Defenses against malicious host attacks

Jeff Stuckman
Defenses against malicious host attacks

• Watermarking
  – Embed a watermark in a program with a secret key

• Obfuscation
  – Protect a program from reverse engineering

• Tamper-proofing
  – Embed a secret within a program, and cause the program to fail if the secret is removed
• [Link](http://books.google.com/books?id=F0aliCz5_kC&lpg=PA19&ots=mUcjCdkenB&pg=PA19#v=onepage&q&f=false)
Software protection dongles
Properties of a successful obfuscation

- Given a set of obfuscating transformations $T$ and a program $P$ consisting of source code artifacts $S$...
- $P'$ has same observable behavior as $P$ (semantics-preserving)
- Obscurity of $P'$ maximized
- Resilience of $T$ maximized
- Stealth of $T$ maximized
- Penalty of $P'$ in execution time/space (cost) is minimized
Opaque predicates

• Some situations are problematic for static analysis
  – Flow-insensitive may-Alias Analysis – determining which pairs of expressions may refer to the same location – is NP-hard (Horwitz, 1997)

• When static analyzers fail, removing watermarks and reversing obfuscation becomes more difficult
Control transformation obfuscation through opaque predicate insertion

Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection
Obfuscation through objects and aliases

Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs
Obfuscation through data transformation

<table>
<thead>
<tr>
<th>g(V)</th>
<th>f(p,q)</th>
<th>2p + q</th>
<th>AND[A,B]</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>q</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>False</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>True</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>True</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>False</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

(a)

(b)

(1) bool A, B, C;
(2) B = False;
(3) C = False;
(4) C = A & B;
(5) C = A & B;
(6) if (A) …;
(7) if (B) …;

(1') short a1, a2, b1, b2, c1, c2;
(2') b1=0; b2=0;
(3') c1=1; c2=1;
(4') x=AND[2*a1+a2, 2*b1+b2]; c1=x/2; c2=x&b2;
(5') c1=(a1 ^ a2) & (b1 ^ b2); c2=0;
(5') x=2*a1+a2; if ((x==1) || (x==2)) …;
(7') if (b1 ^ b2) …;

(c)

Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection
Static and Dynamic Watermarking

• **Static stealth:** Static analysis detects no statistical difference between original and marked program.
• **Dynamic stealth:** Execution traces reveal no differences between programs.
• **Static watermark:** Stored and extracted from the executable.
  – Easy to defeat through distortion or obfuscation.
• **Dynamic watermark:** Watermark only constructed at runtime.
• **Similarities with steganography**
Dynamic Watermarking

• A predetermined, highly unusual input sequence allows the watermark to be read from the program’s state

• Execution trace watermarks
  – Watermark is extracted from the branching structure or variable state of the program

• Data structure watermarks
  – Watermark is extracted from a special-purpose data structure built over the program’s execution
  – Opaque predicates disguise the purpose of the data structure
Requirements for watermarks

• Functionality Preservation
  – Semantics of the stegoprogram should be identical to those of the subject program

• Performance Preservation
  – Performance of the subject program should not be significantly degraded

• Universality Preservation
  – If the subject program is universal, no special hardware should be required for executing the stegoprogram

(An Abstract Interpretation-Based Framework for Software Watermarking)
Requirements for watermarks

• Unbounded signature size

• Credibility and False Recognition
  – Extraction of signature from programs without a signature should fail

• Secrecy
  – Watermark should not be detectable by an average observer

• Impossible or computationally hard to embed or extract the watermark without the secret stegokey

(An Abstract Interpretation-Based Framework for Software Watermarking)
Requirements for watermarks

- Robustness/Perennility
  - Watermarks should be hard to remove without damaging the program
- Multiple watermarks
  - Marking with a new signature should not delete previous signatures
Requirements for watermarks

• Low computational complexity to embed or extract

• Resistance to counterfeiting
  – Watermarking should resist program transformations that do not change observable semantics of program

(An Abstract Interpretation-Based Framework for Software Watermarking)
Attacks against watermarks

• Subtractive attacks
  – Remove the code containing the signatures

• Distortive attacks
  – Ensure the signatures cannot be extracted

• Additive attacks
  – Add watermark with new signature

• Collusive attack
  – Removal of signatures based on comparison of program versions

(An Abstract Interpretation-Based Framework for Software Watermarking)
Embedding watermarks

• Semantics-preserving code reordering
• Inserting new code that encodes a watermark number
• Manipulating instruction frequencies
A Formal Model of Software Watermarking

Definition 1 (Software Watermark) Let \( \mathcal{W} \) be a set of mathematical structures, and \( p \) a predicate such that \( \forall w \in \mathcal{W} : p(w) \). We choose \( p \) and \( \mathcal{W} \) such that the probability of \( p(x) \) for a random \( x \not\in \mathcal{W} \) is small.

