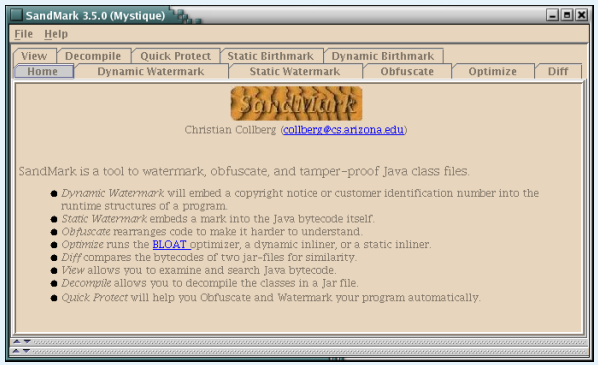


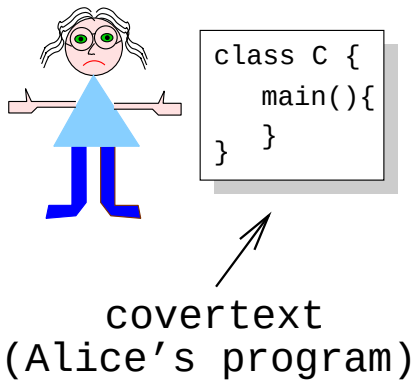

# Software Watermarking

Christian Collberg
University of Arizona

September 17, 2006

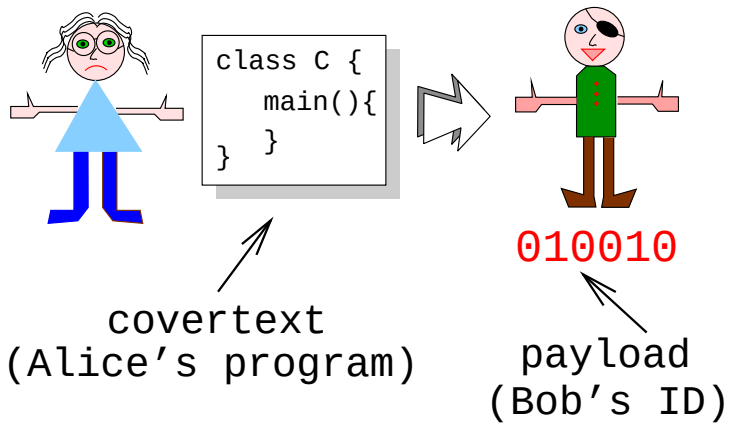
# Hiding Secrets in Software





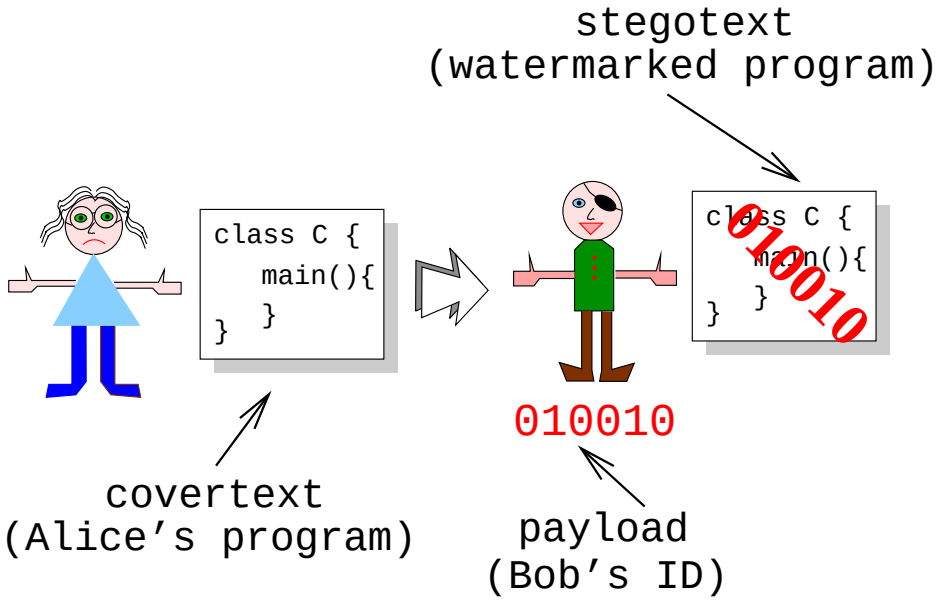
# Hiding Secrets in Software





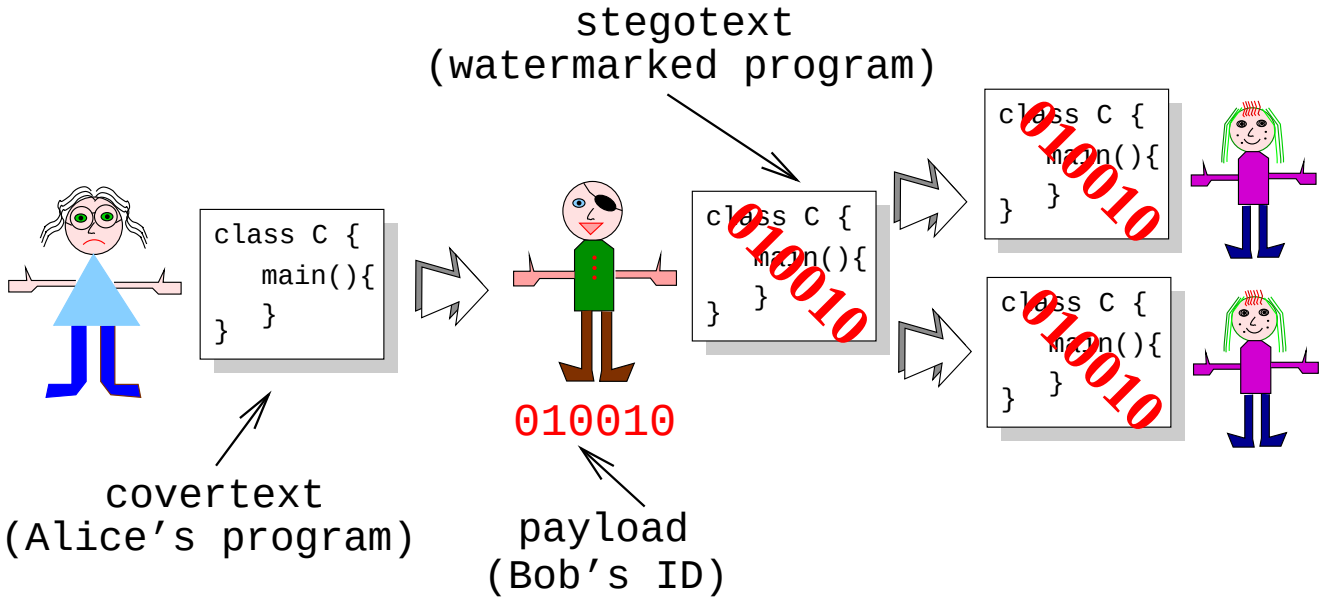
# Hiding Secrets in Software





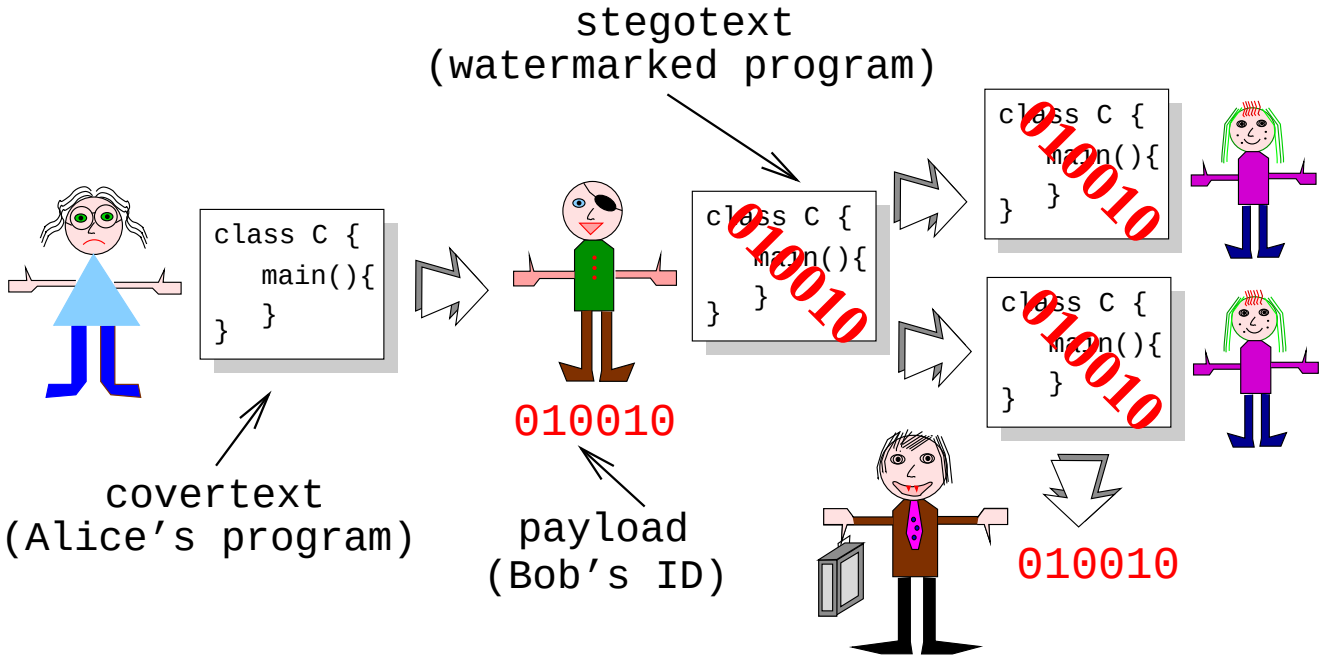
# Hiding Secrets in Software





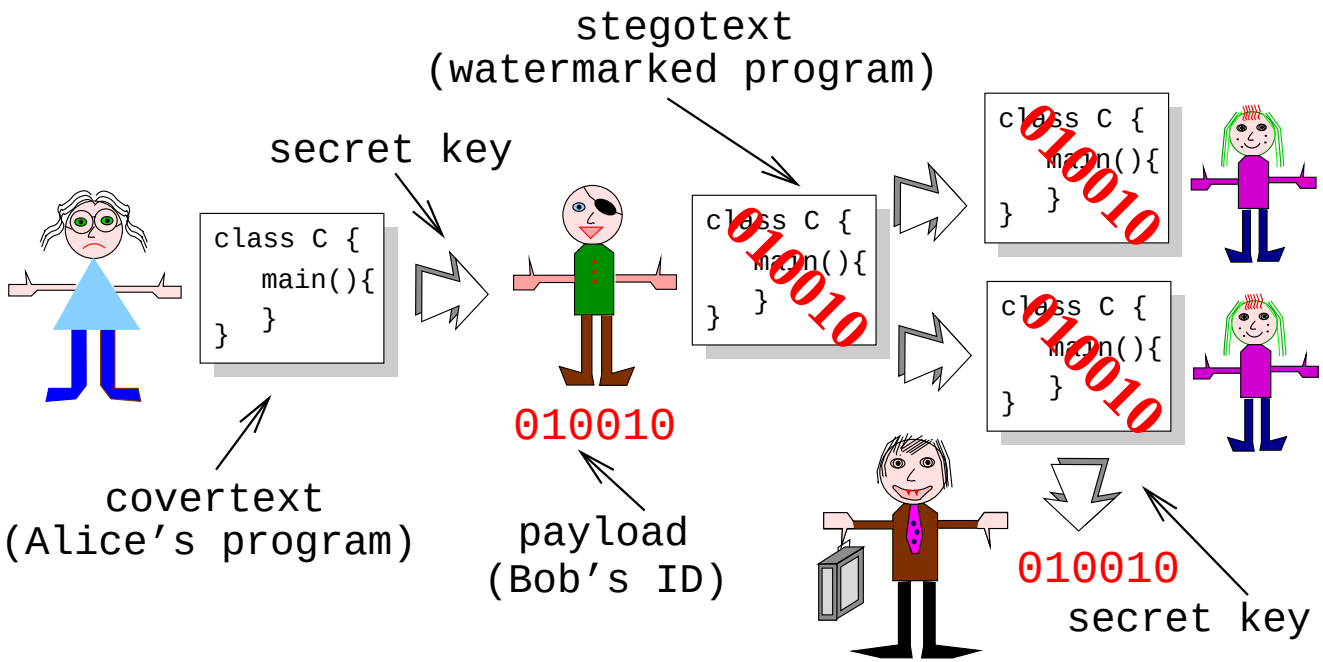
# Hiding Secrets in Software





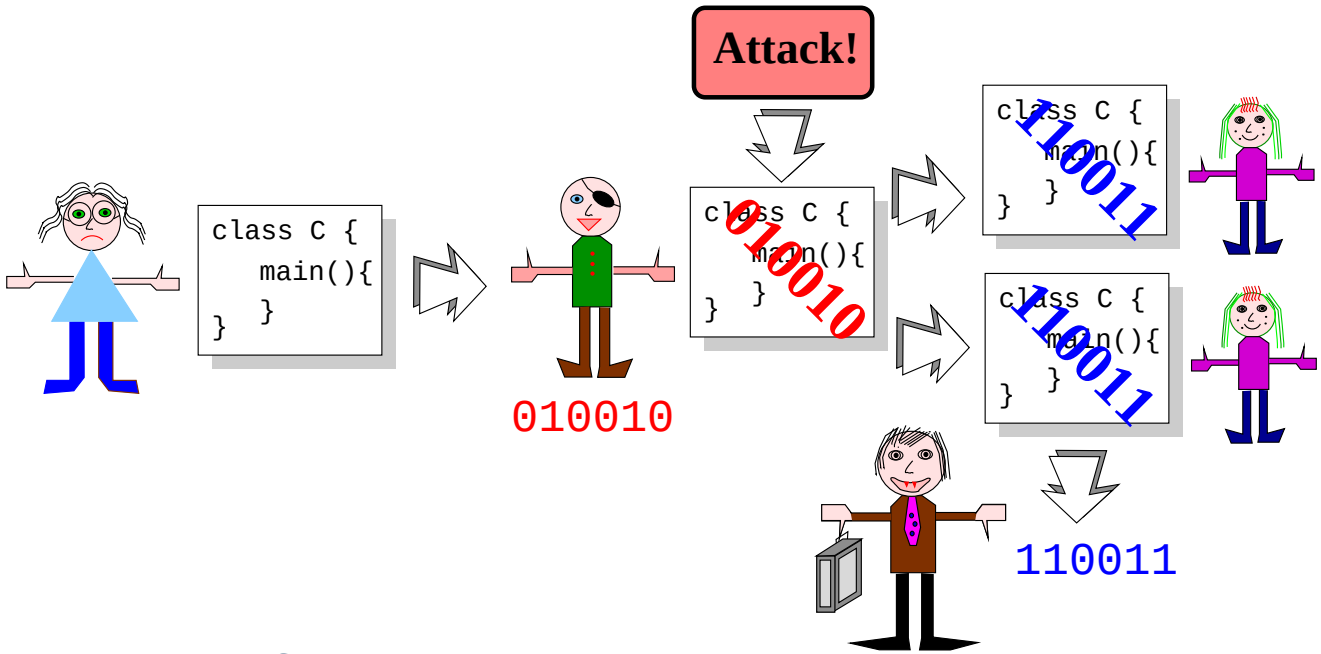
# Hiding Secrets in Software





# Hiding Secrets in Software

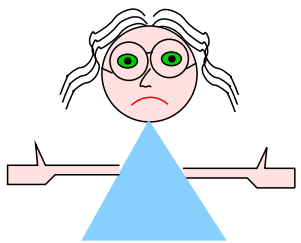


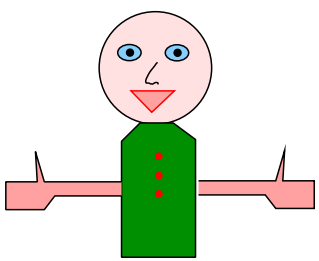



- Applications: Software fingerprinting, steganography,. . .

# Prisoners' Problem





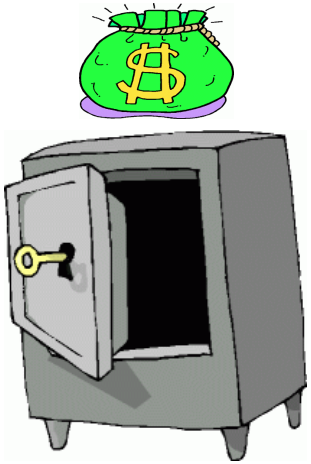



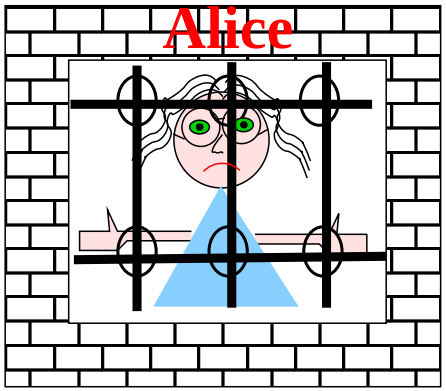
# Prisoners' Problem

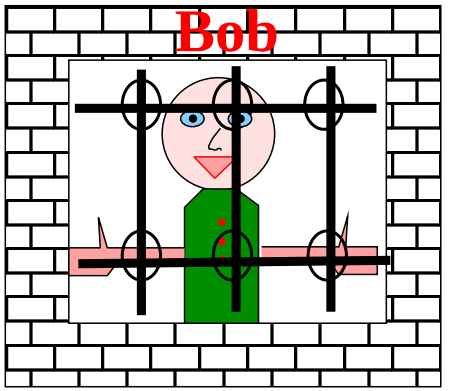


# Prisoners' Problem



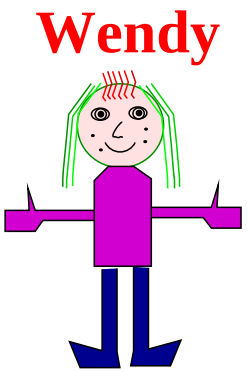


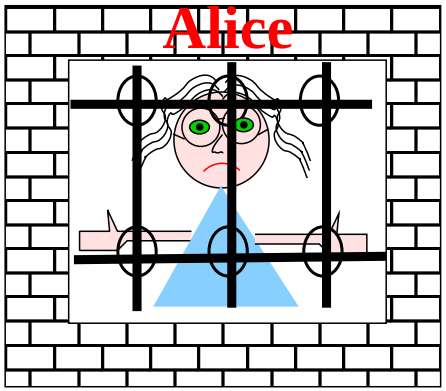


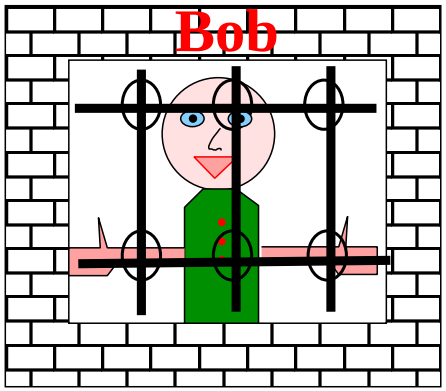



# Prisoners' Problem



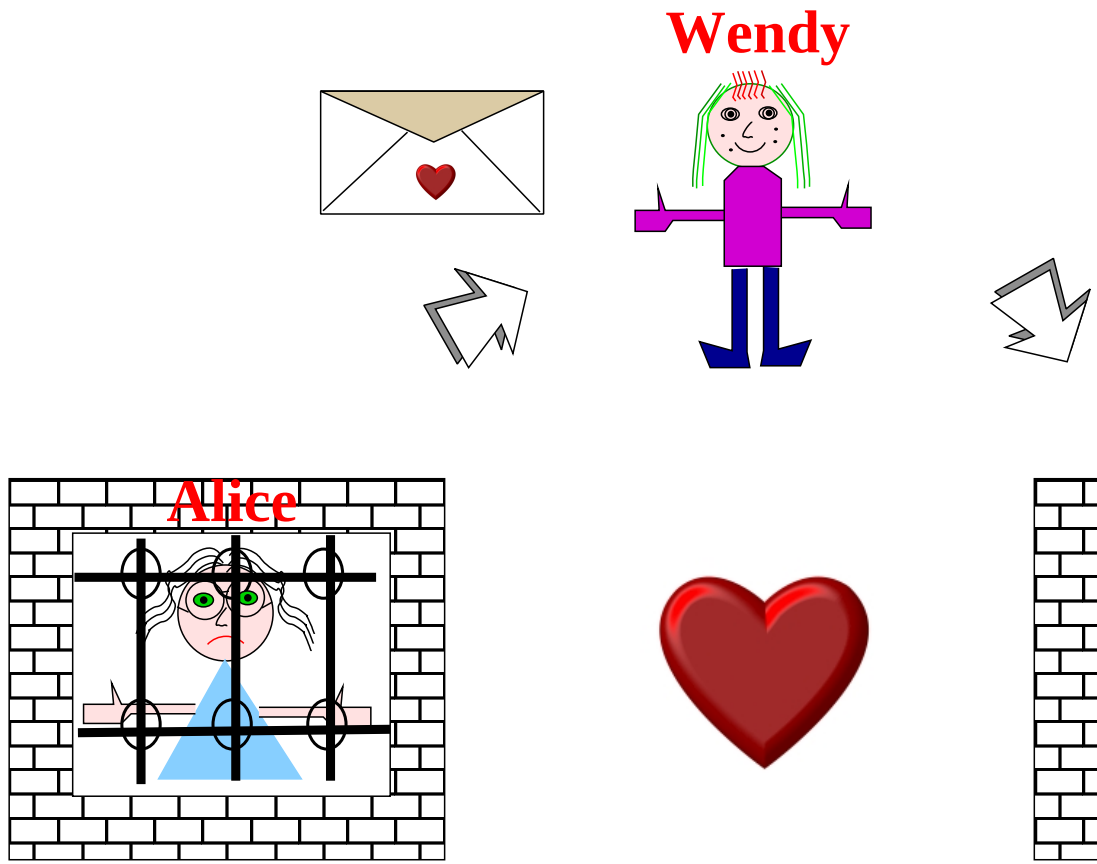


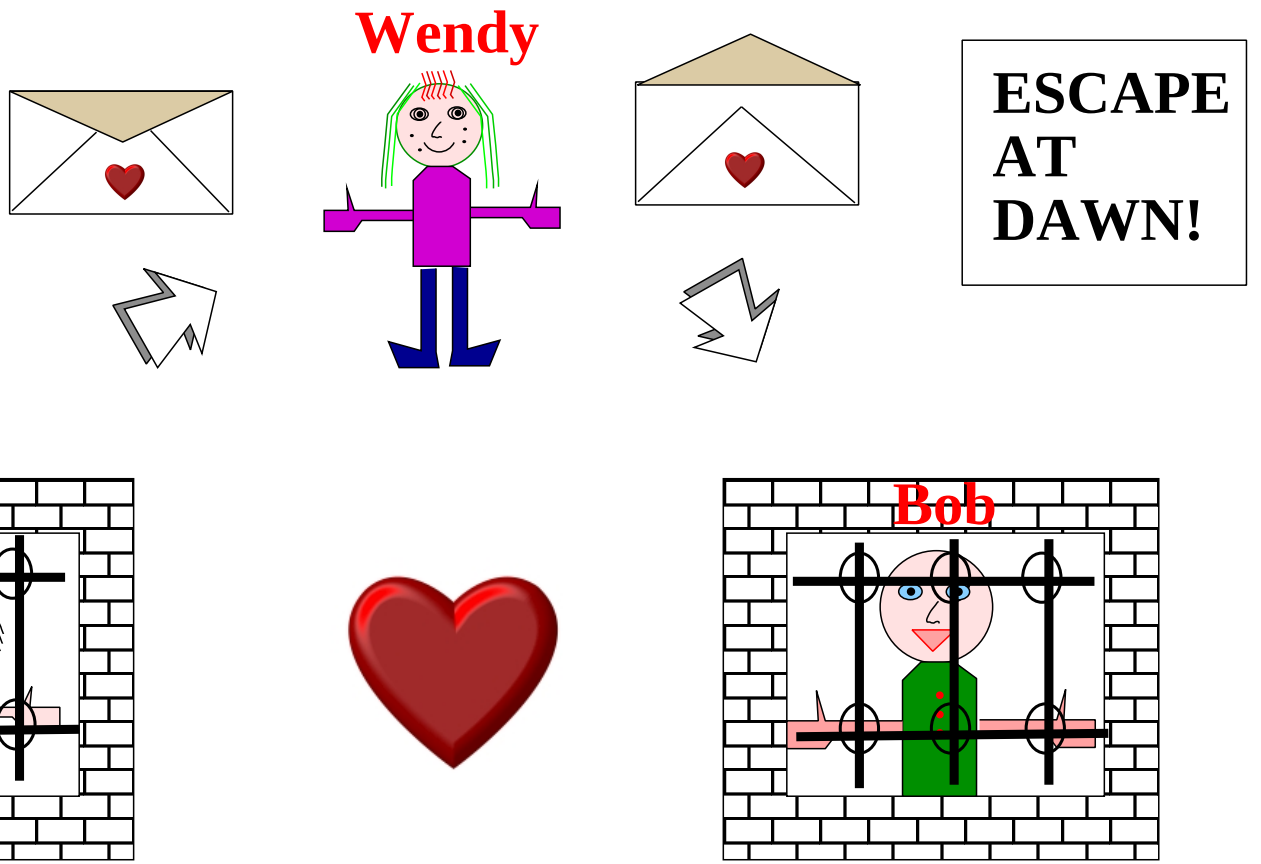
# Prisoners' Problem





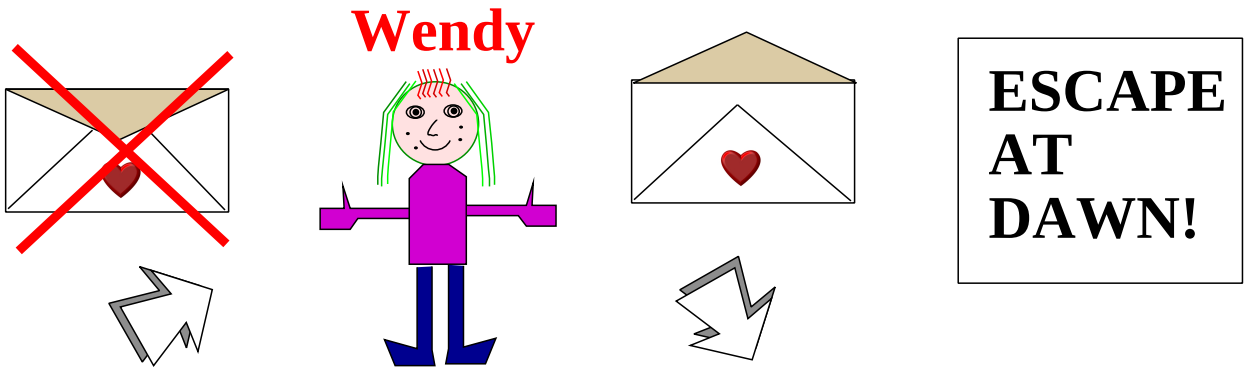
# Prisoners' Problem





# Prisoners' Problem

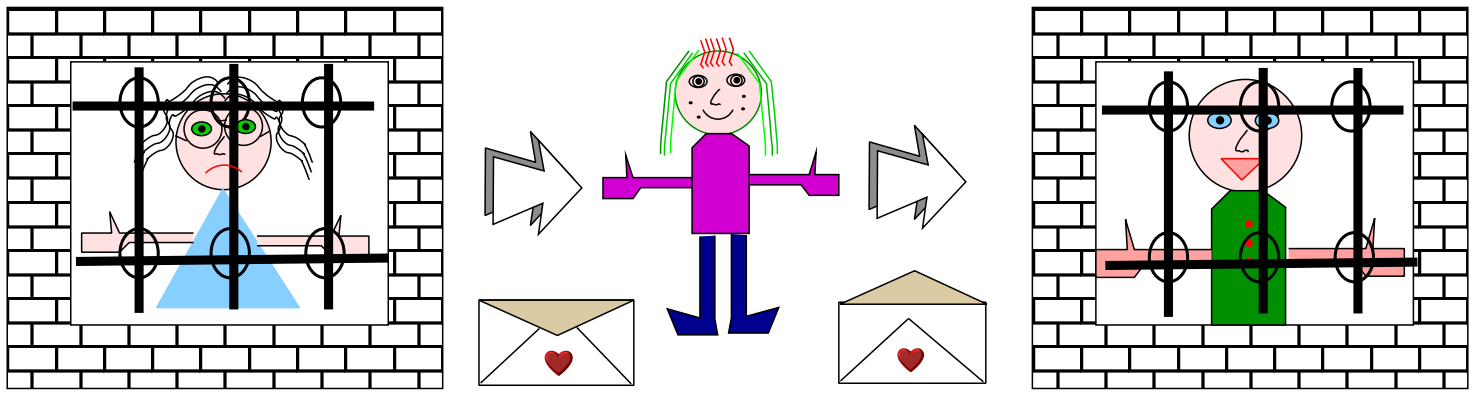





# First Try — Cryptography

ESCAPE AT DAWN





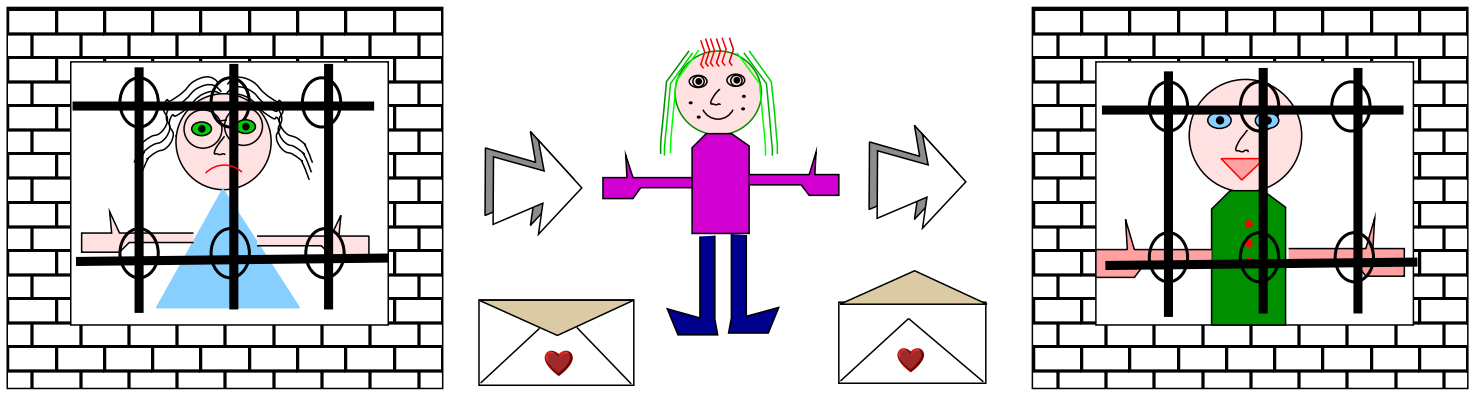
# First Try — Cryptography



ESCAPE AT DAWN

Encrypt

HJ*^(KDLF*^(DJHGA



# First Try — Cryptography

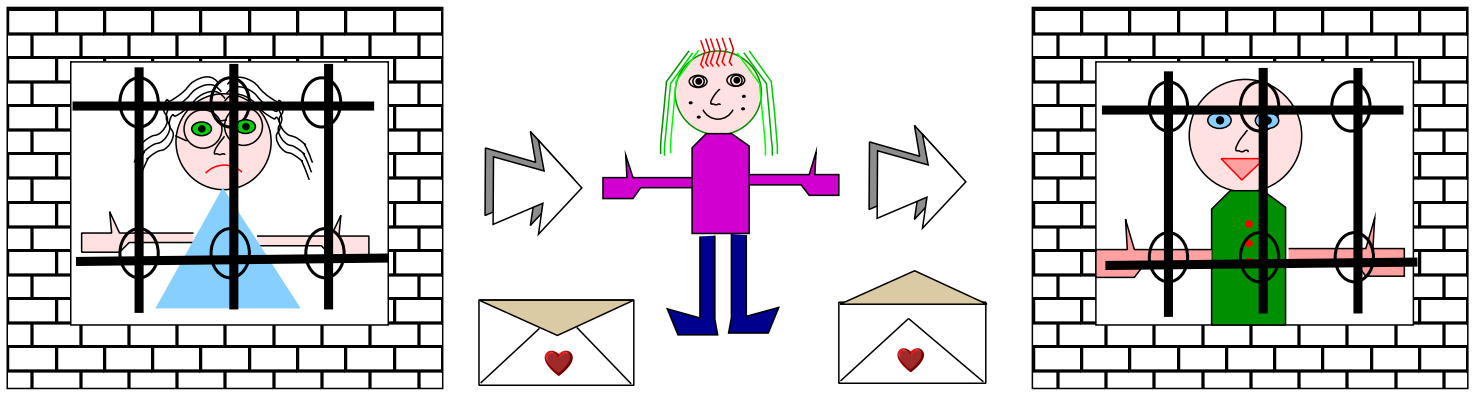
ESCAPE AT DAWN

Encrypt

HJ*^(KDLF*^(DJHGA

HJ*^(KDLF*^(DJHGA





# First Try — Cryptography



ESCAPE AT DAWN

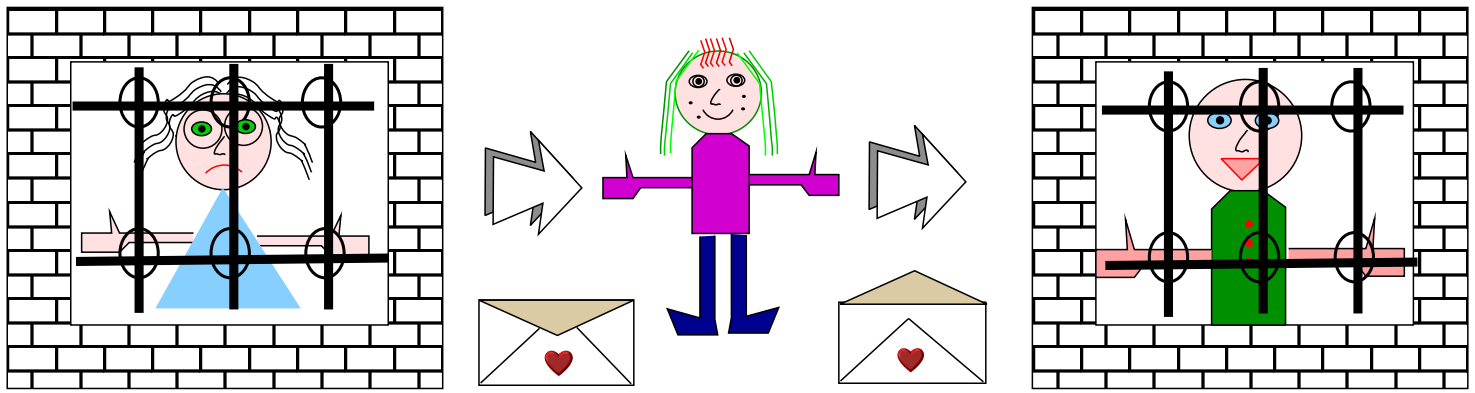
Encrypt

HJ*^(KDLF*^(DJHGA

HJ*^(KDLF*^(DJHGA

Decrypt

ESCAPE AT DAWN



# First Try — Cryptography



ESCAPE AT DAWN

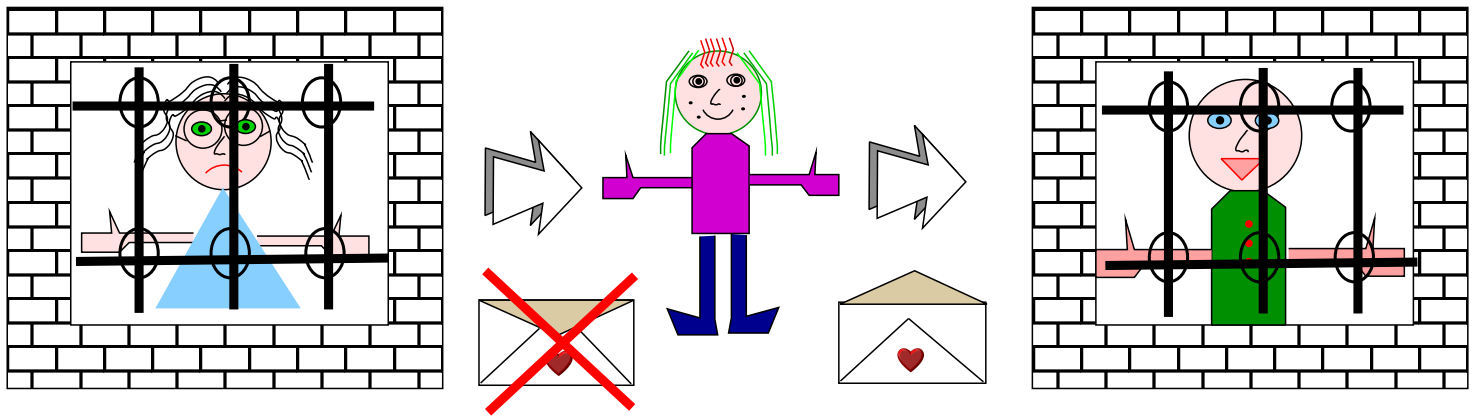
Encrypt

HJ*^(KDLF*^(DJHGA

HJ*^(KDLF*^(DJHGA

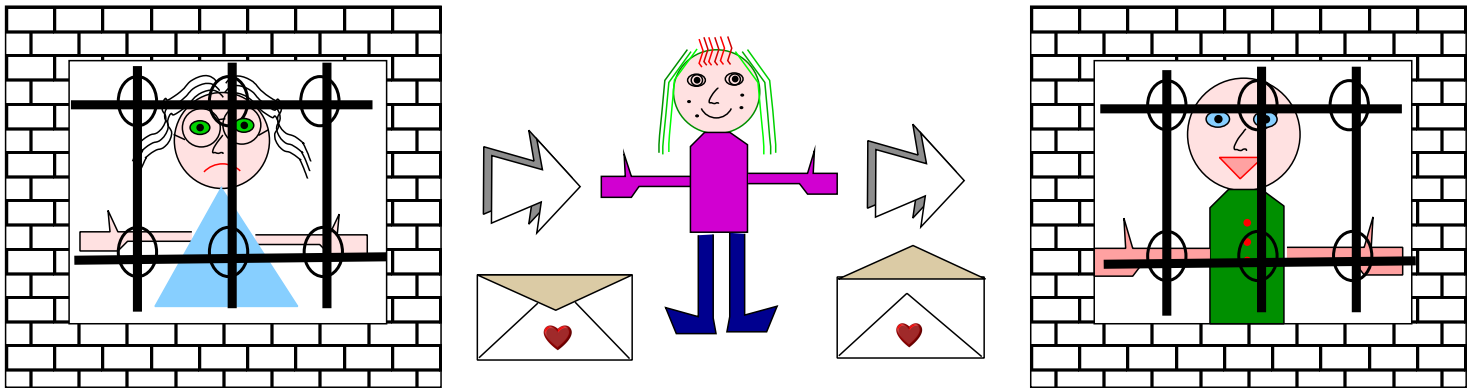
Decrypt

ESCAPE AT DAWN



# Second Try — Steganography (A Null Cipher)

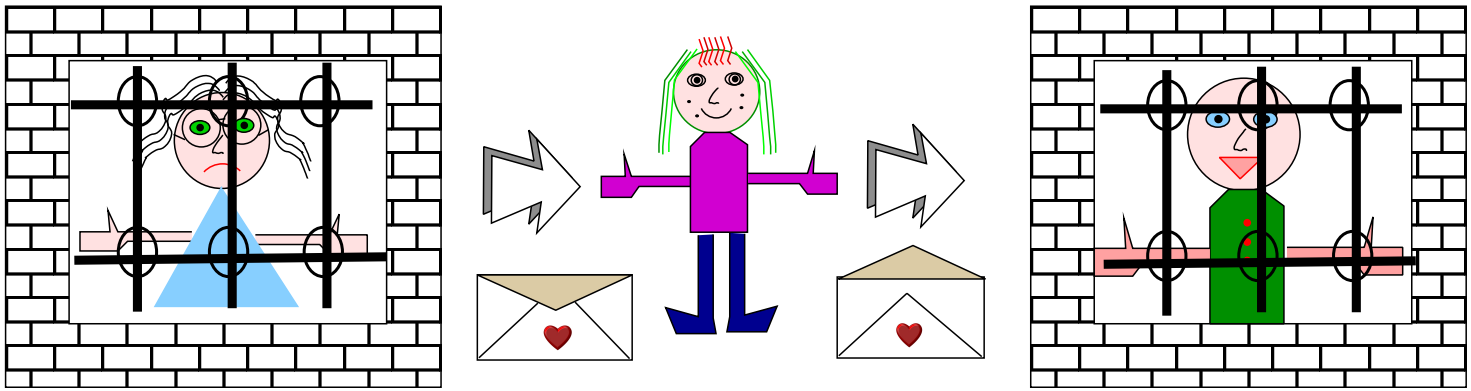




# Second Try — Steganography (A Null Cipher)



Easter is soon, dear!
So many flowers!
Can you smell them?
Are you cold at night?
Prison food stinks!
Eat well, still!
Are you lonely?
Take care!
Don't worry!
All is well!
Wendy is nice!
Need you! ):

