set of values to be embedded into the program
  - These values are embedded in the form of additional branch instructions and other code



Dynamic path-based software watermarking. Collberg et al. PLDI 2004

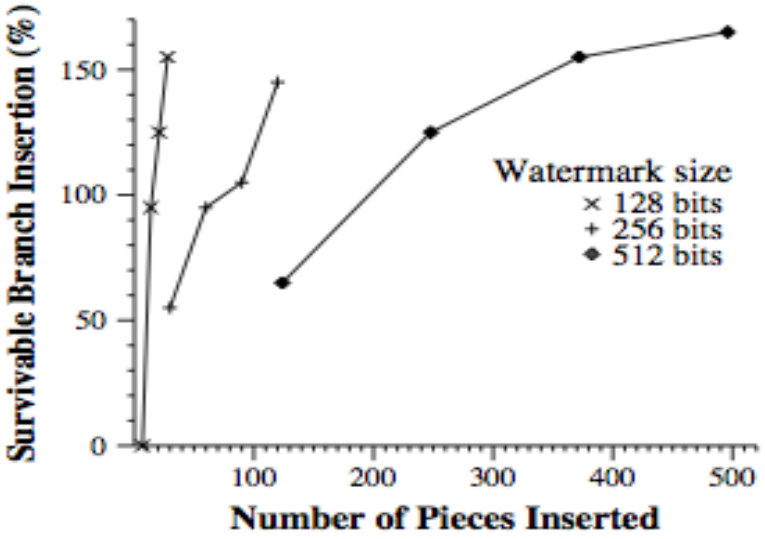
Path-Based Watermarking Extraction

- Extraction:
  - The program is run with the secret input
  - Branches are monitored and a bitstream is extracted
  - The watermark is then extracted from the bitstream



Path-Based Watermarking: the attack

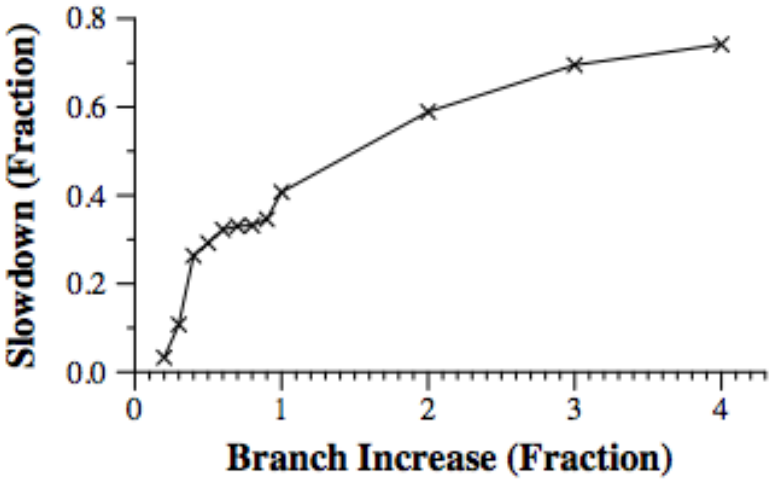
- Attack model: randomly add bogus conditional branches to P
- The more redundant is the watermark w the more an attacker has to destroy the CFG in order to remove w



- With a 256-bit watermark and 100 pieces: double the branches in order to destroy the watermark

Path-Based Watermarking: the attack

* Attack By doubling the number of branches the attacker slows down the program by 40%!!



Abstract watermarking

- Embed a "*large*" number into the program
- Idea: In order to extract the watermark you need to *analyze* the code:
  - ✓The analysis is an abstraction of concrete program executions
  - ✓*Static* since you do not execute the program to extract the mark
  - ✓*Dynamic* since the mark is embedded in the semantics

Abstract watermarking

- Use the Chinese reminder theorem for expressing a big integer **c** in terms of smaller ones $c_1 \ldots c_1$ in terms of pairwise co-prime numbers

Let $n_1, \ldots, n_\ell$ be $\ell \geq 1$ positive integers which are pairwise co-prime (meaning $\gcd(ni, nj) = 1$ whenever $i \neq j$) then $\mathbb{Z}/_{n_1 \ldots n_\ell \mathbb{Z}}$ is isomorphic to the cartesian product $\mathbb{Z}/_{n_1 \mathbb{Z}} \times \ldots \times \mathbb{Z}/_{n_\ell \mathbb{Z}}$.

$$c = \sum_{i=1}^{\ell} \left(\left(\prod_{j=1}^{i-1} n_j \right) . c_i . \left(\prod_{j=i+1}^{\ell} n_j \right) \right) \pmod{\prod_{j=1}^{\ell} n_j}$$

$$\ell \text{ keys } \langle c_1, \ldots, c_\ell \rangle \in [0, n_1 - 1] \times \ldots \times [0, n_\ell - 1]$$

✓ **n_i relatively prime numbers**

✓ **$c \leq n_1 \cdot \ldots \cdot n_l$**

Abstract watermarking

* Each c_i is embedded in a *stegomark* built up in three parts:
  - ✓ *declaration* (introducing a new auxiliary variable `W` hiding the secret key c_i)

```
int W
```

  - ✓ *initialization*

```
W = P(1)
```

  - ✓ *iteration*

```
W = Q(W)
```

  - ✓ such that `P(1)` = c_i in $\mathbb{Z}/n_i\mathbb{Z}$ and c_i = `Q(`c_i`)` in $\mathbb{Z}/n_i\mathbb{Z}$
  - ✓ Thus, once initialized the variable `W` is *constant* in $\mathbb{Z}/n_i\mathbb{Z}$ even if its value keeps on changing during execution. This is the invariant property that will be exploited for extracting the signature.

