• Currently Associate Professor at

• One year at

• Five years on the faculty at

• PhD from

Introduction

About me

• Intellectual property protection of software (sandmark.cs.arizona.edu).

• A tool for self-plagiarism detection (splat.cs.arizona.edu).

• No-cost static linking (slinky.cs.arizona.edu).

• Automatic retargeting of compilers.

In my free time play in The Zax (zaxband.com).

A search-engine for computer scientists (algovista.cs.arizona.edu).

Software Protection

Code Transformation Techniques
Overview of this talk

Software protection

Scenarios

Malicious reverse engineering

Bob wants to protect d’s privacy and integrity

Alice wants to protect d’s algorithms (reverse engineering)

* redistribute software piracy
* remove d’s license checks (tampering)
* discover d’s algorithms (reverse engineering)

Techniques for protecting against these attacks:

- Code obfuscation for protecting privacy
- Tamperproofing for protecting integrity
- Software watermarking for combating software piracy

Man-at-the-end attack scenarios.

Bullet points:

- Computer Games Industry: stealing 3rd party modules for
  their own games (code lifting).
- Bob, a rival developer, extracts and incorporates module M.
- Alice’s program contains a valuable trade secret (a clever
  algorithm or design).

Bob wants to

1. discover d’s algorithms (reverse engineering)
2. remove d’s license checks (tampering)
3. redistribute d’s software piracy

Alice wants to protect d’s privacy and integrity

Bob extracts module M into his own program (code lifting).
Scenario: Software piracy

Alice is a software developer.
Bob buys one copy of Alice's program.
Bob illegally sells copies to his friends.

Scenario: Digital rights management (DRM)

A DRM media player contains cryptographic keys that unlock and play encrypted music files.

Scenario: Mobile agent attack

Alice's mobile shopping agent visits online stores to find the best deal for a CD.
Bob manipulates the agent's code such that it returns his higher price as the best one.

Scenario: Grid computing

Bob buys cycles from Bob's supercomputer.
Bob snoops on confidential data/algorithms or tampers with Alice's program.

Scenario: License check tampering

if (today() > "Aug 17") abort()

Bob removes license checks to be able to run the program whenever he wants.

Alice protects her program so that it won’t run after being tampered with.

Scenario: Protocol discovery

Alice sells voice-over-IP call minutes.

Bob examines the VoIP client to discover proprietary protocols to build his own rival client.

Scenario: Protecting networked computer games

Ensure well-behaved TCP window flow control.

Bob Alice

To build his own rival client, Bob examines the VoIP client to discover proprietary protocols.

Alice sells voice-over-IP call minutes.

Scenario: Protecting Internet infrastructure

Ensure well-behaved TCP window flow control.

TCP flow control.

Ensure well-behaved TCP window flow control.

TCP flow control.

Ensure well-behaved TCP window flow control.

TCP flow control.
Scenario: Wireless sensor networks

- Sensor networks are common in military scenarios.
- The enemy can intercept/analyze/modify sensors.
- Sensor networks are common in military scenarios.

Scenario: Advanced Metering Infrastructure

- Disconnect Electrical Grid
- Smart Meter
- HACKED!

Scenario: Protecting military software

- The military and intelligence communities would like to be able to track the whereabouts of classified software.
- Much Air Force anti-tamper funding.

Scenario: Protecting medical records

- Medical records must be protected from improper access and modification.
- Medical data
- Confidential

Selective black-outs, consumers can adjust usage based on current costs, small-scale energy production...
Scenario: Software plagiarism

Student Bob copies a piece of Alice's program

Q: Who has copied from whom?

similarity (Q, P) = 80%

similarity (Q, R) = 20%

similarity (P, Q') = 10%

Scenario: Software forensics

Who wrote program S?

Extract features likely to identify each programmer:

similarity (f(Alice), f(S)) = 20%

similarity (f(Bob), f(S)) = 80%

similarity (f(Carol), f(S)) = 10%

Scenario: Teathered vs. Unteathered Scenarios

In a teathered scenario, Alice can detect tampering.

In an unteathered scenario, the program must protect itself.

Who wrote program Q:

Student Bob copies a piece of Alice's program?
Remainder of this talk

What can we do about man-at-the-end attacks?

Code Obfuscation

Then, I’ll show simple algorithms from each category.

First, an overview of these techniques:

- Code obfuscation for combating software piracy.
- Tamperproofing for protecting integrity.
- Software watermarking for combating software piracy.

Bob

P

P

′

We build tools to protect P against attack.

And infinite energy and patience.

And dynamic analysis tools: debuggers, tracers, emulators, slicers...

The adversary has full access to P.

The adversary has full access to P.

And infinite energy and patience.

And dynamic analysis tools: debuggers, tracers, emulators, slicers...

He has static analysis tools: disassemblers, decompilers, ...

Optimize for security, not speed! Programs will be larger.

We look for building a compiler.

Slower...

What can we do about man-at-the-end attacks?

1. Code obfuscation for protecting privacy;
2. Tamperproofing for protecting integrity;

First, an overview of these techniques.

Then, I’ll show simple algorithms from each category.

Choose one of these techniques.

Then, I’ll show simple algorithms from each category.

Choose one of these techniques.
To obfuscate a program means to transform it into a form that is more difficult for an adversary to understand or change than the original code.

Vague definition of difficult: The obfuscated program requires more human time, more money, or more computing power to analyze than the original program.

Malicious reverse engineering

Bob extracts algorithms/code from Alice’s program.
Alice obfuscates her program to slow down the attack.
Alice needs to obfuscate her code to prevent Bob from stealing her.

- Alice buys cycles from Bob's supercomputer.
- She obfuscates her code to prevent Bob from stealing her.
- Bob manipulates the agent's code such that it returns his higher price as the best one.
- Alice obfuscates the code to make it harder for Bob to
  analyze.
- Alice uses obfuscation to randomize her code such that a malicious agent will not be able to take advantage of a known vulnerability.
- A DRM media player contains cryptographic keys that unlock and play encrypted music files.
- Alice uses obfuscation to randomize her code such that a malicious agent will not be able to take advantage of a known vulnerability.

