A SEMANTICS-BASED APPROACH TO CODE OBFUSCATION

Mila Dalla Preda
University of Verona, Italy

19 December 2006, RE-TRUST meeting
Code Obfuscation

SW

Malicious Reverse Engineering

malicious user
A Semantics-Based Approach to Code Obfuscation – p.2
Motivation

Drawback of Code Obfuscation: weak theoretical basis
⇒ difficult to formally study and certify effectiveness

Goal: investigate the semantic effects of code obfuscation in order to provide a formal definition of code obfuscation based on program semantics and abstract interpretation
Abstract Interpretation

Design approximate semantics of programs [Cousot & Cousot '77, '79].

\[ \langle C, \alpha, \gamma, A \rangle \]

Galois Connection: \langle C, \alpha, \gamma, A \rangle, A \text{ and } C \text{ are complete lattices.}

\[ \langle \text{Abs}(C), \sqsubseteq \rangle \text{ set of all possible abstract domains, } \]
\[ A_1 \sqsubseteq A_2 \text{ if } A_1 \text{ is more concrete than } A_2 \]
\textit{Sign} is an abstraction of $\wp(\mathbb{Z})$:
Sign is an abstraction of $\varphi(\mathbb{Z})$: 

\[ \varphi(\mathbb{Z}) \]

\[
\begin{align*}
\ldots & \quad 0 - \\
{1, -1} & \\
{1, -3, -4} & \quad \ldots \quad {2, 3, 5} \\
\ldots & \\
0 & 1 \quad \ldots \\
\emptyset & \\
\end{align*}
\]
Compare Abstractions

\[ \varnothing(\mathbb{Z}) \]

Sign

\[ \mathbb{Z} \]

Sign\(^+\)

Sign is more abstract than Sign\(^+\): Sign\(^+\) \(\sqsubseteq\) Sign
Let \( \langle A, \alpha, \gamma, C \rangle \) be a GC, \( f : C \to C \) and \( f^\# : A \to A \), then:

\[
\top \xrightarrow{f} f(x) \xrightarrow{\alpha} \alpha(f(x)) \xrightarrow{f^\#} f^\#(\alpha(x)) \xrightarrow{\top} \top
\]

Soundness: \( \alpha \circ f(x) \leq^A f^\# \circ \alpha(x) \)
Let \( \langle A, \alpha, \gamma, C \rangle \) be a GC, \( f : C \rightarrow C \) and \( f^\#: A \rightarrow A \), then:

\[
\gamma(f^\#(x)) \\
\gamma(f(\gamma(x))) \\
\gamma(x) \\
\bot \\
C
\]

\[
\gamma(f^\#(x)) \\
\gamma(f(\gamma(x))) \\
\gamma(x) \\
\bot \\
A
\]

\[
\text{Soundness: } \alpha \circ f(x) \leq_A f^\# \circ \alpha(x) \\
\text{Soundness: } f \circ \gamma(x) \leq_C \gamma \circ f(x)
\]

\[
\text{Best Correct Approximation: } \alpha \circ f \circ \gamma
\]
Backward-Completeness

Let \( \langle A, \alpha, \gamma, C \rangle \) be a GC, \( f : C \to C \) and \( f^\#: A \to A \), then:

\[
\alpha(f(x)) = f^\#(\alpha(x))
\]

\( B \)-Completeness: \( \alpha \circ f(x) = f^\# \circ \alpha(x) \)
Forward-Completeness

Let \( \langle A, \alpha, \gamma, C \rangle \) be a GC, \( f : C \rightarrow C \) and \( f^\# : A \rightarrow A \), then:

\[
f(\gamma(x)) = \gamma(f^\#(x))
\]

\[B\text{-Completeness: } \alpha \circ f(x) = f^\# \circ \alpha(x)\]

\[\mathcal{F}\text{-Completeness: } f \circ \gamma(x) = \gamma \circ f^\#(x)\]
Forward-Completeness

Let \( \langle A, \alpha, \gamma, C \rangle \) be a GC, \( f : C \rightarrow C \) and \( f^\#: A \rightarrow A \), then:

\[
f(\gamma(x)) = \gamma(f^#(x))
\]