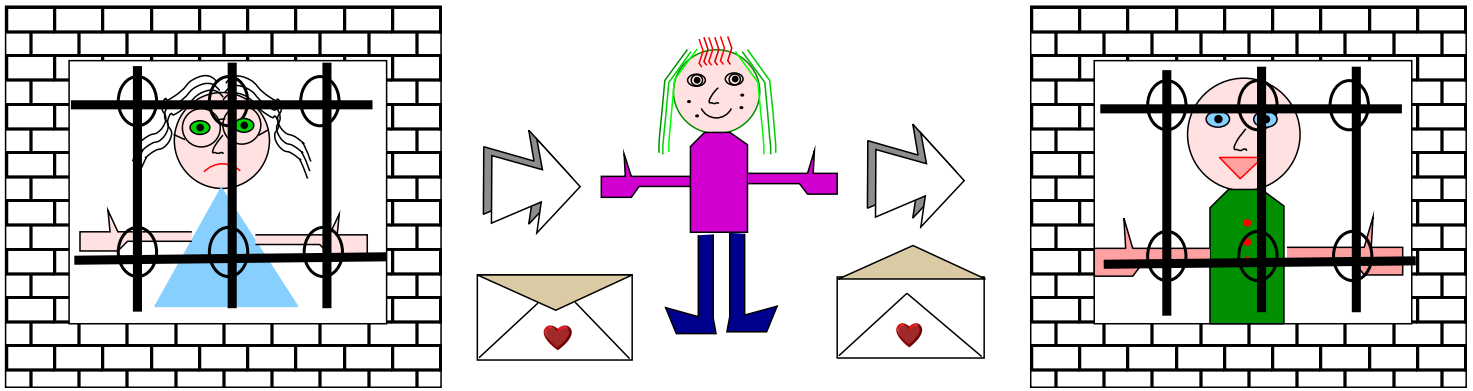


# Second Try — Steganography (A Null Cipher)



Easter is soon, dear!
So many flowers!
Can you smell them?
Are you cold at night?
Prison food stinks!
Eat well, still!
Are you lonely?
Take care!
Don't worry!
All is well!
Wendy is nice!
Need you! ):

ESCAPE AT DAWN



# Second Try — Steganography (A Null Cipher)

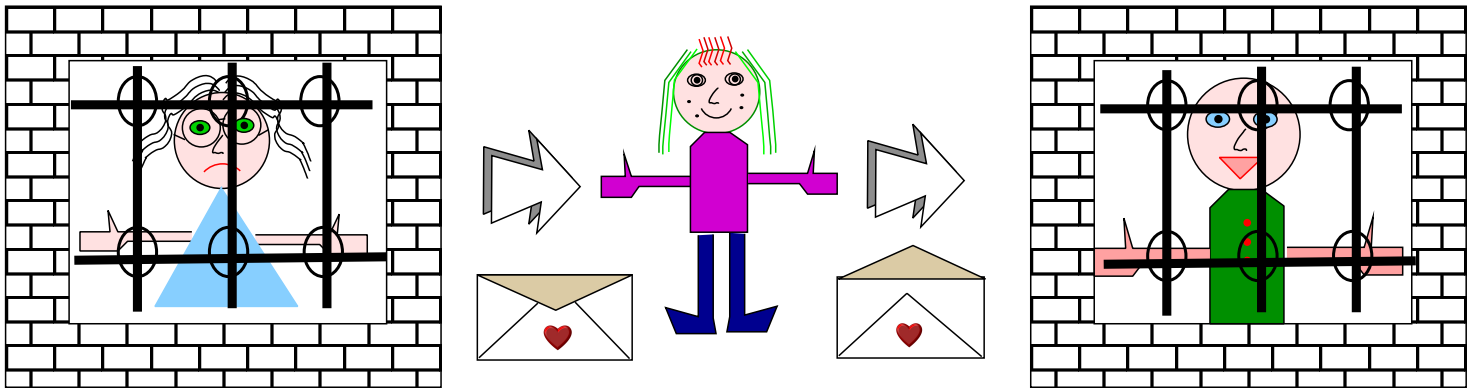


Easter is soon, dear!
So many flowers!
Can you smell them?
Are you cold at night?
Prison food stinks!
Eat well, still!
Are you lonely?
Take care!
Don't worry!
All is well!
Wendy is nice!
Need you! ):

**Covertext**

**Hidden text (payload)**

ESCAPE AT DAWN



# Second Try — Steganography (A Null Cipher)

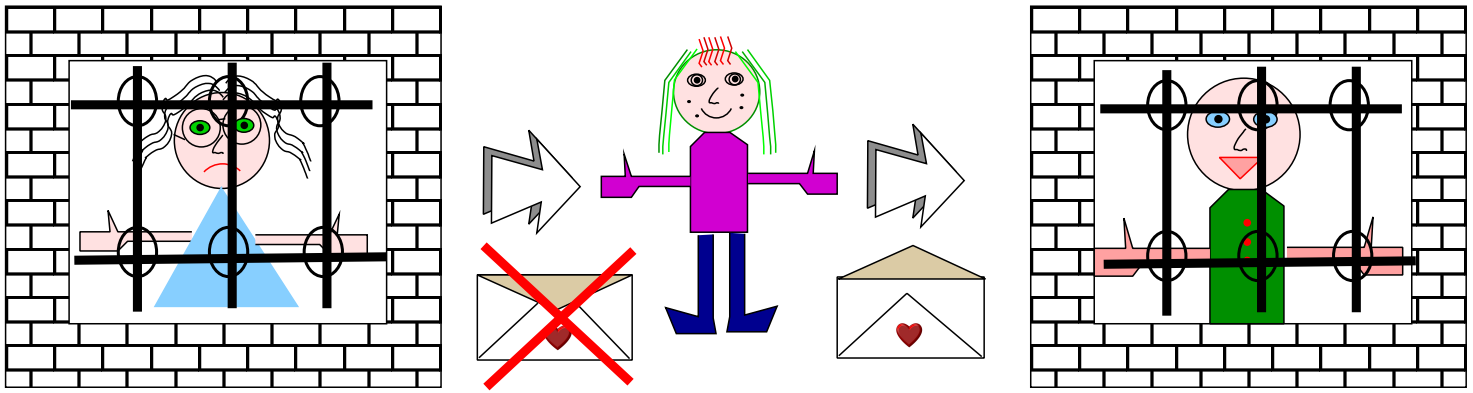


Easter is soon, dear!
So many flowers!
Can you smell them?
Are you cold at night?
Prison food stinks!
Eat well, still!
Are you lonely?
Take care!
Don't worry!
All is well!
Wendy is nice!
Need you! ):

