Obscuring Code

Unveiling and Veiling Information in Programs

Roberto Giacobazzi

University of Verona

&

irdeto
Open Platforms Pose an Opportunity & a Risk

Ongoing trend towards openness (even TVs and STBs)

Open devices/platforms attract more developers and consumers

Unfortunately, they decrease the required hacker skill level

---

Low Security Threat

<table>
<thead>
<tr>
<th>Feature</th>
<th>Phone</th>
<th>Game Console</th>
<th>Symbian Phone</th>
<th>BlackBerry</th>
<th>iPad iPhone</th>
<th>Mac</th>
<th>Windows PC</th>
<th>Android Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone</td>
<td>Blackberry</td>
<td>Symbian</td>
<td>iPhone</td>
<td>iPhone</td>
<td>iPad</td>
<td>Mac</td>
<td>Windows</td>
<td>Android</td>
</tr>
<tr>
<td>TV</td>
<td>STB</td>
<td>Game Console</td>
<td>Blackberry</td>
<td>Blackberry</td>
<td>iPad</td>
<td>Mac</td>
<td>Windows</td>
<td>Android</td>
</tr>
</tbody>
</table>

Ubiquitous

High

Required Hacker Skill Level

Low
Crypto Assumption

Black Box Attacks or Grey Box Attacks

Man-In-The-Middle Attack (Indirect, side-channel)

Alice and Bob each have exclusive control over their own computers
No information leaves from or store into their computers without their approval

Perimeter Defenses
Network

Trusted Inside Black Box

Alice
Software

Bob
Software

Network
White-Box Attacks

**Attackers have open-end powers to do**
- Trace every program instruction
- View the contents of memory and cache
- Stop execution at any point and run an off-line process
- Alter code or memory at will
- Do all of this for as long as they want, whenever they want, in collusion with as many other attackers as they can find

**Attacking has much less limitation than protection**
- Device and environment are untrusted
- Attacker has direct access to the machine and software no matter whether it's running or not

**Bob is the Attacker**

**Man-At-The-End Attack**

**Network**

**Alice**

**Software**
Just Like in Museums
The state of the art
Obfuscation

“Anything that can be learned from the obfuscated form, could have been learned by merely observing the program’s input-output behavior (i.e., by treating the program as a black-box)”

A nondeterministic algorithm $O$ is a TM obfuscator if three following conditions hold:

- **(functionality)** For every TM $M$, the string $O(M)$ describes the same function as $M$.
- **(polynomial slowdown)** The description length and running time of $O(M)$ are at most polynomially larger than that of $M$.
- **(“virtual black box” property)** For any PPT $A$, there is a PPT $S$ and a negligible function $\alpha$ such that for all TMs $M$

$$\left| Pr[A(O(M)) = 1] - Pr[S^M(1|M|) = 1] \right| \leq \alpha(|M|).$$
Impossibility

“Anything that can be learned from the obfuscated form, could have been learned by merely observing the program’s input-output behavior (i.e., by treating the program as a black-box)”

A nondeterministic algorithm $O$ is a **TM obfuscator** if three following conditions hold:

⇒ **(functionality)** For every TM $M$, the string $O(M)$ describes the same function as $M$.

⇒ **(polynomial slowdown)** The description length and running time of $O(M)$ are at most polynomially larger than that of $M$.

⇒ **(“virtual black-box” property)** For any PPT $A$, there is a PPT $S$ and a negligible function $\alpha$ such that for all TMs $M$:

$$\left| Pr[A(O(M)) = 1] - Pr[S^M(1|M|) = 1] \right| \leq \alpha(|M|).$$

**Barak’s et al. JACM 2012**
What exactly do we mean when we say that we have obfuscated a program?

Indistinguishability:
If P and Q compute the same function then $\mathcal{O}(P) \approx \mathcal{O}(Q)$

1. Program obfuscation re-cast as a rigorous mathematical science
2. The adversary can have full knowledge of the obfuscating theory but still cannot de-obfuscate

Garg, Gentry, Halevy, Raykova, Sahai, Waters, 2013 “Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits”.
The value

- **Secured Input**: Authentication, validation, integrity, confidentiality of input data
- **Secured Output**: Authentication, validation, integrity, confidentiality of output data
- **Hide Algorithms & Computations**
- **Tamper Resistance**: Makes it hard to modify the code’s data and control flow
- **Tamper Detection**
- **Hide Internal Data**: Including internally initialized data
- **Damage Mitigation**
- **Anti Bug**
A different view from PL…
**Attack Effectiveness**

- **Input / output**
  - Black-box
    - Timing analysis
    - Power analysis
    - Fault injection
  - Grey Box
  - White-box
    - Debuggers
    - Emulators
    - Other attack tools