- A DRM media player contains cryptographic keys that unlock and play encrypted music files.
Types of obfuscation

1. Abstraction transformations
   Destroy module structure, classes, functions, etc.

2. Data transformations
   Replace data structures with new representations

3. Control transformations
   Destroy if-, while-, repeat-, etc.

4. Dynamic transformations
   Make the program change at runtime

Obfuscation example: original program
```
int main () {
  int y = 6;
  y = foo (y);
  bar (y, 42);
}
```

Obfuscation example: after abstraction transformation
```
int main () {
  int y = 6;
  y = foobar (y, 99, 1);
  foobar (y, 42, 2);
}
```

Obfuscation example: after data transformation
```
int main () {
  int y = 12;
  y = foobar (y, 99, 1);
  foobar (y, 36, 2);
}
```

Obfuscation example: after abstraction transformation
```
int main () {
  int y = 6;
  y = foobar (y, 99, 1);
  foobar (y, 42, 2);
}
```

Obfuscation example: after data transformation
```
int main () {
  int y = 12;
  y = foobar (y, 99, 1);
  foobar (y, 36, 2);
}
```

Obfuscation example: after control transformation
```
int main () {
  int y = 6;
  y = foobar (y, 99, 1);
  foobar (y, 42, 2);
}
```

Obfuscation example: after dynamic transformation
```
int main () {
  int y = 6;
  y = foobar (y, 99, 1);
  foobar (y, 42, 2);
}
```
Obfuscation example: After control transformation

```c
int foobar (int x, int z, int s)
{
    char * next = && cell0;
    int retVal = 0;
    cell0 : next = (s ==1)?&& cell1 :&& cell2; goto * next;
    cell1 : retVal = (x *37)%51; goto end;
    cell2 : next = (s ==2)?&& cell3 :&& end; goto * next;
    cell3 : next = (x==z)?&& cell4 :&& end; goto * next;
    cell4 : {
        int x2 = x*x % 51, x3 = x2*x % 51;
        int x4 = x2*x2 % 51, x8 = x4*x4 % 51;
        int x11 = x8*x3 % 51;
        printf("%i\n", x11); goto end;
    }
end : return retVal;
}
```

Tamperproofing

Tamperproofing has to do two things:

- Detect tampering
- Respond to tampering

Detection: but this is too unseemly!

```
if (tampering_detected()) abort
```

Essentially:

Tamperproofing makes the program useless to Bob if he tries to
modify it!

Necessary for digital rights management systems,
license checking code

Tamperproofing makes the program useless to Bob if he tries to modify it!

Two phases of tamperproofing

Tamperproofing has to do two things:

- Detect tampering
- Respond to tampering

```
Tamperproofing
```
Scenario: Digital rights management (DRM)

- Software Player
- Cleartext media
- Crypto keys
- Encrypted media

Alice needs to tamperproof to prevent Bob from modifying the player so that he can play without paying.

Scenario: License check tampering

- License check:
  - Alice needs to tamperproof to prevent Bob from removing the player so that he can play without paying.

Scenario: Protecting Internet infrastructure

- TCP−Client windowSz++
- if (congestion) Window
  - windowSz=4
- if (false) windowSz++

Alice needs to tamperproof to prevent Bob from modifying the TCP stack.

Scenario: Wireless sensor networks

- Radioactivity? Chemicals? Troup movements?
- Sensor
- Wifi
- CPU
- Code

Alice needs to tamperproof to prevent the sensor from sending bogus information to headquarters.

The TCP stack:

- Alice needs to tamperproof to prevent Bob from modifying the player so that he can play without paying.
Scenario: Advanced Metering Infrastructure

Disconnection
Electrical Grid
Smart Meter
HACKED!

Alice needs to tamperproof to prevent the meters from sending bogus information to the rest of the network.

Scenario: Protecting medical records

[Code snippet]

public class tamper {
    public static int checksum_self () {
        File file = new File("tamper.class");
        FileInputStream fr = new FileInputStream(file);
        DigestInputStream sha = new DigestInputStream(fr,
            MessageDigest.getInstance("SHA").digest());
        while (fr.read() != -1) {}
        byte[] digest = sha.getMessageDigest().digest();
        int result = 12;
        for (int i = 0; i < digest.length; i++)
            result = (result + digest[i]) % 16;
        return result;
    }

    public static void main(String[] args) {
        if (checksum_self() != 9) System.exit(-1);
    }
}

public class tamper {
    public static int checksum_self () {
        float celsius = Integer.parseInt(args[0]);
        float farenheit = checksum_self() * celsius / 5 + 32;
        System.out.println(celsius + "C = " + farenheit + "F");
    }

    public static void main(String[] args) {
        float celsius = Integer.parseInt(args[0]);
        float farenheit = checksum_self() * celsius / 5 + 32;
        System.out.println(celsius + "C = " + farenheit + "F");
    }
}

Sending bogus information to the rest of the network.

Alice needs to tamperproof to prevent the meters from being bypassed.

Alice needs to tamperproof to prevent authentication devices from being bypassed.

Sending: Protecting medical records

Scenario: Advanced Metering Infrastructure
Software Piracy

Bob buys one copy of Alice’s program.
Bob illegally sells copies to his friends.
Alice watermarks/fingerprints her program.
Alice uses the fingerprint to trace the program back to Bob.
Alice’s lawyer sues for software piracy!

Trivial static watermark

Embed the watermark as string constants included in the source of a program.

```java
public class Fibonacci {
    String copyright = " Copyright © Alice";
    public int fibonacci ( int n ) {
        if ( n <= 2 )
            return 1;
        else
            return fib ( n - 1 ) + fib(n - 2);
    }
}
```

Watermarking API

A watermarking system consists of two functions embed and extract.

- Bob wants to destroy the mark before reselling the object!
- Disturb the extract function so that Alice can no longer get the mark.
- Example: Bob can obfuscate the program to destroy the mark!
Permutation-Based Watermarking

```java
switch (e) {
    case 0 : return "voided ";
    case 1 : return "countersign ";
    case 2 : return "transfer ";
    case 3 : return "billing ";
    case 4 : return "overdraft ";
    case 5 : return "balance ";
    case 6 : return "check ";
    default : return "Bogus !";
}
```

Attacks against software watermarks — Rewrite attack

Alice has to assume that Bob will try to destroy her marks before trying to resell the program!

One attack will always succeed...

Ideally, this is the only effective attack.

An additive attack can help Bob to cast doubt in court as to whose watermark is the original one.

Bob can also add his own watermarks to the program:

A distortive attack applies semantics-preserving transformations to try to disturb Alice's recognizer:

Transformations: code optimizations, obfuscations,...

Transformations

Attacks against software watermarks — Distortive attack

Ideally, this is the only effective attack.