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

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

Let \( S(P, I) \) be the internal state of program \( P \) (drawn from a set of states \( \mathcal{S} \)) after having processed input \( I \). Let \( |S(P, I)| \) be the size of this state, in accessible words.
The resilience of a watermark $w$ in a program $P_w$ will be defined in terms of the de-watermarking attacks that can be launched against $P_w$. Attacks are *program transformations* that can be *semantics-preserving* (if they preserve input-output behavior), *state-preserving* (if internal state is preserved), or *cropping* (if input-output behavior is not preserved).

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

$T_{\text{sem}} \subset \mathbb{T}$ is the set of *semantics-preserving* transformations, $T_{\text{stat}} \subset \mathbb{T}$ is the set of *state-preserving* transformations, and $T_{\text{crop}} \subset \mathbb{T}$ is the set of transformations which do not preserve semantics:

$$T_{\text{sem}} = \{ t : \mathbb{T} \mid P \in \mathbb{P}, I \in \text{dom}(P), \quad \text{dom}(P) = \text{dom}(t(P)), \quad \text{out}(P, I) = \text{out}(t(P), I) \}.$$  

$$T_{\text{stat}} = \{ t : \mathbb{T} \mid P \in \mathbb{P}, I \in \text{dom}(P), \quad S(P, I) = S(t(P), I) \}.$$  

$$T_{\text{crop}} = \{ t : \mathbb{T} \mid \exists P \in \mathbb{P}, \exists I \in \text{dom}(P), \quad (I \notin \text{dom}(t(P))) \lor \text{out}(P, I) \neq \text{out}(t(P), I) \}.$$  

State-preservation implies semantics-preservation but many transformations (such as code optimizing transformations) will preserve semantics but not state, i.e. $T_{\text{stat}} \subset T_{\text{sem}}$. 

\[\square\]
**Definition 4 (Watermark Recognition)** A watermark \( w \in \mathcal{W} \) in a program \( P_w \in \mathcal{P} \) is recognizable wrt a set of transformations \( T \subseteq \mathcal{T} \) if there exists a recognizer

\[
\mathcal{R}_T : (\mathcal{P} \times \mathcal{S}) \to \mathcal{W}
\]

and an input \( I \) such that

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

This definition allows us to distinguish several useful subclasses of recognizers:

- \( \mathcal{R}_\emptyset(P_w, S(P_w, I)) \) is a **trivial** recognizer that is not guaranteed to recognize \( w \) if any transformations have been performed on \( P_w \).

- \( \mathcal{R}_{\text{sem}}(P_w, S(P_w, I)) \) is a **strong** recognizer that is resilient to any semantics-preserving transformation.

- \( \mathcal{R}_T(P_w, S(P_w, I)) \) is an **ideal** recognizer that is resilient to any transformation.

- \( \mathcal{R}_T(P_w, \emptyset) \) is a **static** recognizer that can only examine the text of \( P_w \), not its execution state.

- \( \mathcal{R}_T(\emptyset, S(P_w, I)) \) is a **pure dynamic** recognizer that can only examine the execution state of \( P_w \), not its text.
**Definition 5 (Watermark \( r \)-Space Resilience)** A watermark \( w \in \mathbb{W} \) in a program program \( P_w \) is \( r \)-space resilient wrt a set of transformations \( T \subset \mathbb{T} \) if there exists a recognizer \( \mathcal{R}_T \) and an input \( I \) such that

\[
\forall t \in T : (p(\mathcal{R}_T(t(P_w), S(t(P_w), I)))) \neq p(w) \Rightarrow \\
\frac{|S(t(P_w), I)|}{|S(P_w, I)|} \geq r
\]

Note that any 1-space resilient watermark has a strong recognizer. However, if \( r > 1 \), our parameter \( r \) is a measure of the weakness of the watermarking system.