**Covertext**

**Hidden text (payload)**

ESCAPE AT DAWN

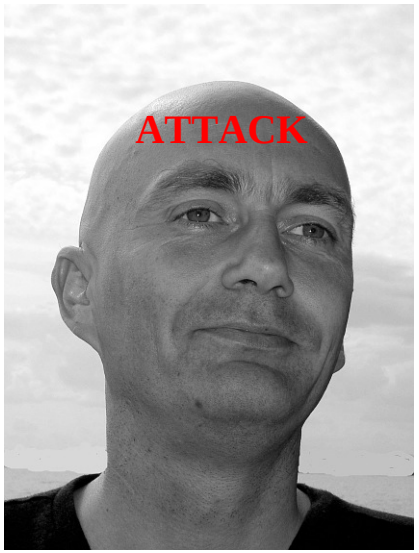


# Classic Steganography



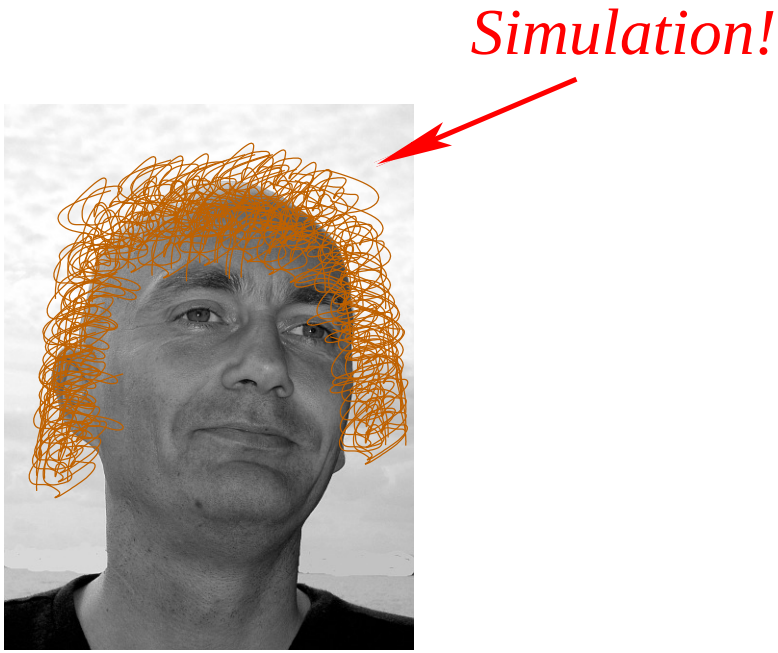
# Classic Steganography





# Classic Steganography





# Classic Steganography



# Classic Steganography

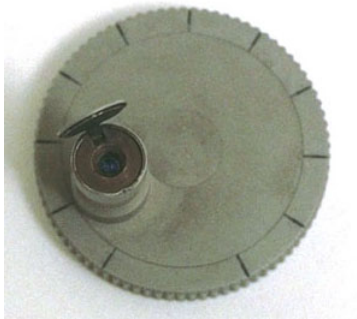


# Classic Steganography



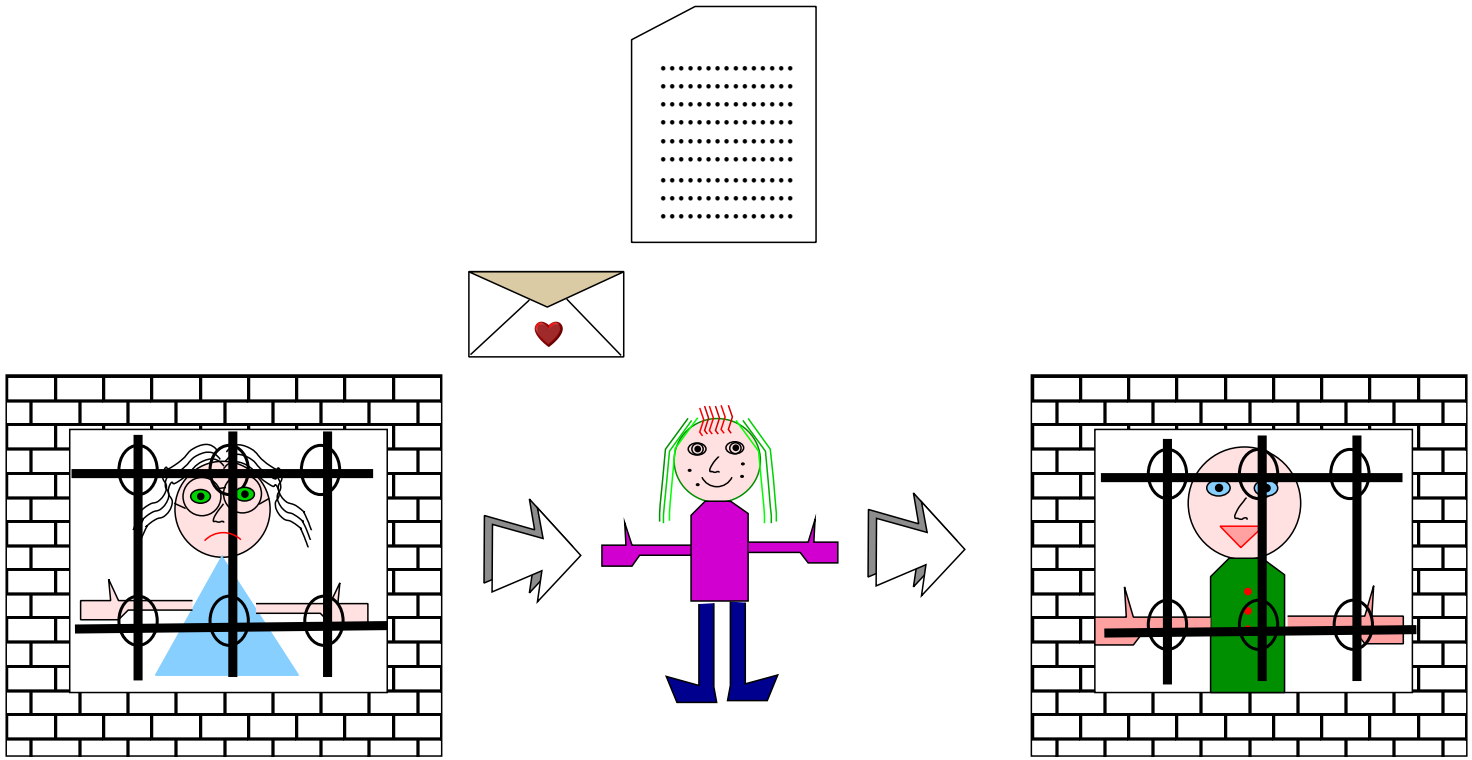

# Classic Steganography



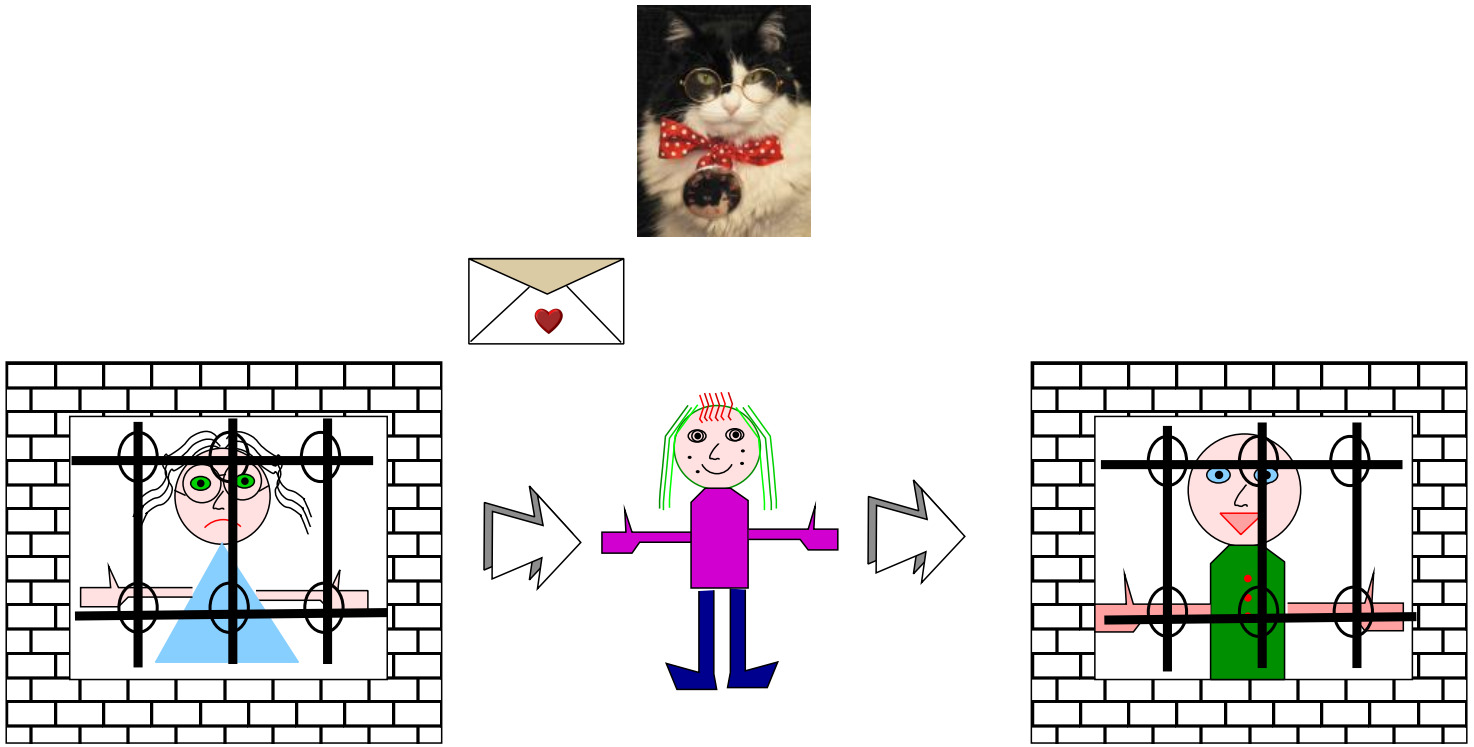
# Choice of Covertext





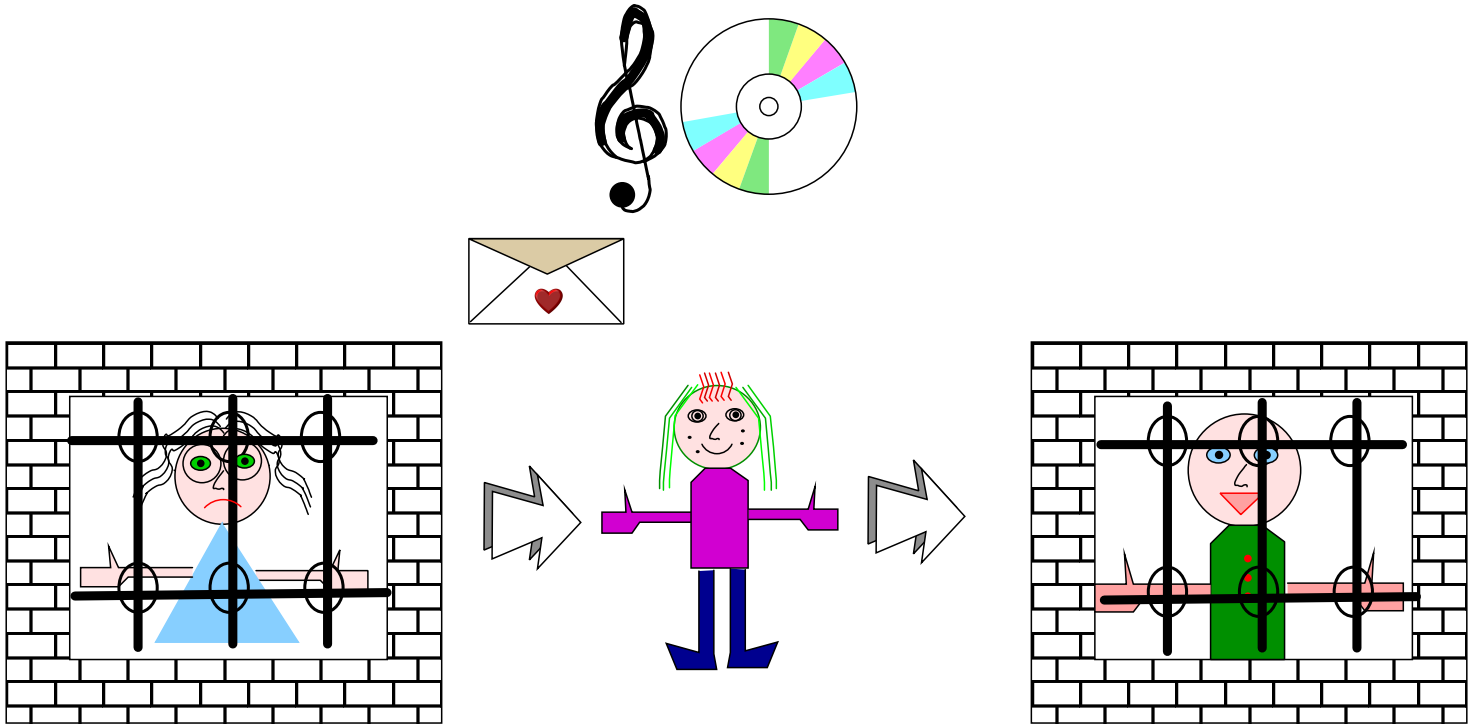
# Choice of Covertext





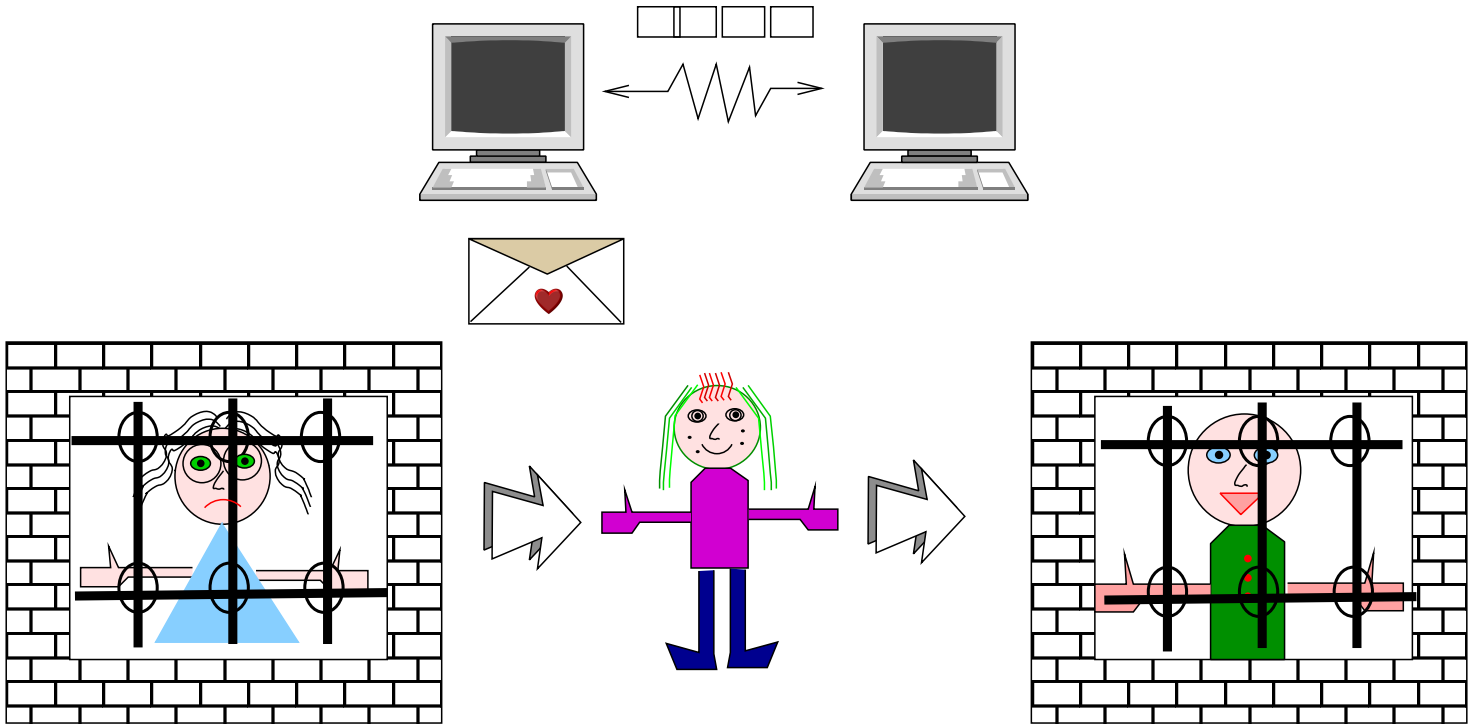
# Choice of Covertext





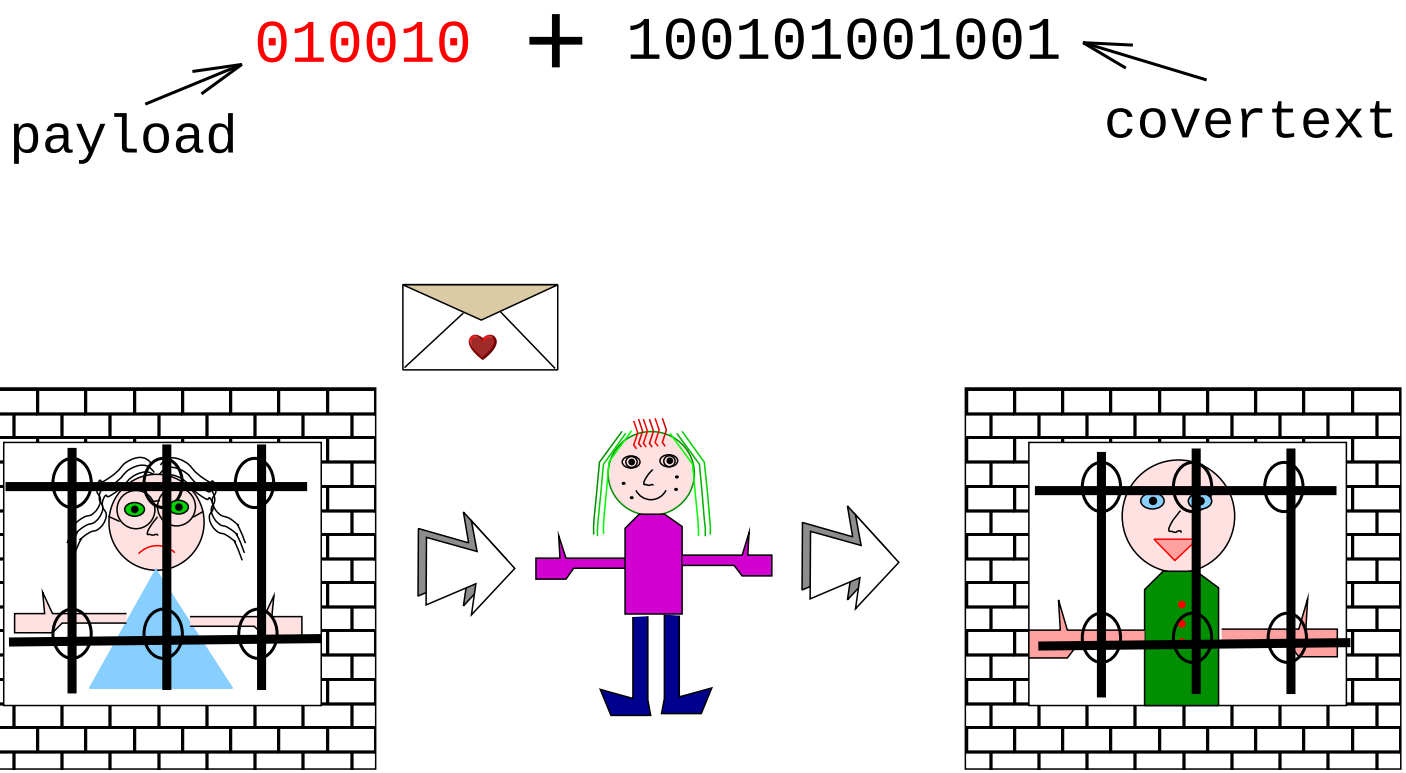
# Choice of Covertext





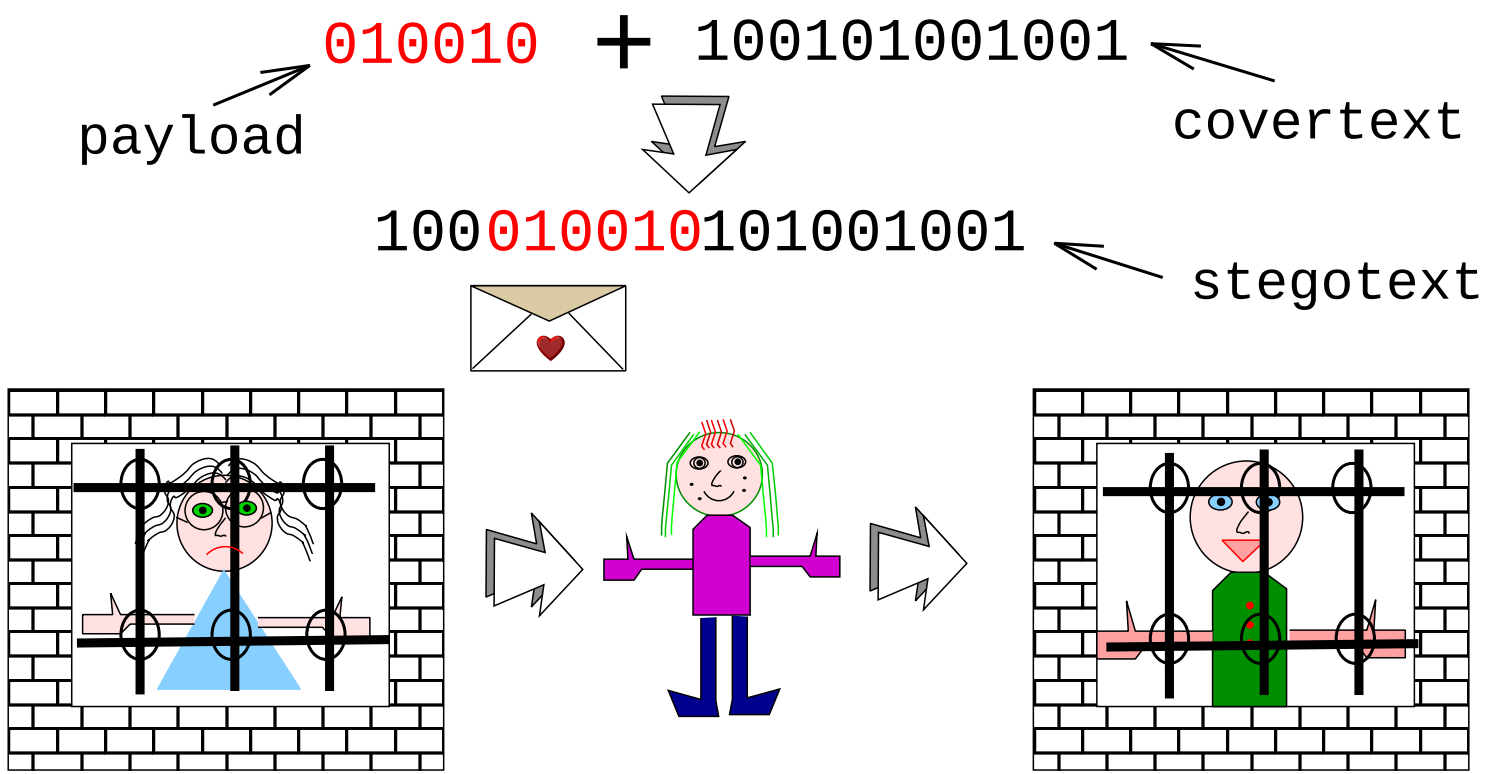
# Choice of Covertext





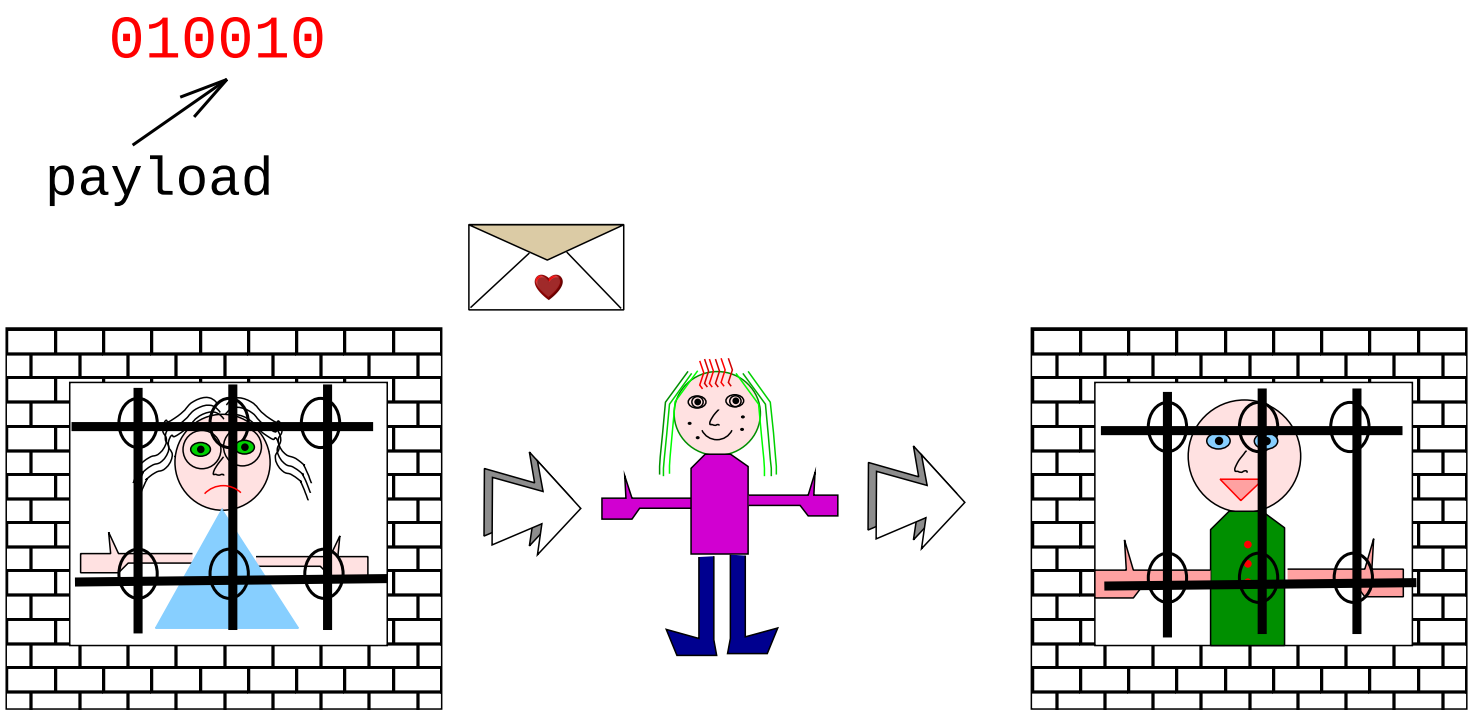
# Choice of Covertext





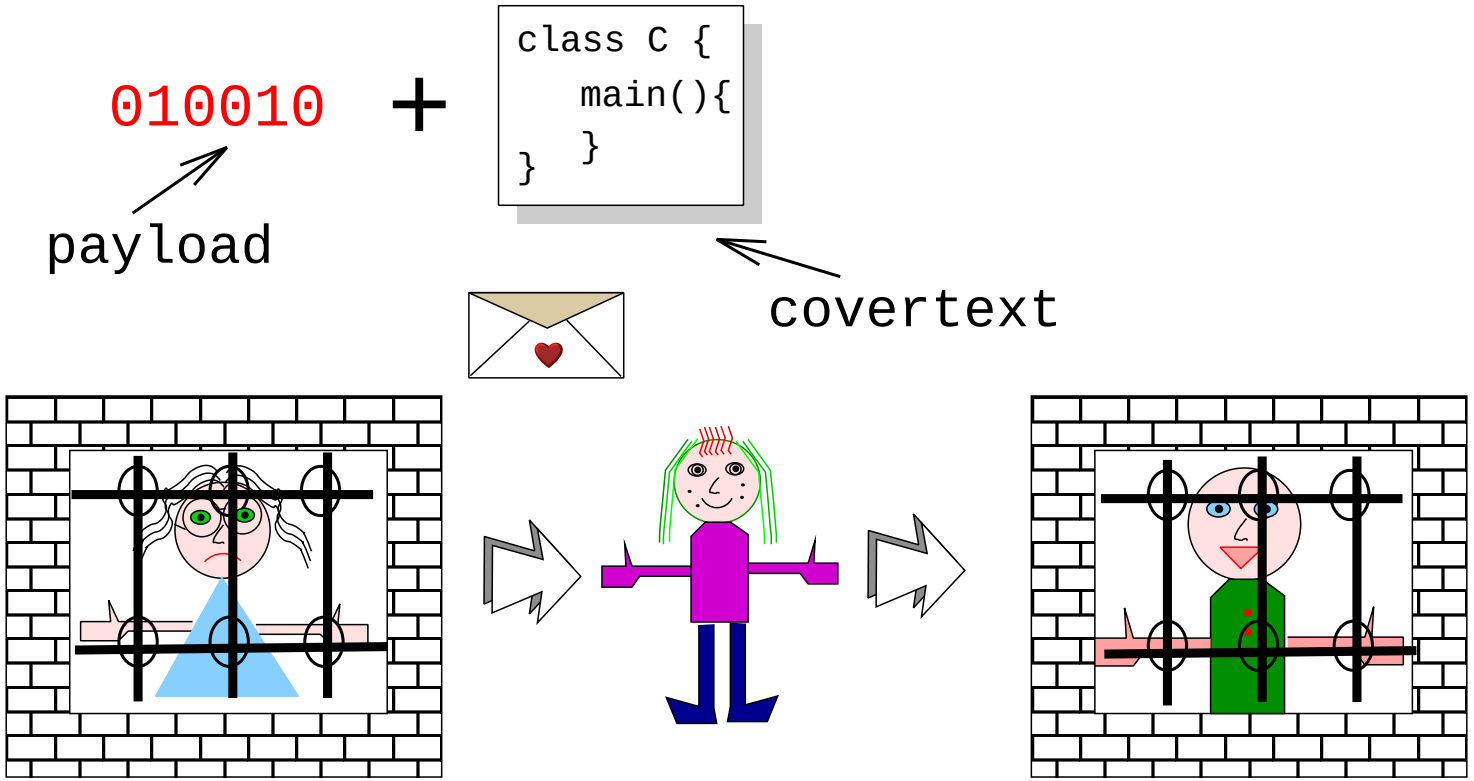
# Choice of Covertext





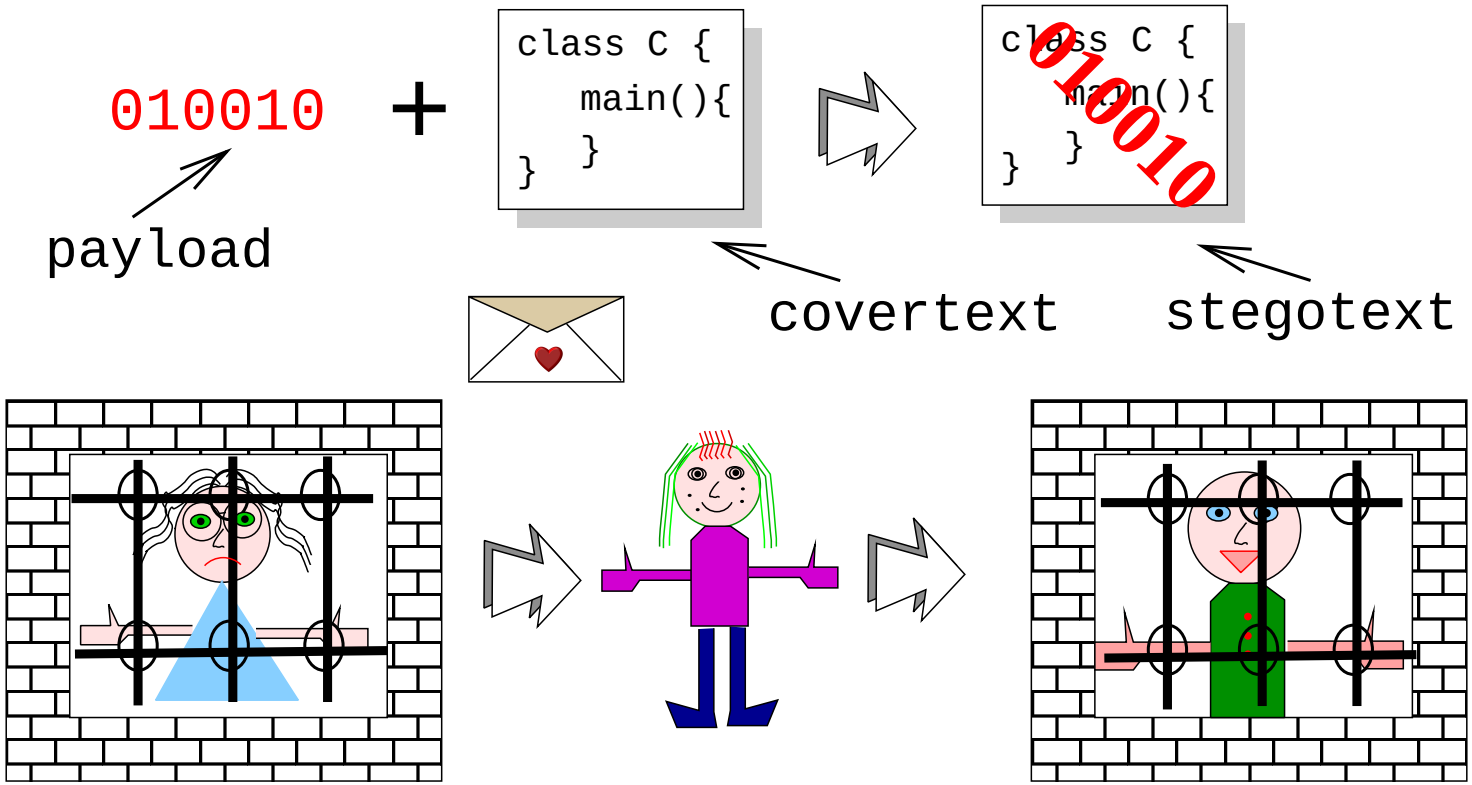
# Choice of Covertext





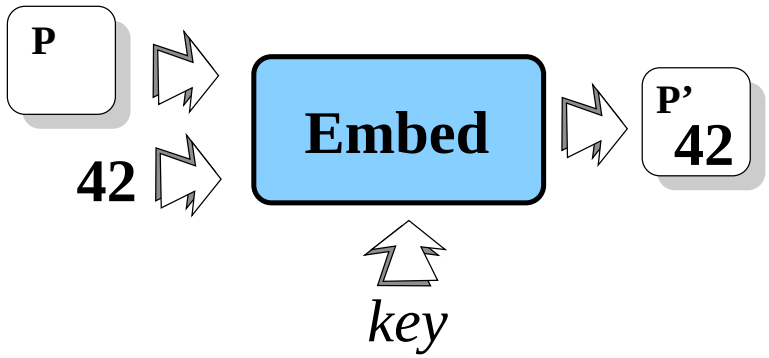
# Choice of Covertext





# Software Watermarking



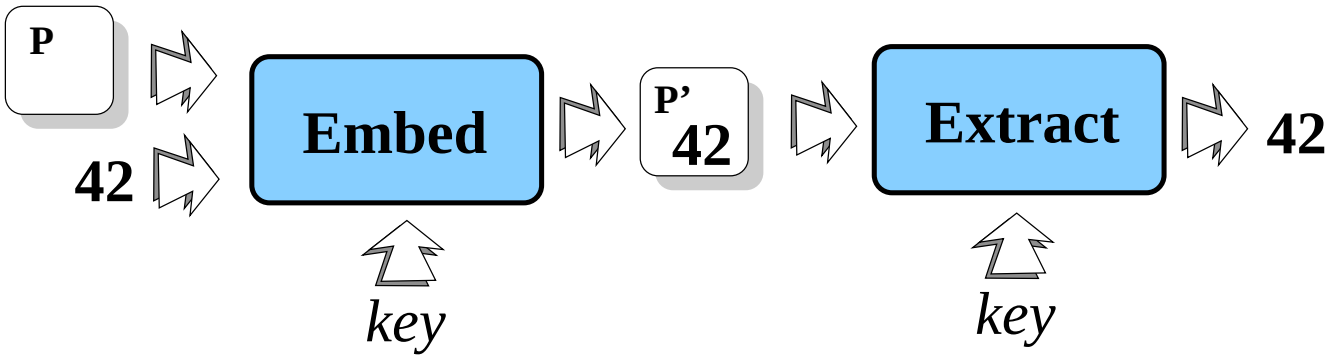



Embed an integer $W$ in program $P$ such that

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

# Software Watermarking



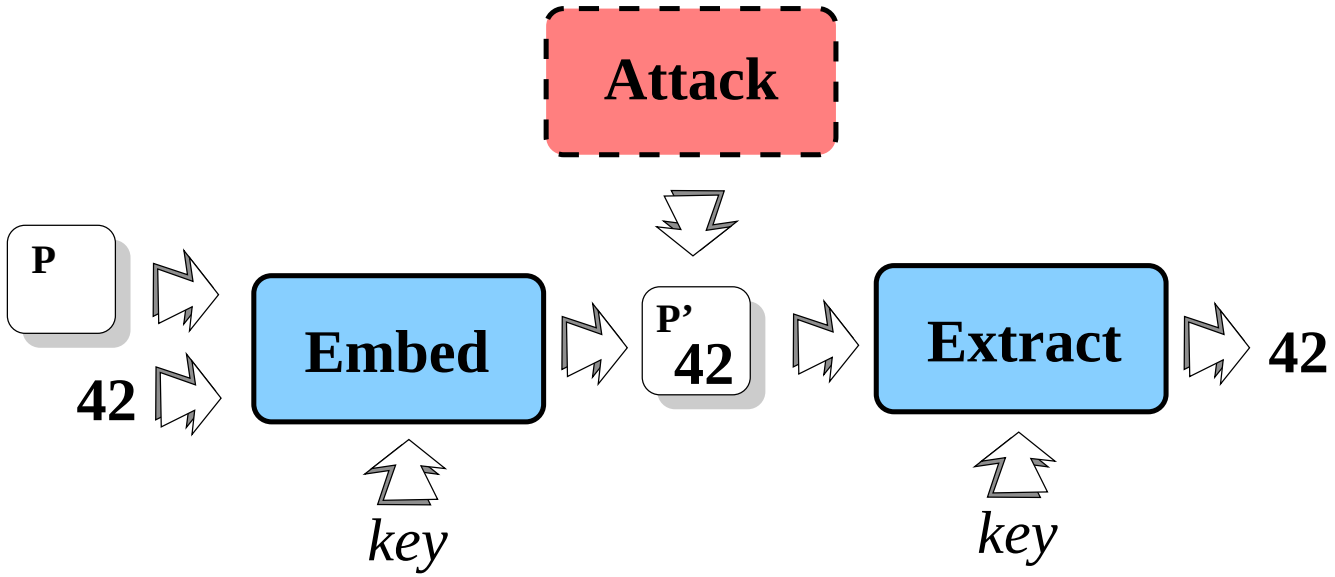



Embed an integer $W$ in program $P$ such that

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

# Software Watermarking



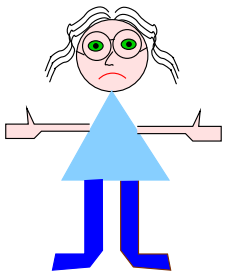


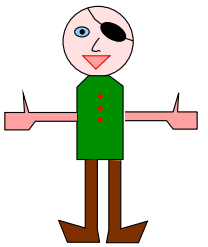

Embed an integer $W$ in program $P$ such that

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

# Attacks on Software Watermarks

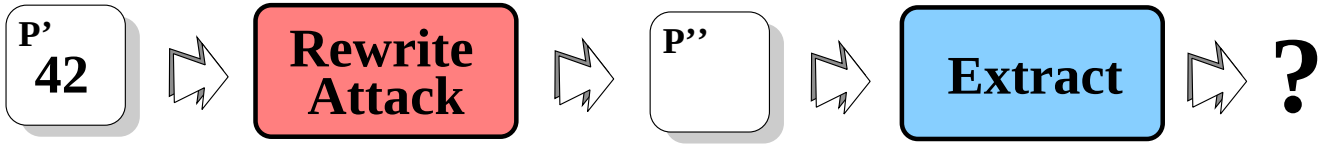


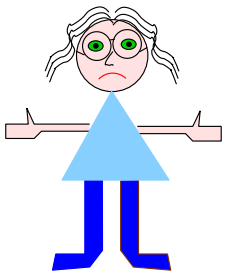


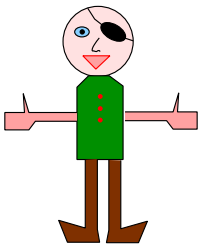


# Attacks on Software Watermarks



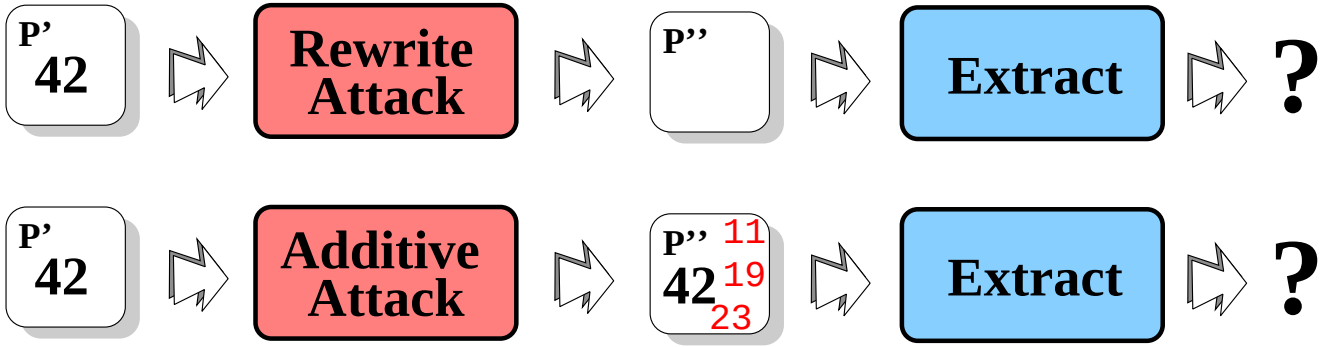


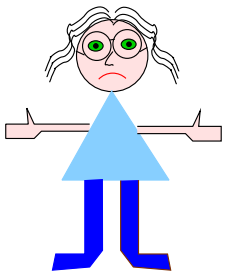



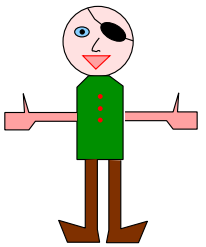


# Attacks on Software Watermarks



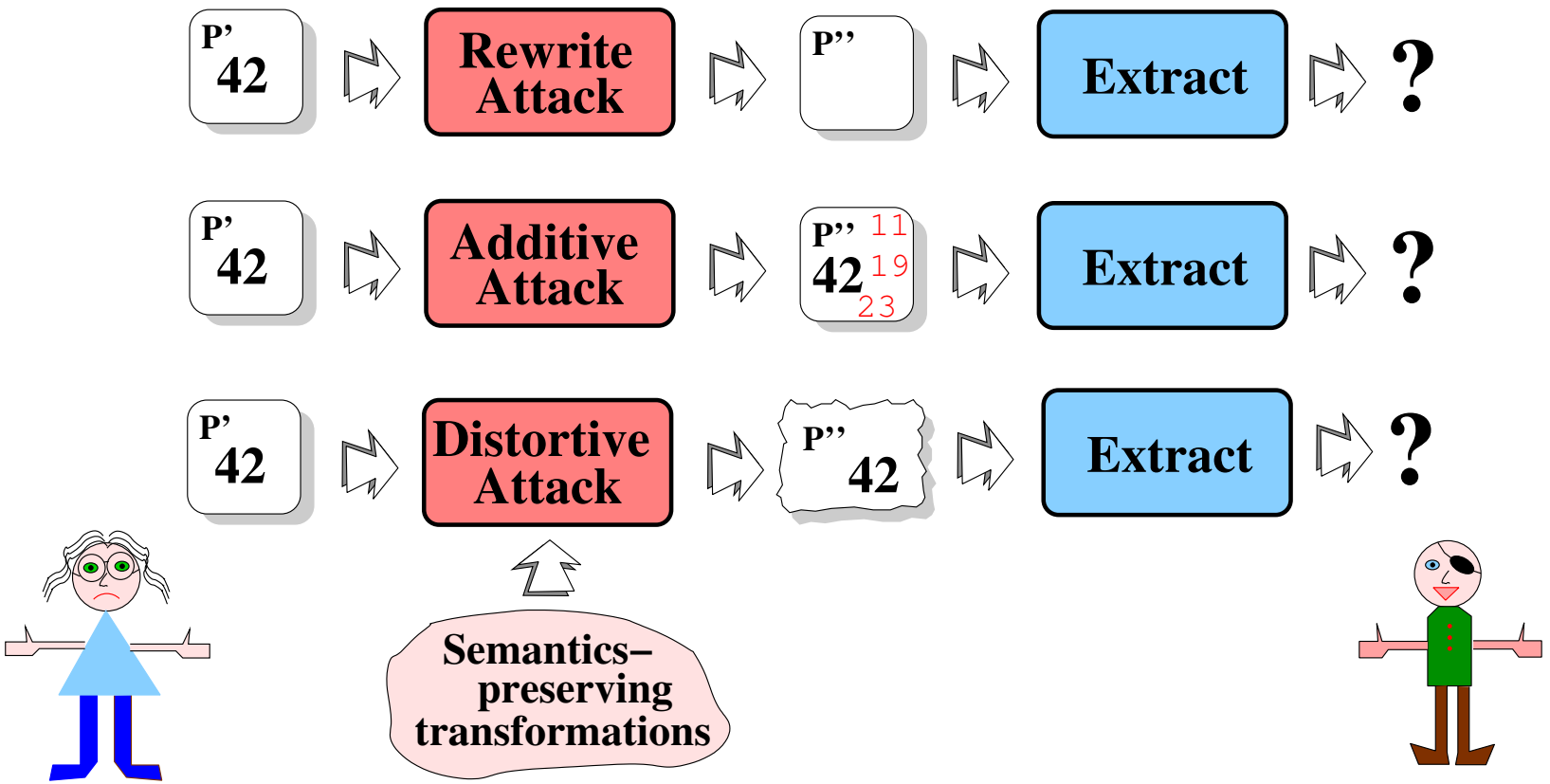




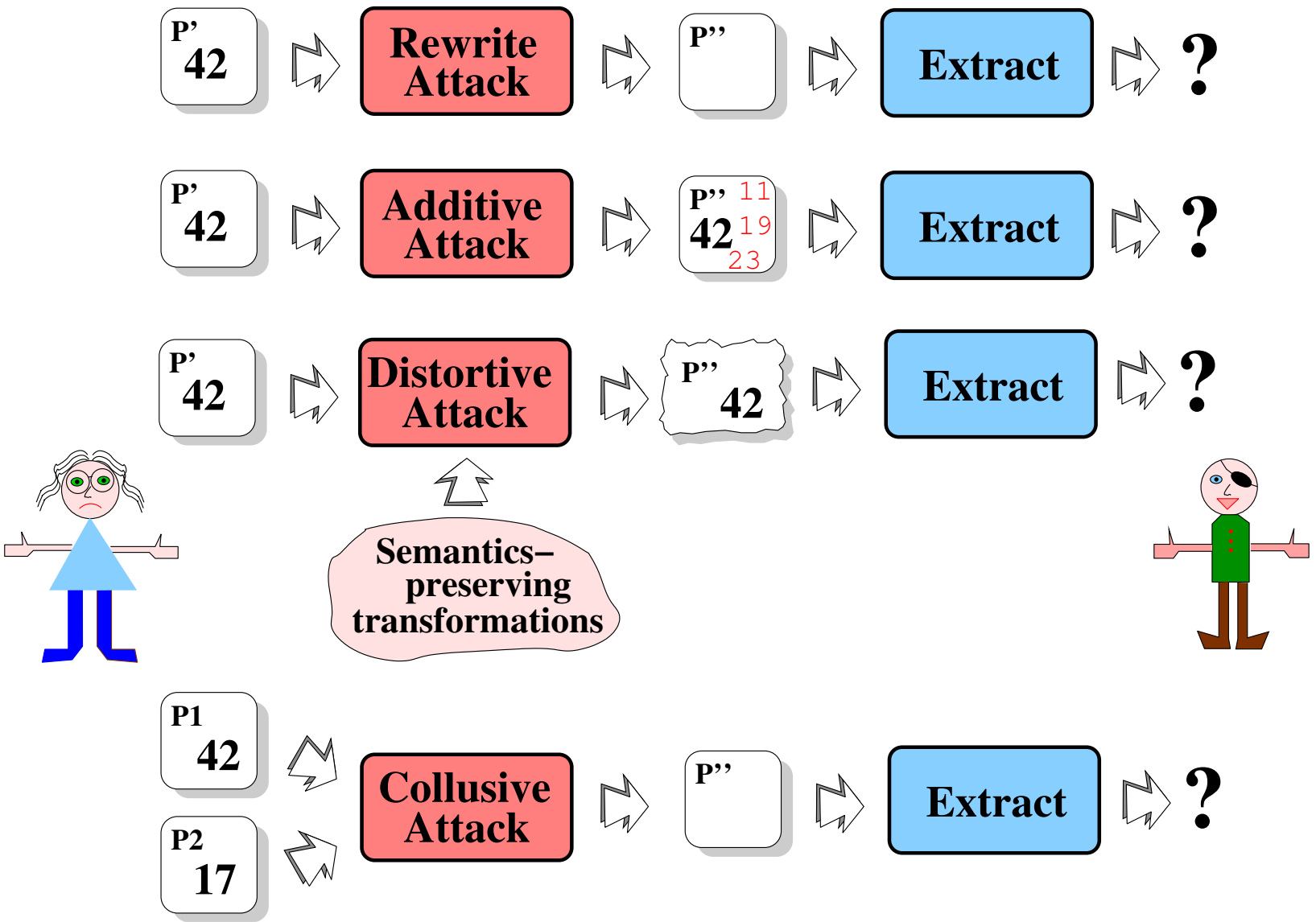
# Attacks on Software Watermarks





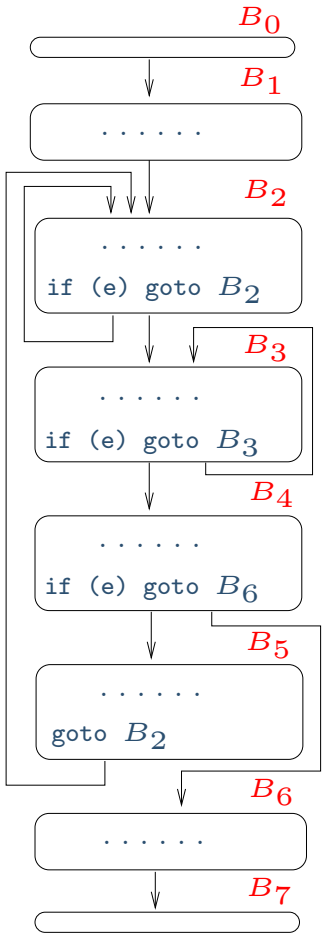
# Attacks on Software Watermarks





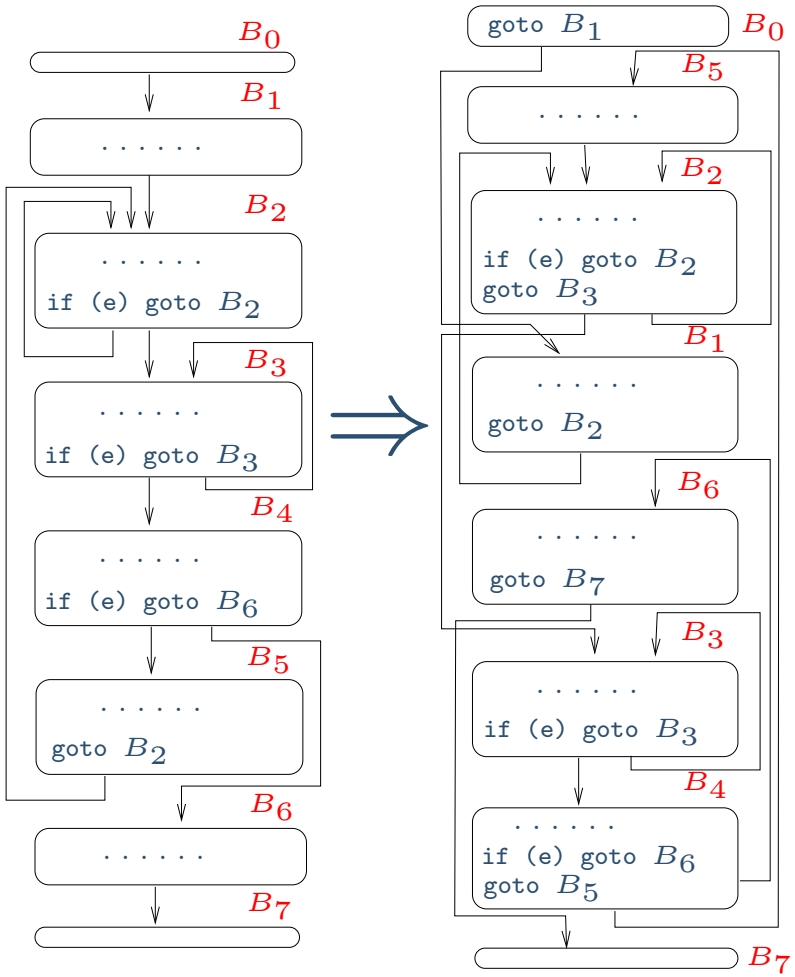

# — Davidson & Myhrvold






# — Davidson & Myhrvold

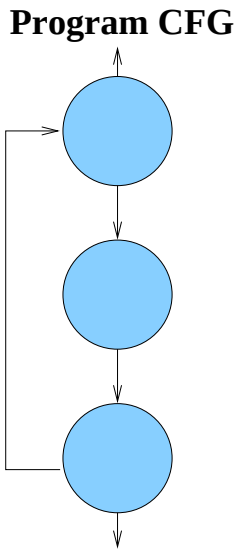




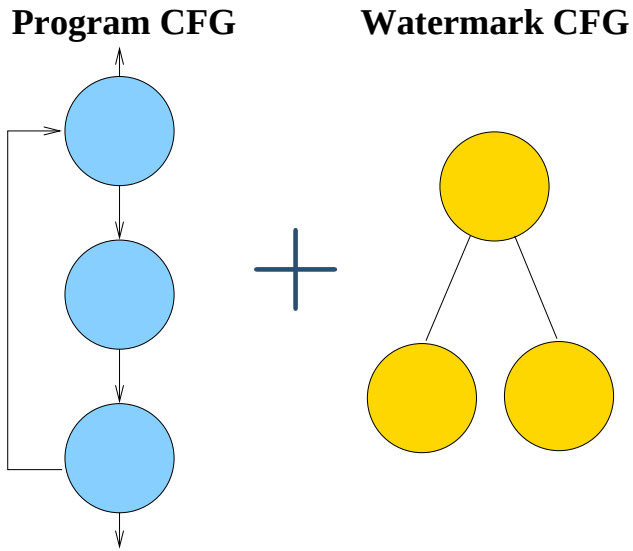

- The watermark is encoded in the basic block sequence $\langle B_5, B_2, B_1, B_6, B_3, B_4 \rangle$.

US Patent 5,559,884, 1996, Microsoft


## — Venkatesan et al.

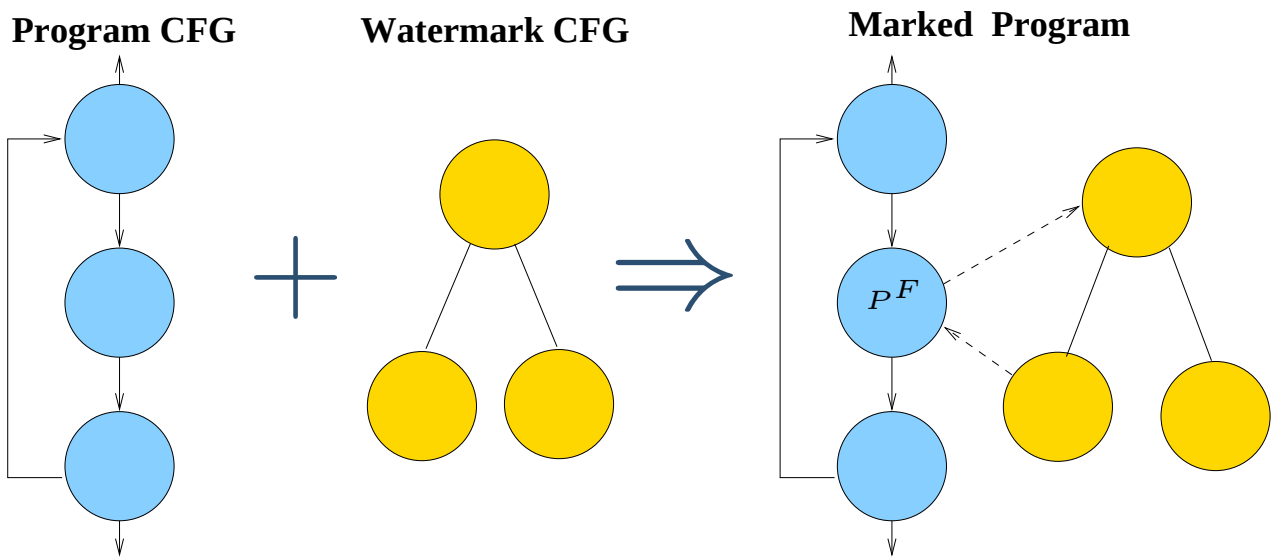



## — Venkatesan et al.





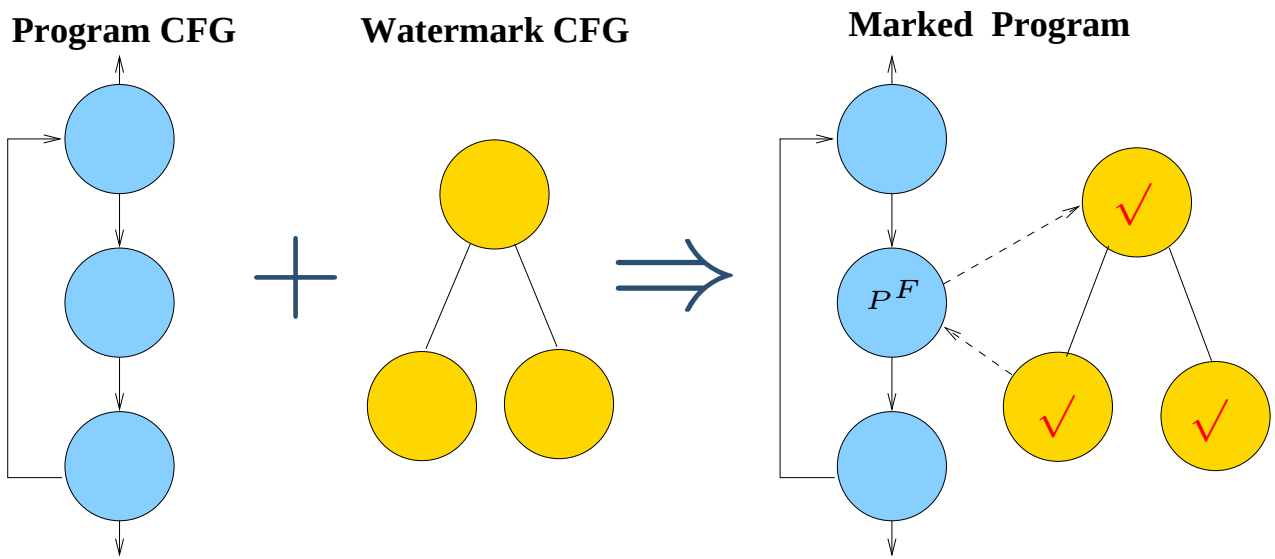

## — Venkatesan et al.




- Bogus branches tie the watermark CFG to the program.