B-Completeness: \( \alpha \circ f(x) = f^\# \circ \alpha(x) \)

\( F \)-Completeness: \( f \circ \gamma(x) = \gamma \circ f^#(x) \)

How to get complete domains? Completeness domain refinement

[Giacobazzi et al. 2000]
Example

\[ \text{Sign}(10) \oplus \text{Sign}(-5) = 0 + \oplus 0- = \mathbb{Z} \]

\[ \text{Sign}(10 + (-5)) = \text{Sign}(5) = 0+ \]

\[ \text{Sign} \text{ is not } B\text{-complete for addition} \]
Example

\[ \text{Parity}(10) \oplus \text{Parity}(-5) = \text{even} \oplus \text{odd} = \text{odd} \]

\[ \text{Parity}(10 + (-5)) = \text{Parity}(5) = \text{odd} \]

\( \text{Parity} \) is \( \mathcal{B} \)-complete for addition
Syntactic transformation: $T = p \circ t \circ S$
$T : P \rightarrow P$ is a code obfuscation if:

1. $T$ is potent, i.e. $T[P]$ is more complex than $P$

2. $T$ preserves the observational behaviour of programs, i.e. the input-output behaviour (denotational semantics $DenSem$)

[C. Collberg et al. '97, '98]
**Code Obfuscation**

[C. Collberg et al. '97, '98]

\[ T : P \rightarrow P \] is a code obfuscation if:

1. **T** is potent, i.e. \( T[P] \) is more complex than \( P \)

2. **T** preserves the observational behaviour of programs, i.e. the input-output behaviour (denotational semantics *DenSem*)

AI describes the relation among semantics at different levels of abstraction:

Hierarchy of semantics defined by [Cousot '00]:

\[ S \llbracket P \rrbracket \] trace semantics
\[ D \] Denotational semantics
T : P → P is potent if there is a property ϕ ∈ Abs(Sem) such that:

ϕ(S[P]) ≠ ϕ(S[T[P]])
$T : P \rightarrow P$ is potent if there is a property $\varphi \in Abs(Sem)$ such that:

$$\varphi(S[P]) \neq \varphi(S[T[P]])$$

The most concrete property preserved by $T$ is:

$$\delta_T = \sqcap \left\{ \varphi \in Abs(Sem) \left| \varphi(S[P]) = \varphi(S[T[P]]) \right. \right\}$$
**Code Obfuscation and Program Semantics**

[M. Dalla Preda and R. Giacobazzi ICALP’05]

$T : P \rightarrow P$ is **potent** if there is a property $\varphi \in \text{Abs}(\text{Sem})$ such that:

$$\varphi(S[P]) \neq \varphi(S[T[P]])$$

The most concrete property preserved by $T$ is:

$$\delta_T = \sqcap \left\{ \varphi \in \text{Abs}(\text{Sem}) \mid \varphi(S[P]) = \varphi(S[T[P]]) \right\}$$

$$O_{\delta_T} = \left\{ \varphi \mid \varphi \ominus (\varphi \sqcup \delta_T) \neq T \right\}$$

$$O_{\delta_T} = \left\{ \varphi \mid \delta_T \not\sqsubseteq \varphi \right\}$$
Semantic Code Obfuscation

[M. Dalla Preda and R. Giacobazzi ICALP’05]
Semantic Code Obfuscation

[M. Dalla Preda and R. Giacobazzi ICALP’05]

Semantics-based Definition

\[ T : \mathcal{P} \to \mathcal{P} \text{ is a } \delta\text{-obfuscator if:} \]
\[ \delta_T = \delta \text{ and } O_\delta \neq \emptyset \]
Semantic Code Obfuscation

Semantics-based Definition

\[ T : \mathcal{P} \rightarrow \mathcal{P} \text{ is a } \delta-\text{obfuscator if:} \]
\[ \delta_T = \delta \text{ and } O_\delta \neq \emptyset \]

EXAMPLE: \( X \mapsto 2X \) obfuscates the parity and preserves the sign of \( X \)
Semantic Code Obfuscation