Extract

Rewrite

Distortive attack

Extract

Rewrite

Attacks against software watermarks — Rewrite attack

Attacks against software watermarks — Additive attack

An additive attack can help Bob to cast doubt in court as to whose watermark is the original one.

Bob can also add his own watermarks to the program:

Additive attack

Extract

Distortive

Rewrite

Additive

Extract

Additive

Additive attack

Extract

Rewrite

Additive attack
Attacks against software watermarks — Collusive attack

Bob buys two differently marked copies and comparing them to discover the location of the fingerprint:

- Alice should apply a different set of obfuscations to each distributed copy, so that comparing two copies of the same program will yield little information.

Hardware for software protection

- You ship your program along with a hard to clone physical object (dongle).
- The program won’t run without the dongle.
- Annoying to the user, expensive for the manufacturer.
- Still need to protect (obfuscate/tamperproof) the communication with the dongle!

Preventing piracy: Hardware token

- The code of your obfuscated, watermarked, and tamperproofed program will always be available to the attacker — the user needs the code in order to run it!
- Hardware-based protection provides a safe haven for data, code, and/or execution.
Protecting integrity: Encrypted execution

- Physical attack
- Snooping attack
- Environmental data

Encrypted data
- Encrypt/decrypt
- Address bus
- Data bus

CPU
- RAM
- Code

Alice
- Bob
- Pub
- Priv

Encrypt the program. CPU contains key.

Add sensors for temperature, voltage, clock-signal, radiation.
Add layers of shielding.

Discussion

- The IBM 4758 is $4000 and pretty tamperproof.
- A smart card is <$40 and pretty easy to tamper with.

Protecting the hardware: Physical barriers

- Penetration
- Temperature
- Radiation
- Power
- Clock

Attacker abuses the crypto-processor to force it to reveal its key.

3DES, RSA, SHA-1

EEPROM, RAM

VCC, RST, VPP, CLK, I/O

The IBM 4758 is $4000 and pretty tamperproof.

Protecting the hardware: Tamper-resistant processors

The IBM 4758 is $4000 and pretty tamperproof.
Reasons to use software protection

- Short-time protection
- Dealing with unique programs
- High-value programs
- Help legal prosecution
- Development cycle
- Performance
- User annoyance
- Bugs

...and reasons not to!

- More complexity for engineers
- More protection → more overhead
- Some techniques make every distributed copy unique
- Run program on several machines
- Transfer program to new machine
- Corporations buy 10 licenses, use 50...
- Trace leaks of software...
- Code transformations can be buggy
- Code transformations can tickle bugs

Development cycle
- More complexity for engineers
- Some techniques make every distributed copy unique
- More protection → more overhead
- Short-time protection
- Dealing with unique programs
- High-value programs
- Help legal prosecution
- Performance
- User annoyance
- Bugs

...and reasons not to!

- More complexity for engineers
- More protection → more overhead
- Some techniques make every distributed copy unique
- Run program on several machines
- Transfer program to new machine
- Corporations buy 10 licenses, use 50...
- Trace leaks of software...
- Code transformations can be buggy
- Code transformations can tickle bugs

So, which algorithms should I use?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Effort to implement</th>
<th>Effort to defeat</th>
<th>Parallelizable?</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belt-and-suspenders</td>
<td>Uh, we don’t know...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If your program is crack-worthy, it will be cracked</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Be prepared to update your defenses</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, which algorithms should I use?

- Belt-and-suspenders (defense-in-depth)
- If your program is crack-worthy, it will be cracked
- Be prepared to update your defenses

How do I evaluate the effectiveness of an algorithm?

- Uh, we don’t know...

Give me a table like this!

- Belt-and-suspenders (defense-in-depth)
- If your program is crack-worthy, it will be cracked
- Be prepared to update your defenses
Control Flow Obfuscation

Transformations that make it difficult for an adversary to analyze the flow-of-control:

1. Insert bogus control-flow.
2. Complicate the flow-of-control.

None of these transformations are immune to attacks!

Opaque Expressions

Simply put, an expression whose value is known to you as the defender (at obfuscation time) but which is difficult for an attacker to figure out.

Graphical notation:

\[ P \] for an opaquely true predicate
\[ !P \] for an opaquely false predicate
\[ ?P \] for an opaquely indeterminate predicate
\[ E = v \] for an opaque expression of value \( v \)

Building blocks for many obfuscations.

opaque expressions
Opaque Expressions

An opaquely true predicate:
\[ \text{truefalse} \]
\[ (x^2 + x) \]

An opaquely indeterminate predicate:
\[ \text{false true} \]
\[ x \mod 2 = 0 \]

Simple Opaque Predicates

Look in number theory textbooks, in the problems sections:

\[ \forall x, y \in \mathbb{Z}: x^2 - 34y^2 \neq 1 \]
\[ \forall x \in \mathbb{Z}: 2 \mid x^2 + x \]

Algorithm obfCTJ bogus: Inserting bogus control-flow

1. dead branches which will never be taken
2. superfluous branches which will always be taken
3. branches which will sometimes be taken and sometimes not, but where this doesn’t matter

It seems that the blue block is only sometimes executed:

\[ \text{an opaquely indeterminate predicate:} \]
\[ \text{an opaquely true predicate:} \]
Algorithm \text{obfCTJ\_bogus}: Inserting bogus control-flow

- Sometimes execute the blue block, sometimes the green block.
- The green and blue blocks should be semantically equivalent.

A bogus block (green) appears as it might be executed while, in fact, it never will.

Algorithm \text{obfWHKD}: Control-flow flattening

- Removes the control-flow structure of functions.
- Known as chenxify, chenxification.
- Put each basic block as a case inside a switch statement, and wrap the switch inside an infinite loop.

Extend a loop condition $P$ by conjoining it with an opaquely true predicate $P'$.
```c
int modexp ( int y, int x[], int w, int n) {
    int R, L; int k = 0; int s = 1;
    while (k < w) {
        if (x[k] == 1)
            R = (s*y) % n;
        else
            R = s;
        s = R*R % n; L = R; k ++;
    }
    return L;
}
```

Performance penalty

Replacing 50% of the branches in three SPEC programs slows them down by a factor of 4 and increases their size by a factor of 2. Why?

Use gcc's labels-as-values to construct a jump table that lets you jump directly to the next basic block.

Keep tight loops as one switch entry.

Jump table check on the next variable and an indirect jump through a

The for loop incurs one jump, the switch statement a bounds

Why?