## — Venkatesan et al.

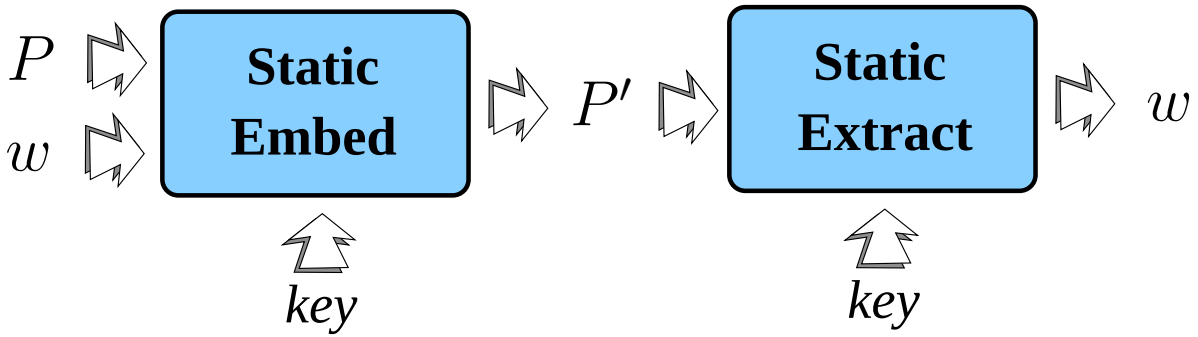



- Bogus branches tie the watermark CFG to the program.
- Basic blocks are marked so the watermark graph can be found.

Venkatesan et al., 4th Information Hiding Workshop, IHW'01

Collberg et al., 6th Information Hiding Workshop, IHW'04

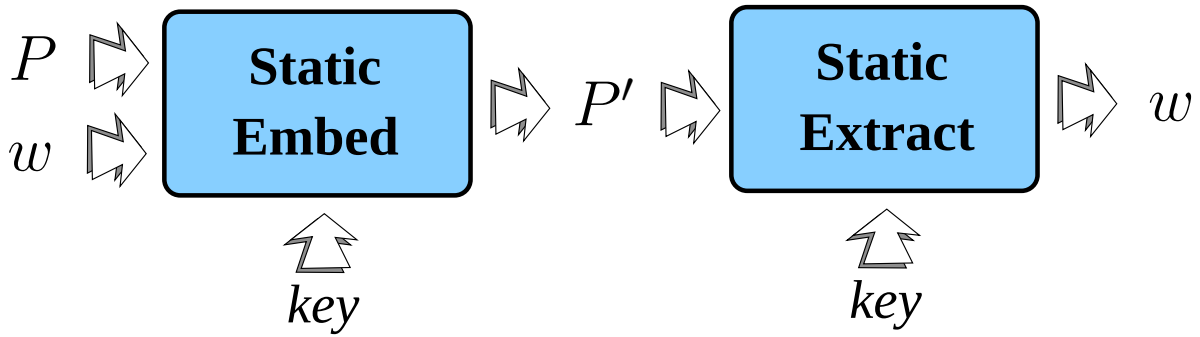
# Static vs. Dynamic Watermarking



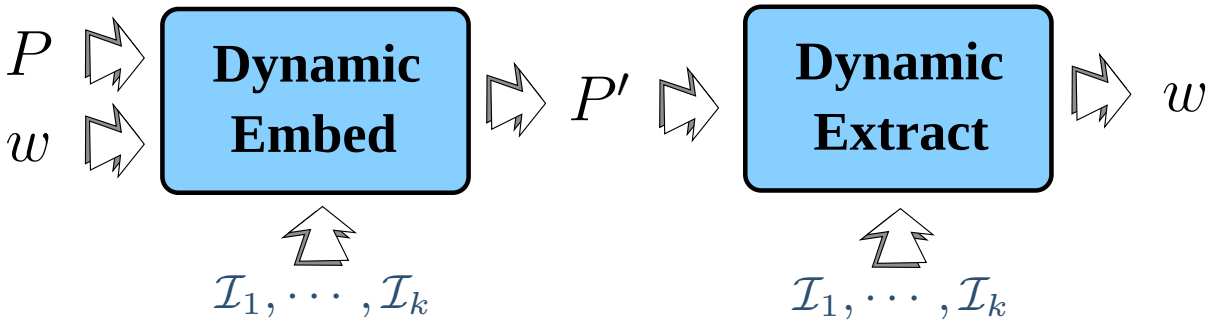

- Static algorithms are vulnerable to semantics-preserving code transformations.

# Static vs. Dynamic Watermarking





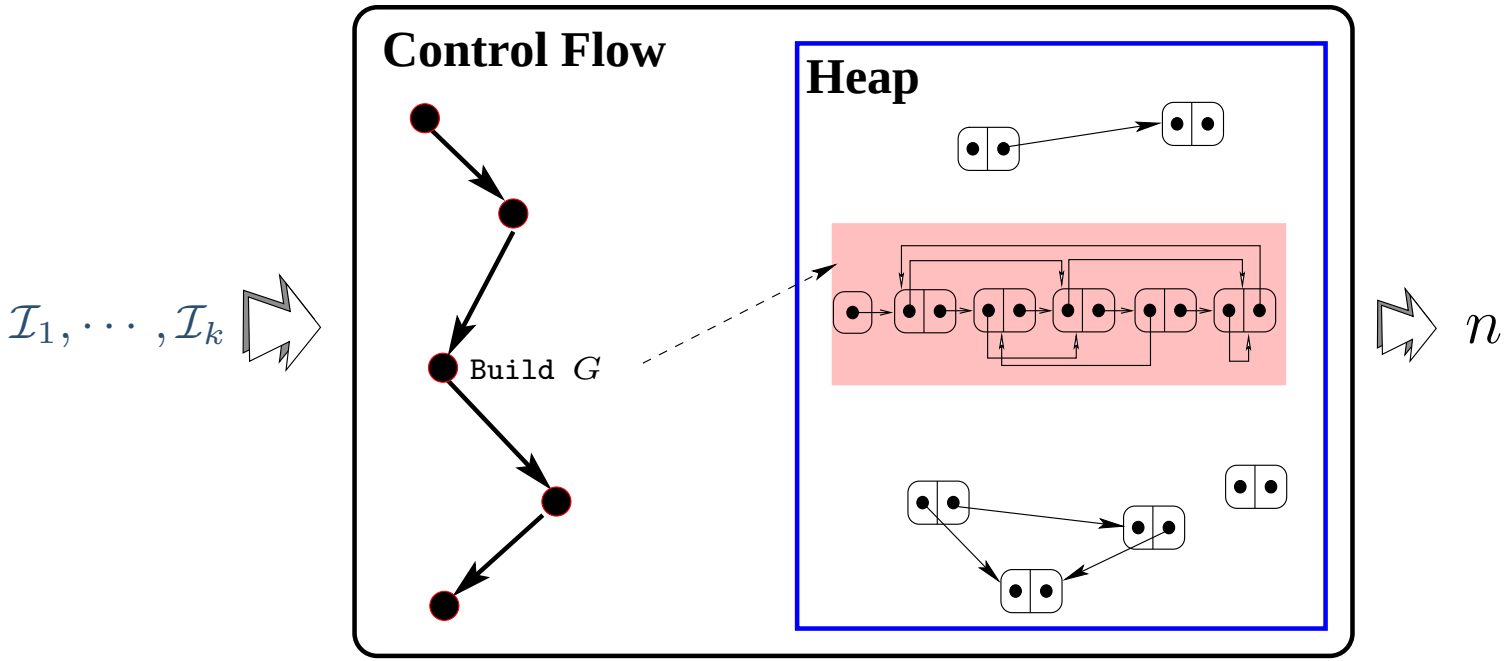

- Static algorithms are vulnerable to semantics-preserving code transformations.




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


# — Graph-Based Watermark






- 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.

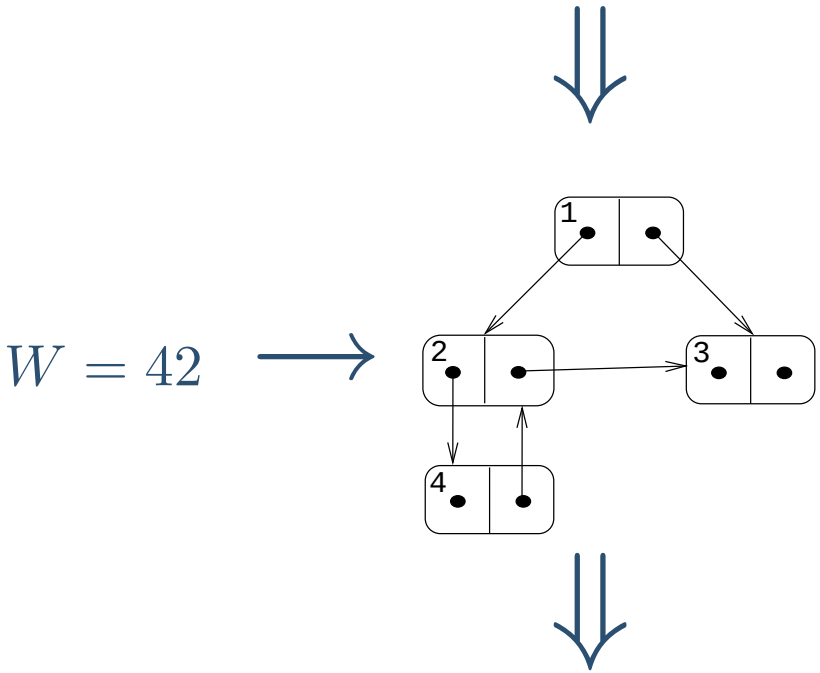
Collberg & Thomborson, ACM POPL'99

# CT — Example

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

# CT — Example

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



# CT — Example...

```
public class Node {public Node left, right;}
```

# CT — Example...

```
public class Node {public Node left, right;}
```

```
public class Main {

    public static void main(String args[]){
        System.out.println("Hello " + args[0]);









    }
}
```

# CT — Example...

```
public class Node {public Node left,right;}
```

+

```
public class Main {

    public static void main(String args[]){
        System.out.println("Hello " + args[0]);

            Node n4=new Node();
            Node n2=new Node();
            n2.left=n4; n4.right=n2;
            Node n3=new Node();
            n2.right=n3;
            Node n1=new Node();
            n1.left=n2; n1.right=n3;

    }
}
```

# CT — Example...

```
public class Node {public Node left,right;}
```

+

```
public class Main {

    public static void main(String args[]){
        System.out.println("Hello " + args[0]);
        if (i.equals("World")) {
            Node n4=new Node();
            Node n2=new Node();
            n2.left=n4; n4.right=n2;
            Node n3=new Node();
            n2.right=n3;
            Node n1=new Node();
            n1.left=n2; n1.right=n3;
        }
    }
}
```

# CT — Example...

```
public class Node {public Node left, right;}
```

+

```
public class Main {
    public static Node root;
    public static void main(String args[]){
        System.out.println("Hello " + args[0]);
        if (i.equals("World")) {
            Node n4=new Node();
            Node n2=new Node();
            n2.left=n4; n4.right=n2;
            Node n3=new Node();
            n2.right=n3;
            Node n1=new Node(); root=n1;
            n1.left=n2; n1.right=n3;
        }
    }
}
```

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?
3. bogus field addition?

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?
3. bogus field addition?
4. attacks by advanced alias analysis?

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?
3. bogus field addition?
4. attacks by advanced alias analysis?
5. global variables?

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?
3. bogus field addition?
4. attacks by advanced alias analysis?
5. global variables?
6. introducing extra unstealthy classes?

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?
3. bogus field addition?
4. attacks by advanced alias analysis?
5. global variables?
6. introducing extra unstealthy classes?
7. weak cuts?

# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?
3. bogus field addition?
4. attacks by advanced alias analysis?
5. global variables?
6. introducing extra unstealthy classes?
7. weak cuts?
8. static collusive attacks?

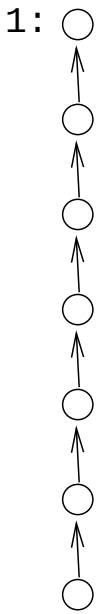
# Problems

Algorithm looks easy enough on paper, but there are lots of problems!

How do we avoid

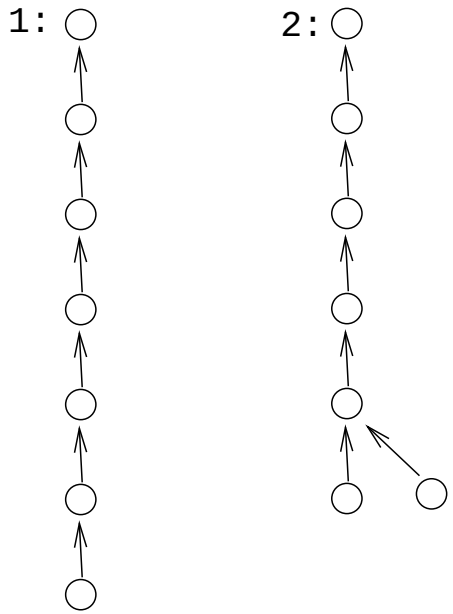
1. huge (unstealthy) graphs?
2. attacks by small graph perturbations?
3. bogus field addition?
4. attacks by advanced alias analysis?
5. global variables?
6. introducing extra unstealthy classes?
7. weak cuts?
8. static collusive attacks?
9. dynamic collusive attacks?

# Avoiding huge graphs — Enumeration Encoding



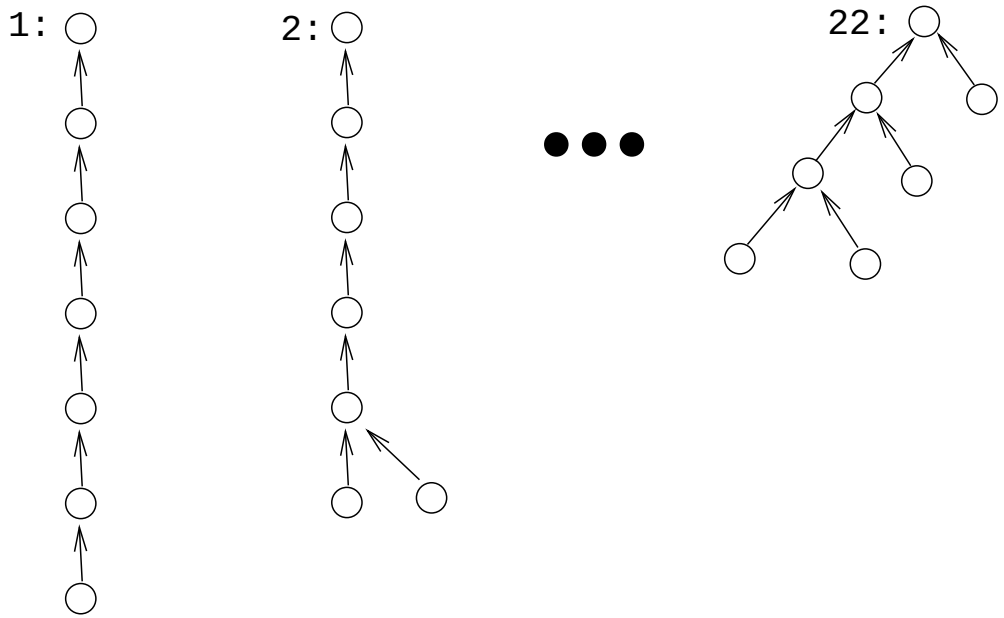

- $n$ is represented by the *index* of the the graph $G$ in some enumeration.
- We must, efficiently, be able to
  1. given $n$, generate the $n$:th graph,
  2. given $G$, find $G$s index $n$.
- Oriented parent-pointer trees $\Rightarrow$ bit-rate: 1.56 bits per word.

# Avoiding huge graphs — Enumeration Encoding



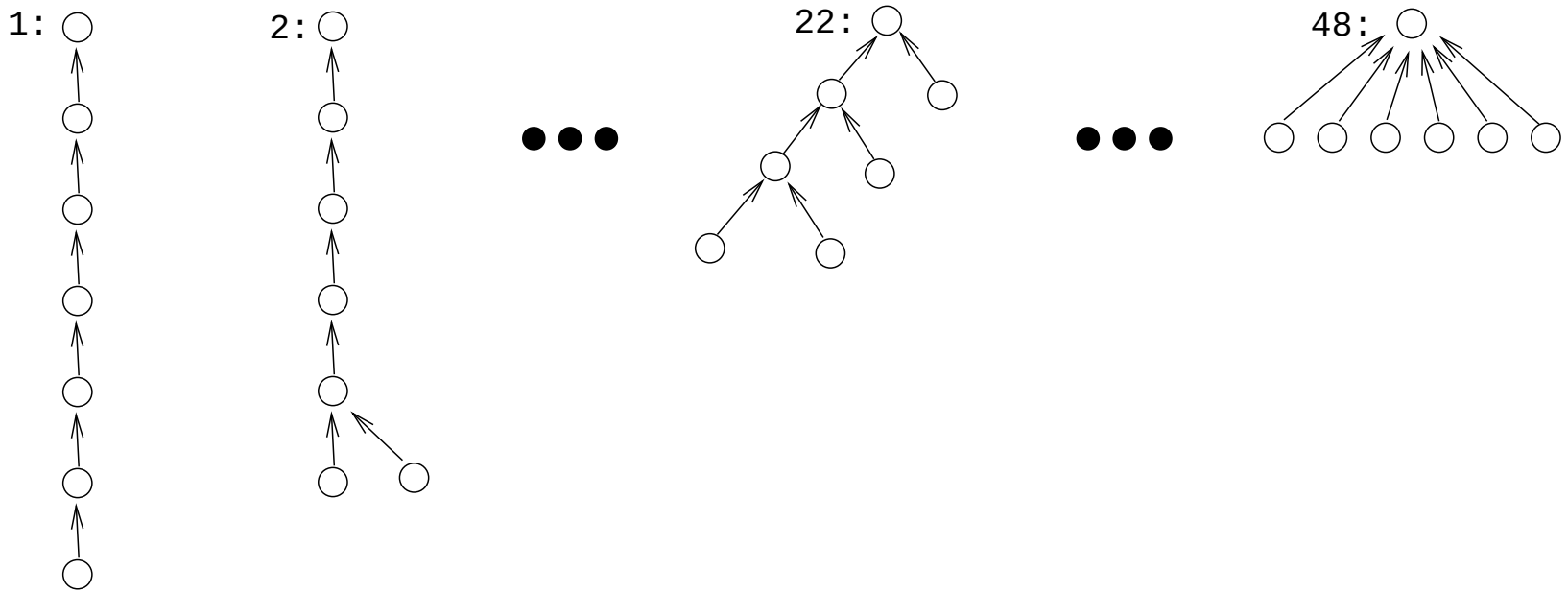

- $n$ is represented by the *index* of the the graph $G$ in some enumeration.
- We must, efficiently, be able to
  1. given $n$, generate the $n$:th graph,
  2. given $G$, find $G$s index $n$.
- Oriented parent-pointer trees $\Rightarrow$ bit-rate: 1.56 bits per word.

# Avoiding huge graphs — Enumeration Encoding



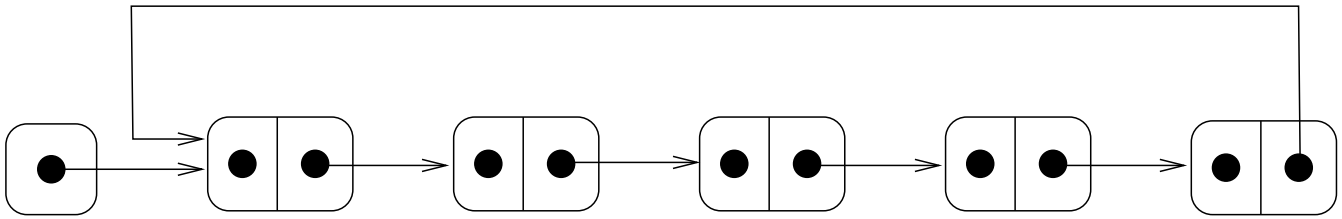

- $n$ is represented by the *index* of the the graph $G$ in some enumeration.
- We must, efficiently, be able to
  1. given $n$, generate the $n$:th graph,
  2. given $G$, find $G$s index $n$.
- Oriented parent-pointer trees $\Rightarrow$ bit-rate: 1.56 bits per word.

# Avoiding huge graphs — Enumeration Encoding




- $n$ 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 $G$s index $n$.
- Oriented parent-pointer trees $\Rightarrow$ bit-rate: 1.56 bits per word.

# Avoiding huge graphs — Radix-$k$ Encoding



$$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0 = 4453$$

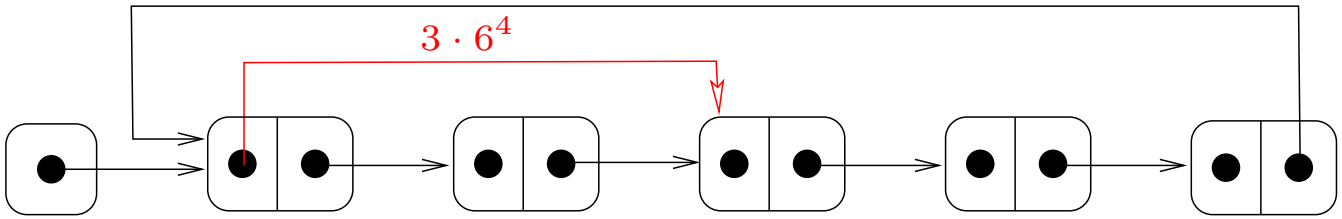
- $G$ is a circular linked list where an extra pointer field encodes a base-$k$ digit:

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

- Bit-rate is 4 bits per word.

# Avoiding huge graphs — Radix-$k$ Encoding






$$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0 = 4453$$

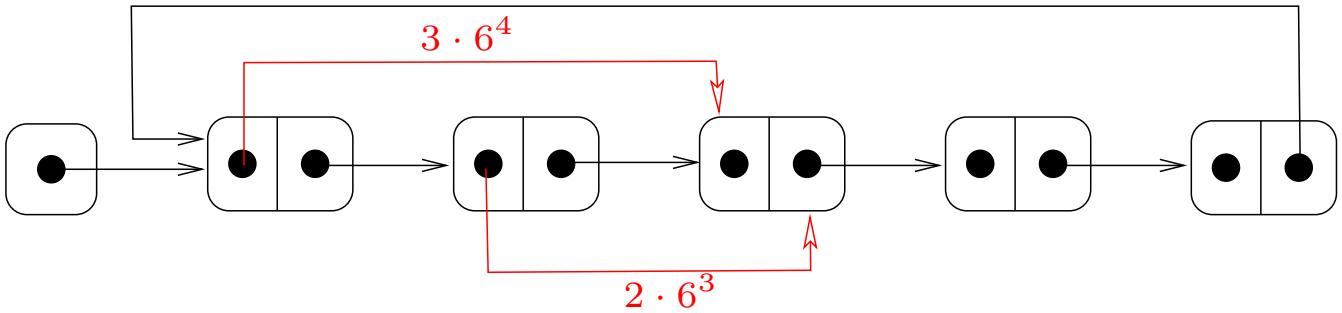
- $G$ is a circular linked list where an extra pointer field encodes a base-$k$ digit:

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

- Bit-rate is 4 bits per word.

# Avoiding huge graphs — Radix-$k$ Encoding






$$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0 = 4453$$

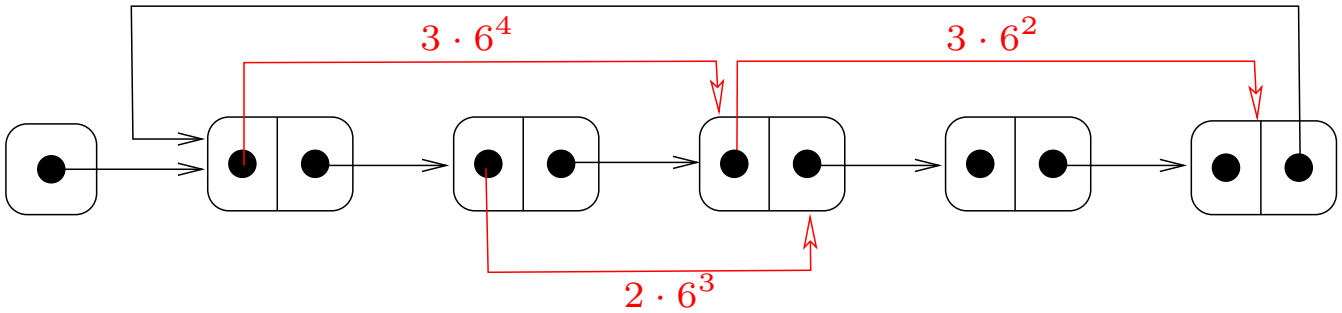
- $G$ is a circular linked list where an extra pointer field encodes a base-$k$ digit:

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

- Bit-rate is 4 bits per word.

# Avoiding huge graphs — Radix-$k$ Encoding






$$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0 = 4453$$

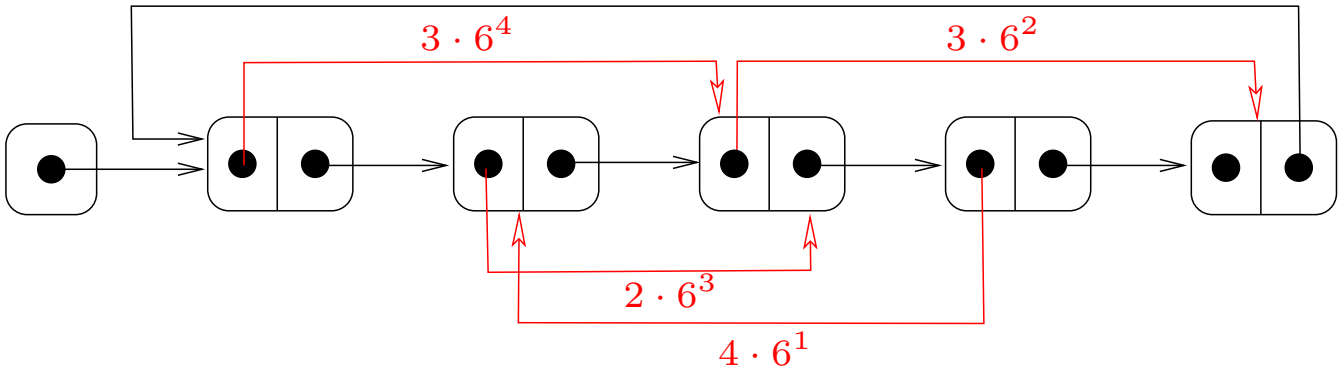
- $G$ is a circular linked list where an extra pointer field encodes a base-$k$ digit:

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

- Bit-rate is 4 bits per word.

# Avoiding huge graphs — Radix-$k$ Encoding






$$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0 = 4453$$

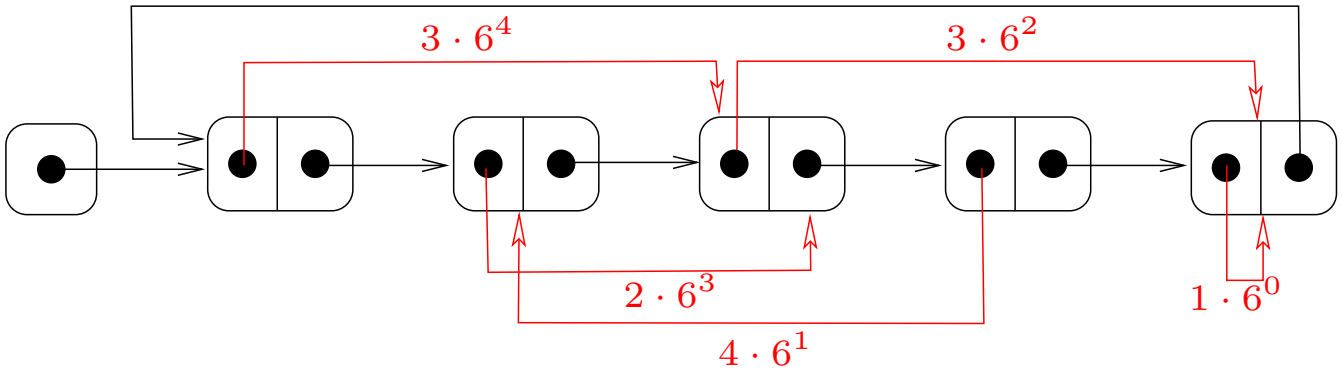
- $G$ is a circular linked list where an extra pointer field encodes a base-$k$ digit:

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

- Bit-rate is 4 bits per word.

# Avoiding huge graphs — Radix-$k$ Encoding






$$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0 = 4453$$

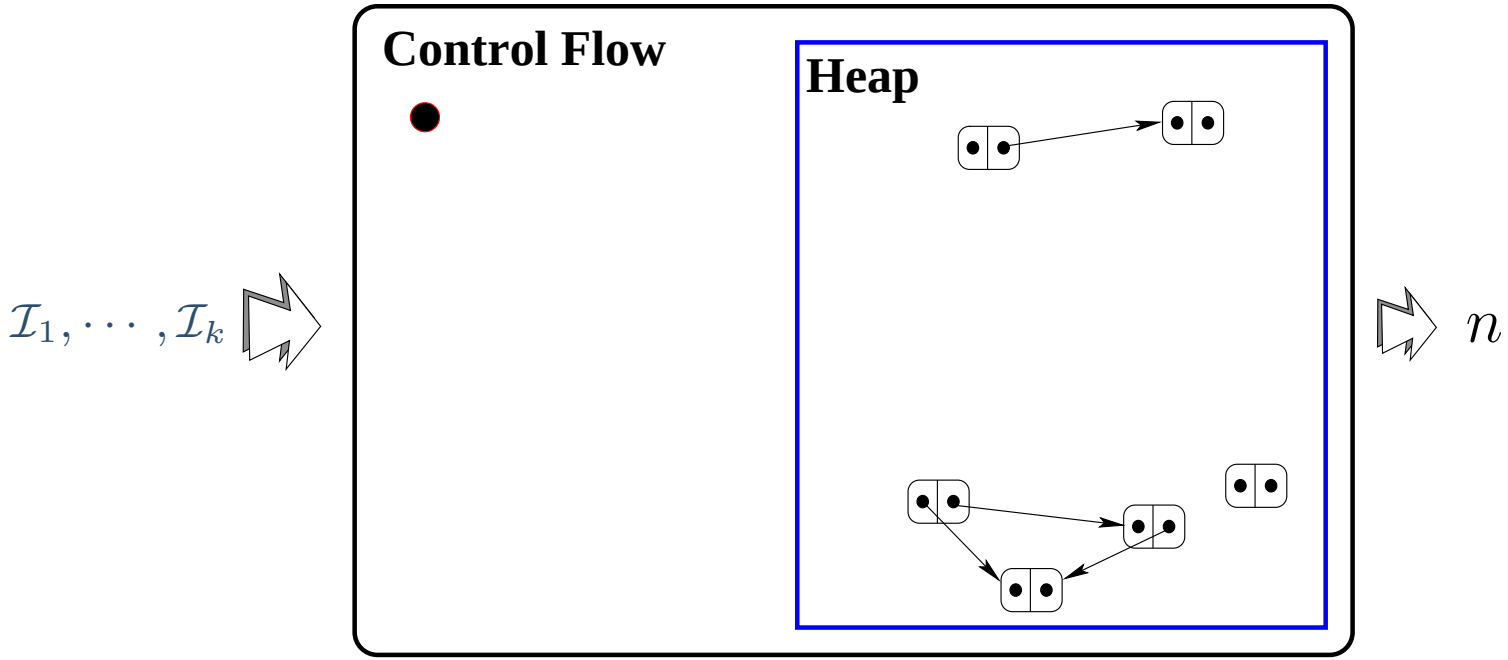
- $G$ is a circular linked list where an extra pointer field encodes a base-$k$ digit:

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

- Bit-rate is 4 bits per word.

# Avoiding huge graphs — Splitting the graph

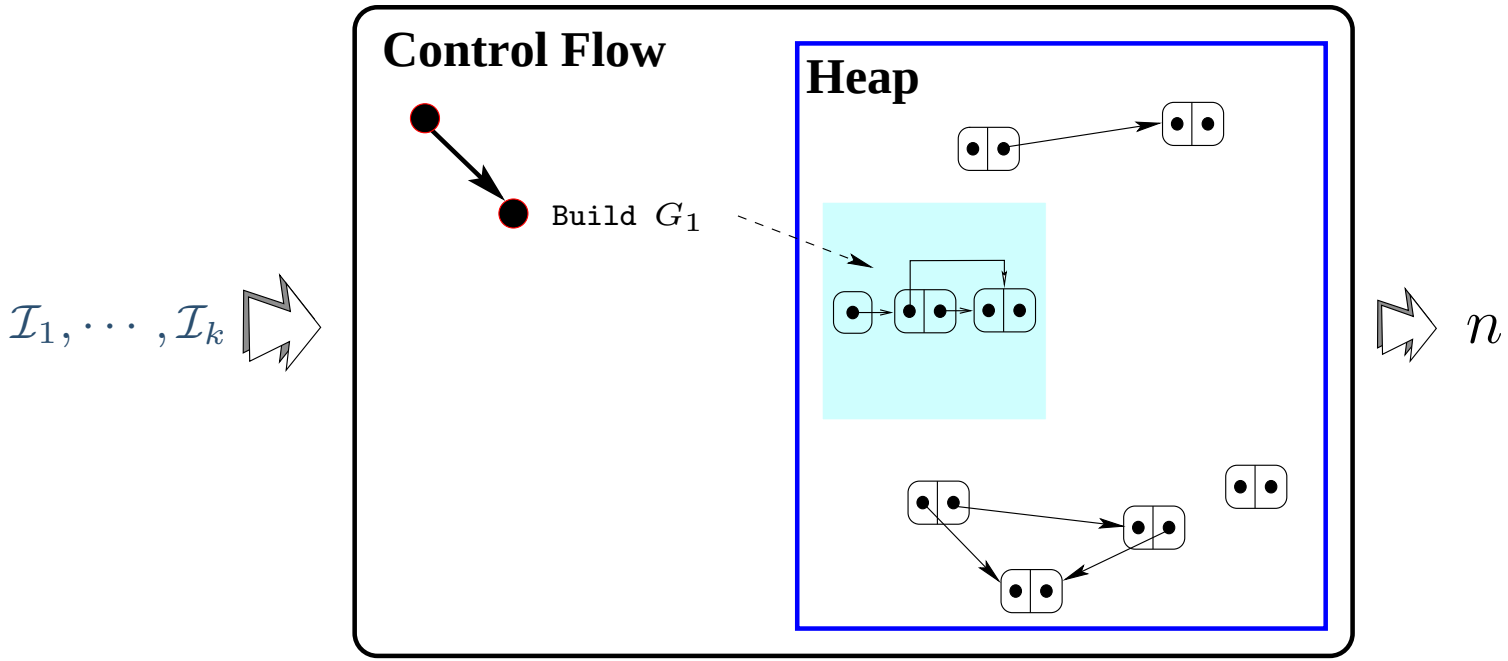





- Build the graph in pieces ⇒ increase the watermark size without decreasing stealth.

# Avoiding huge graphs — Splitting the graph

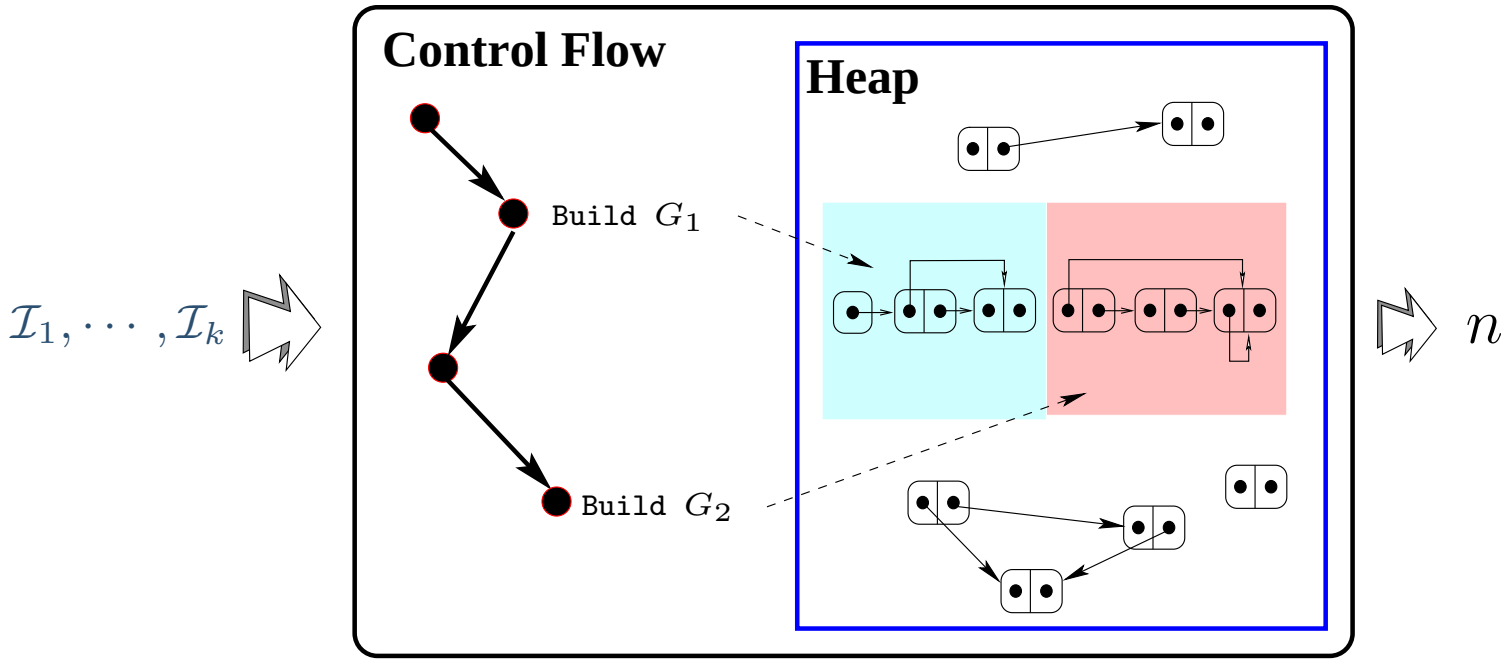





- Build the graph in pieces ⇒ increase the watermark size without decreasing stealth.