[M. Dalla Preda and R. Giacobazzi ICALP’05]

Semantics-based Definition

\[ T : \mathbb{P} \rightarrow \mathbb{P} \text{ is a } \delta-\text{obfuscator if:} \]

\[ \delta_T = \delta \text{ and } O_\delta \neq \emptyset \]

EXAMPLE: \( X \mapsto 2X \) obfuscates the parity and preserves the sign of \( X \)

Collberg’s Obfuscators = \( \{ \delta \text{ – obfuscators} \mid \delta \subseteq \text{DenSem} \} \)
Semantic Code Obfuscation

[ M. Dalla Preda and R. Giacobazzi ICALP’05 ]

Semantics-based Definition

\[ T : \mathcal{P} \rightarrow \mathcal{P} \text{ is a } \delta - \text{obfuscator if:} \]
\[ \delta_T = \delta \text{ and } O_\delta \neq \emptyset \]

**EXAMPLE:** \( X \rightsquigarrow 2X \) obfuscates the parity and preserves the sign of \( X \)

Collberg’s Obfuscators = \( \{ \delta - \text{obfuscators} \mid \delta \subseteq \text{DenSem} \} \)

Compare obfuscators w.r.t. potency: \( T_1 \) is more potent than \( T_2 \) iff \( \delta_2 \subseteq \delta_1 \)
Semantic Code Obfuscation

[M. Dalla Preda and R. Giacobazzi ICALP’05]

Semantics-based Definition

\[ T : \mathbb{P} \rightarrow \mathbb{P} \text{ is a } \delta-\text{obfuscator if:} \]
\[ \delta_T = \delta \text{ and } O_\delta \neq \emptyset \]

EXAMPLE: \( X \leadsto 2X \) obfuscates the parity and preserves the sign of \( X \)

Collberg’s Obfuscators = \( \{ \delta-\text{obfuscators} \mid \delta \subseteq \text{DenSem} \} \)

Compare obfuscators w.r.t. potency: \( T_1 \) is more potent than \( T_2 \) iff \( \delta_2 \subseteq \delta_1 \)

Constructive characterization of \( \delta_T \)
Control Code Obfuscation affects the control flow of the program often by inserting opaque predicates.

\[
\begin{align*}
P & \quad \Downarrow \quad C_1; C_2; \ldots; C_i \\
& \quad \Downarrow \quad C_{i+1}; \ldots; C_n \\
& \quad \Downarrow \quad p^T \quad F \\
\end{align*}
\]
Attacks and Completeness

**Precise detection of Opaque Predicates:**

An attacker $\varphi$ breaks an opaque predicate $Pr$ if $\varphi$ recognises the always true value of $Pr$. 
Precise detection of Opaque Predicates:

An attacker $\varphi$ breaks an opaque predicate $Pr$ if $\varphi$ recognises the always true value of $Pr$

$\forall x \in \mathbb{Z} : g(x) = h(x)$

[M. Dalla Preda and R. Giacobazzi SEFM’05]: if attacker $\varphi$ is $\mathcal{F}$-complete for $g$ and $h$, then $\varphi$ breaks the opaque predicate $\forall x \in \mathbb{Z} : g(x) = h(x)$

$\forall x \in \mathbb{Z} : n \ mod \ f(x)$

[M. Dalla Preda and R. Giacobazzi AMAST’06]: if attacker $\varphi$ is $\mathcal{B}$-complete for the elementary functions composing $f$ then $\varphi$ breaks the opaque predicate $\forall x \in \mathbb{Z} : n \ mod \ f(x)$
Precise detection of Opaque Predicates:

An attacker $\varphi$ breaks an opaque predicate $Pr$ if $\varphi$ recognises the always true value of $Pr$

$\forall x \in \mathbb{Z}: g(x) = h(x)$

[M. Dalla Preda and R. Giacobazzi SEFM'05]: if attacker $\varphi$ is $\mathcal{F}$-complete for $g$ and $h$, then $\varphi$ breaks the opaque predicate $\forall x \in \mathbb{Z}: g(x) = h(x)$