Switch (next)

{ }

```
switch (next)
{ }
```
Algorithm OBFWHKD_alter: Control-flow flattening

```c
int modexp (int y, int x[], int w, int n) {
    int R, L, k, s;
    char * jtab[] = {case0, case1, case2, case3, case4, case5, case6};
    goto * jtab[0];
    case0: k = 0; s = 1; goto * jtab[1];
    case1: if (k < w) goto * jtab[2]; else goto * jtab[6];
    case2: if (x[k] == 1) ... goto * jtab[5];
    case4: R = s; goto * jtab[5];
    case5: s = R * R % n; L = R; k++; goto * jtab[1];
    case6: return L;
}
```

Invariants:
- Every third cell (in pink), starting with cell 0, is \( \equiv 1 \mod 5 \);
- Cells 2 and 5 (green) hold the values 1 and 5, respectively;
- Every third cell (in blue), starting with cell 1, is \( \equiv 2 \mod 7 \);
- Cells 8 and 11 (yellow) hold the values 2 and 7, respectively.

You can update a pink element as often as you want, with any value you want, as long as you ensure that the value is always \( \equiv 1 \mod 5 \).
int g[] = {36, 58, 1, 46, 23, 5, 16, 65, 2, 41, 2, 7, 1, 37, 0, 11, 16, 2, 21, 16};

if((g[3] % g[5]) == g[2])
    printf("true!

g[5] = (g[1] * g[4]) % g[11] + g[6] % g[5];
g[14] = rand();
g[4] = rand() * g[11] + g[8];

int six = (g[4] + g[7] + g[10]) % g[11];
int seven = six + g[3] % g[5];
int fortytwo = six * seven;

Opaque Predicates

Constructing opaque predicates

Construct them based on number theoretic results

∀ x, y ∈ Z: x^2 - 34y^2 ≠ 1
∀ x ∈ Z: 2 | x^2 + x
X' ∈ Z: x' ≠ 3x
number theoretic results

protect team by
the hardness of concurrency analysis
the hardness of aliasing

Initialize at runtime!
Green: an opaque value 42.
Blue: g is constantly changing at runtime.
Pink: opaque true predicate.

Algorithm obfCTJ:
alias: Opaque predicates from pointer
Algorithm obfCTJ alias

Creates opaque predicates from pointer analysis problems.

The algorithm tries to go beyond the capabilities of known analysis algorithms:

Despite a great deal of work on both flow-sensitive and context-sensitive algorithms [...], none has been shown to scale to programs with millions of lines of code, and most have difficulty scaling to 100,000 lines of code.

Alias analysis algorithms are designed to perform well on "normal code" written by humans:

Breaking opaque predicates

1. Insert opaque queries such as \((q_1 \neq q_2)\) into the code.
2. Perform enough operations to confuse even the most precise alias analysis algorithms.
3. "If \(q_1\) points to a node in \(G_1\) and \(q_2\) points to a node in \(G_2\)"
4. "\(G_1\) and \(G_2\) are circular linked lists."
5. Two invariants:

```
  ... goto q 
  ... \(q \rightarrow q\)
  ... 
  ... \(\cdots \rightarrow \cdots\) 
  ... \(\cdots \rightarrow \cdots\)
```
Breaking opaque predicates

int x = some complicated expression;
int y = 42;

boolean b = (34* y*y -1)== x*x;
if b goto ...

1 Compute a backwards slice from b,
2 Find the inputs (x and y),
3 Find range of x and y,
4 Use number-theory/brute force to determine \( b \equiv false \).

Algorithm rePMBG: Breaking \( \forall x \in \mathbb{Z}: p(x) \)

Attack opaque predicates confined to a single basic block.

Start at a conditional jump instruction \( j \) and incrementally extend it with the 1, 2, ... instructions until an opaque predicate is found.

Assume that the instructions that make up the predicate are contiguous.

Brute force evaluate, or use abstract interpretation.

101 / 210

Breaking opaque predicates

How to make attacker’s task more difficult? Make it harder to find \( f(x_1, x_2, \ldots) \); find the inputs \( x_1, x_2, \ldots \) to \( f \); find the ranges \( R_1, R_2, \ldots \) of \( x_1, x_2, \ldots \); or determine the outcome of \( f \) for all argument values.

102 / 210

Brute force evaluate, or use abstract interpretation.

Predicate (or beginning of basic block) is found.

extend it with the 1, 2, ... instructions until an opaque predicate is found.

Assume that the instructions that make up the predicate are contiguous.

Attack opaque predicates confined to a single basic block.

Brute force evaluate, or use abstract interpretation.
Using Abstract Interpretation

Consider the case when $x$ is an even number:

$x = \text{even number}$;

$y = x \times x$;

$y = y + x$;

$z = y \% 2$;

$b = z ==0$;

if $b$ . . .

Consider the case when $x$ starts out being odd:

$x = \text{odd number}$;

$y = x \times a$;

$x = \text{even}$;

$y = y + x$;

$z = y \% 2$;

$b = z ==0$; = true

if $b$ . . .

Regardless of whether $x$'s initial value is even or odd, $b$ is true!

Dynamic Obfuscation

You've broken the opaque predicate, efficiently!!

Static vs. Dynamic obfuscation

Static obfuscations transform the program at runtime.

Dynamic algorithms transform the program prior to execution.

Static obfuscation counter attacks by static analysis.

Dynamic obfuscation counter attacks by dynamic analysis.

Dynamic Obfuscation
int modexp (int y, int x[], int w, int n, int mode) {
    int R, L, k = 0, s = 1, t;
    char *p = open_begin; while (p < (char *)open_end) *p++ ^= 99;
    if (mode == 1) return 0;
    while (k < w) {
        begin:
        if (x[k] == 1) R = (s * y) % n;
        else R = s;
        s = R * R % n; L = R;
        end:
        k ++;
    }
    p = open_begin; while (p < (char *)open_end) *p++ ^= 99;
    return L;
}

int main () {
    makeCodeWritable (···);
    modexp (0, NULL, 0, 0, 1);
    · · ·
}

Modeling dynamic obfuscation — compile-time

A dynamic obfuscator runs in two phases:

1. Transformer T creates p's initial configuration.
2. Transformer T continuously modifies p at runtime.

Modeling dynamic obfuscation — runtime

A dynamic obfuscator turns a "normal" program into a self-modifying one.

- At compile-time, the obfuscator transforms the program to an initial configuration.
- At runtime, the obfuscator runs the program with calls to the runtime transformer.