# Avoiding huge graphs — Splitting the graph

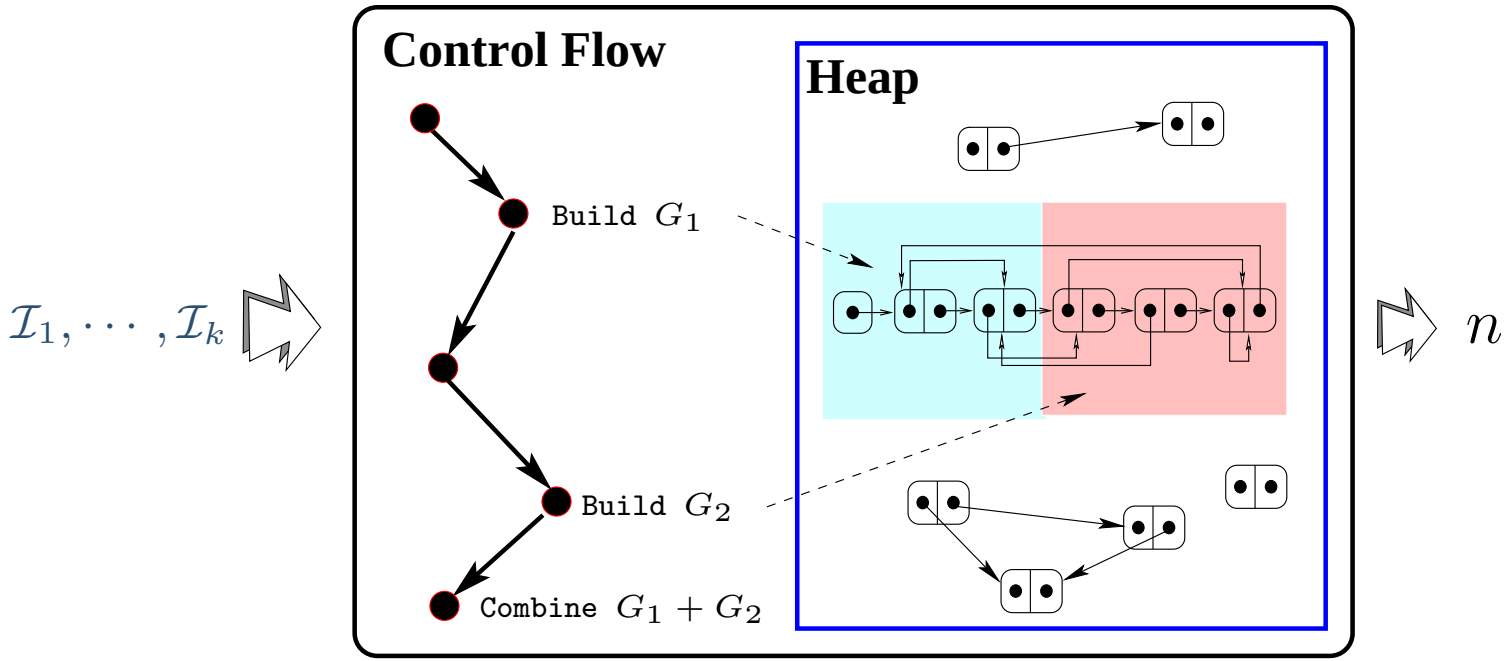





- Build the graph in pieces ⇒ increase the watermark size without decreasing stealth.

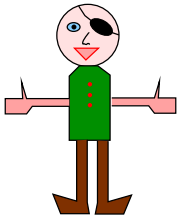
# Avoiding huge graphs — Splitting the graph

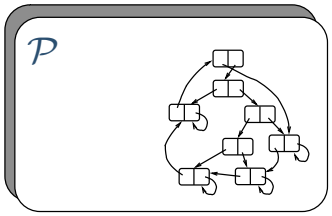




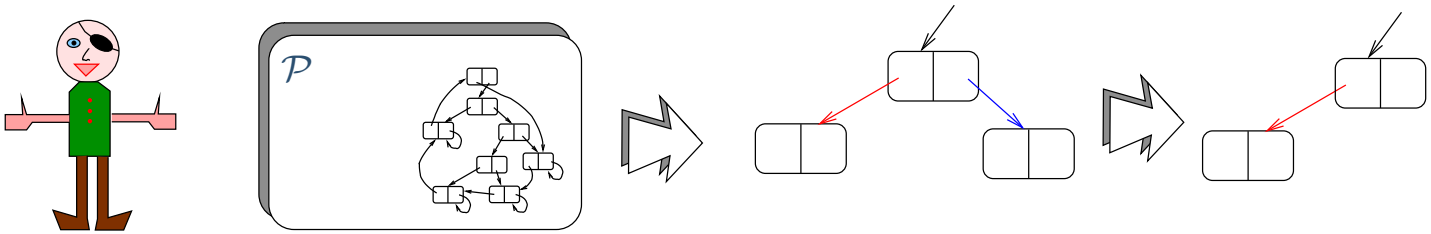

- Build the graph in pieces ⇒ increase the watermark size without decreasing stealth.
- More pieces ⇒ increase the code necessary to combine pieces!

# Avoiding attacks by small graph perturbations



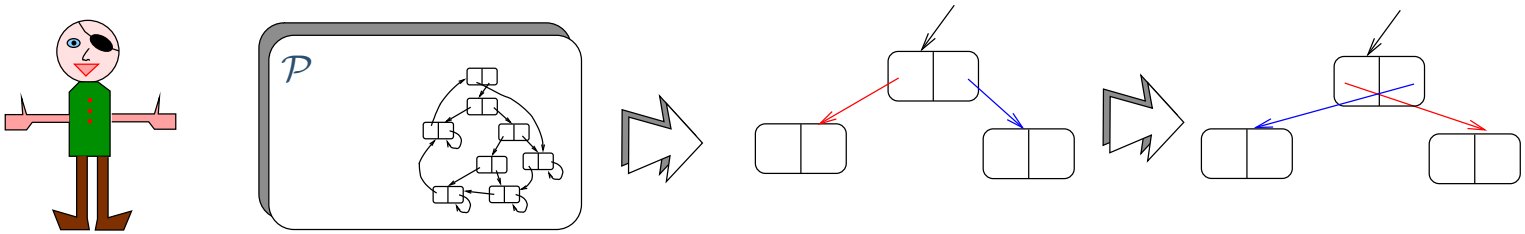


# Avoiding attacks by small graph perturbations




- Node deletion

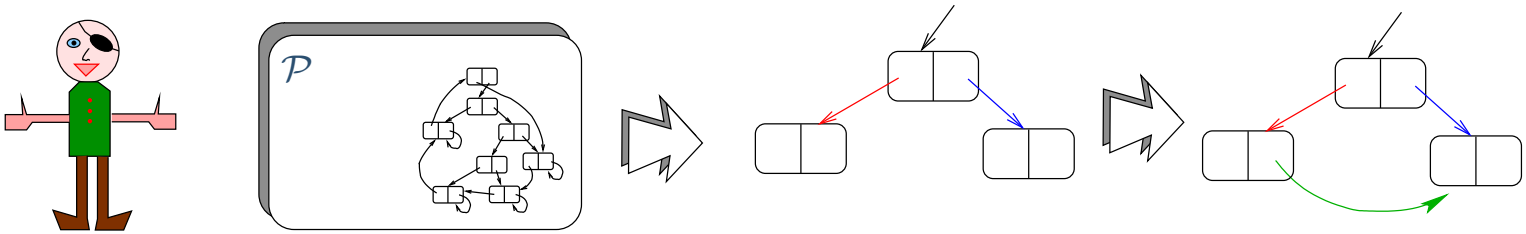
# Avoiding attacks by small graph perturbations






- Node deletion
- Edge flip

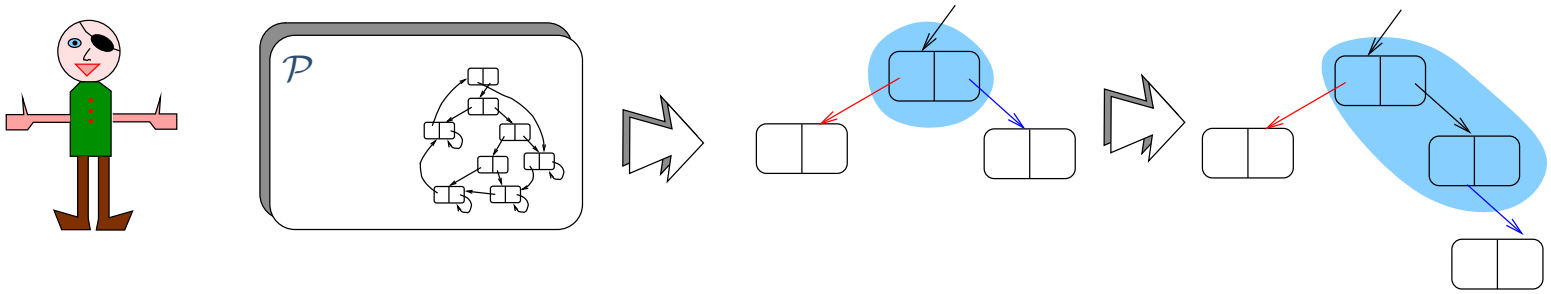
# Avoiding attacks by small graph perturbations






- Node deletion
- Edge flip
- Edge addition

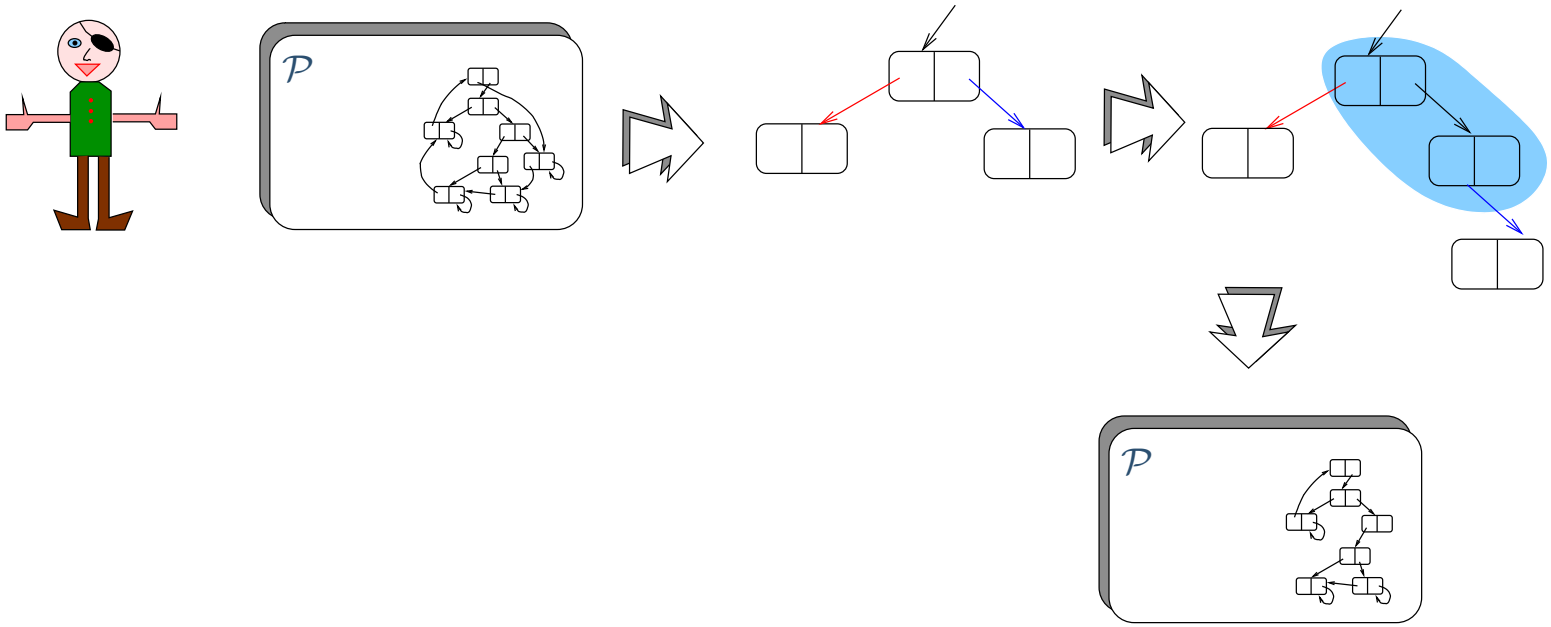
# Avoiding attacks by small graph perturbations






- Node deletion
- Edge flip
- Edge addition
- Node split

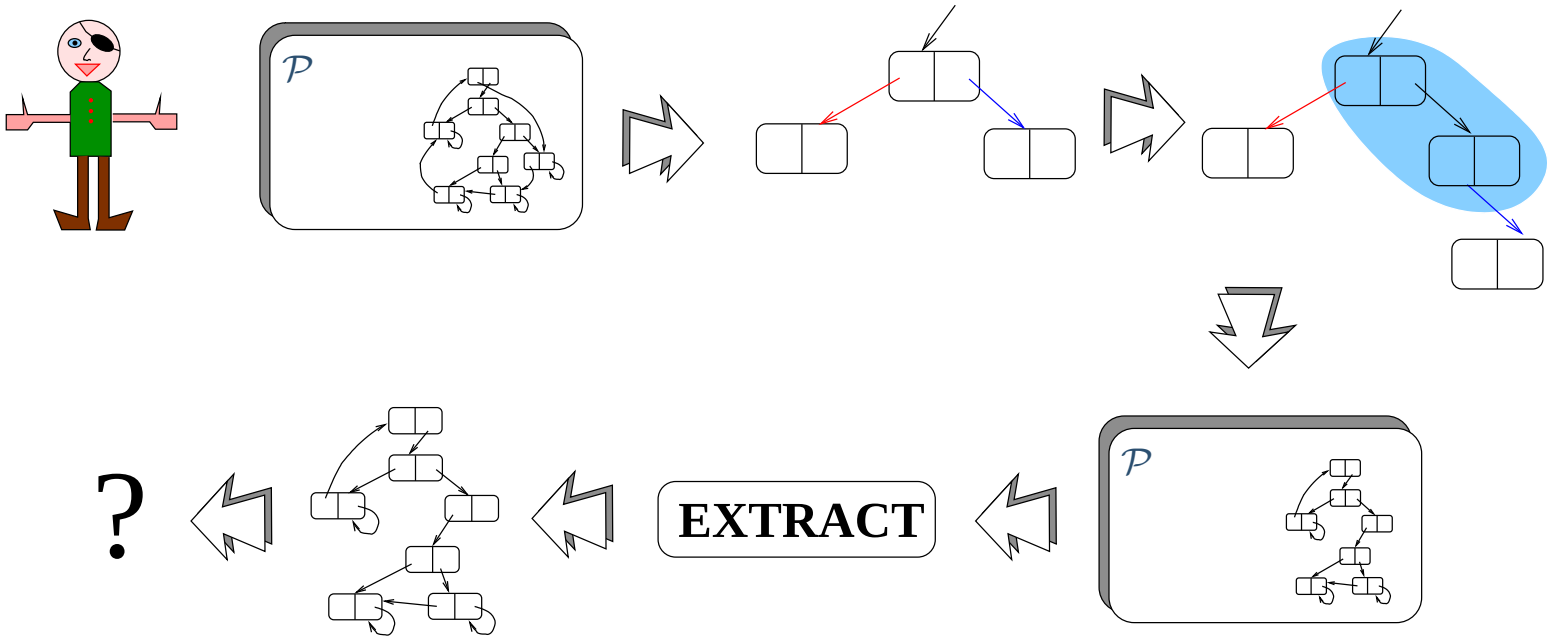
# Avoiding attacks by small graph perturbations




- Node deletion
- Edge flip
- Edge addition
- Node split

# Avoiding attacks by small graph perturbations

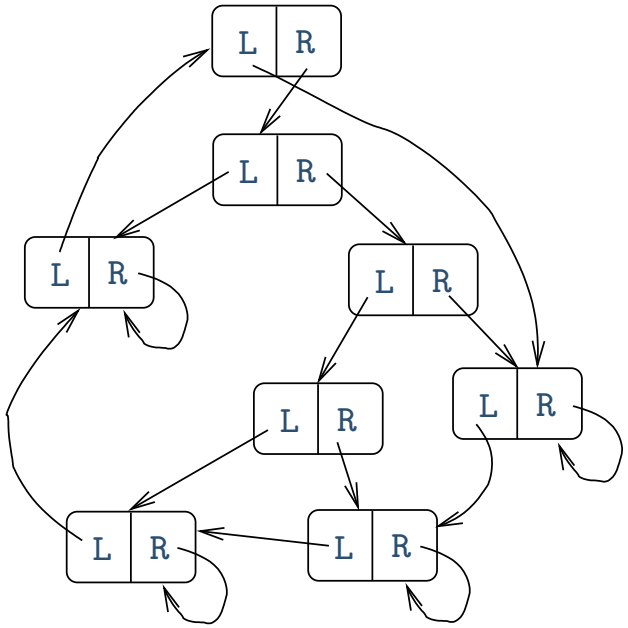





- Node deletion
- Edge flip
- Edge addition
- Node split

# Error-Correcting Graphs — PPCT

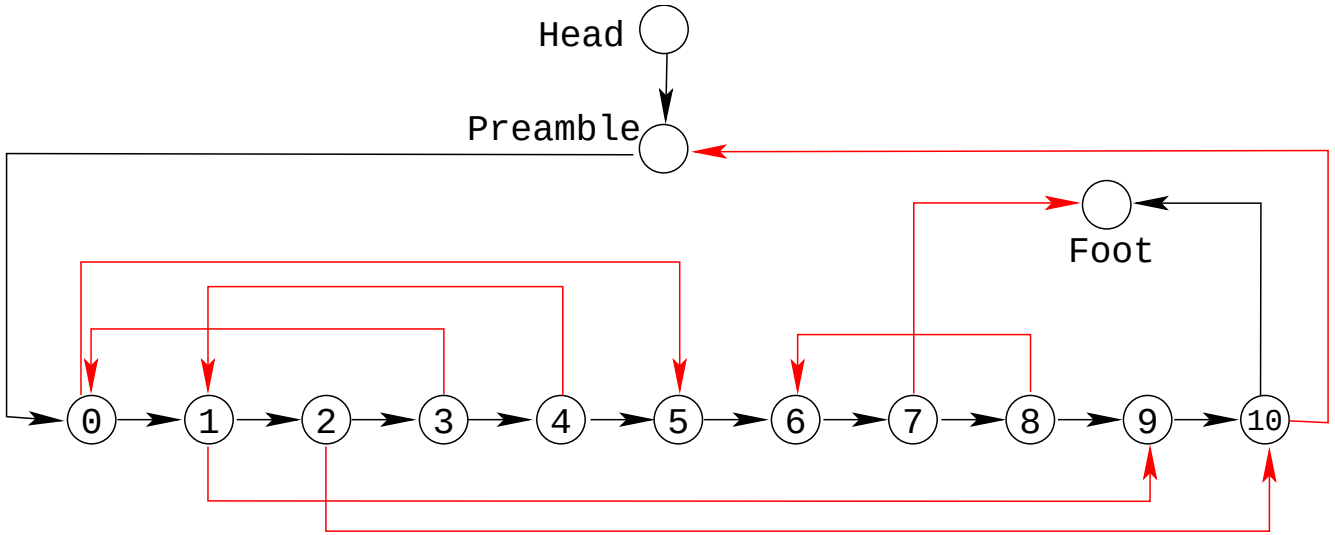





- Planted Plane Cubic Trees can detect and correct one occurrence of node deletion or insertion.

# Error-Correcting Graphs — RPG

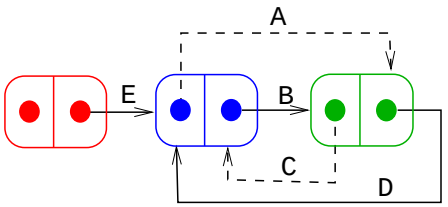





- Reducible Permutation Graphs can be decoded and corrected for edge-flips in polynomial time.

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

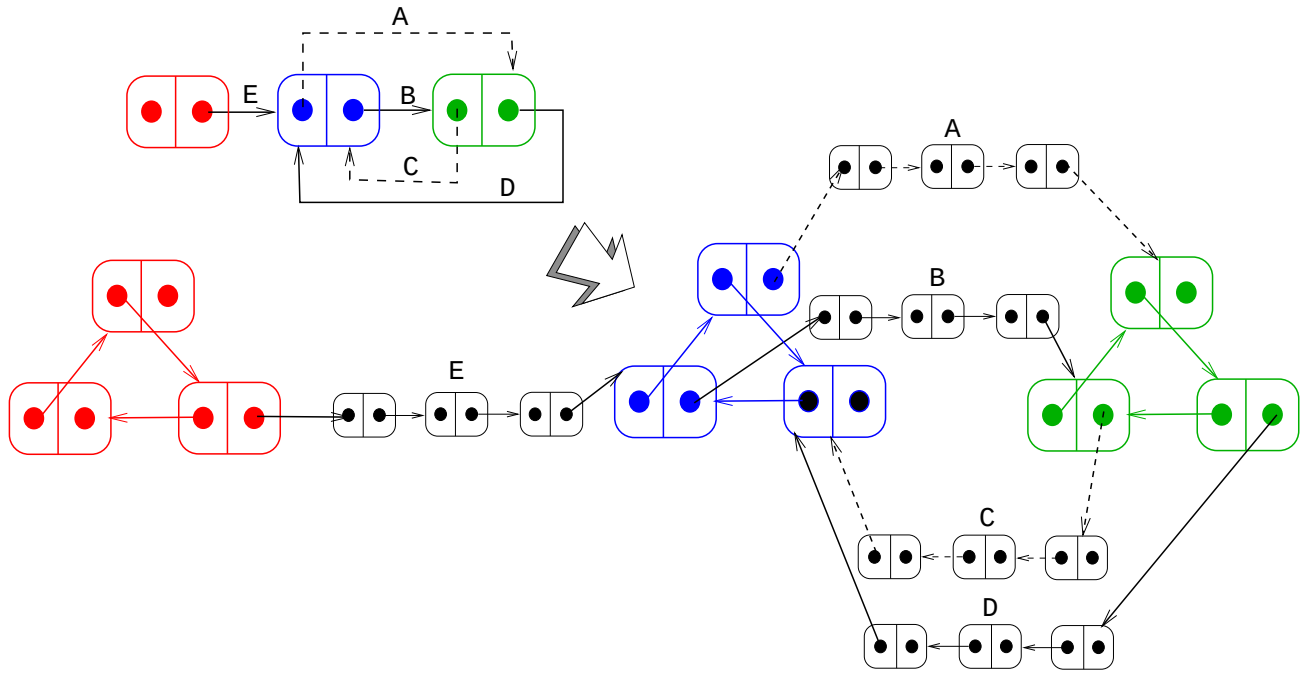
# Error-Correcting Graphs — Cycled Graphs




- Turn every node into a 3-cycle and every edge into a path of length 3 during embedding.
- During extraction the cycles and paths are contracted back to the original graph.

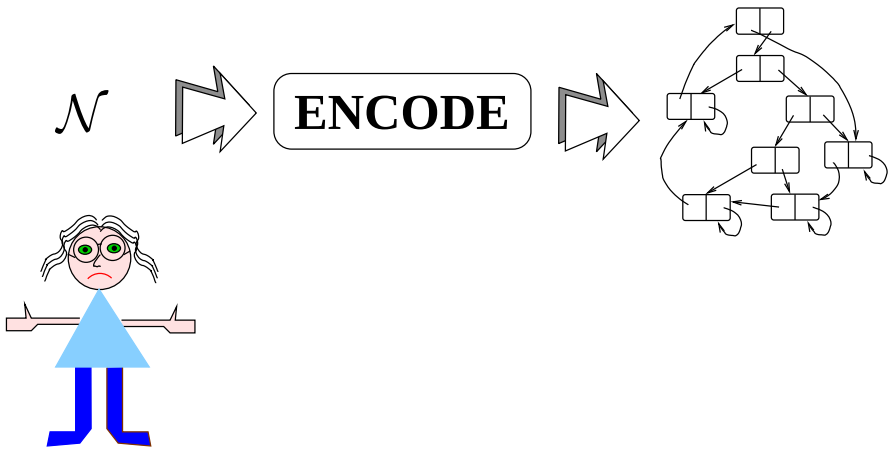
# Error-Correcting Graphs — Cycled Graphs






- Turn every node into a 3-cycle and every edge into a path of length 3 during embedding.
- During extraction the cycles and paths are contracted back to the original graph.

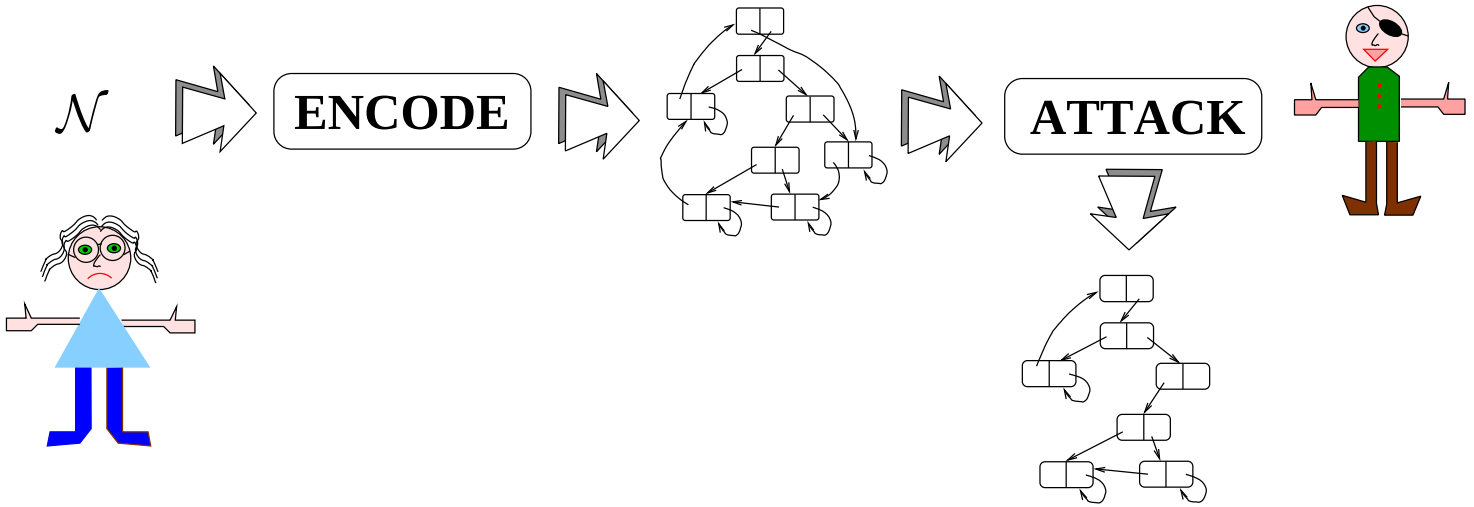
# Research problem: Graph Encoding Properties






- **ENCODE** should produce small, directed multigraphs with an ordering on the outgoing edges and low max out-degree.

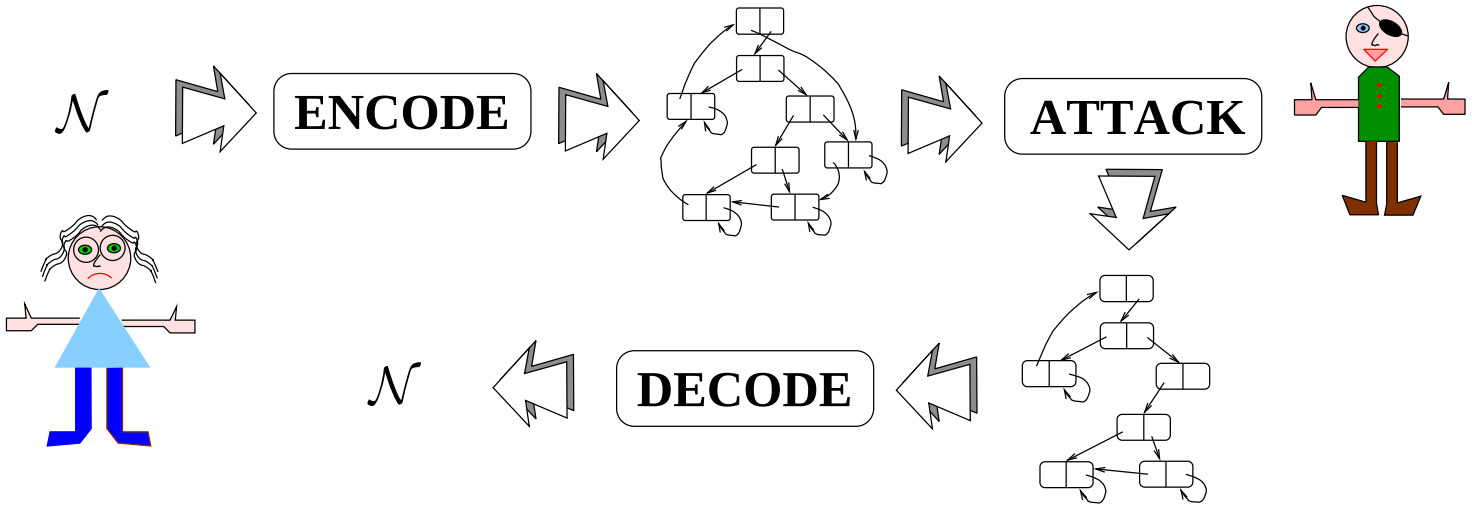
# Research problem: Graph Encoding Properties




- **ENCODE** should produce small, directed multigraphs with an ordering on the outgoing edges and low max out-degree.

# Research problem: Graph Encoding Properties






- **ENCODE** should produce small, directed multigraphs with an ordering on the outgoing edges and low max out-degree.
- **DECODE** should be insensitive to small perturbations and be polynomial time efficient.

# Avoiding Bogus Field Addition

- Assume that we have a graph node `Node`:

```
class Node {
    public int a;
    public Node left, right;
}
```

# Avoiding Bogus Field Addition

- Assume that we have a graph node `Node`:

```
class Node {
    public int a;
    public Node left, right;
}
```

- Reflection lets us check the integrity of this type at runtime:

```
Field[] F = Node.class.getFields();
if (F.length != 3) die();
if (F[1].getType() != Node.class) die();
```

# Avoiding Bogus Field Addition

- Assume that we have a graph node `Node`:

```
class Node {
    public int a;
    public Node left, right;
}
```

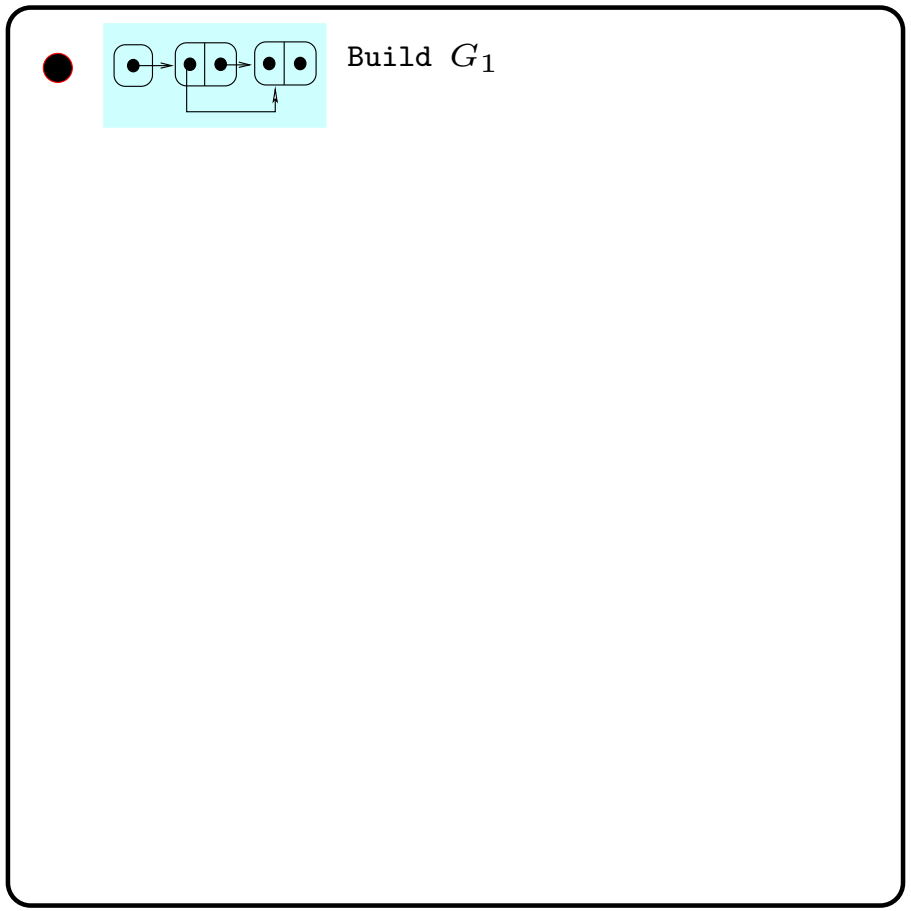
- Reflection lets us check the integrity of this type at runtime:

```
Field[] F = Node.class.getFields();
if (F.length != 3) die();
if (F[1].getType() != Node.class) die();
```

- Unfortunately, this type of code is unstealthy.

# Avoiding Alias Analysis

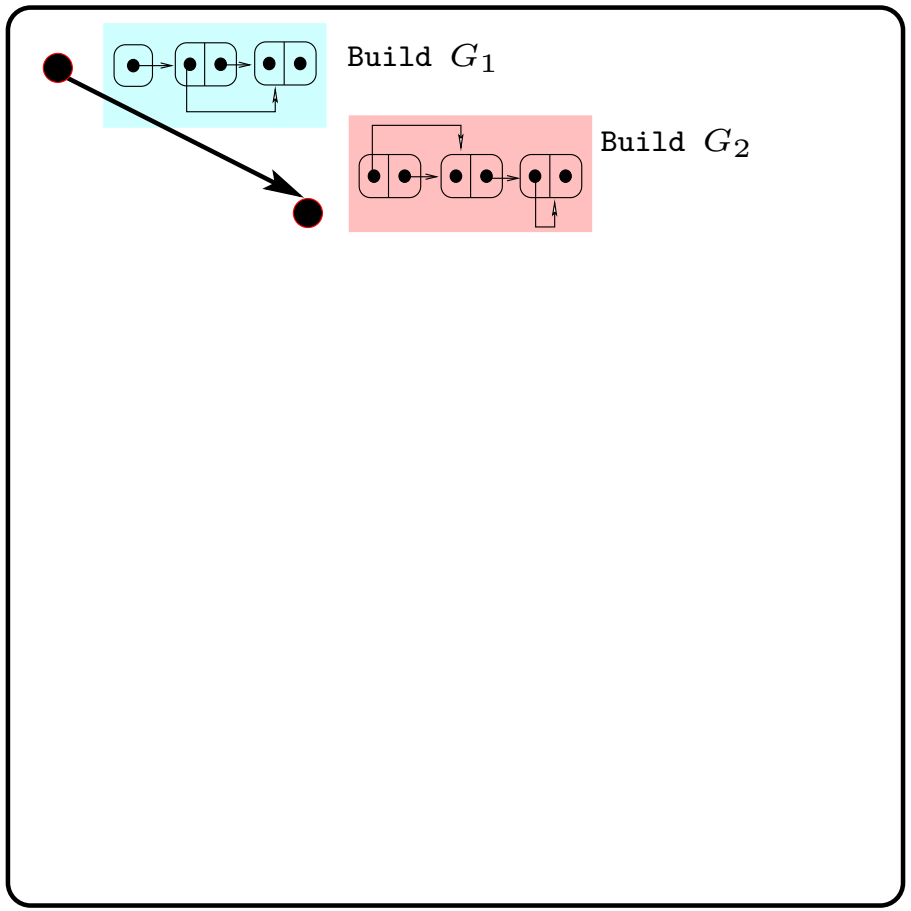





- Why? Static analysis algorithms fail on dynamically changing structures.