$\forall x \in \mathbb{Z}: \text{mod } f(x)$

[M. Dalla Preda and R. Giacobazzi AMAST'06]: if attacker $\varphi$ is $\mathcal{B}$-complete for the elementary functions composing $f$ then $\varphi$ breaks the opaque predicate $\forall x \in \mathbb{Z}: \text{mod } f(x)$

consider $\text{OP}_1$ and $\text{OP}_2$: if $\mathcal{C}_{\text{OP}_1}(\varphi) \subseteq \mathcal{C}_{\text{OP}_2}(\varphi)$, then $\text{OP}_1$ is more resilient than $\text{OP}_2$ in contrasting $\varphi$
Obfuscation and Malware

MALICIOUS TRANSFORMATION

MALWARE
- Input
- Deobfuscation
- signature matching
- Output
- malware detector
A program $P$ is infected by malware $M$, denoted $M \leftrightarrow P$ if (a part) of $P$ execution is similar to that of $M$.
A program $P$ is infected by malware $M$, denoted $M \leftrightarrow P$ if (a part) of $P$ execution is similar to that of $M$:

$$S[M] \subseteq S[P]$$
A program $P$ is infected by malware $M$, denoted $M \leftrightarrow P$ if (a part) of $P$ execution is similar to that of $M$:

$$\exists \text{ restriction } r : S[M] \subseteq \alpha_r(S[P])$$
A program $P$ is infected by malware $M$, denoted $M \rightarrow P$ if (a part) of $P$ execution is similar to that of $M$:

$$\exists \text{ restriction } r : S[M] \subseteq \alpha_r(S[P])$$

**Vanilla Malware** i.e. not obfuscated malware
Obfuscated Malware

[M. Dalla Preda et al. POPL'07]

\[ O : P \rightarrow P \text{ obfuscating transformation} \]

\[ \alpha_O : \text{Sem} \rightarrow A \text{ abstraction} \text{ that discards the details changed by the obfuscation while preserving maliciousness} \]

\[ \exists \text{ restriction } r : \alpha_O(S[M]) \subseteq \alpha_O(\alpha_r(S[P])) \]
Obfuscated Malware

\[ \mathcal{O} : \mathcal{P} \rightarrow \mathcal{P} \text{ obfuscating transformation} \]

\[ \alpha_\mathcal{O} : \text{Sem} \rightarrow A \text{ abstraction that discards the details changed by the obfuscation while preserving maliciousness} \]

\[ \exists \text{ restriction } r : \alpha_\mathcal{O}(S[M]) \subseteq \alpha_\mathcal{O}(\alpha_r(S[P])) \]
Sound vs Complete

Precision of the SMD depends on the choice of $\alpha_0$
Precision of the SMD depends on the choice of $\alpha$.

A SMD on $\alpha$ is **complete** w.r.t. a set $\mathcal{O}$ of transformations if $\forall \mathcal{O} \in \mathcal{O}$:

$$\mathcal{O}(M) \rightarrow P \Rightarrow \begin{cases} \exists \text{ restriction } r : \\ \alpha_{\mathcal{O}}(S[M]) \subseteq \alpha_{\mathcal{O}}(\alpha_r(S[P])) \end{cases}$$

always detects programs that are infected (no false negatives).
Sound vs Complete

Precision of the SMD depends on the choice of $\alpha_\mathcal{O}$

A SMD on $\alpha_\mathcal{O}$ is **complete** w.r.t. a set $\mathcal{O}$ of transformations if $\forall \mathcal{O} \in \mathcal{O}$:

$$\mathcal{O}(M) \leadsto P \Rightarrow \begin{cases} \exists \text{ restriction } r : \\ \alpha_\mathcal{O}(S[M]) \subseteq \alpha_\mathcal{O}(\alpha_r(S[P])) \end{cases}$$

always detects programs that are infected (no false negatives)