We'd like an infinite, non-repeating series of configurations.
In practice, the configurations repeat.
The Dynamic Primitive — Aucsmith

The cells are divided into two regions in memory, upper and lower. A function is split into cells.

\[ C_0 : C_5 \]
\[ C_4 \]
\[ C_3 \]
\[ C_2 \]
\[ C_1 \]
\[ C_0 \]

Dynamic obfuscation: Aucsmith’s algorithm

Scheduling transformations

XOR!
```c
void xor (caddr_t from, caddr_t to, int len) {
    int i;
    for (i = 0; i < len; i++) {
        *to ^= *from;
        from++;
        to++;
    }
}

void swap (caddr_t from, caddr_t to, int len) {
    int i;
    for (i = 0; i < len; i++) {
        char t = *from;
        *from = *to;
        *to = t;
        from++;
        to++;
    }
}

#define CELLSIZE 64
#define ALIGN asm volatile (" .align 64 \n")

void P() {
    static int firsttime = 1;
    if (firsttime) {
        xor (&& cell5, && cell2, CELLSIZE);
        xor (&& cell0, && cell3, CELLSIZE);
        swap (&& cell1, && cell4, CELLSIZE);
        firsttime = 0;
    }

    char *a[] = {&& align0, && align1, && align2, ...
                 , && align5};
    char *next[] = {&& cell0, && cell1, && cell2, && cell3, && cell4, && cell5};
    goto *next[0];
    ....
}

int main (int argc, char *argv[]) {
    makeCodeWritable (...);
    P(); P();
    ....
}
```

Discussion

Code Obfuscation — What’s it Good For?

Diversification — make every program unique to prevent malware attacks
Prevent collusion — make every program unique to prevent diffing attacks
Code Privacy — make programs hard to understand to protect algorithms
Data Privacy — make programs hard to understand to protect secret data (keys)
Integrity — make programs hard to understand to make them

Performance:

Dynamic algorithms generate self-modifying code. Bad for:
flush instruction pipeline
write data caches to memory

Diversification vs. Dynamic Obfuscation

Static obfuscations confuse static and dynamic analyses.

Static obfuscations confuse static and dynamic analyses.

Common Obfuscating Transformations

Many obfuscating transformations are built on some simple operations:

Apply these basic operations to:

Duplication
Indirection
Mapping
Reordering
Duplication
Mapping
Splitting/Merging

Abstract structures
Data structures
Control structures

Control structures
Data structures
Abstract structures

Hand to change:

Integrity — make programs hard to understand to prevent
protect secret data (keys)

Data Privacy — make programs hard to understand to
protect algorithms

Code Privacy — make programs hard to understand to
prevent
malware attacks

Prevent collusion — make every program unique to prevent
malware attacks

Diversification — make every program unique to prevent
What is tamperproofing?

Ensure that a program executes as intended, even in the presence of an adversary who tries to disrupt, monitor, or change the execution.

A tamperproofing algorithm

1. Makes tampering difficult
2. Detects when tampering has occurred

From your program:
- Prevent an adversary from removing license checking code
- Prevent an adversary from adding missing functionality

What are typical applications/scenarios?

Prevent an adversary from removing license checking code.

- DRM media players and TV-set-top boxes change you to watch movies or listen to music.
- Voice-over-IP clients charge you money to make phone calls.
- Adobe’s free PDF reader lets you fill in a form and print it.
- Some evaluation products don’t allow you to print.
- But not save it.
- Some evaluation copies stop working after a certain period of time.
- Voice-over-IP clients charge you money to make phone calls.

What semantics to maintain/hack?

Adobe’s free PDF reader lets you fill in a form and print it.

- Games don’t provide you with an infinite supply of ammunition.
- Voice-over-IP clients charge you money to make phone calls.
- Evaluation copies stop working after a certain period of time.
- Voice-over-IP clients charge you money to make phone calls.
- Some evaluation products don’t allow you to print.
- But not save it.
- Some evaluation products don’t allow you to print.
- Voice-over-IP clients charge you money to make phone calls.
- Evaluation copies stop working after a certain period of time.
- Voice-over-IP clients charge you money to make phone calls.
- Some evaluation products don’t allow you to print.
- But not save it.

Prevent an adversary from removing license checking code.

- Adobe’s free PDF reader lets you fill in a form and print it.
- Some evaluation products don’t allow you to print.
- But not save it.
Definitions

How do we respond to tampering?

1. Terminate the program.
2. Restore the program to its correct state, by patching the tampered code.
3. Deliberately return incorrect results, maybe deteriorate slowly over time.
4. Degrade the performance of the program.
5. Report the attack, for example by "phoning home".
6. Punish the attacker by destroying the program or objects in its environment.

What do we want?

1. Unadulterated hardware and operating system.
2. Unmodified P's code.
3. Not running under emulation.
4. The right dynamic libraries have been loaded.
5. Not being modified by a debugger.

How does the adversary attack P?

1. Modify files:
   - P's executable file
   - Dynamic linker
   - Dynamic libraries
   - P's executable file

2. Modify the operating system

3. Run P under emulation

4. Modify P while running under debugging

5. Modify P's code.
Augment the program with functions that compute a hash over a code region to compare to an expected value.

How can we be sure that the attacker won’t tamper with the hash computation itself?

1. Build up a network of checkers and responders.
2. Repair code that has been tampered with.
3. Checkers check the code and check each other as well.
4. Expected value.
5. Checkers compute a hash over a region and compare to the expected value.

Can we attack all introspection algorithms?

1. Hide the hash values.
2. Repair code that has been tampered with.
3. Build up a network of checkers and responders.

How can we be sure that the attacker won’t tamper with the over a code region to compare to an expected value.

Augment the program with functions that compute a hash.
Algorithm Chang & Atallah: Checker network

The input to the protection tool is the guard graph. It shows the relationship between regions to be protected, and the checkers and responders that check them.

Algorithm REWOS: Attacking self-hashing algorithms

Attack can just as well be external to the program.
4. Remove or disable them without destroying the rest of the program.
5. Analyze the code to locate the responders, then
6. Analyze the code to locate the checkers, or
7. How to attack introspection algorithms?

---

```c
uint32 A_COPY[] = {0x83e58955, 0x72b820ec, 0xc7080486, ...};
uint32 B_COPY[] = {0x83e58955, 0xaeb820ec, 0xc7080486, ...};

int main(int argc, char* argv[]) {
    A();
    int A_hash = hash(A);
    if (A_hash != 0x105AB23F) {
        memcpy(A, A_COPY);
    }
    A();
    int B() {
        ...
    }
    int B_hash = hash(B);
    if (B_hash != 0x4f4205a5) {
        memcpy(B, B_COPY);
    }
    B();
    ...
}
```

---

Algorithm reWOS: Attacking self-hashing algorithms

1. Analyze the code to locate the checkers, or
2. Analyze the code to locate the responders, then
3. Remove or disable them without destroying the rest of the program.
4. Attack can just as well be external to the program!