Abstract watermarking

- The stegomark should be *obfuscated* in order to hide the signature c_i
- if `W` is a *dead variable* we do not need to declare it
- initialization `W=P(1)` we use a second-degree polynomial `P`

  $$P(x) = x^2 + k_1 * x + k_0$$

  where $k_1 = -(1+c_i)$ and $k_0 = 2*c_i$
- In order to hide c_i these coefficient are incremented a random times the modulo n_i
  - ✓ $k_1 = -(1+c_i) + r_1 * n_i$
  - ✓ $k_0 = 2*c_i + r_0 * n_i$
- Of course in $\mathbb{Z}/n_i\mathbb{Z}$ we have that $P(1) = c_i$

Abstract watermarking

- Iteration $W = Q(W)$
- Q is chosen to be a second-degree polynomial

  $$Q(x) = ax^2 + b*x + c$$

  where a and b are (not too large) non-zero random numbers and c is chosen to ensure that $c_i = Q(c_i)$ in Z/n_iZ

$$c = c_i - (b*c_i + a*c_i^2)$$

Abstract watermarking

To compute the polynomials use the Horner method:

$P(x) = x^2 + k_1 * x + k_0$

```
W=1;
T=W+k1;
T=W*T;
W=T+k0;
```

$Q(x) = ax^2 + b*x + c$

```
T=W*a;
T=T+b;
T=T*W;
W=T+e;
```

* Initialization

```
W = P(1)
```

* Iteration:

```
W = Q(W)
```

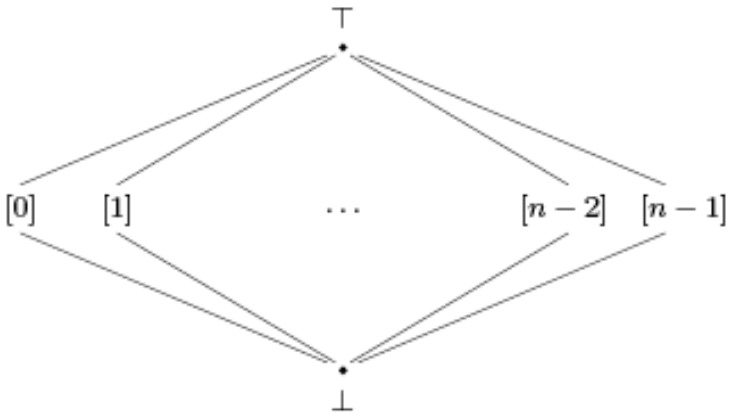
Regola di Horner

$$P(x) = (\dots(p_n * x + p_{n-1}) * x + p_{n-2}) * x + \dots + p_1) * x + p_0$$

Abstract watermarking

Watermark *extraction*:

- Analyze $\mathbf{P_w}$ for constant propagation (modulo $\mathbf{n_i}$)
- *Key*: You have to know $\mathbf{n_i}$ (the primes used for encoding **w**)



Abstract watermarking

- High data-rate it allows us to embed large numbers
- Not stealthy: large numbers are not common in programs
- Hard to destroy: you have to know that those numbers are useless (I.e. Static program analysis)
- Easy to erase: Program slicing of variables `W` and `T`
- Counterattack: obfuscate (es. include bogus aliasing)

Abstract watermarking

```
int f = -158657;
for (…)
…
f = 81351;
…
```

initialization

update

W = 21349
key = 3001

Abstract watermarking

```
int f = -158657;
for (…)
…
f = 81351;
…
```

f = {21349}

f = {21349}

W = 21349
key = 3001

to extract the mark we perform constant propagation reducing the computations modulo the secret key

158657 mod 3001 = 21349
81351 mod 3001 = 21349

Abstract watermarking

```
int n = 1;
for (int i=2;i<=10;i++)
n=n*i;
```

+

```
int f = -158657;
for (…)
…
f = 81351;
…
```

```
int n = 1;
int f = -158657;
for (int i=2;i<=10;i++)
n=n*i;
f = 81351;
```

Look in the program in order to find a function with a loop where to insert the watermark

Abstract watermarking

```
int n = 1;
int f = -158657;
for (int i=2;i<=10;i++)
int g = f*4;
g = g + 1566;
n=n*i;
g = g* f
f = g + 21494;
```

We obfuscate it in order to make it more stealth and difficult to slice away

Birthmark algorithms