A SMD on $\alpha_\mathcal{O}$ is **sound** w.r.t. a set $\mathcal{O}$ of transformations if:

$$\exists \text{ restriction } r : \\
\alpha_\mathcal{O}(S[M]) \subseteq \alpha_\mathcal{O}(\alpha_r(S[P])) \Rightarrow \exists \mathcal{O} \in \mathcal{O} : \mathcal{O}(M) \leadsto P$$

never erroneously claims a program is infected (no false positives)
Main Results

[M. Dalla Preda et al. POPL'07]

- Classification of obfuscating transformations w.r.t. their effects on program semantics
  - conservative
  - non-conservative
- Abstraction that is both sound and complete for conservative obfuscation
- Possible strategies in order to handle non-conservative obfuscations
- Flexibility of the abstract interpretation-based approach
- Prove completeness of the semantics-aware malware detector
  [Christodorescu et al. 2005]
Open issues (related to RE-TRUST)

- Metamorphic viruses and monitor factory
- Composition of elementary obfuscations
- The checker may verify the satisfaction of some properties of program behaviour (abstract semantics)
Thank you!
If $\alpha_\mathcal{O}$ is preserved by $\mathcal{O}$ then the SMD on $\alpha_\mathcal{O}$ is complete w.r.t. $\mathcal{O}$:

$$\forall P \in \mathcal{P} : \alpha_\mathcal{O}(S[P]) = \alpha_\mathcal{O}(S[O[P]])$$
If \( \alpha_O \) is preserved by \( O \) then the SMD on \( \alpha_O \) is complete w.r.t. \( O \):

\[
\forall P \in P : \alpha_O(S[P]) = \alpha_O(S[O[P]])
\]

Given an abstraction \( \alpha \), consider the set \( O \) of transformations such that \( \forall P, T \in P : \):

\[
(\alpha(S[T])) \subseteq \alpha(S[P]) \Rightarrow (\exists O \in O : S[O[T]]) \subseteq S[P]
\]

then, the SMD on \( \alpha \) is sound w.r.t. \( O \)
Classifying Transformations

$O : P \rightarrow P$ is a **conservative** transformation if

$\forall \text{trace}_1 \in S[P], \exists \text{trace}_2 \in S[O[P]]: \text{trace}_1 \text{ is sub-sequence of } \text{trace}_2$

$O : P \rightarrow P$ is a **non-conservative** transformation if $O$ is not conservative
Suitable abstraction for conservative transformations:

\[ \alpha_c[X](Y) = X \cap \text{SubSequences}(Y) \]

returns all the traces in \( X \) that are sub-sequences of a trace in \( Y \)

\[ \alpha_c[S[M]](S[O_c[M]]) = S[M] \]

\( O_c(M) \leftrightarrow P \text{ iff } \exists \text{ restriction } r: \alpha_c[S[M]](S[M]) \subseteq \alpha_c[S[M]](\alpha_r(S[P])) \)
Suitable abstraction for conservative transformations:

\[ \alpha_c[X](Y) = X \cap \text{SubSequences}(Y) \]

returns all the traces in \( X \) that are sub-sequences of a trace in \( Y \)

\[ \alpha_c[S[M]](S[O_c[M]]) = S[M] \]

\[ O_c(M) \leftrightarrow P \text{ iff } \exists \text{ restriction } r: \alpha_c[S[M]](S[M]) \subseteq \alpha_c[S[M]](\alpha_r(S[P])) \]
Suitable abstraction for conservative transformations:

$$\alpha_c[X](Y) = X \cap \text{SubSequences}(Y)$$

returns all the traces in $X$ that are sub-sequences of a trace in $Y$

$$\alpha_c[S[M]](S[O_c[M]]) = S[M]$$
Conservative Transformations

[M. Dalla Preda et al. POPL’07]

The property of being conservative is preserved by composition
Conservative Transformations

The property of being conservative is preserved by composition

Opaque Predicate Insertion

Code Reordering: changes the order in which commands are written while maintaining the execution order

Semantic NOP insertion: inserts irrelevant commands (es. $x := x + 0$)