---

```
... {
    int B() {
        ...
    }
    if (B) {
        memcpy(B, B_COPY);
        if (B_hash = hash(B)) {
            B = B_hash = hash(B);
        }
    }
    A();
    if (A) {
        memcpy(A, A_COPY);
        if (A_hash = hash(A)) {
            A = A_hash = hash(A);
        }
    }
    int main(char argc, char* argv[]) {
        int B_COPY[] = {0x83e58955, 0x72b820ec, 0xc7080486, ...};
        int A_COPY[] = {0x83e58955, 0x72b820ec, 0xc7080486, ...};
    }
```
Processors treat code and data differently. TLBs (Translation Lookaside Buffers) and caches are split into separate parts for code and data. In the hash-based algorithms, code is accessed as code (when it's being executed) and as data (when it's being hashed), sometimes a function will be read into the I-cache and sometimes a function will be read into the D-cache.

A TLB miss will either generate an exception or throw a TLB miss. The hardware gives the OS control on a TLB miss and allows it to decide whether to throw an exception or continue processing.

Typical memory management system:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Page Frame</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>Page Index</td>
<td>Virtual Address</td>
</tr>
</tbody>
</table>

Page Tables

TLB

TLB hit

On a TLB hit, the operating system kernel fetches the new virtual-to-physical address mapping.

Attacking self-hashing algorithms:

Algorithm reWOS: Attacking self-hashing algorithms

Attack(P, K):
1. Copy program P to Porig.
2. Modify P as desired to a hacked version P′.
3. Modify the operating system kernel K such that data reads are redirected to Porig and instruction reads to P′.

ATTACK(K):
Algorithm \textsc{reWOS}: Attacking self-hashing algorithms

1. Copy \( P \) and modify \( P \), however you like.
2. Arrange the physical memory such that frame \( i \) comes from
   the hacked \( P \) and frame \( i + 1 \) is the corresponding original
   page from \( P_{\text{orig}} \).
3. Modify the kernel: if a page table lookup yields a \( v \rightarrow p \)
   virtual-to-physical address mapping, I-TLB is updated with
   \( v \rightarrow p \) and D-TLB with \( v \rightarrow p + 1 \).

The program tries to read its own code in order to execute it
\( \Rightarrow \) the processor throws an I-TLB-miss exception, the OS
updates the I-TLB to refer to the modified page.

The attacker has modified the program to bypass a
license-expired check.

The original program pages are in blue.

The modified program pages are in pink.

The program tries to read its own code in order to execute it
\( \Rightarrow \) the processor throws a D-TLB-miss exception, and the OS
updates the D-TLB to refer to the original, unmodified, page.
Oblivious hashing

Algorithm

IDEA: overlap basic blocks of x86 instructions. When one block executes it also computes a hash over the second block! The hash is computed without reading the code! Invulnerable to memory splitting attacks!

```
B0: shll 2, %eax
    incl %eax
    ret

B1: decl %eax
    shrl 3, %eax
    ret
```

Merge the blocks by interleaving the instructions, inserting jumps to maintain semantics:

```
B0: shll 2, %eax
    jmp I1

B1: decl %eax
    jmp I2
```

Want the two blocks also to share instruction bytes. Replace the jmp with xorl that takes a 4-byte literal argument:

```
B0: shll $2, %eax
    xorl %ecx,
    next 4 bytes // used to be jmp
    ret

B1: decl %eax
    jmp I2
    nop
call %eax
```

The xorl instruction has, embedded in its immediate operand, the four bytes from decl;jmp;nop:

```
48 C1 E0 02 40 C3 0 1 2 3 4
```

The xorl instruction is computed without reading the code! When one block executes it also computes a hash over the second block!

To maintain semantics:

```
ret
    shll 2, %eax
    jmp %eax

ret
    shll 2, %eax
    jmp %eax
```

IDEA: overlap basic blocks of x86 instructions.
Remote Tampering

Algorithm TPZGC: Move client code server-side

 Overhead: up to 3x slowdown.
int f(int x, int y) {
    int a = 4*x + y;
    int c;
    if (y < 5)
        c = a*x + 4;
    else
        c = 2*x + 4;
    int sum = 0;
    for (int i = a; i < 10; i++)
        sum += i;
    return x*(sum + c);
}

int client(int x, int y) {
    f1(x, y);
    int c;
    if (!f2(y, x))
        c = 2*x + 4; f3(c);
    int sum = 0; f4(sum);
    f5();
    return x*f6();
}

int Ha = 5; int Hc = 0; int Hsum = 0;
void f1(int x, int y) {
    Ha = 4*x + y;
}

boolean f2(int y, int x) {
    if (y < 5)
        Hc = Ha*x + 4;
    return true;
}

void f3(int c) {
    Hc = c;
}

void f4(int sum) {
    Hsum = sum;
}

void f5() {
    for (int i = Ha; i < 10; i++)
        Hsum += i;
}

int f6() {
    return Hsum + Hc;
}
Example Function $f$ is the original one. You want to hide variable $a$.

Compute a forward slice on $a$ (pink).

You want to protect all the pink code $\Rightarrow$ put it on the server in functions $H_{f1}$, $\ldots$, $H_{f6}$.

The client accesses the hidden functions by making RPCs.

$c$ is a partially hidden variable. It resides both on the client and the server, but the code that updates it is split between them.

The client accesses the hidden functions by making RPCs.

### Algorithm TPZG: Performance

- **Runtime overhead from 3 to 58%**
- **Depends on the amount of protection that is added**
- **Can I trust my program when it’s running on an untrusted site?**

#### Trustworthiness

- Tamperproofing is about **trustworthiness**.
- For us to trust $P$, the adversary cannot modify $P$'s environment.
- For us to trust $P$, the adversary cannot modify $P$'s code.
- For us to trust $P$, the adversary cannot add/remove/change $P$'s code.
- Can I trust my program when it's running on an untrusted site?

#### Discussion

**Trustworthiness**

- Tamperproofing is about trustworthiness:
  - Can I trust my program when it's running on an untrusted site?