# Avoiding Alias Analysis

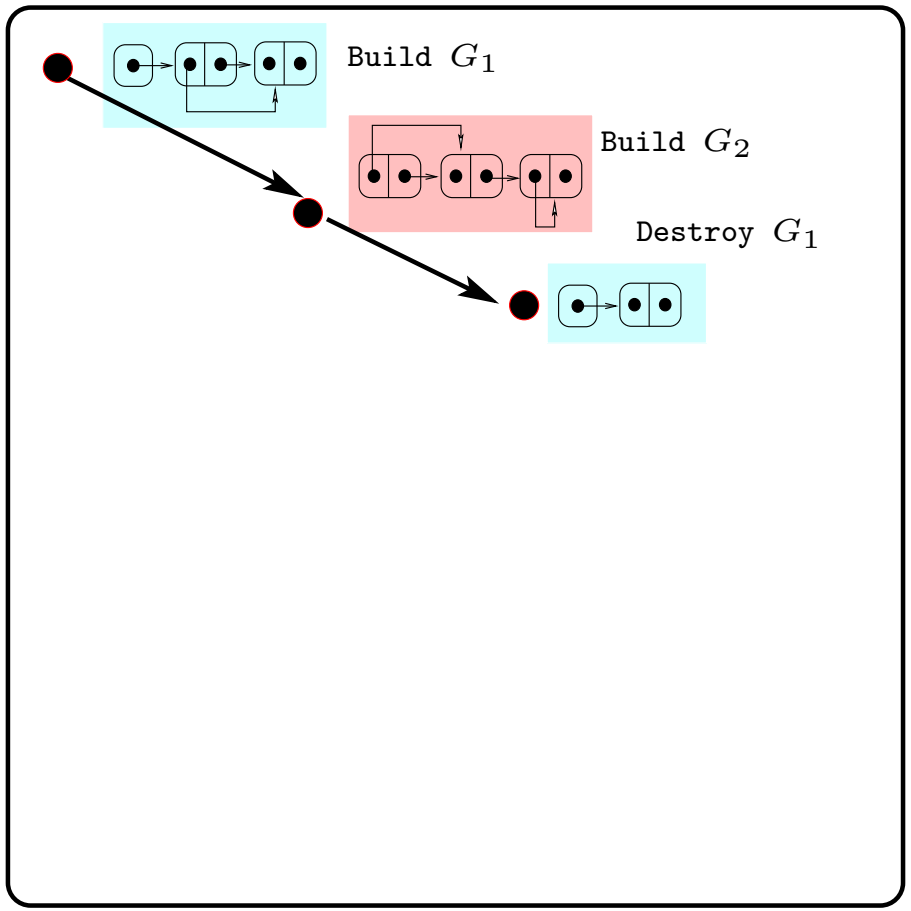





- Why? Static analysis algorithms fail on dynamically changing structures.

# Avoiding Alias Analysis

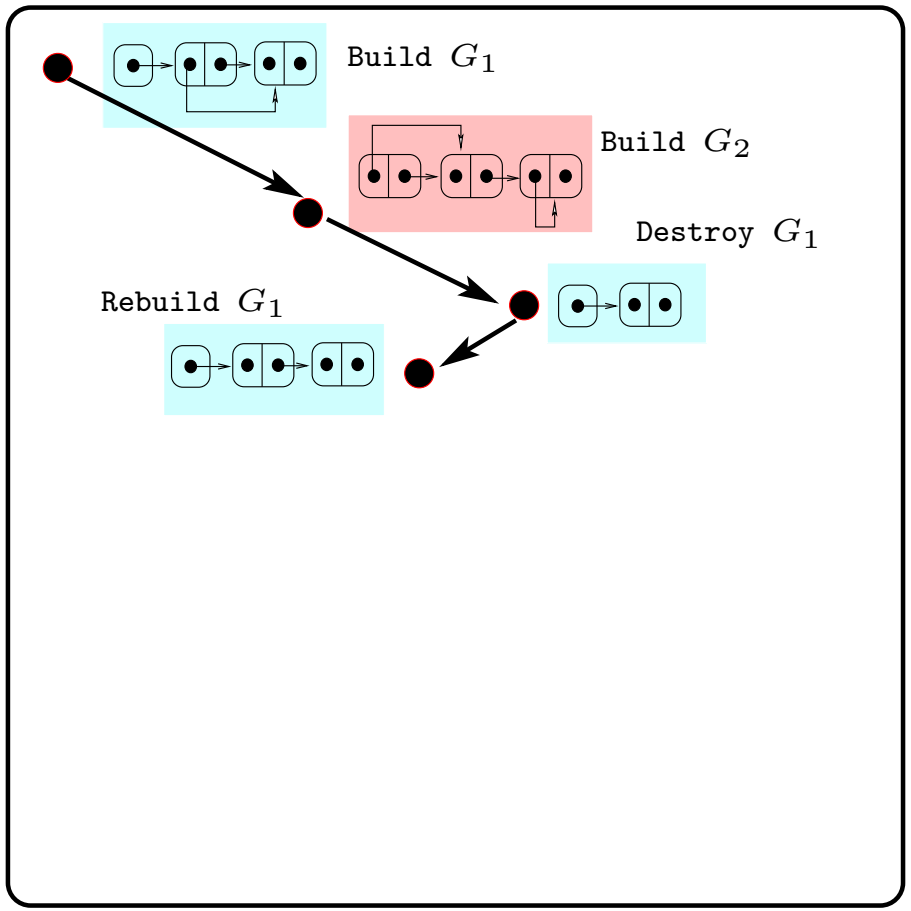





- Why? Static analysis algorithms fail on dynamically changing structures.

# Avoiding Alias Analysis

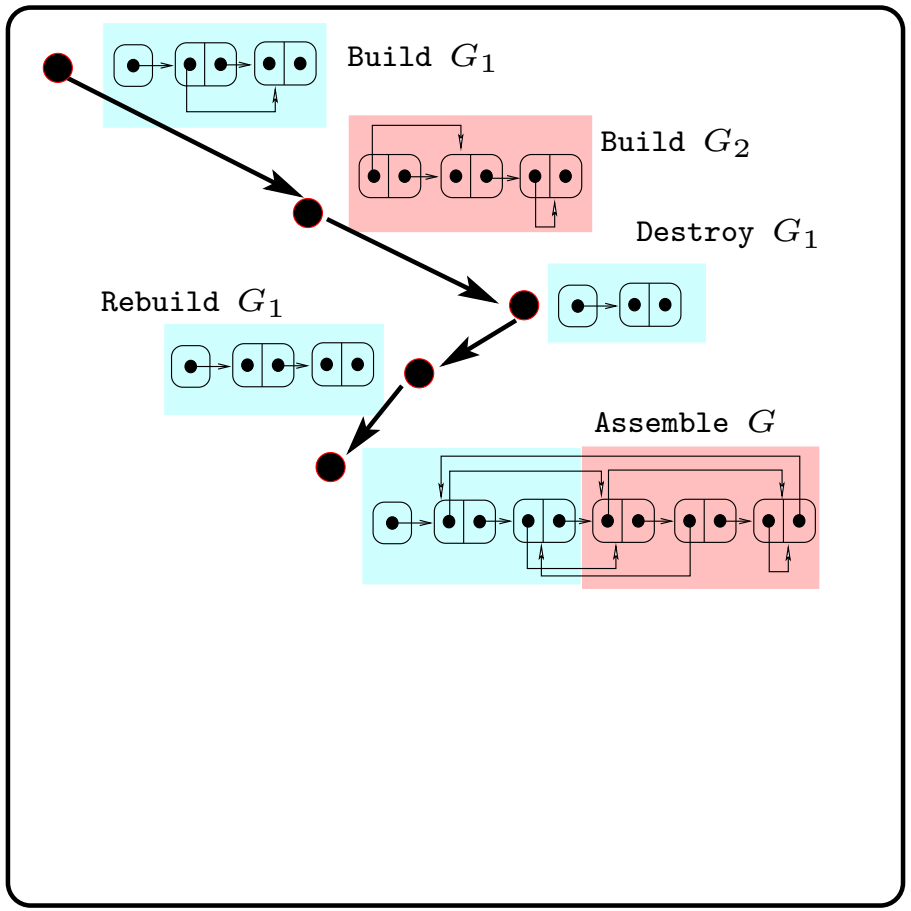





- Why? Static analysis algorithms fail on dynamically changing structures.

# Avoiding Alias Analysis

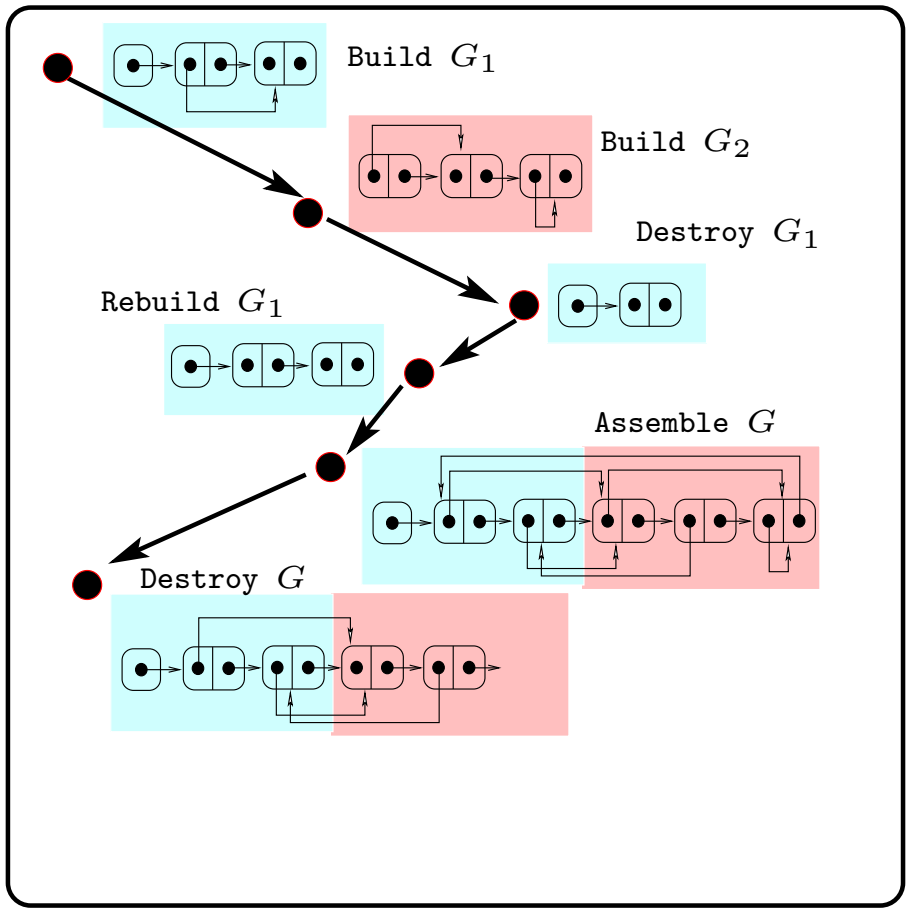





- Why? Static analysis algorithms fail on dynamically changing structures.

# Avoiding Alias Analysis

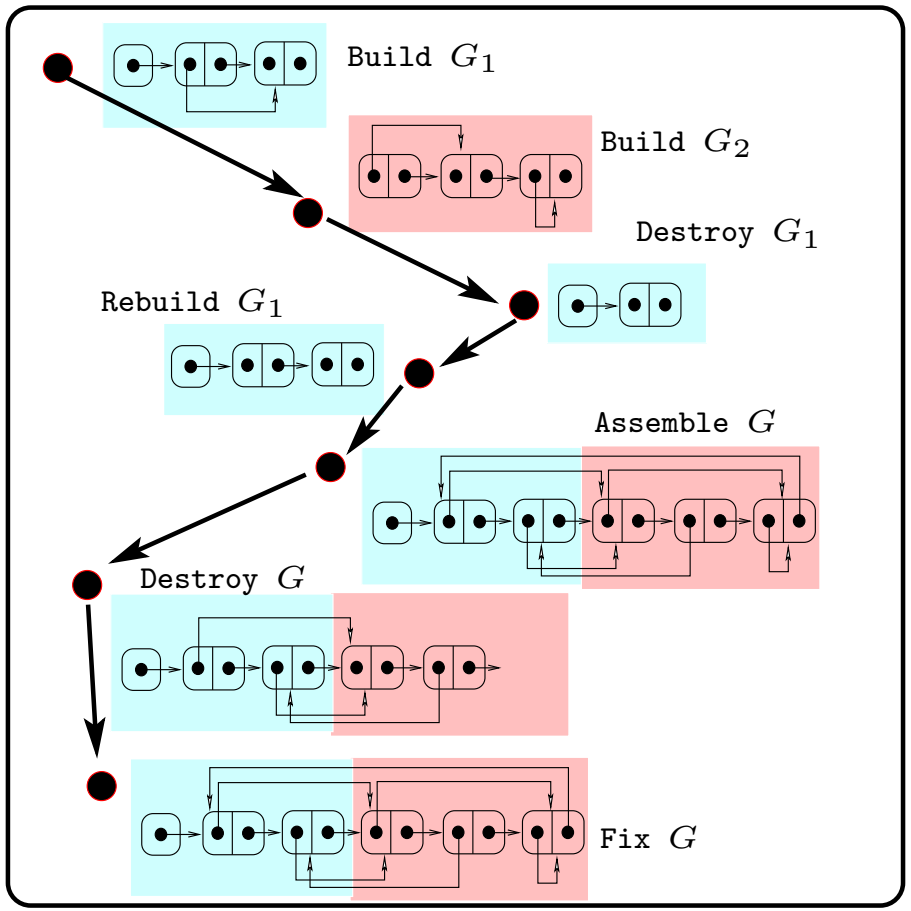





- Why? Static analysis algorithms fail on dynamically changing structures.

# Avoiding Alias Analysis

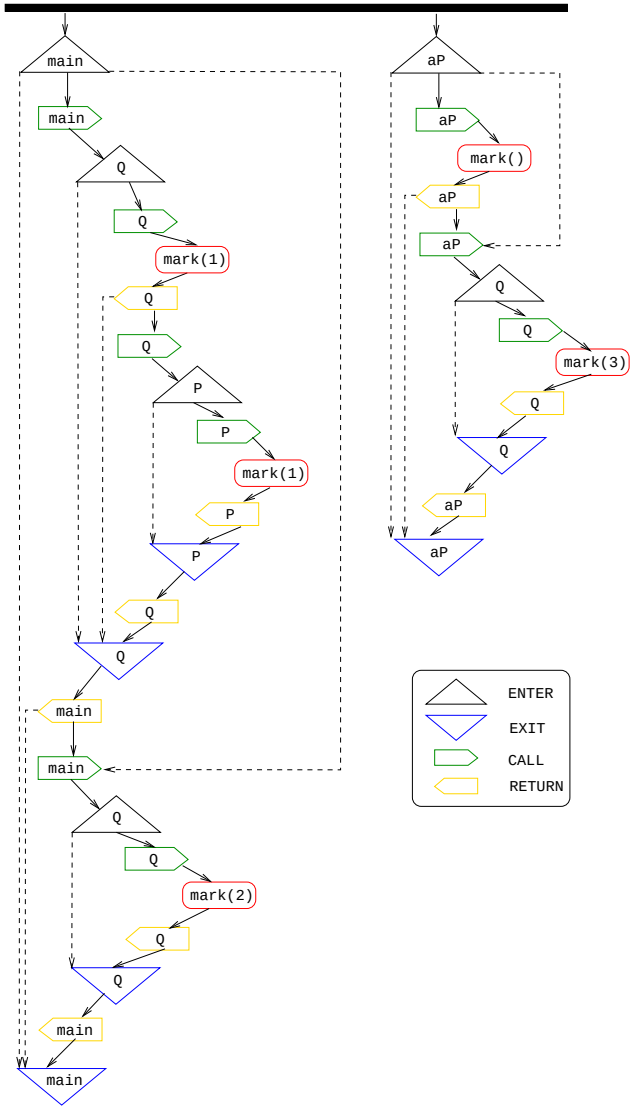





- Why? Static analysis algorithms fail on dynamically changing structures.

# Avoiding Global Variables






```
public class Main {
    public static Node root;

    public static void main(String args[]){
        Node n2 = new Node();
        Node n1 = new Node();
        n1.left = n2;  n1.right = n2;
        root = n1;
    }
}
```

- We shouldn't store secrets in globals, since there are few of them!
- Analyze the flow to pass secrets along in bogus method arguments.

# Avoiding Unstealthy Node Classes

- Don't create unstealthy extra watermark classes:

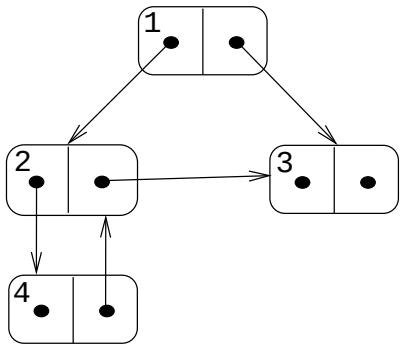
```
class Node{
    public Node left,right;
}
```

# Avoiding Unstealthy Node Classes

- Don't create unstealthy extra watermark classes:

```
class Node{
    public Node left,right;
}
```

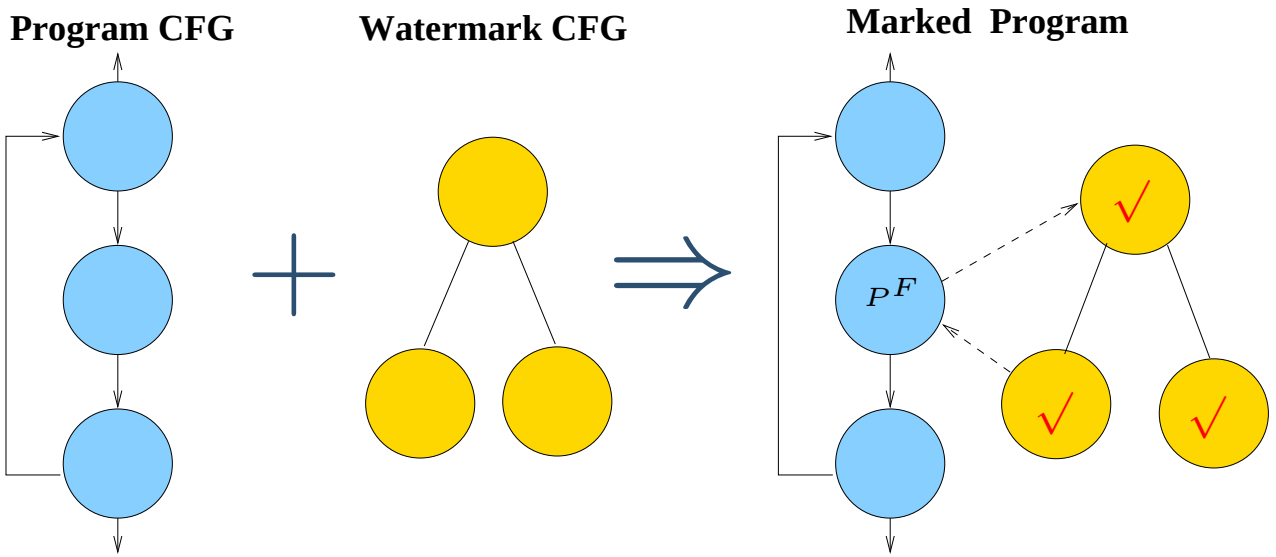
- Instead reuse user or standard library classes:




```
LinkedList n4 = new LinkedList();
n4.add(null);
LinkedList n2 = new LinkedList();
n2.add(n4);
n4.add(n2);
Event n3 = new Event(null,0,null);
n2.add(n3);
Object[] n1 = {n2,n3};
```

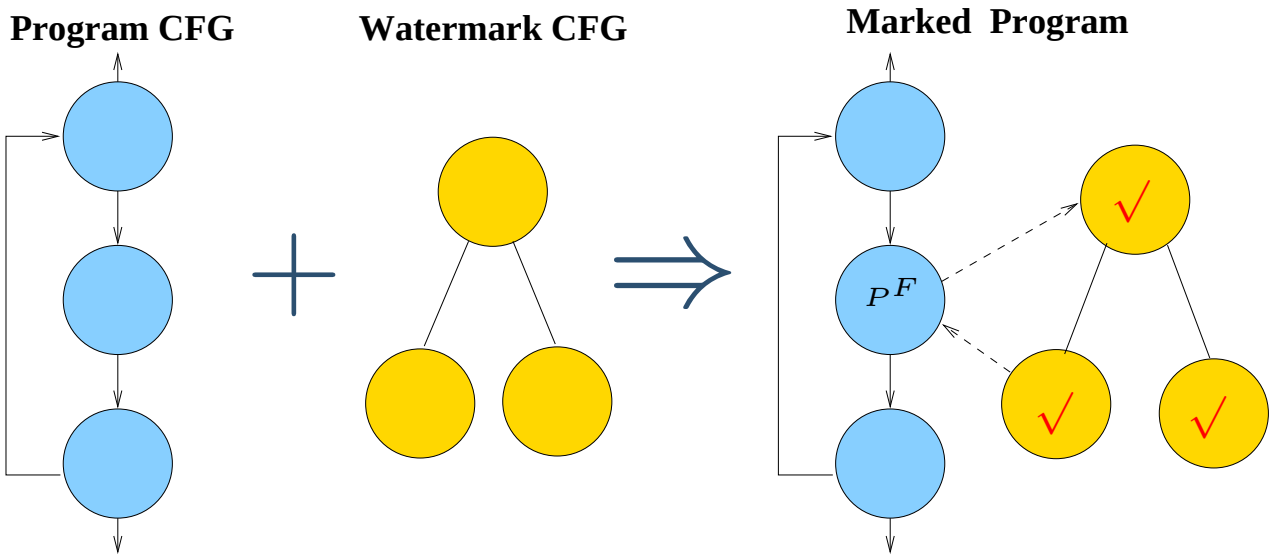
# Avoiding Weak Cuts

- We can avoid weak cuts by adding many bogus jumps across the code, a la Venkatesan:

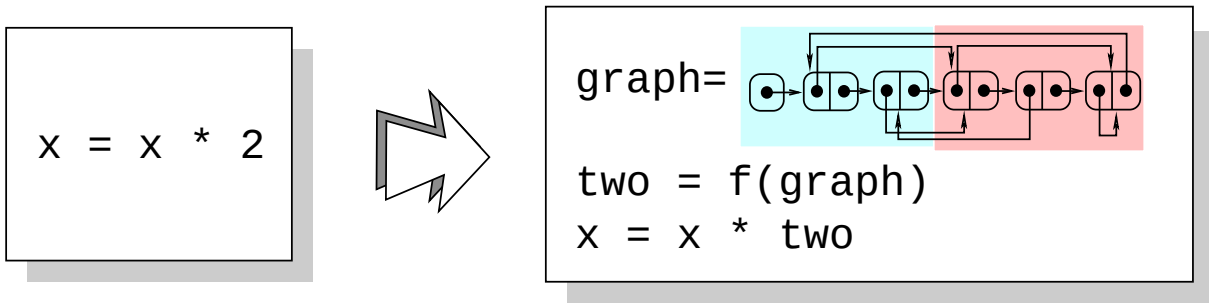


# Avoiding Weak Cuts

- We can avoid weak cuts by adding many bogus jumps across the code, a la Venkatesan:

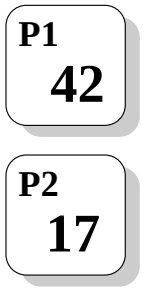



- Compute a value from the graph and use it in the code:



# Avoiding static collusive attacks

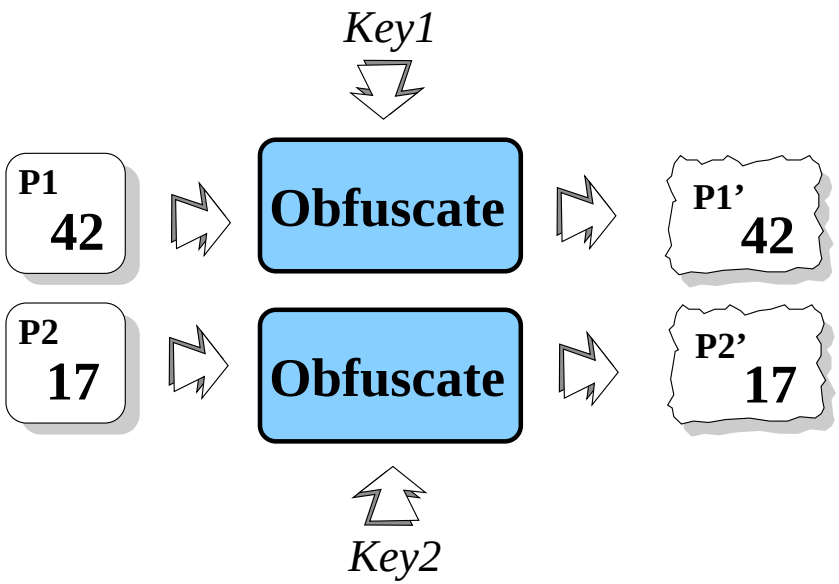





- Obfuscation can be used to protect against static collusive attacks.

# Avoiding static collusive attacks

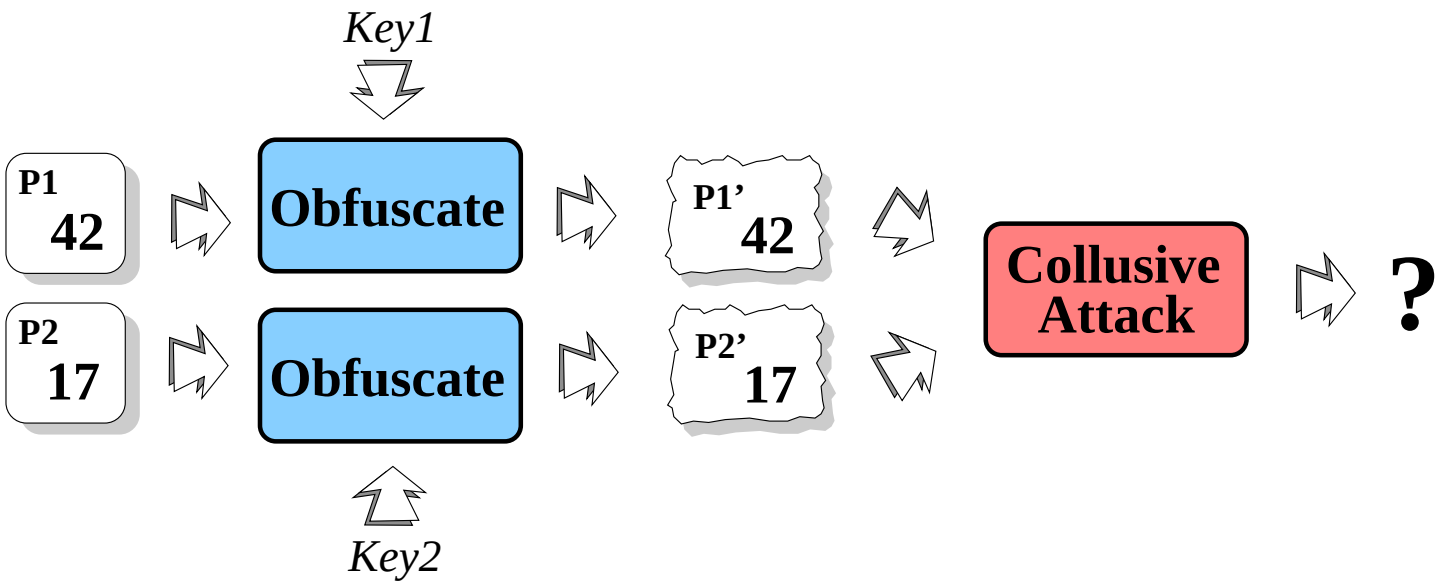





- Obfuscation can be used to protect against static collusive attacks.

# Avoiding static collusive attacks

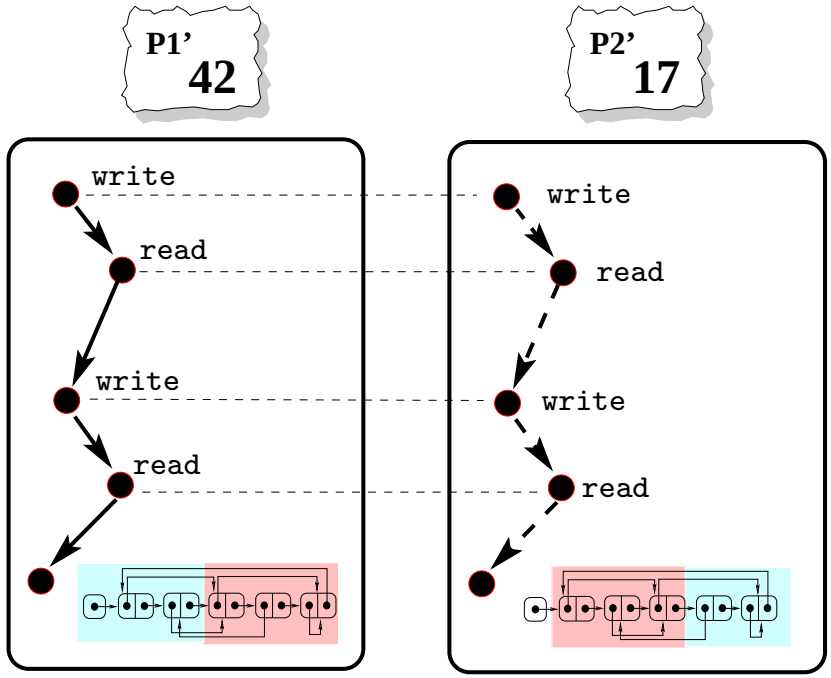





- Obfuscation can be used to protect against static collusive attacks.

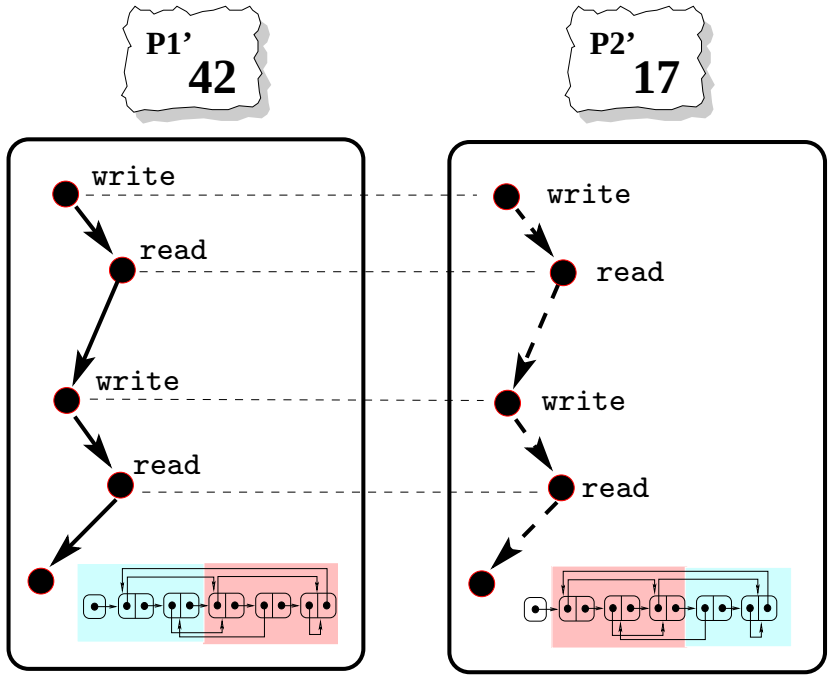
# Avoiding Dynamic Collusive Attacks

- The adversary can run two differently obfuscated versions of two differently fingerprinted programs, lining up code points at system calls:



# Avoiding Dynamic Collusive Attacks

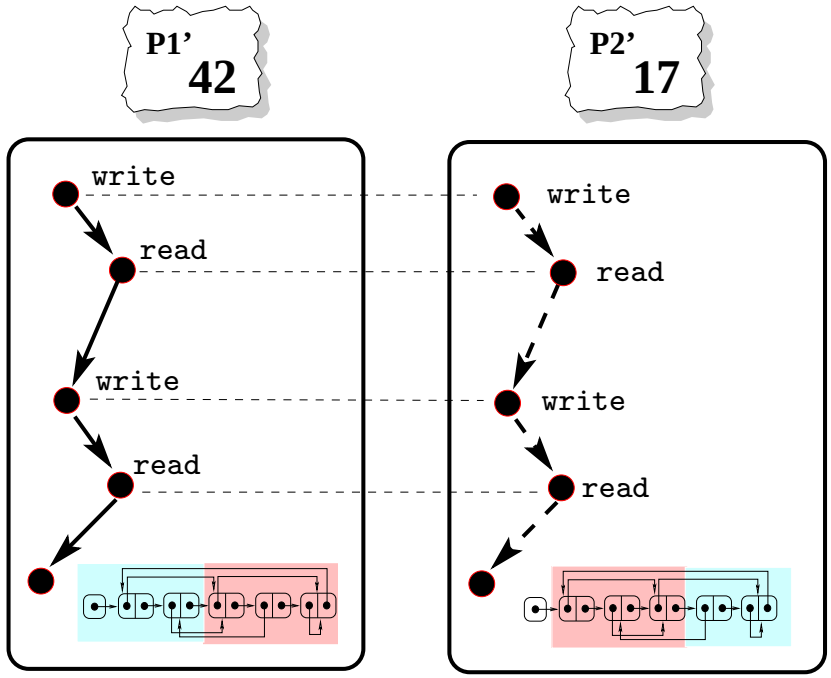
- The adversary can run two differently obfuscated versions of two differently fingerprinted programs, lining up code points at system calls:




- Now he only needs to analyze the code between system calls - much less work!

# Avoiding Dynamic Collusive Attacks

- The adversary can run two differently obfuscated versions of two differently fingerprinted programs, lining up code points at system calls:




- Now he only needs to analyze the code between system calls - much less work!
- Hard to insert extra system calls to confuse him!

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)
6. introducing extra unstealthy classes? (`user/lib classes`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)
6. introducing extra unstealthy classes? (`user/lib classes`)
7. weak cuts? (`compute values from graph`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)
6. introducing extra unstealthy classes? (`user/lib classes`)
7. weak cuts? (`compute values from graph`)
8. static collusive attacks? (`obfuscation`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)
6. introducing extra unstealthy classes? (`user/lib classes`)
7. weak cuts? (`compute values from graph`)
8. static collusive attacks? (`obfuscation`)
9. dynamic collusive attacks? (`uhm?`)

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)
6. introducing extra unstealthy classes? (`user/lib classes`)
7. weak cuts? (`compute values from graph`)
8. static collusive attacks? (`obfuscation`)
9. dynamic collusive attacks? (`uhm?`)

- Attention to detail is important!

# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)
6. introducing extra unstealthy classes? (`user/lib classes`)
7. weak cuts? (`compute values from graph`)
8. static collusive attacks? (`obfuscation`)
9. dynamic collusive attacks? (`uhm?`)

- Attention to detail is important!
- Stealth is hard!