Substitution of equivalent commands
Example

$\mathcal{O}_c [M]$

- $M$
  - $L_1 : \text{assign}(L_B, B) \rightarrow L_2$
  - $L_2 : \text{assign}(L_A, A) \rightarrow L_c$
  - $L_c : \text{cond}(A) \rightarrow \{L_T, L_F\}$
  - $L_T : B := \text{Dec}(A) \rightarrow L_{T_1}$
  - $L_{T_1} : \text{assign}(\text{succ}(B), B) \rightarrow L_{T_2}$
  - $L_{T_2} : \text{assign}(\text{succ}(A), A) \rightarrow L_C$
  - $L_F : \text{skip} \rightarrow L_B$
  - $L_2 : \text{skip} \rightarrow L_4$
  - $L_c : \text{cond}(A) \rightarrow \{L_O, L_F\}$
  - $L_4 : \text{assign}(L_A, A) \rightarrow L_5$
  - $L_5 : \text{skip} \rightarrow L_c$
  - $L_O : P^T \rightarrow \{L_N, L_k\}$
  - $L_N : X := X - 3 \rightarrow L_{N_1}$
  - $L_{N_1} : X := X + 3 \rightarrow L_T$
  - $L_T : B := \text{Dec}(A) \rightarrow L_{T_1}$
  - $L_{T_1} : \text{assign}(\text{succ}(B), B) \rightarrow L_{T_2}$
  - $L_{T_2} : \text{assign}(\text{succ}(A), A) \rightarrow L_C$
  - $L_k : \ldots$
  - $L_F : \text{skip} \rightarrow L_B$
Identify the set of all possible modifications induced by a non-conservative transformation and fix a **canonical** one

- **Variable renaming**
- Canonical rename: $V_1$ first variable, $V_2$ second variable...
- Apply variable renaming with canonical renaming to $\alpha_r(S[P])$ and to $S[M]$

- Verify infection
- Complete and Sound

Derive the most concrete preserved property as seen before

Use further abstractions
Completeness

- SMD on $\alpha_1$ is complete for $\mathcal{O}_1$ and SMD on $\alpha_2$ is complete for $\mathcal{O}_2$
- $\alpha_1 \circ \alpha_2 = \alpha_2 \circ \alpha_2$
  \[ \Rightarrow \text{SMD on } \alpha_1 \circ \alpha_2 \text{ is complete w.r.t. } \{ \mathcal{O}_1 \circ \mathcal{O}_2, \mathcal{O}_2 \circ \mathcal{O}_1 \} \]

Soundness

- SMD on $\alpha_1$ is sound for $\mathcal{O}_1$ and SMD on $\alpha_2$ is sound for $\mathcal{O}_2$
- $\alpha_1(X) \subseteq \alpha_1(Y) \Rightarrow X \subseteq Y$
  \[ \Rightarrow \text{SMD on } \alpha_1 \circ \alpha_2 \text{ is sound w.r.t. } \mathcal{O}_1 \circ \mathcal{O}_2 \]
Interesting States:

Let $I$ be the set of interesting states:

$$M \hookrightarrow P \text{ if } \exists r : \alpha_I(S[M]) \subseteq \alpha_I(\alpha_T(S[P]))$$

Example: reordering of independent statements
Further Abstraction

[M. Dalla Preda et al. POPL’07]

Interesting Behaviours: consider a significant set of behaviours \( X \subseteq S[M] \):

\[
M \leftrightarrow P \text{ if } \exists r : X \subseteq \alpha_r(S[P])
\]

Interesting Actions: consider a significant set of program actions **Bad**

\[
M \leftrightarrow P \text{ if } \exists r : \alpha_a(S[M]) \subseteq \alpha_a(\alpha_r(S[P]))
\]

\[
\sigma = \langle L : A_i \rightarrow L', \rho_i \rangle
\]
Identifying the class of obfuscators for which a malware detector is resilient can be a complex and error-prone task.

There exists many obfuscation technique often defined using different languages.

Obfuscators and detectors can be expressed on executions traces:

- express the malware detector as an algorithm on traces
- prove soundness and completeness w.r.t. a class of obfuscating techniques

Case study: Semantics-Aware Malware Detection Algorithm proposed by [Christodorescu et al. 2005]
Future work

Malware Detection:

1. Systematically derive a suitable abstraction $\alpha_O$ (Data mining)

Model Checking

1. Abstraction $\alpha_O$ identifies sets of program traces
1. Express such set of traces as formulae in some linear-branching temporal logic

Program semantics is in general not computable, what happens when considering CFG or dependency graphs?