Some watermarks that are written in the program text or static data are susceptible to attacks that increase the static size of the code:
Definition 6 (Watermark r-Size Resilience) A watermark \( w \in \mathbb{W} \) in a program program \( P_w \) is r-size resilient wrt a set of transformations \( T \subseteq \mathbb{T} \) if there exists a recognizer \( R_T \) and an input \( I \) such that

\[
\forall t \in T : (p(R_T(t(P_w), S(t(P_w), I))) \neq p(w)) \Rightarrow
\]

\[
\frac{|t(P_w)|}{|P_w|} \geq r
\]

\[\square\]

Finally, many attacks on a watermarked program \( P_w \) will increase its runtime; however if the runtime is increased too much (for at least one input in the domain of \( P \)) then the attack is not particularly worrisome:
Definition 7 (Watermark $r$-Runtime Resilience) A watermark $w \in \mathbb{W}$ in a program $P_w$ is $r$-runtime resilient wrt a set of transformations $T \subseteq \mathbb{T}$ if there exists a recognizer $R_T$ and an input $I$ such that

$$\forall i \in T : (p(R_T(t(P_w), S(P_w, I))) \neq p(w)) \Rightarrow$$

$$\exists i \in \text{dom}(P) \frac{\text{Time}(t(P_w), i)}{\text{Time}(P_w, i)} \geq r$$

□
Dynamic graph watermarking

- Pointer-aliasing is hard to statically analyze, so obfuscators cannot safely tamper with the watermark
- Embed the signature in the topology of a graph

Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection
Path-based watermarking -- Embedding watermarks in Java code

- Insert “useless” code that encodes the watermark in its branching structure when run
- Profile code when run against a secret input
- Add loops that create the desired branch structure when executed

```java
int bits = 0xa;
int counter = 5;
for (int i = 0; i < counter; i++, bits >>= 1)
    if ((bits & 1) == 1)
        bits |= 1;
```

Dynamic Path-Based Software Watermarking
Embedding watermarks in Java code

- Add branches with known outcomes based on the profiling of the program

```java
int tmp = 0;
if(c == d)
    tmp++;
if(a == b)
    tmp++;
if(c == d)
    tmp++;
if(a == b)
    tmp++;
```
Embedding watermarks in machine code

- Implement an obfuscated branch function by using perfect hashing and XORing the return address on the stack
- Convert an unconditional jump into a series of forward and backward branch function calls to encode each bit
Splitting a watermark signature

• Under some schemes, a signature must be extracted from the program or its execution
• In some watermark implementations, the signature must be split into smaller chunks for successful embedding
• Choose pairwise coprime positive integers \( n_1, n_2, \ldots, n_l \) such that the maximum value is less than \( n_1 \cdot n_2 \cdot n_3 \cdot \ldots \cdot n_l \)
• Split the signature \( c \) into constants (chunks) \( c_1, c_2, \ldots, c_l \) by computing chunks \( c_i = c \mod n_i \)
• Signature can be reconstructed per the Chinese Remainder Theorem

(An Abstract Interpretation-Based Framework for Software Watermarking)
The Chinese Remainder Theorem

- $C = 57$
- $C \equiv 0 \mod 3$
- $C \equiv 2 \mod 5$
- $C \equiv 2 \mod 11$
- $11 \cdot 5 \cdot 3 = 165 \geq 57$
- Ensures that chunks of encoded signature are recoverable
The chunk enumeration algorithm

Each statement \( W \equiv x_k \mod p_{i_k} p_{j_k} \) is turned into a single integer by an enumeration scheme. In our scheme,

\[
\omega_k = x_k + \sum_{n=1}^{i_k-1} \sum_{m=n+1}^{r} p_n p_m + \sum_{m=1}^{j_k-1} p_{i_k} p_m.
\]
Embedding and recognition

• To convert a trace into a bitstring, generate a 0 when the outcome of a branch matches the first time it was encountered; 1 otherwise

• Split the signature into 64-bit chunks, encrypt them, and embed them

• Extract signature chunks from the bitstring, decrypt them, and identify signature chunks that are consistent with the others
Reconstructing the signature

Let $V = \{v_0, v_1, \ldots, v_m\}$ be the set of statements on $W$ we are given. In step 5, we construct two graphs, $G$ and $H$, on $V$. (The figure only shows $G$.) Two vertices are adjacent in $G$ iff they are inconsistent. Two vertices are adjacent in $H$ iff they are consistent because the $x$’s agree mod $p_i$ for some $i$, not if they are consistent by the Chinese Remainder Theorem. For step 5, we initialize $U := \emptyset$ and repeat the following steps until $G$ is a coclique:

1. Let $v$ be some vertex in the set $V - U$ of maximum degree in $H$. This vertex is presumed to be a true statement about $W$.


Once $G$ is a coclique, we have a set of statements about $W$ that are consistent and can be combined by the Generalized Chinese Remainder Theorem [8] in step 5.