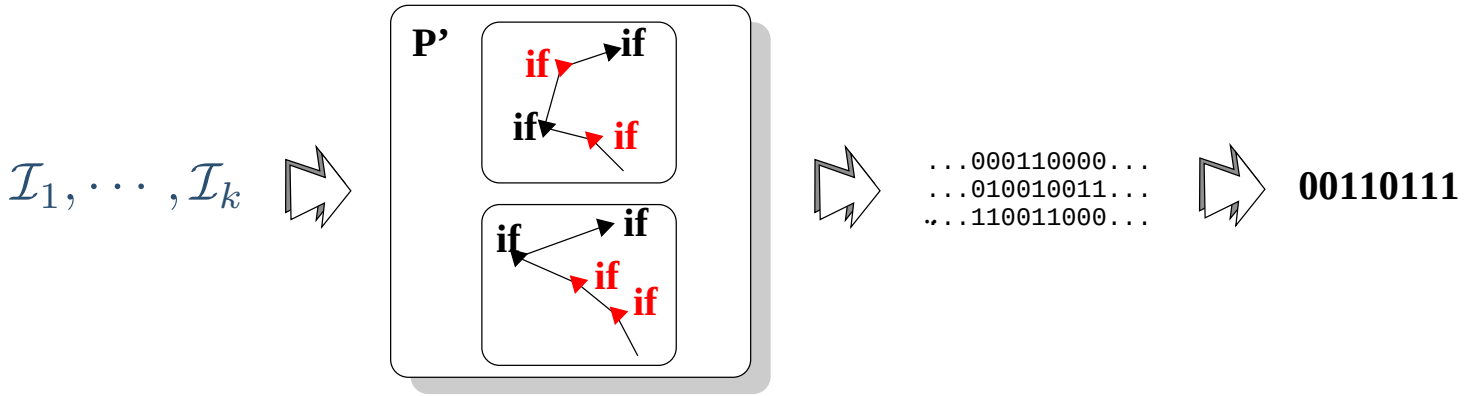
# Problems Revisited

How do we avoid

1. huge graphs? (`new graph classes`)
2. attacks by small graph perturbations? (`ECGs`)
3. bogus field addition? (`reflection`)
4. attacks by alias analysis? (`destructive updates`)
5. global variables? (`method params`)
6. introducing extra unstealthy classes? (`user/lib classes`)
7. weak cuts? (`compute values from graph`)
8. static collusive attacks? (`obfuscation`)
9. dynamic collusive attacks? (`uhm?`)

- Attention to detail is important!
- Stealth is hard!
- Dynamic attacks are harder!


# — Path-Based Watermarking



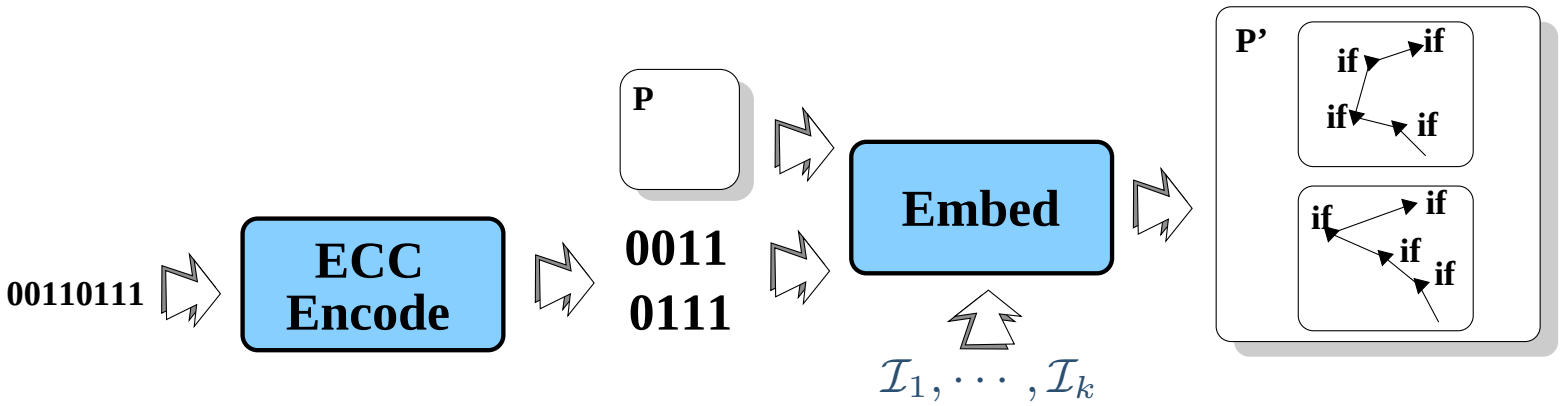

- The branches executed for the secret input generate a stream of 0s and 1s from which the watermark is extracted.
- An attacker can easily insert new branches:

  **Java** $\Rightarrow$ Use an Error Correcting Code

  **x86** $\Rightarrow$ 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



```
void main() {
   int a=25,b=10;
   while((a%b)!=0){
      int t=b%a;
      b = a;
      a = t;
   }
   println(b);
}
```

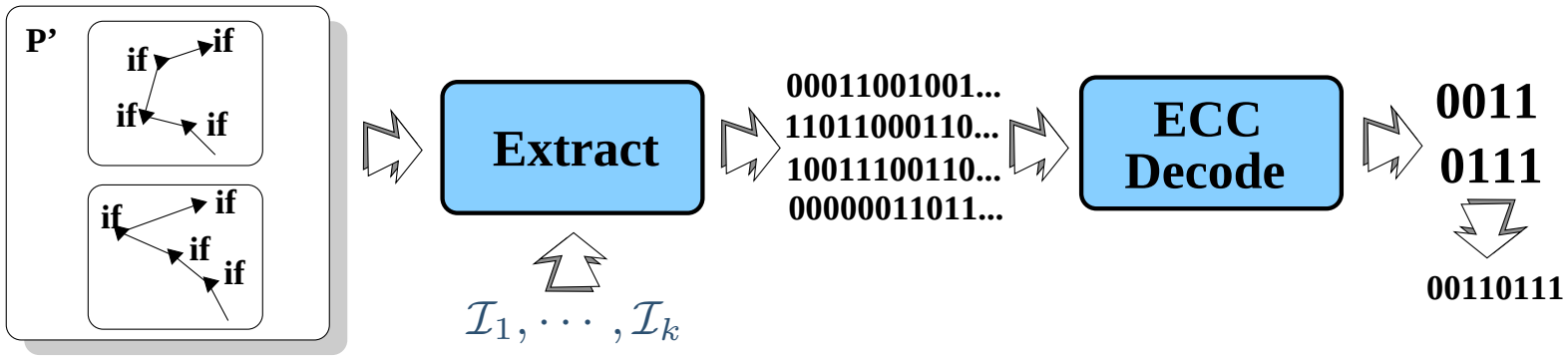
00110111

Embed

```
void main()  {
   int a=25,b=10;
   int u=0,x=0x1a,c=8;
   while((a%b)!=0) {
       int t = b % a;
       b = a; a = t;
       if (t==a) u++;
       if (t==a) u++;
       if (a==10) u++;
       if (a==10) u++;
   }
   for(int i=0; i<c;i++,x>>=1)
      if ((x&1)==1) x|=1;
   println(b);
}
```

- 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

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

$$W = 17 \Rightarrow \begin{array}{rcl} W & \equiv & 5 \bmod p_1 p_2 \\ W & \equiv & 7 \bmod p_1 p_3 \\ W & \equiv & 2 \bmod p_2 p_3 \end{array}$$

- Choose $p_1, \ldots, p_k$ pairwise relatively prime, split watermark into $\frac{k(k-1)}{2}$ pieces of the form $W \equiv r \bmod p_i p_k$.

# PBW — ECC Encode

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

$$
\begin{array}{rrcl@{\quad}rcl}
 & W & \equiv & 5 \bmod p_1p_2 & 5 & = & 5 \\
W = 17 \Rightarrow & W & \equiv & 7 \bmod p_1p_3 \Rightarrow & p_1p_2 + 7 & = & 13 \\
 & W & \equiv & 2 \bmod p_2p_3 & p_1p_2 + p_1p_3 + 2 & = & 18
\end{array}
$$

- Choose $p_1, \ldots, p_k$ pairwise relatively prime, split watermark into $\frac{k(k-1)}{2}$ pieces of the form $W \equiv r \bmod p_i p_k$.
- Use an enumeration scheme to turn these into integers, run through a block-cipher, embed into program.

# PBW — ECC Encode

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

$$
\begin{array}{rrcl@{\quad}rcl@{\quad}l}
 & W & \equiv & 5 \bmod p_1p_2 & 5 & = & 5 \\
W = 17 \Rightarrow & W & \equiv & 7 \bmod p_1p_3 & \Rightarrow\ p_1p_2 + 7 & = & 13 \quad \Rightarrow \\
 & W & \equiv & 2 \bmod p_2p_3 & p_1p_2 + p_1p_3 + 2 & = & 18
\end{array}
$$

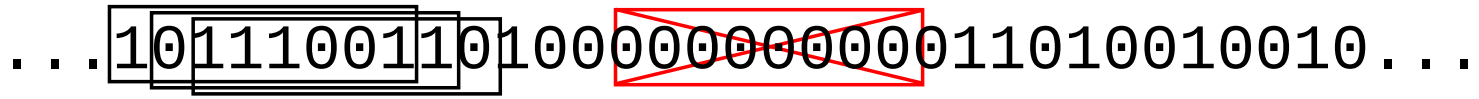
$$
\overbrace{11\cdots01}^{64}
\overbrace{01\cdots11}^{64}
\overbrace{10\cdots00}^{64}
$$



- Choose $p_1, \ldots, p_k$ 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.

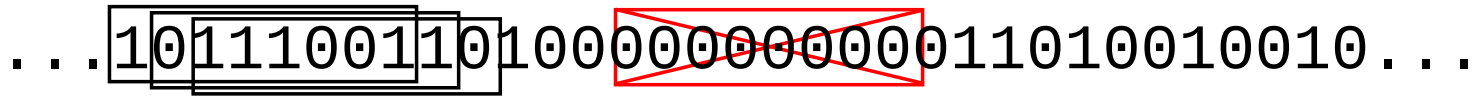
# PBW — ECC Decode

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



# PBW — ECC Decode

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

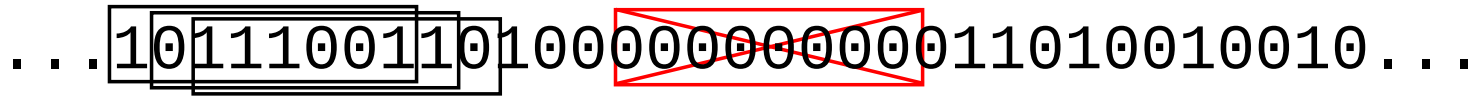



- Reconstruct $W \equiv r \bmod p_i p_k$ by inverting enumeration scheme.

$$\begin{array}{l} 11\cdots01 \\ 01\cdots11 \\ 10\cdots00 \\ 10\cdots11 \end{array} \quad \Rightarrow \quad \begin{array}{c} 5 \\ 13 \\ 17 \\ 0 \end{array} \quad \Rightarrow \quad \begin{array}{lcl} W & \equiv & 5 \bmod p_1 p_2 \\ W & \equiv & 7 \bmod p_1 p_3 \\ W & \equiv & 1 \bmod p_2 p_3 \\ W & \equiv & 0 \bmod p_1 p_2 \end{array}$$

# PBW — ECC Decode

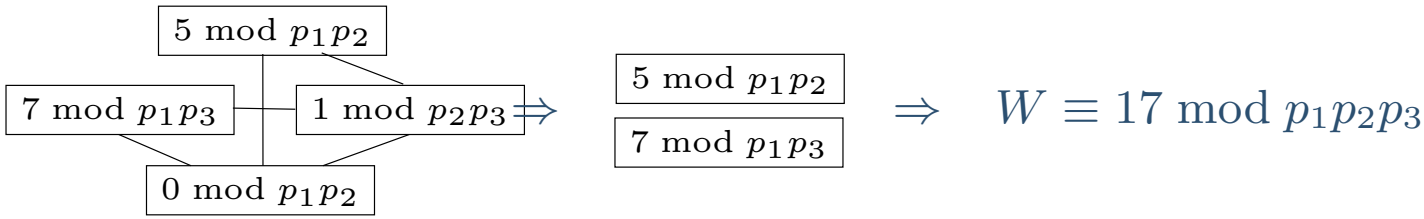
- Slide a 64-bit window across the bitstream. Throw out those that don't meet randomness criteria.




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

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

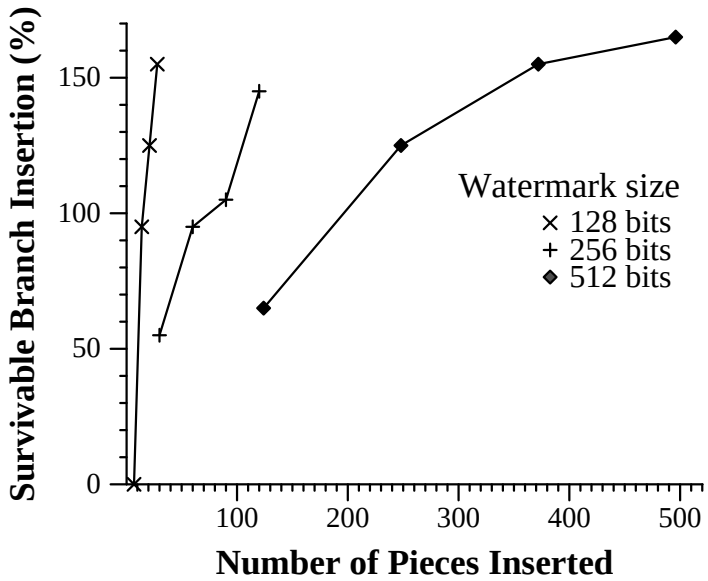
- Build a graph of statements inconsistent wrt to GCRT. Compute "most consistent" subgraph.



# 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



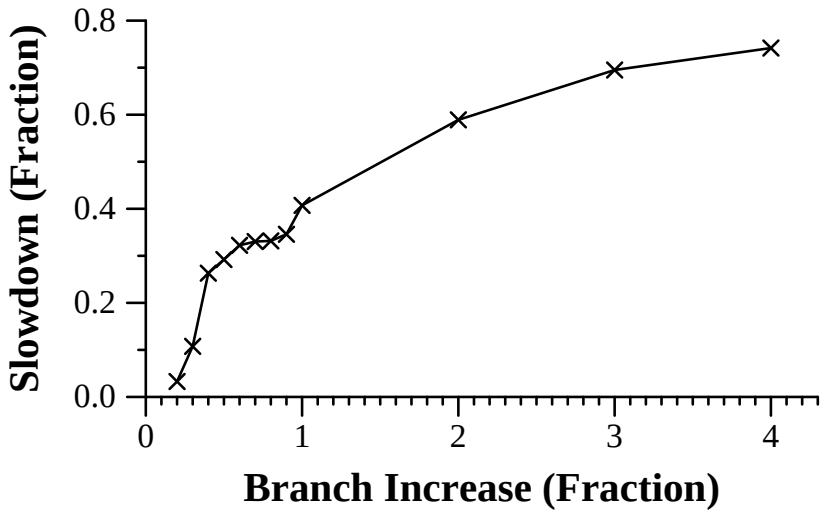



- With a 256-bit mark and 100 pieces, the attacker needs to double the number of branch instructions in the program in order to destroy the mark.

# PBW — Adding Branches Attack

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



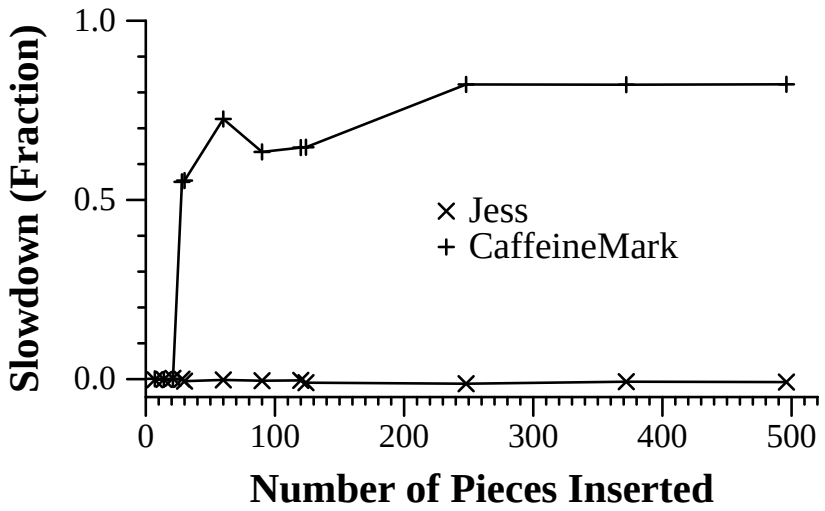



- By doubling the number of branches, the attacker slows down the program by about 40%.

# 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.

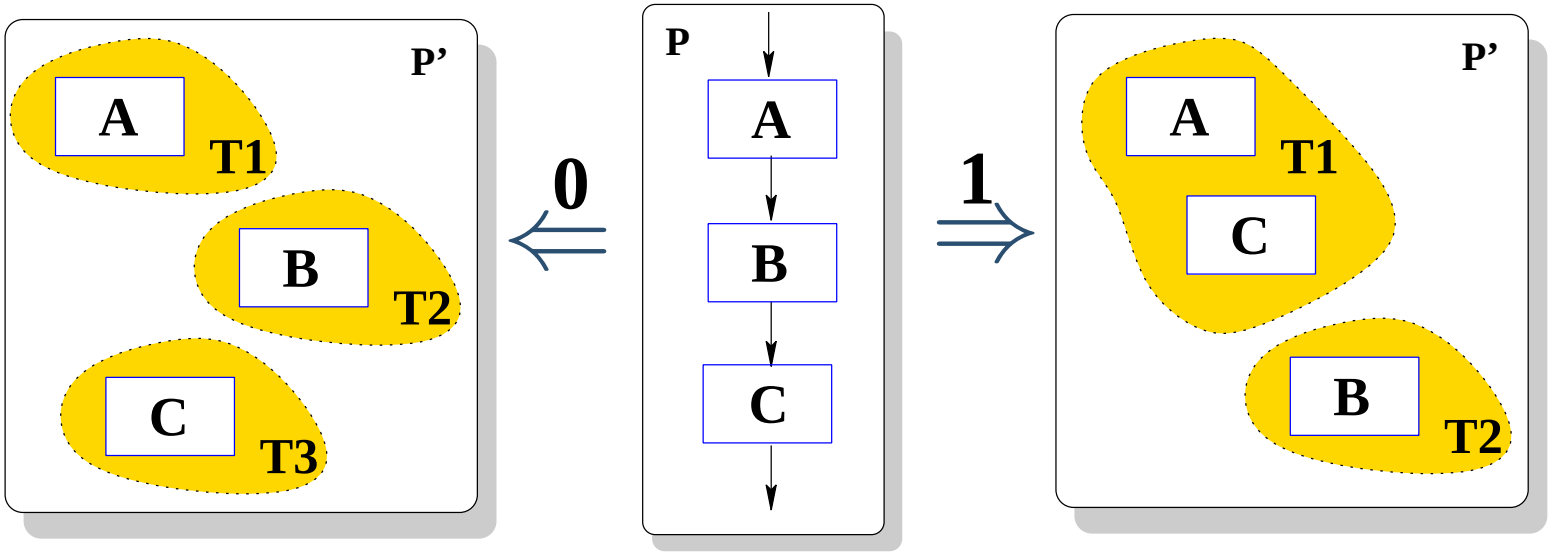





- For Jess we avoid the hotspots, so slowdown is negligible.
- For CaffeineMark we can't avoid the hotspots, so slowdown is $> 50\%$.



# — Thread-Based Watermark

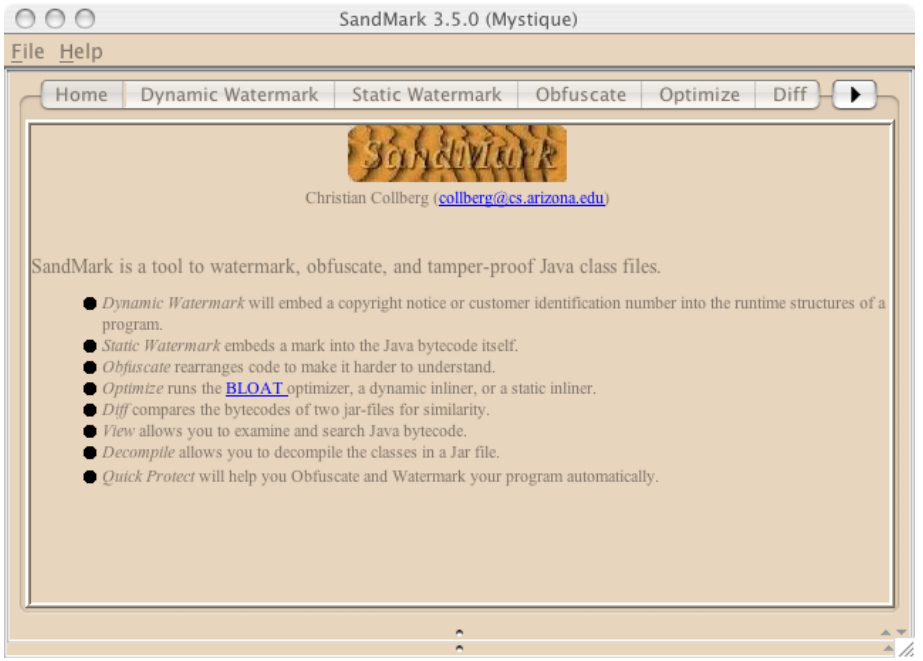



- Embed mark in which threads execute which basic blocks.
- Can have huge performance degradation.
- Why? Parallelism-analysis is hard.

Nagra & Thomborson, 6th Information Hiding Workshop, IHW'04

# SandMark — A Software Protection Tool





# SandMark — A Software Protection Tool

33 Obfuscation Algorithms

# SandMark — A Software Protection Tool

33 Obfuscation Algorithms

16 Watermarking Algorithms

# SandMark — A Software Protection Tool

33 Obfuscation Algorithms

16 Watermarking Algorithms

6 Birthmarking Algorithms

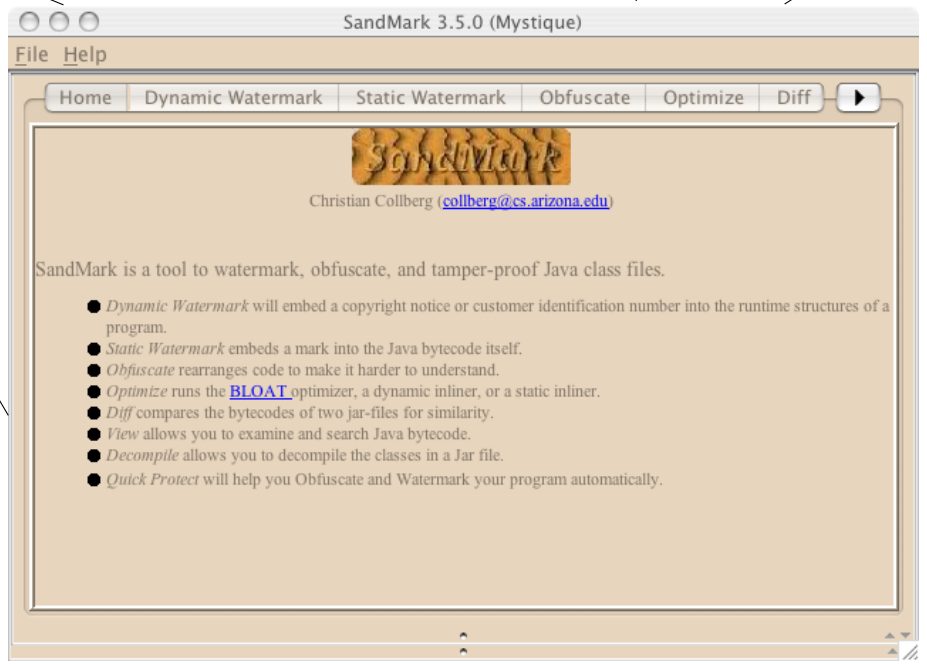
# SandMark — A Software Protection Tool

33 Obfuscation Algorithms

16 Watermarking Algorithms

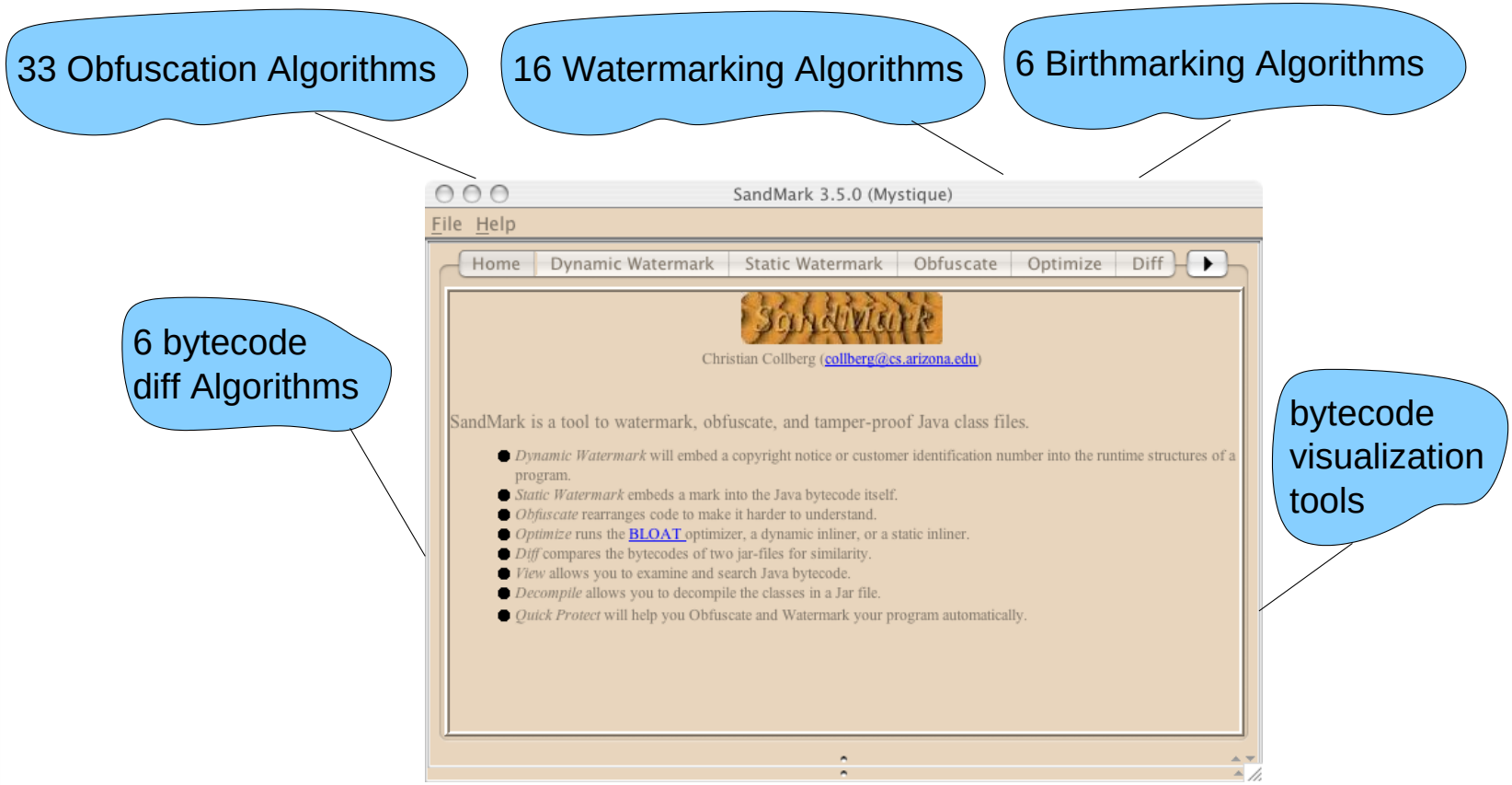
6 Birthmarking Algorithms

6 bytecode diff Algorithms



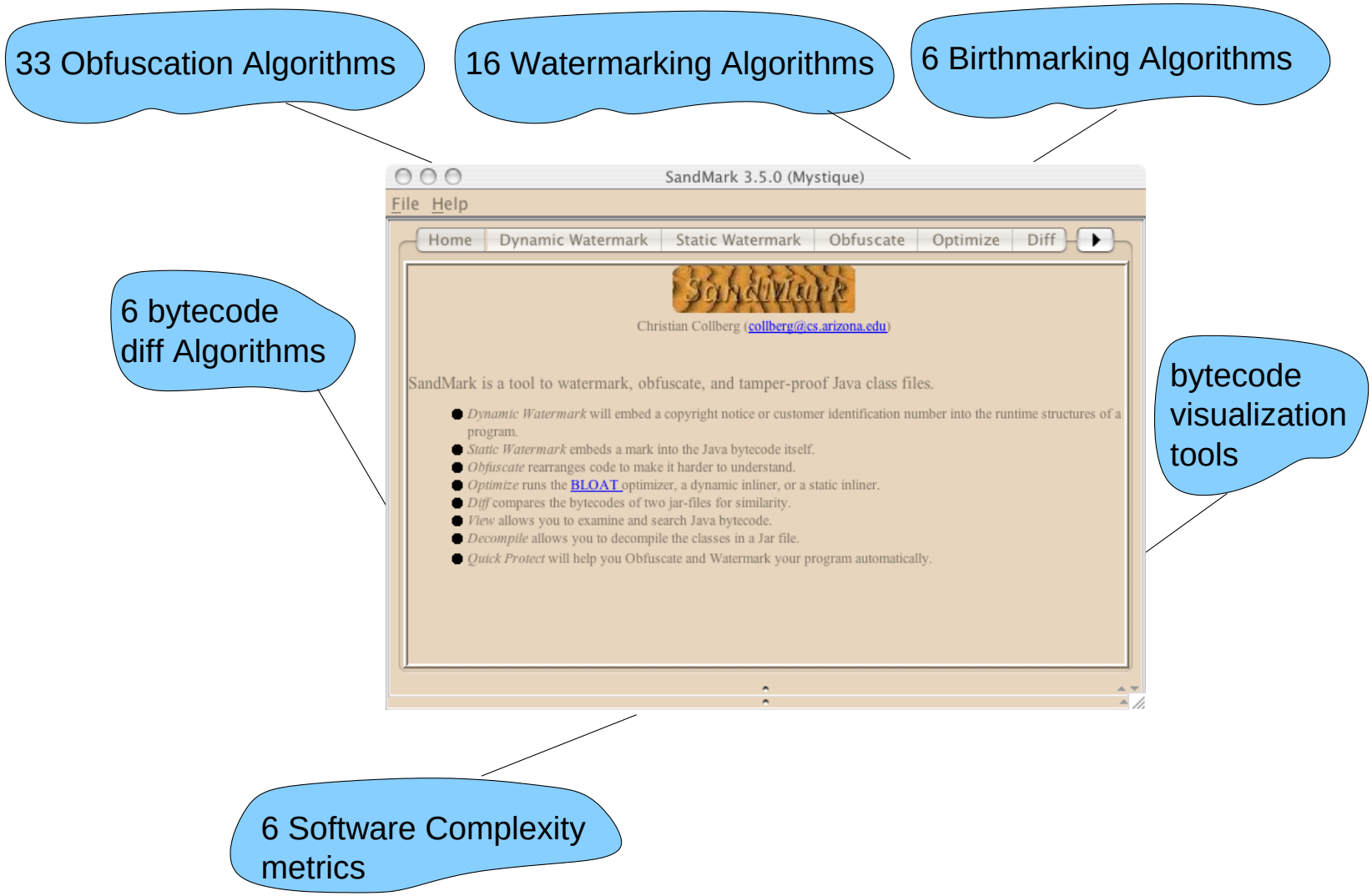
# SandMark — A Software Protection Tool





# SandMark — A Software Protection Tool





# SandMark — A Software Protection Tool

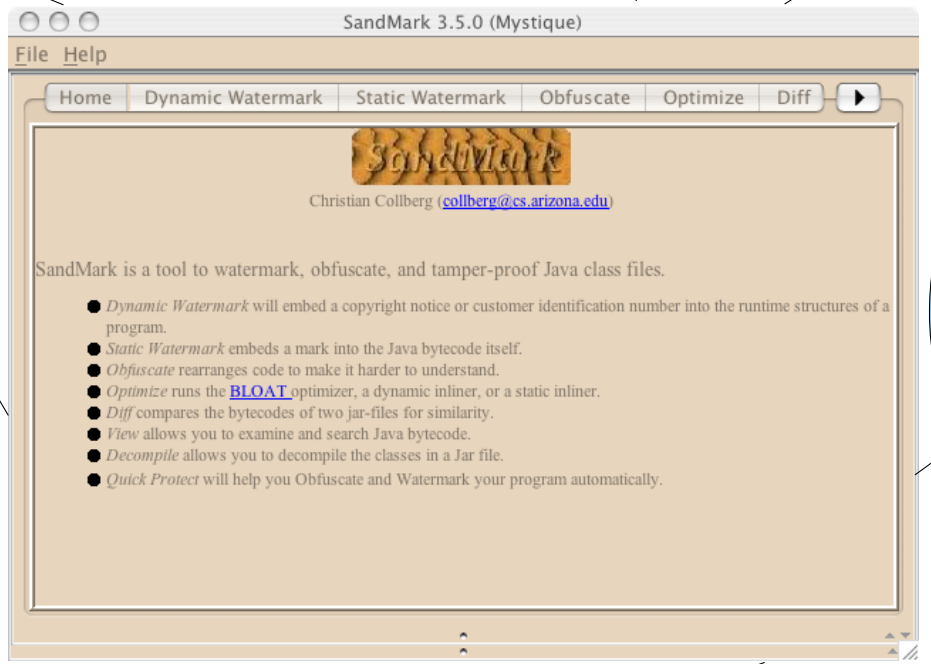


33 Obfuscation Algorithms

16 Watermarking Algorithms

6 Birthmarking Algorithms

6 bytecode diff Algorithms



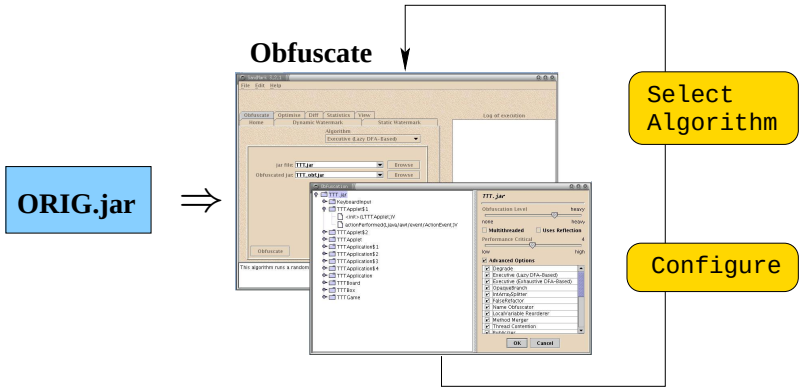

bytecode visualization tools

6 Software Complexity metrics

Large toolbox (graphs, static analysis,etc)

# A Session with SandMark

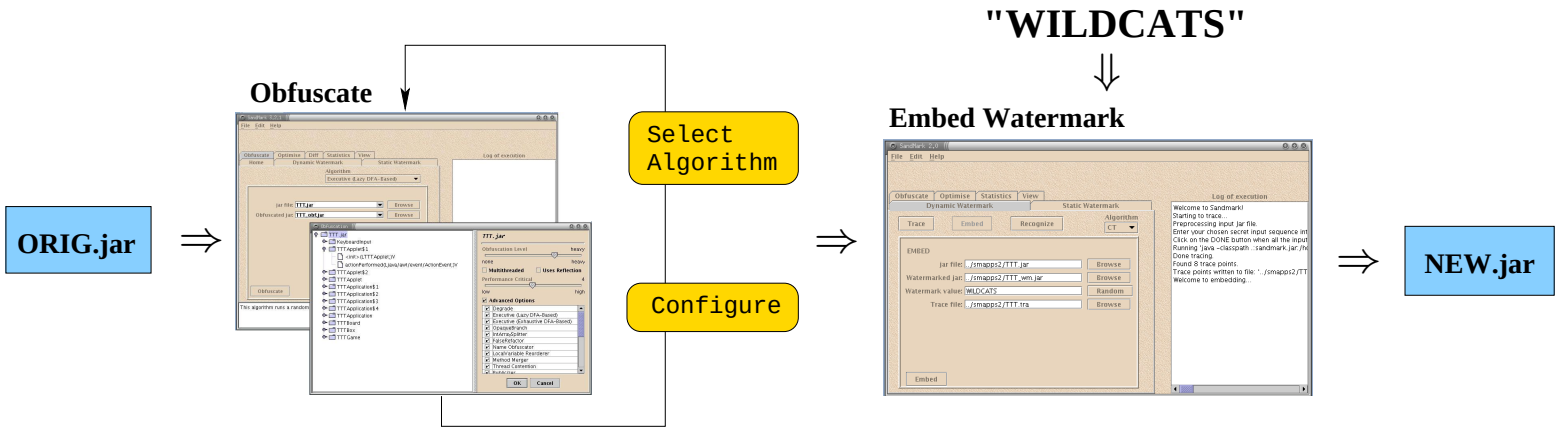





- We obfuscate to protect against attacks by
  1. reverse engineering
  2. collusive de-watermarking



# A Session with SandMark



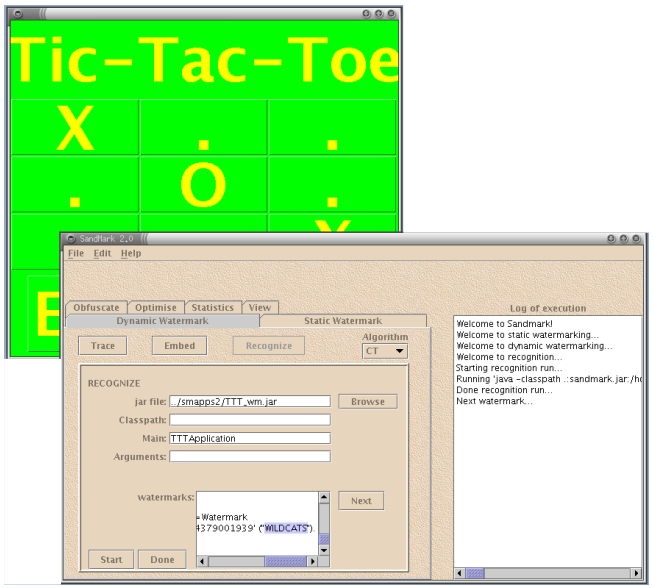

- We obfuscate to protect against attacks by
  1. reverse engineering
  2. collusive de-watermarking

# A Session with SandMark



**NEW.jar** ⇒

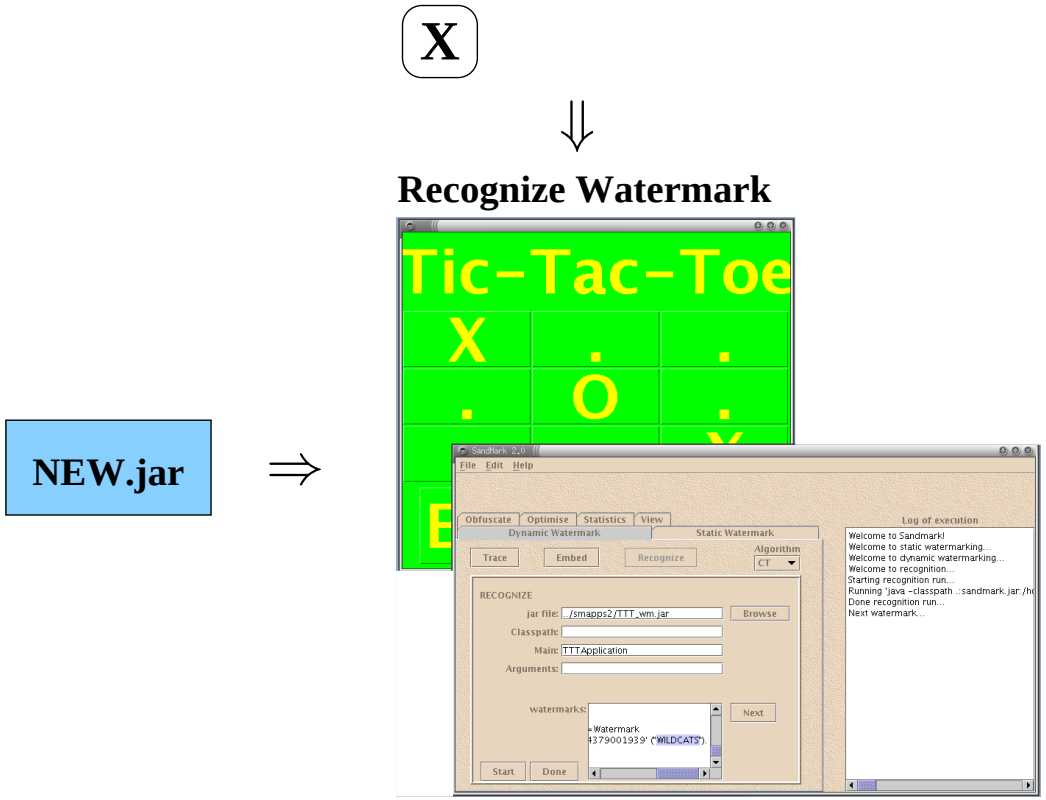
**Recognize Watermark**




- We extract the watermark to prove ownership.