Investigate code obfuscation composition

Semantic Obfuscation

1. Given an attacker $\varphi$ derive an obfuscation (possibly the simplest) able to defeat it.
Future Work

Opaque Predicates Detection

1. Investigate the composition of opaque predicates

2. Investigate a wider class of opaque predicates

3. Use complex abstract domain to design opaque predicates (Polyhedra)
Brute Force Detection

[M. Dalla Preda et al. AMAST’06]

Example: $\forall x \in \mathbb{Z}: 2 \mod (x + x)$ is decomposed into $x$ and $x + y$

16-bit x86 environment: 3 instructions and each variable: $x = 2^{16}$

Time 8.83 seconds

Hybrid Static-Dynamic attack is time consuming
Abstract Detection

∀x ∈ ℤ : 2 \text{ mod } (x + x)

Abstract domain (Attacker): \text{Parity} = \{ ℤ, \text{even}, \text{odd}, ∅ \}

... 

y = x 

z = x + y

\text{cond } z \% 2 

\text{jump if zero}

x = \text{even, odd}

x, y = \text{even, odd}

z = \text{even, odd}
Abstract Detection Results

[M. Dalla Preda et al. AMAST’06]

SPECInt2000 benchmarks obfuscated with:

\[
\forall x \in \mathbb{Z} : 2 \ mod \ (x^2 + x) \quad \text{and} \quad \forall x \in \mathbb{Z} : 2 \ mod \ (x + x)
\]

Hybrid static-dynamic attack 8.83 sec to deobfuscate one opaque predicate
Let $\perp$ denote the undefined function. $\alpha_c[\alpha_e(S[M])]\alpha_e(S[M])$ is given by:
\[
(\perp, \perp) \\
((B \leadsto L_B), \perp) \\
((B \leadsto L_B, A \leadsto L_A), \perp)^2 \\
((B \leadsto L_B, A \leadsto L_A), (\rho(B) \leftarrow \text{Dec}(A))) \\
((B \leadsto \text{succ}(m(\rho(B))), A \leadsto L_A), (\rho(B) \leftarrow \text{Dec}(A))) \\
((B \leadsto \text{succ}(m(\rho(B))), A \leadsto \text{succ}(m(\rho(A)))), (\rho(B) \leftarrow \text{Dec}(A))) \\
\ldots
\]

while $\alpha_e(S[O_c(M)])$ given by:
\[
(\perp, \perp) \\
((B \leadsto L_B), \perp)^2 \\
((B \leadsto L_B, A \leadsto L_A), \perp)^5 \\
((B \leadsto L_B, A \leadsto L_A), (\rho(X) \leftarrow X + 3)) \\
((B \leadsto L_B, A \leadsto L_A), (\rho(X) \leftarrow X + 3, \rho(X) \leftarrow X - 3)) \\
((B \leadsto L_B, A \leadsto L_A), (\rho(B) \leftarrow \text{Dec}(A))) \\
((B \leadsto \text{succ}(m(\rho(B))), A \leadsto L_A), (\rho(B) \leftarrow \text{Dec}(A))) \\
((B \leadsto \text{succ}(m(\rho(B))), A \leadsto \text{succ}(m(\rho(A)))), (\rho(B) \leftarrow \text{Dec}(A))) \\
\ldots
\]
Comparing attackers

\[ \mathcal{R}_{P_1^T}(\varphi) = \mathcal{R}_{P_2^T}(\psi) \]

\[ \mathcal{R}_{P_1^T}(\varphi) \]

\[ \mathcal{R}_{P_2^T}(\varphi) \]

\( P_1^T \) obstructs \( \psi \) more than \( \varphi \)