<table>
<thead>
<tr>
<th>Target site</th>
<th>Ms</th>
<th># Hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>cse.asu.edu</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td><a href="http://www.stanford.edu">www.stanford.edu</a></td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td><a href="http://www.usp.ac.fj">www.usp.ac.fj</a></td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td><a href="http://www.eltech.ru">www.eltech.ru</a></td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td><a href="http://www.tsinghua.edu.cn">www.tsinghua.edu.cn</a></td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>cse, asu.edu</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>rotorito.cs.arizona.edu</td>
<td>0.2</td>
<td>1</td>
</tr>
</tbody>
</table>

Packet transmission times:

- **$\approx 0.2$ s**
- **$\approx 5$ s**
- **$\approx 153$ s**
- **$\approx 201$ s**
- **$\approx 209$ s**

### Algorithms TPZG: Example

- Compute a forward slice on a (pink).
- You want to hide variable $a$.
- You want to hide variable $a$.
Basic operations

- Check P's environment:
  - Am I running under a debugger?
  - Am I running under emulation?
  - Has the OS been hacked?

- Check P's code:
  - Have the executable bits been changed?
  - Is P in a legal executable state?

- Check P's dynamic data:
  - Simple attack: trace back from failure!
  - Simple attack: detect reads of executable pages!

In practice...

- Use a combination of operations!
- Check the environment
- Check the code
- Check the state
- You must check the checking code!
- Simple attack: remove the checkers!
- The response must be stealthy!

You must check the checking code!

Simple attack: remove the checkers!

The response must be stealthy!

In practice...

Use a combination of operations!

Check the environment
Check the code
Check the state
You must check the checking code!

Simple attack: remove the checkers!

The response must be stealthy!

You must check the checking code!

Simple attack: remove the checkers!

The response must be stealthy!

Watermarking

- Embed a unique identifier into the executable of a program.
- A watermark is much like a copyright notice.
- Won't prevent an attacker from reverse engineering or pirating it the program.
- Allows us to show that the program the attacker claims to be his, is actually ours.
- Software fingerprinting: every copy you sell will have a different unique mark in it.
- Trace the copy back to the original owner, and take legal action.

Watermarking

Embed a unique identifier into the executable of a program.

A watermark is much like a copyright notice.

Won't prevent an attacker from reverse engineering or pirating it the program.

Allows us to show that the program the attacker claims to be his, is actually ours.

Software fingerprinting: every copy you sell will have a different unique mark in it.

Trace the copy back to the original owner, and take legal action.

Watermarking

Embed a unique identifier into the executable of a program.

A watermark is much like a copyright notice.

Won't prevent an attacker from reverse engineering or pirating it the program.

Allows us to show that the program the attacker claims to be his, is actually ours.

Software fingerprinting: every copy you sell will have a different unique mark in it.

Trace the copy back to the original owner, and take legal action.
Audio marking: Echo hiding

Embed echoes that are short enough to be imperceptible to the human ear.

Audio: Least Significant Bit

Alter without adversely affecting quality!

- Attack: randomly replace the least significant bit of every sample.

Image: Patchwork

Embed a single bit by manipulating the brightness of pixels.

Meta-data mark (visible, fragile)

Authorship mark ((in)visible, robust)

Licensing mark (invisible, robust)

Fingerprint mark (invisible, fragile)

Filtering mark Secret mark

Cute kitten in window in Venice

PG-13

MD5(kitten.jpg)

attack mice at dawn

0xc6ba8f25d2dfc44cf518d7f327c8e83f

© 2006 Collberg

Audio marking: Echo hiding

During embedding adjust the brightness of a up by a small amount, and b down by the same small amount. Use a pseudo-random number sequence to trace out parts of pixels. Embed a single bit by manipulating the brightness of pixels.
Watermarking Text: Syntax

I saw the best minds of my generation, starving hysterical naked.

Watermarking Text: Synonym replacement

I observed the choice intellects of my generation, famished hysterical nude.

Watermarking Text: Formatted ASCII

I saw the best minds of my generation, starving hysterical naked.

Watermarking Text: PDF
Watermarking Software

Ideas for Software Watermark Algorithms

- Encode the watermark in a permutation of a language structure
- In an embedded media object
- In a statistical property of the program
- As a solution to a static analysis problem
- In the topology of a CFG

You care about
- Encoding bitrate
- Stealth
- Resilience to attack

Static watermarks

1. Static watermarks
2. Embed
3. Extract

Dynamic watermarks

1. Dynamic watermarks
2. Embed
3. Extract

Dynamic marks appear more robust, but are more cumbersome to use.
Attacks against software watermarks

- Embed
- Static
- Extract
- Key

The adversary knows the algorithm.
The adversary has complete access to the program.
The adversary doesn't know the key.
The adversary doesn't know the embedding location (it's key-dependent).

1. **Rewrite attack**
   - Alice has to assume that Bob will try to destroy her marks before trying to resell the program!
   - One attack will always succeed.

2. **Additive attack**
   - An additive attack can help Bob to cast doubt in court as to whose watermark is the original.
   - Bob can also add his own watermarks to the program.

3. **Distortive attack**
   - A distortive attack applies semantics-preserving transformations to try to disturb Alice's recognizer:
     - Transformations: code optimizations, obfuscations, ...
   - Ideally, this is the only effective attack.

Attacks against software watermarks.
## Watermarking Algorithms

**Attacks — Collusive attack**

- Bob buys two differently marked copies and comparing them to discover the location of the fingerprint.

**Algorithm wmDM: Reordering Basic Blocks**

- Alice should apply a different set of obfuscations to each distributed copy, so that comparing two copies of the same program will yield little information.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Flag</th>
<th>Performance overhead of 0.11% for three standard high-performance computing benchmarks.</th>
<th>Negligible slowdown for a set of Java benchmarks.</th>
<th>What about stealth?</th>
</tr>
</thead>
<tbody>
<tr>
<td>wmDM</td>
<td>$\log_2(\sqrt{2\pi m/(m/e)^m}) = O(m \log m)$</td>
<td>Watermarking bits.</td>
<td>If you have $m$ items to reorder you can encode $\log_2(\sqrt{2\pi m/(m/e)^m})$ watermarking bits.</td>
<td></td>
</tr>
</tbody>
</table>
**Algorithm wmMC**: Media watermark

```java
void main() {
    Image tudou = Bob uses Stirmark to destroy marks in embedded images!
    Alice encodes part of the program into the image!
}
```

**Algorithm wmVVS**: Watermarks in CFGs

**Basic idea:**
1. Embed the watermark in the CFG of a function.
2. Tie the CFG tightly to the rest of the program.

**Issues:**
1. How do you encode a number in a CFG?
2. How do you find the watermark CFG?
3. How do you attach the watermark CFG to the rest of the program?