# A Session with SandMark

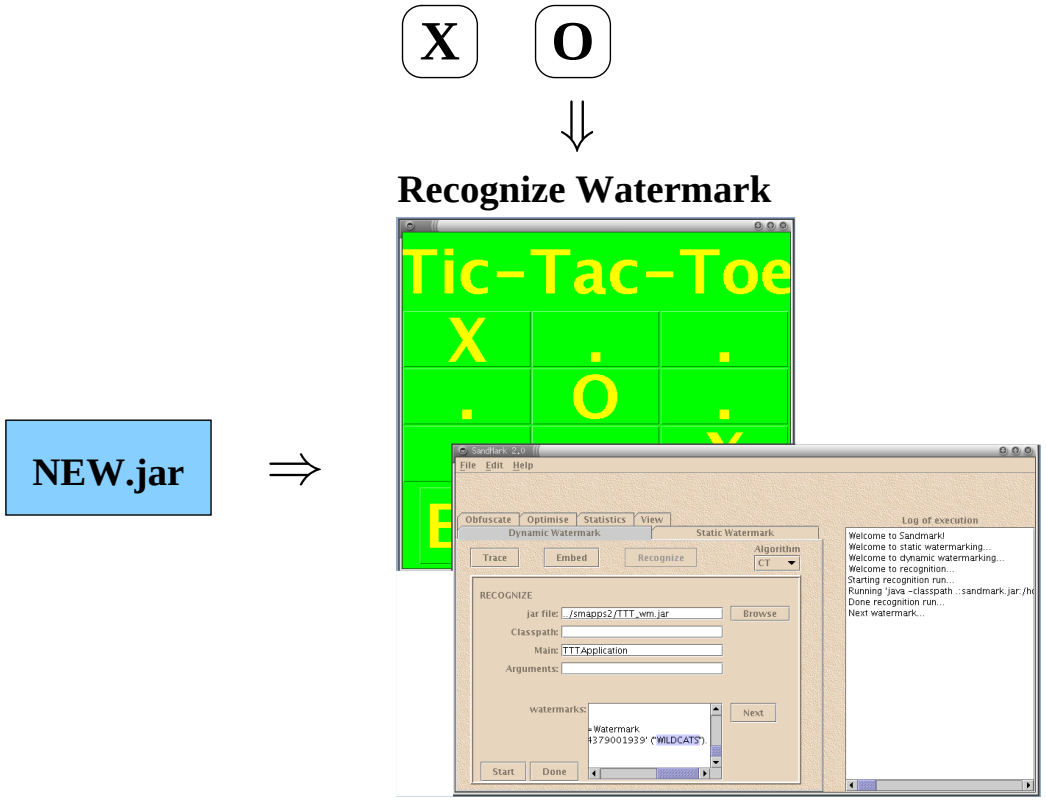





- We extract the watermark to prove ownership.

# A Session with SandMark

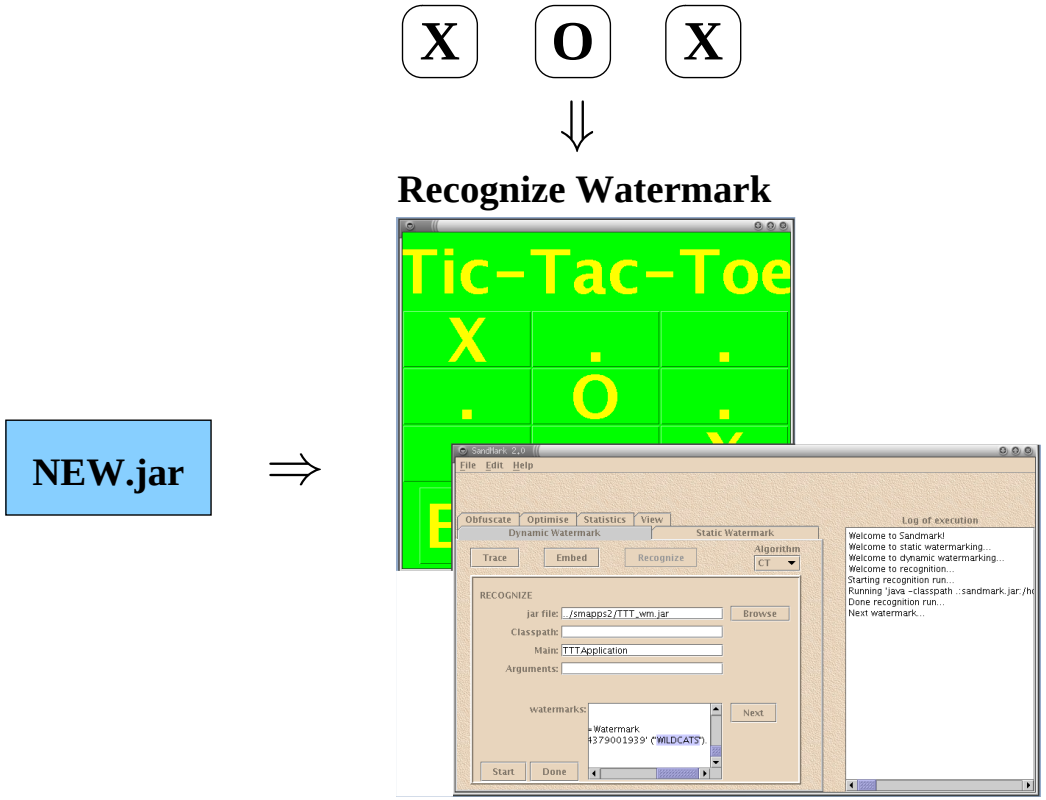





- We extract the watermark to prove ownership.

# A Session with SandMark

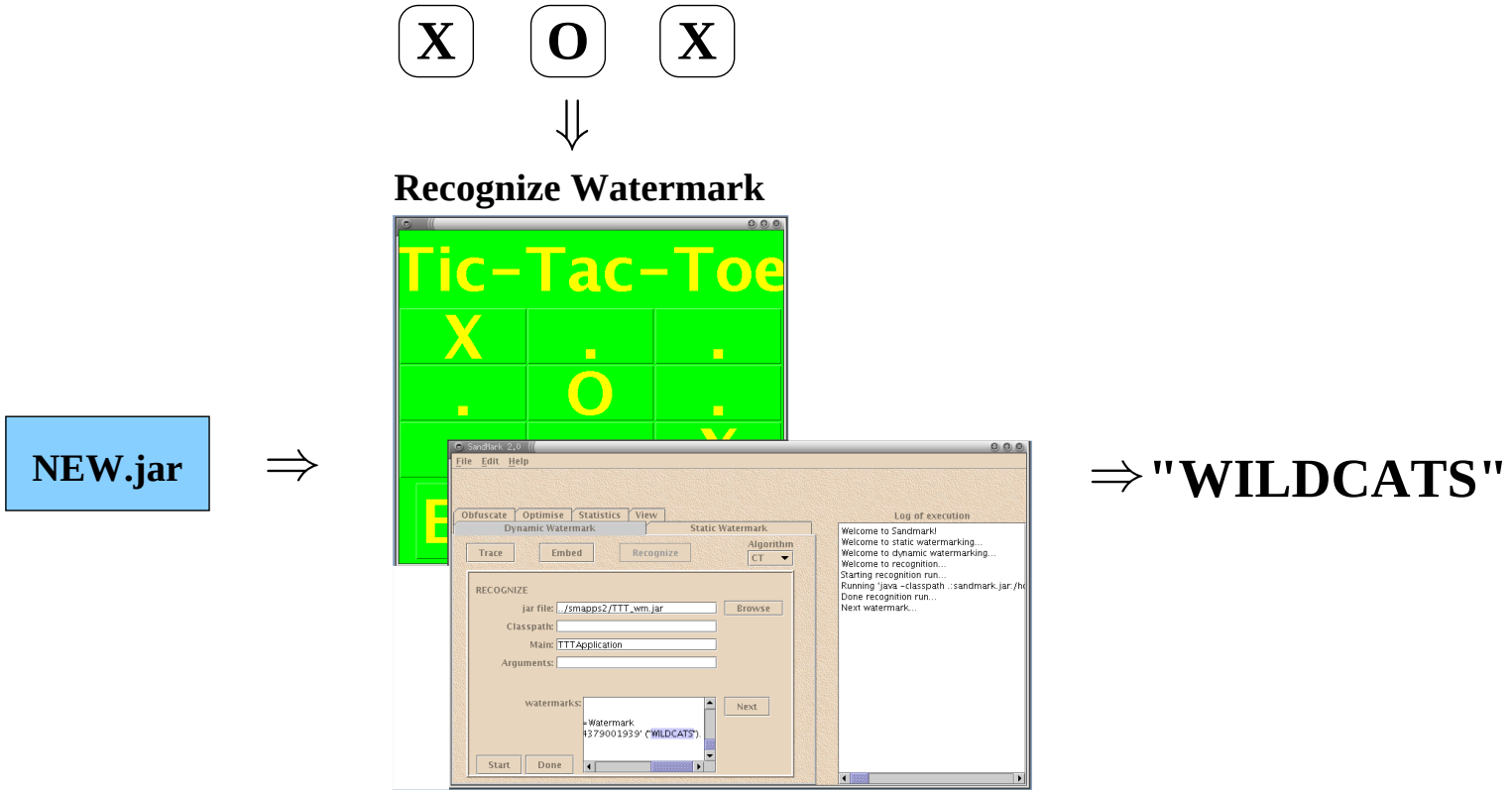





- We extract the watermark to prove ownership.

# A Session with SandMark



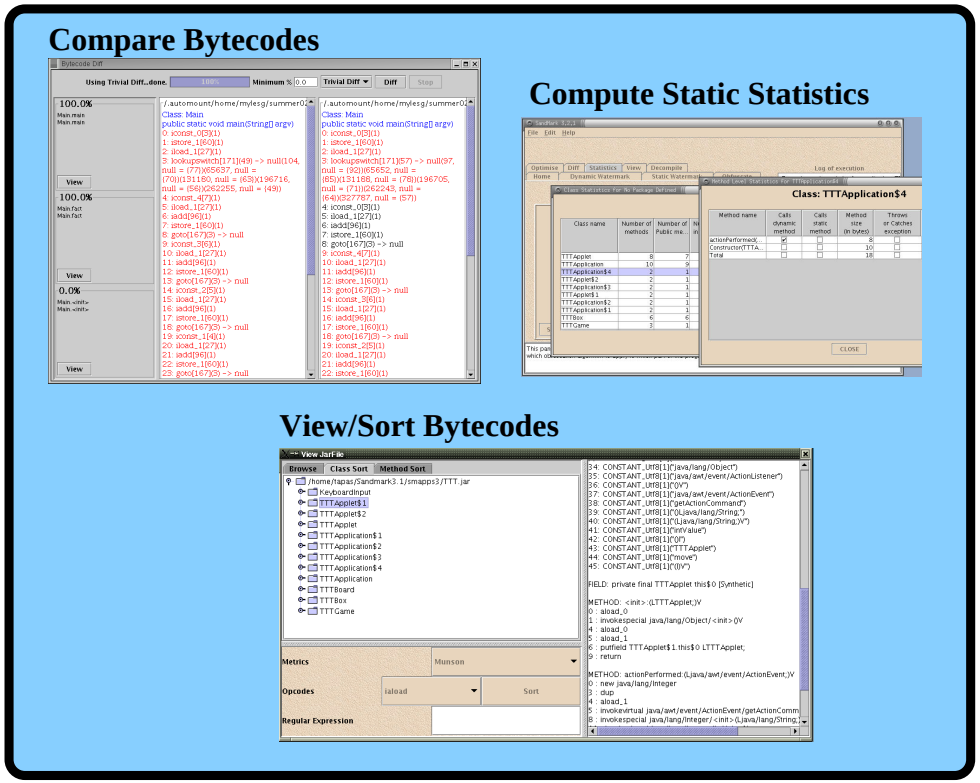



- We extract the watermark to prove ownership.

# A Session with SandMark



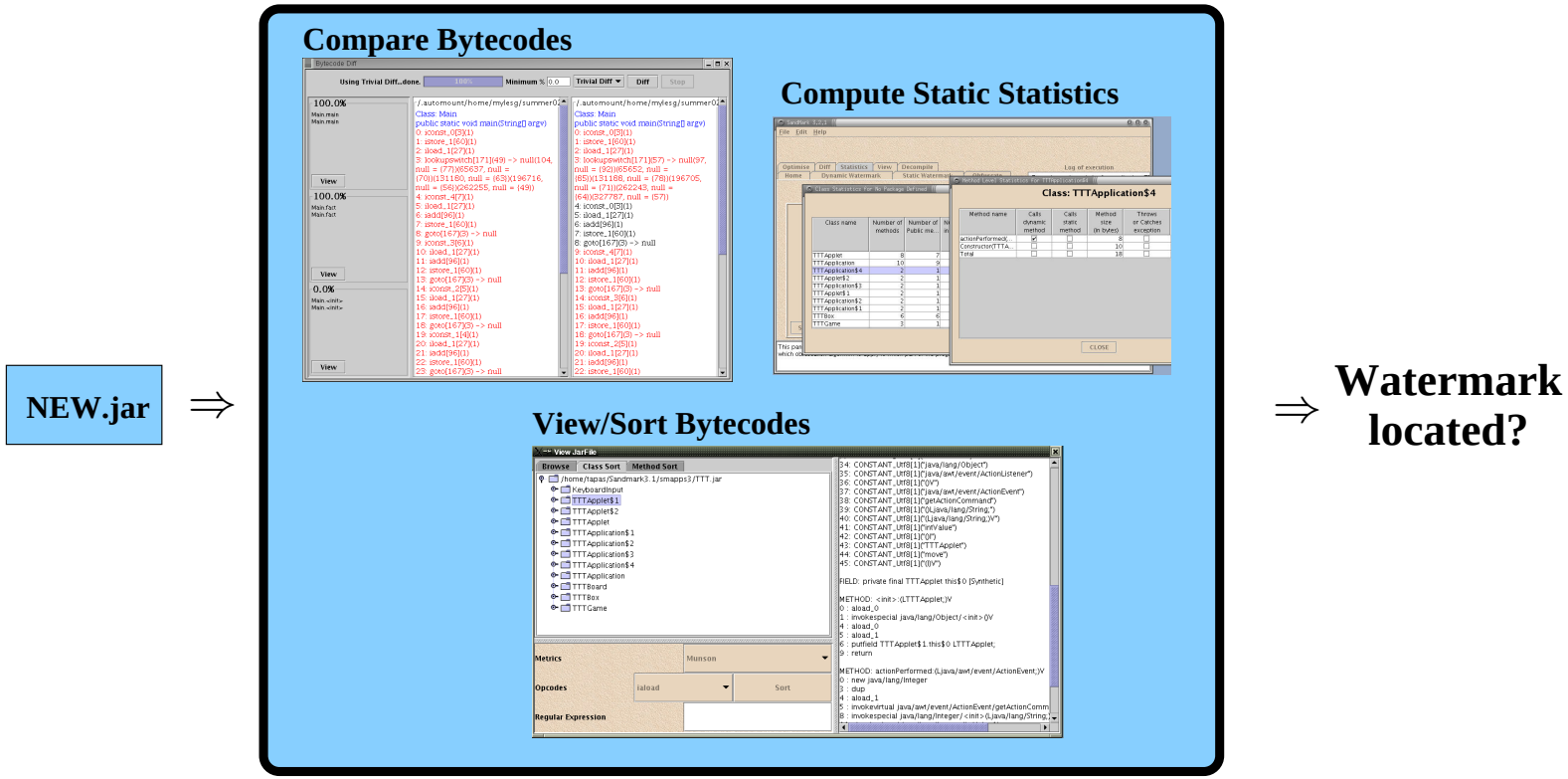
NEW.jar ⇒




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

# A Session with SandMark

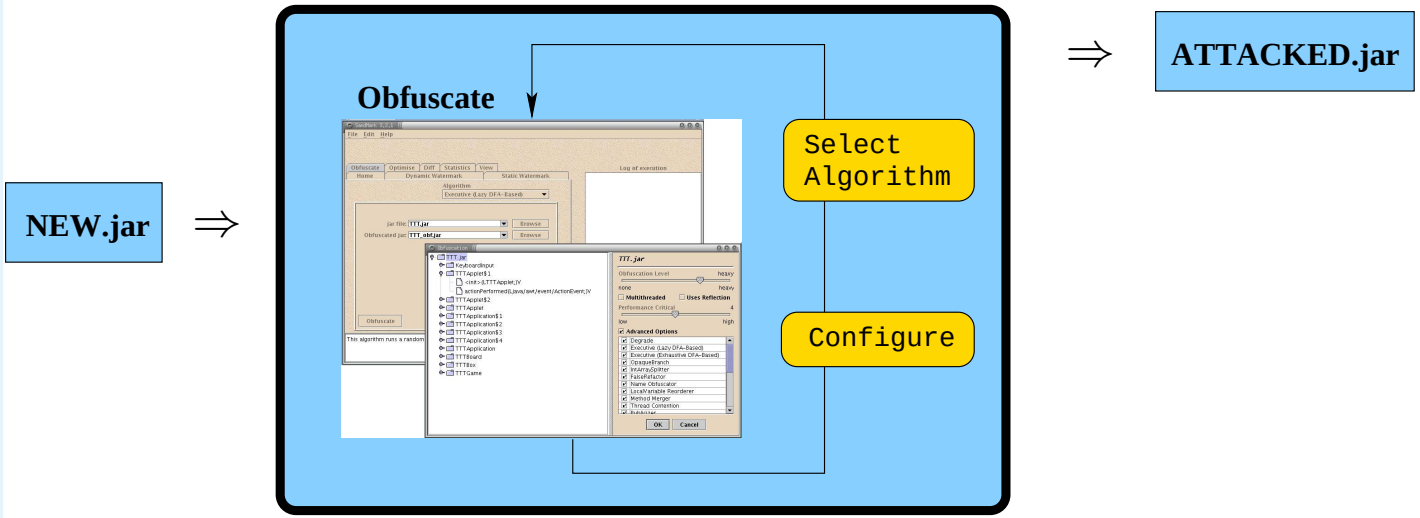





- To simulate a manual attack we examine the obfuscated/watermarked program using various static analysis tools.

# A Session with SandMark

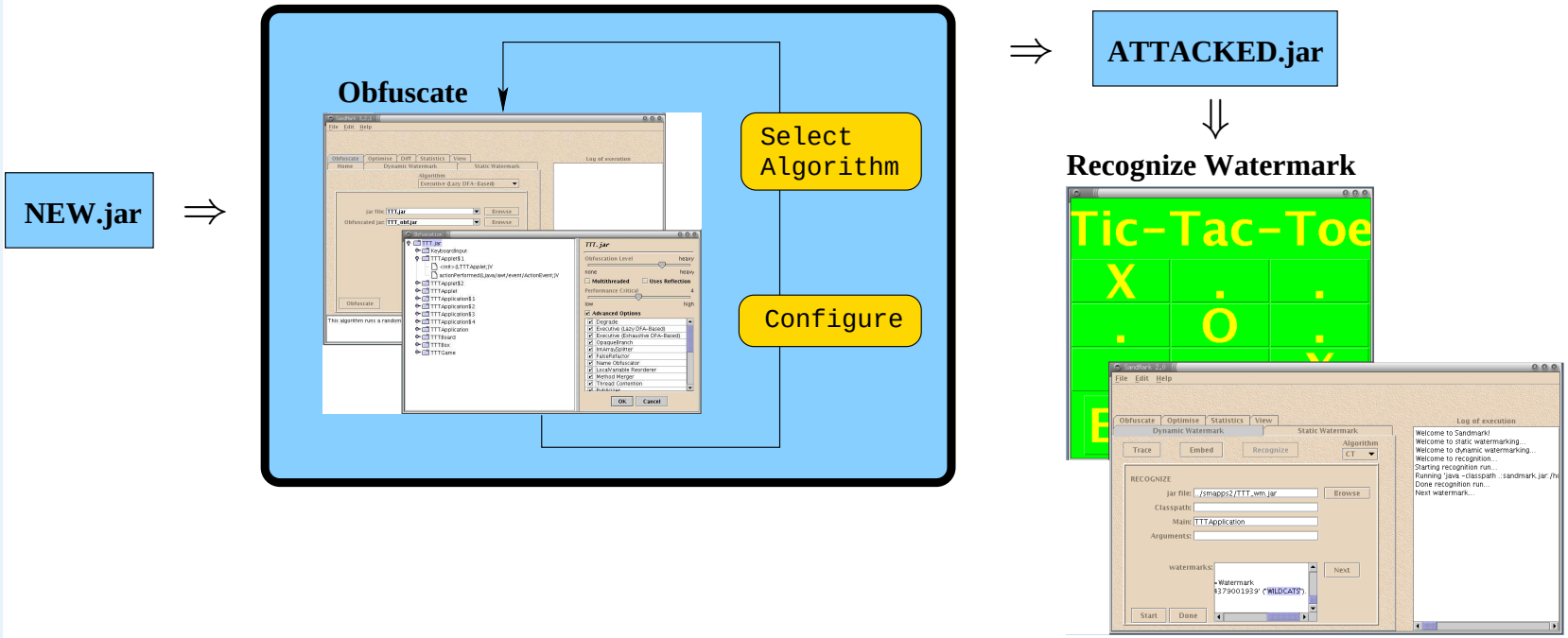





- To simulate an automatic attack we use SandMark's obfuscators ("*SoftStir*") to attack the watermark.

# A Session with SandMark

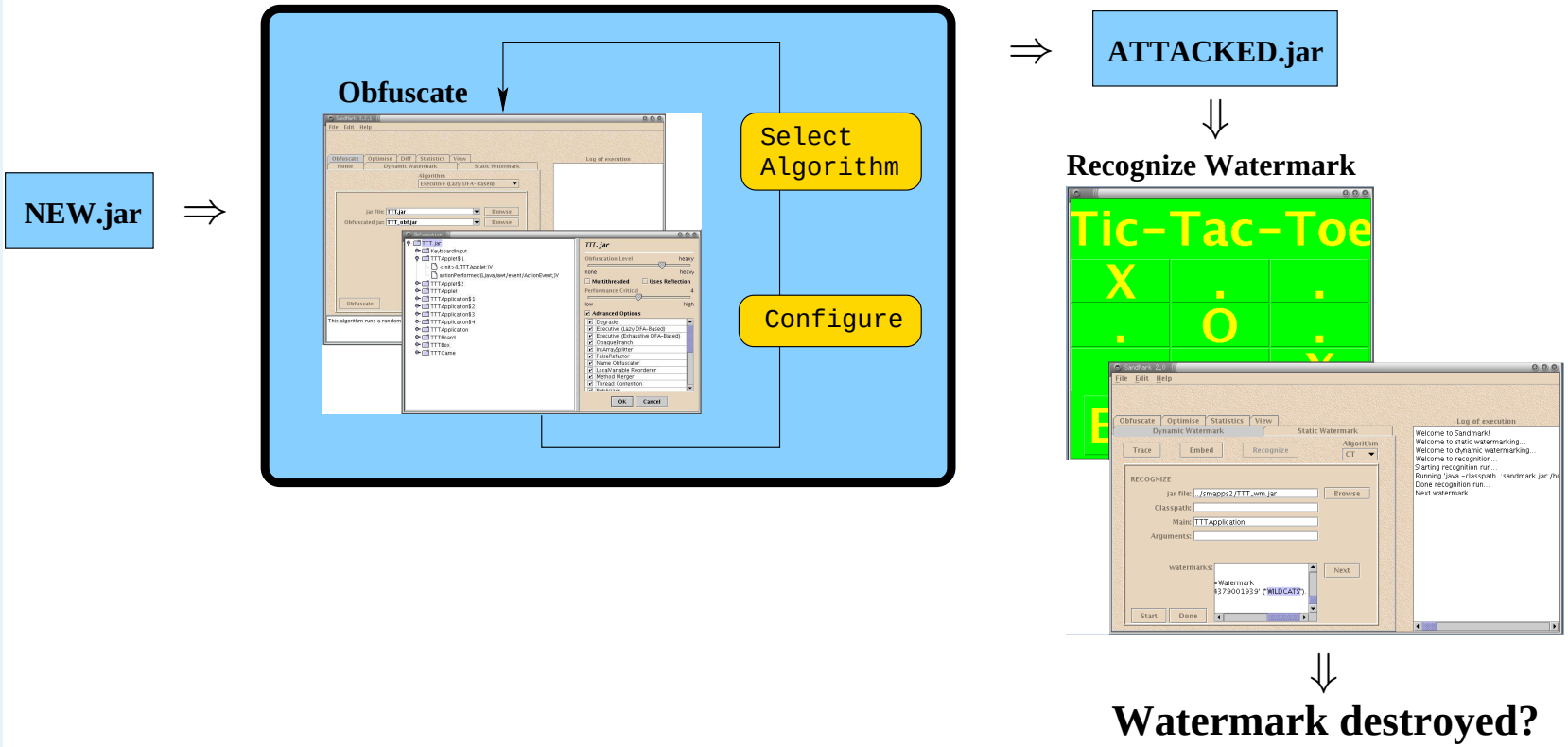





- To simulate an automatic attack we use SandMark's obfuscators ("*SoftStir*") to attack the watermark.

# A Session with SandMark

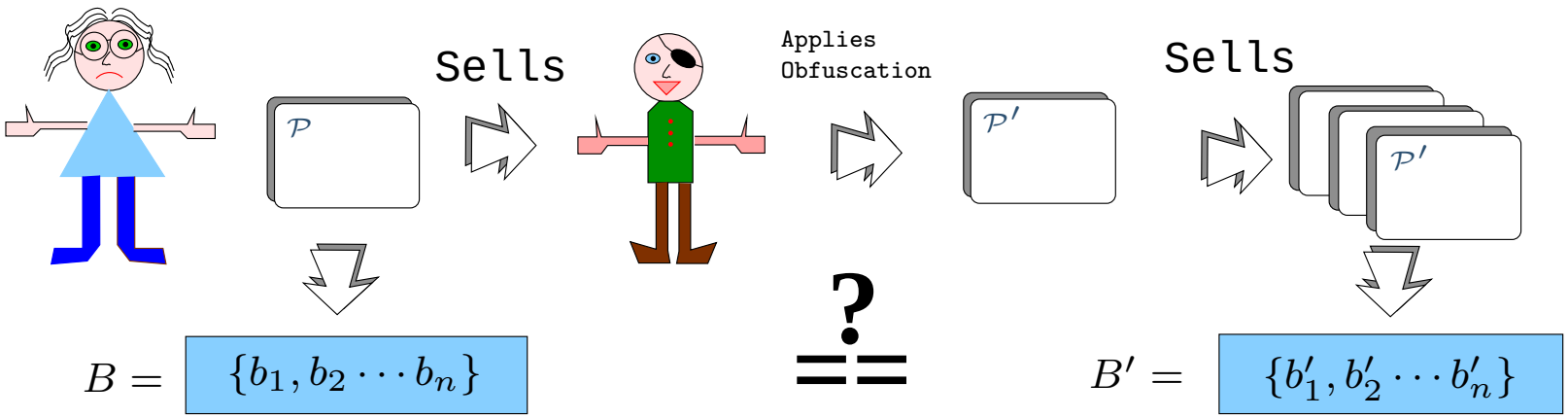





- To simulate an automatic attack we use SandMark's obfuscators ("*SoftStir*") to attack the watermark.

# Work in Progress — Birthmarking



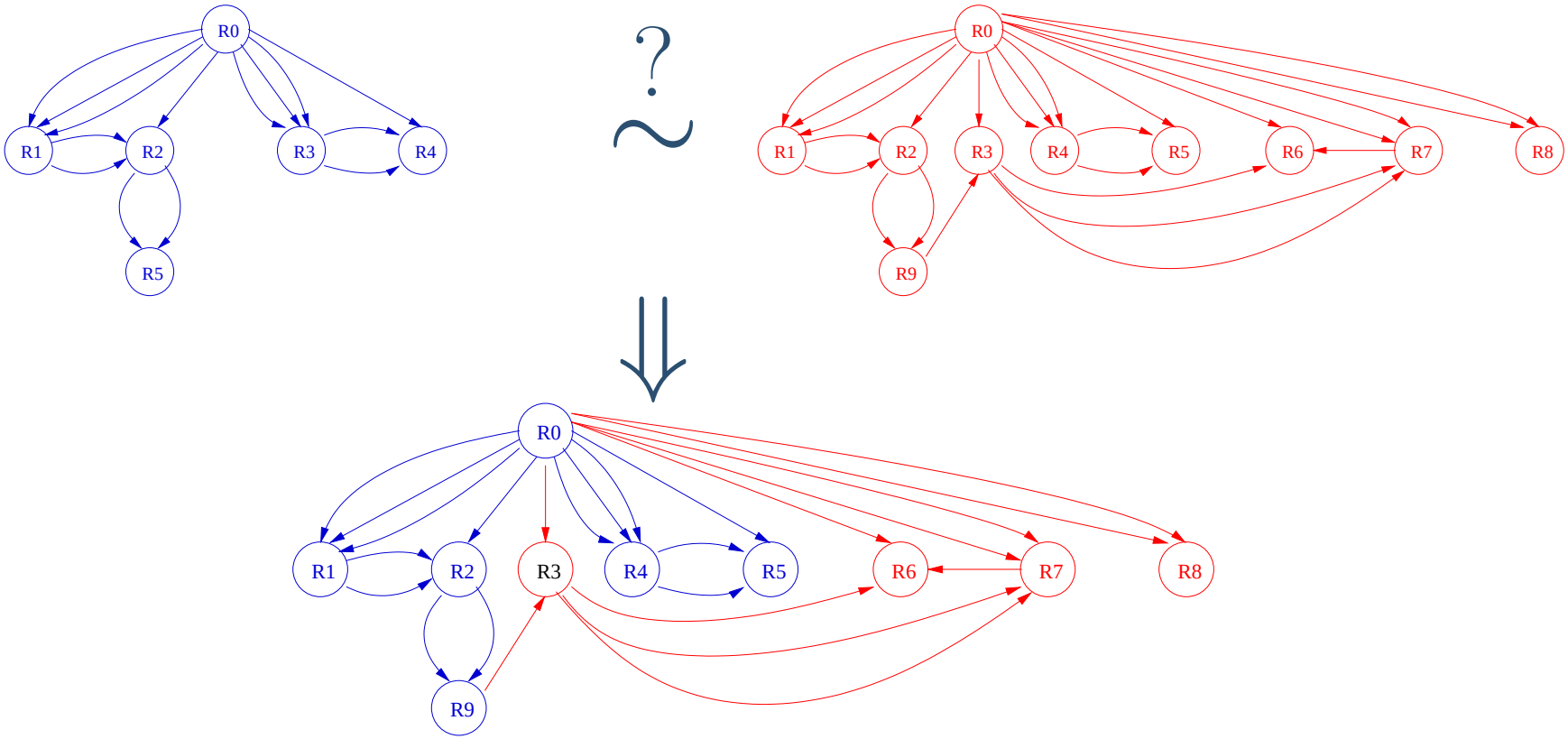



- A software birthmark is a set of unique characteristics of a program.
- Birthmarks are used to detect software theft.
- Characteristics are chosen to be invariant under semantics-preserving transformations.

Myles & Collberg, 7th Information Security Conference (ISC'04)

# Birthmarking — Whole-Program Paths

- We compute dynamic birthmarks using Whole-Program Paths and graph-similarity metrics.



# Conclusion



- Attention to details is important. The attcker will go for the easy target first.
- Implementations that cover *all* the corners are an important part of the evaluation.
- Are dynamic attacks harder to defend against?
- The goal is to make the attacker have to consider the entire program, not just "interesting pieces." Dynamic attacks are useful to chunk up the program.


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