\( P_1^T \) is more efficient in obstructing the attacker \( \varphi \) than \( P_2^T \)
Reordering of independent statements

independent commands

\[ C_j; C_i \text{ is equivalent to } C_i; C_j \]
Reordering of independent statements

\[ C_j; C_i \text{ is equivalent to } C_i; C_j \]

\[
\begin{align*}
C_1 & \\
C_2 & \\
C_3 & \\
C_4 & \\
C_5 & \\
\end{align*}
\]

\[
\begin{align*}
C_1 & \\
C_3 & \\
C_2 & \\
C_4 & \\
C_5 & \\
\end{align*}
\]
Reordering of independent statements

independent commands

\[ C_j; C_i \text{ is equivalent to } C_i; C_j \]

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

\[ C_1 \]
\[ C_3 \]
\[ C_2 \]
\[ C_4 \]
\[ C_5 \]
Reordering of independent statements

independent commands

\[ C_j; C_i \] is equivalent to \[ C_i; C_j \]

\[ C_1 \]
\[ \rightarrow \]
\[ C_2 \]
\[ \rightarrow \]
\[ C_3 \]
\[ \rightarrow \]
\[ C_4 \]
\[ \rightarrow \]
\[ C_5 \]
\[ \rightarrow \]

\[ C_1 \]
\[ \rightarrow \]
\[ C_3 \]
\[ \rightarrow \]
\[ C_2 \]
\[ \rightarrow \]
\[ C_4 \]
\[ \rightarrow \]
\[ C_5 \]
Let $\mathcal{T}$ be a transformation of a source program $P$ into a target program $P'$. $\mathcal{T}$ is an obfuscating transformation if $P$ and $P'$ have the same observational behaviour. More precisely in order for $\mathcal{T}$ to be a legal obfuscating transformation the following conditions must hold:

- If $P$ fails to terminate or terminates with an error condition, then $P'$ may or may not terminate;
- Otherwise, $P'$ must terminate and produce the same output as $P$. 

Barak et al. Definition

[Barak et al. CRYPTO’01]

A probabilistic algorithm $O$ is a **TM obfuscator** if the following conditions hold:

- **functionality**: For every TM $M$, the string $O(M)$ describes a TM that computes the same function as $M$;

- **polynomial slowdown**: The description length and running time of $O(M)$ are at most polynomially larger than that of $M$. That is, there exists a polynomial $p$ such that for every TM $M$, $|O(M)| \leq p(|M|)$, and if $M$ halts in $t$ steps on some input $x$, then $O(M)$ halts within $p(t)$ steps on $x$;

- **virtual black-box property**: anything one can efficiently compute from the obfuscated program, one should be able to efficiently compute given just oracle access to the program (oracle says the input-output behaviour)
Relation with Signature Matching

\[ \mathbb{P} = \wp(\mathbb{C}), \text{ malware signature } S \subseteq M \]

syntactic test: \( S \subseteq P \)

\[ \alpha_s(S[M]) = S[S] \]

semantic test: \( \exists r : \alpha_s(S[M]) \subseteq \alpha_r(S[P]) \)

Proposition: Semantic and syntactic test are equivalent

Semantic test corresponds to SMD when \( S = M \)

All the results still hold if considering abstraction \( \alpha_s(S[M]) \)
[Detecting Malicious Code by Model Checking, Kinder et al. 2005]

Logic CTPL (Computation Tree Predicate Logic), extension of CTL: $p(x_1...x_n)$ where $x_i$ are free variable in universe $U$ or constants.

“In the code there exists a `mov` instruction that loads the constant 937 into some register, later the value contained in this register is always pushed into the stack”:

$$\exists r \text{EF} (\text{mov}(r, 937) \land \text{AF}(\text{push}(r)))$$

Experimental results: carefully written CTPL specifications can apply to several families of worms.

Replace x86 instructions predicated with abstracted forms that capture their operational semantics.
Why Galois Connections?
Why Galois Connections?

best correct representation of a concrete element in the abstract domain
Why Galois Connections?

\[ \varphi(\mathbb{Z}) \]

\[ \mathbb{Z} \]

\[ 0 - \quad 0+ \]

\[ 0 - \quad 0+ \]

\[ \neq 0 \]

Sign

Sign\(^+\)

compare abstractions