**Embedding**: Generate a stealthy watermark CFG:
1. basic blocks have out-degree of one or two
2. basic blocks have out-degree of one or two
3. basic blocks have out-degree of one or two
4. basic blocks have out-degree of one or two
5. basic blocks have out-degree of one or two

Reducible permutation graphs (RPGs)

```
Reducible Permutation Graphs (RPGs)
```

```
void main() {
    Image tudou = Bob uses Stirmark to destroy marks in embedded images!
    Alice encodes part of the program into the image!
}
```
The embedder can split the watermarking into pieces, for example, encode each one into an integer watermark if required. Then the embedder constructs CFGs for the watermark functions. It assumes that any function with more than a certain percentage of blocks marked is a watermark function. For every cover program block, it marks the basic blocks:

Idea:

1. Mark the basic blocks.
2. Assume that any function with more than \( t \% \) blocks marked is a watermark function.
3. Compute the mark value for each basic block in the program.
4. Construct CFGs for the watermark functions.
5. Decode each one into an integer watermark.

Recognition procedure:

A 0 for every cover program block, a 1 for every watermark block.

Recognition:

1. Compute the mark value for each basic block in the program.
2. Assume that any function with more than \( t \% \) blocks marked is a watermark function.
3. Construct CFGs for the watermark functions.
4. Decode each one into an integer watermark.

The embedder can split the watermarking into pieces, for example, encode each one into an integer watermark if required. Then the embedder constructs CFGs for the watermark functions. It assumes that any function with more than a certain percentage of blocks marked is a watermark function. For every cover program block, it marks the basic blocks:

Idea:

1. Mark the basic blocks.
2. Assume that any function with more than \( t \% \) blocks marked is a watermark function.
3. Compute the mark value for each basic block in the program.
4. Construct CFGs for the watermark functions.
5. Decode each one into an integer watermark.

Recognition procedure:

A 0 for every cover program block, a 1 for every watermark block.

Recognition:

1. Compute the mark value for each basic block in the program.
2. Assume that any function with more than \( t \% \) blocks marked is a watermark function.
3. Construct CFGs for the watermark functions.
4. Decode each one into an integer watermark.

The embedder can split the watermarking into pieces, for example, encode each one into an integer watermark if required. Then the embedder constructs CFGs for the watermark functions. It assumes that any function with more than a certain percentage of blocks marked is a watermark function. For every cover program block, it marks the basic blocks:

Idea:

1. Mark the basic blocks.
2. Assume that any function with more than \( t \% \) blocks marked is a watermark function.
3. Compute the mark value for each basic block in the program.
4. Construct CFGs for the watermark functions.
5. Decode each one into an integer watermark.
Watermark Embeddings

Watermarks are short identifiers difficult to locate hard to destroy
The adversary knows that the object is marked knows the algorithm used doesn't know the key is active
You care about data-rate stealth resilience

Steganographic Embeddings

Stegomarks are long identifiers difficult to locate
The adversary wants to know if the object is marked
You care about data-rate stealth

Steganography — Prisoners' Problem

Master is soon, dear! So many flowers! Can you smell them?
Are you cold at night?
Prison food stinks!
Eat well, still!
Are you lonely?
The prison cat is cute!
Don't worry! Don't worry! Don't worry!

HAPPY EASTER DEAR!
Basic idea:
Play compiler!
Whenever the compiler has a choice in which code to generate, or the order in which to generate it, pick the choice that embeds the next bits from the message $W$.

Four sources of ambiguity:
1. Code layout (ordering of chains of basic blocks)
2. Instruction scheduling (instruction order within basic blocks)
3. Register allocation
4. Instruction selection

Construct:
- Codebook $B$ of equivalent instruction sequences
- Statistical model $M$ of real code
- Encrypt with key $K$.

Sequence:
- Replace each instruction with the first alternative from $B$.
- Order instructions in each block in standard order.
- Sort block chains, procedures, modules.
- Encrypt with $K$.
- Construct a statistical model $M$ of real code.

Code Layout: Embed bits from $W$ by reordering code segments within the executable.
- Most don't occur in real code.

Instruction Scheduling:
- Build dependency graph.
- Generate all valid instruction schedules.
- Pick from $W$ by picking a schedule.
- Use $M$ to avoid unusual schedules.

Instruction Selection:
- Use $B$ to avoid unusual instruction sequences.
- Use $B$ to embed bits from $W$ by replacing instructions.
- Use $M$ to embed bits from $W$ by replacing instructions.

Stealth:
- Instruction selection:
  - There are 3078 different encodings of three instructions for $EAX=(EAX/2)$.
  - Most don't occur in real code.
- Instruction scheduling:
  - Avoid bad schedules: no compiler would generate it.
  - Avoid generating different schedules for two blocks with the same dependency graph.
- Code layout:
  - Compilers lay out code for locality: don't deviate too much.
WMSB: Stealth

- Encoding rate
  - Unstealthy code: \( \frac{1}{27} \)
  - Stealthy: \( \frac{1}{89} \)
- Encoding space:
  - 58% from code layout
  - 25% from instruction scheduling
  - 17% from instruction selection
- Real code doesn’t use unusual instruction sequences.
- Real code contains many schedules for the same dependency graph

Wanna design a watermarking algorithm?

- Find a language structure into which to encode the mark (CFGs, threads, dynamic control flow...)
- Construct an encoder/decoder (number \( \leftrightarrow \) CFG, ...)
- Construct a tracer/locator to find locations for the mark (using key, every function, ...)
- Construct a embedder/extractor to tie the mark to surrounding code
- Decide on an attack model.

\( \langle \text{language structure}, \text{encoder/decoder}, \text{tracer/locator}, \text{embedder/extractor}, \text{attacker/protector} \rangle \)

Discussion

- Many important man-at-the-end security scenarios:
  - protecting medical records
  - preventing software piracy
  - ensuring the fairness of networked games
  - tracing virus writers
  - protecting military secrets
  . . . . .
Techniques for protecting against man-at-the-end attacks:

1. Obfuscation
2. Tamperproofing
3. Watermarking
4. Birthmarking
5. Hardware

Much research left to be done:

- What are the theoretical limits on obfuscation?
- Develop new and better algorithms!
- Develop ways to measure the goodness of algorithms!
- Build useful tools for experimentation!
- Develop new and better algorithms!