**Difficulty to Protect Assets**

- **Easiest**
  - Input / output abstraction

- **Hardest**

**Weakest**

- Attack Effectiveness

**Strongest**

**More Concrete**

**White-box**

**Grey Box**

**Black-box**

**Input / output**
Obfuscation

$\text{Input} \xrightarrow{\tau} \text{Output}$

$P \rightarrow \tau[P]$
Obfuscation as **compilation**

(Pseudo-)Code:

\[
\begin{align*}
\text{(Pseudo-)Code:} \\
\text{mov eax, [edx+0Ch]} \\
\text{push ebx} \\
\text{push [eax]} \\
\text{call ReleaseLock}
\end{align*}
\]

\[
\begin{align*}
\text{Obfuscated code (junk + reordering):} \\
\text{mov eax, [edx+0Ch]} \\
\text{jmp +3} \\
\text{push ebx} \\
\text{dec eax} \\
\text{jmp +4} \\
\text{inc eax} \\
\text{jmp -3} \\
\text{call ReleaseLock} \\
\text{jmp +2} \\
\text{push [eax]} \\
\text{jmp -2}
\end{align*}
\]
Abstract Interpretation is a general theory for approximating the semantics of dynamic systems (Cousot & Cousot 1977)

Computing means Interpreting

Abstraction

For large/real programs control/data flow is too complex for being understandable by humans:

Reverse Engineering needs abstraction!
Reverse Engineering needs automated tools!

Interpretation
Reverse Engineering is Interpreting $P ightarrow \neg \neg P$.

Constrained Adversary

Each tool is an Abstract Interpretation.

Disassembler → IDA Pro → BinHunt → BinDiff → BinJuice → Theorem Prover

HexRays → GDB → OllyDbg → SMT → SAT → Proof

Tracing → Decompiler → Profiling → Slicing → Static Analysis → Dynamic Analysis → Monitoring

Static Analysis → Monitoring

Dynamic Analysis

We can quantify the security achieved by looking at proof complexity!
Protecting is obscuring Interpretation

Removing noise means refining abstractions / complicating proofs! (Giacobazzi et al 2000 / 2012)
...the ingredients?
A Model

Bad State

Too complicated, complex, undecidable

No bug!
To understand code we need abstraction: *simpler and computable*
Abstraction

Abstraction $\leadsto$ (sound) loss of precision
Abstraction

Computable Abstraction ➟ (sound) loss of precision

Bad State

No bug!
Completeness

Abstraction $\Rightarrow$ (sound) loss of precision

Incompleteness

Bad State

False Alarms
Completeness

Abstraction $\Rightarrow$ False alarms
Incompleteness

Bad State

$x(t)$

Error

Incompleteness $\neq$ Error

False alarms

$t$
Completeness

\[ \eta \circ f \circ \rho = \eta \circ f \]
(in)Completeness

In-completeness: $\eta \circ f \circ \rho \geq \eta \circ f$
Making Completeness

Refining input domains

[Giacobazzi et al. JACM’00]
Making Completeness

Making ABSTRACTIONS COMPLETE: Simplifying output domains

[Giacobazzi et al. JACM’00]
Making Completeness

A simple example in Interval analysis

A simple domain of intervals

\[ sq(X) = \{ x^2 \mid x \in X \} \]

\{\mathbb{Z}, [0, +\infty], [0, 10]\} is not Backward complete

Same input & output abstraction = fix-point refinement

\[ R_f(\alpha) = \text{gfp}(\lambda X. \alpha \sqcap R_f(X)) \]

\[ R_f \overset{\text{def}}{=} \lambda X. \mathcal{M}(\bigcup_{y \in X} \max(f^{-1}(\downarrow y))) \]
Transformation

A self interpreter int and a specializer spec

Defense  \[ \text{target := [spec]}(\text{int, source}) \]

source

Attack

Design int and spec for obfuscating code

Challenge!
Transformation

A self interpreter \texttt{int} and a specializer \texttt{spec}

\begin{verbatim}
1. input x;  2. pc := 2;
3. while \{p\} c < 6 do
4.   case pc of
5.     y := 2;  6. pc := 3;
7.     if \{x\} \{>0 then 8. pc := 4 else 9. pc := 6;
10.    y := y + 2;  11. pc := 5;
12.   x := x - 1;  13. pc := 3;
endw
14. output y
15. end
\end{verbatim}

Design \texttt{obfuscation} from \texttt{observation}  
\begin{itemize}
\item (\texttt{transformation})
\item (\texttt{abstraction})
\end{itemize}
The Attacker
**WHY ABSTRACT INTERPRETATION?**

*Abstract Interpretation (1977) is the a general model for the (static or dynamic) approximation of semantics of discrete dynamic systems*

Including: Static program analysis, dynamic analysis, profiling, debugging, tracing, compilation, de-compilation, type checking and type inference, model checking and predicate abstraction, trajectory evaluation, testing, proof systems, etc.
Abstract Interpretation

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

Galois Connection: \( \langle C, \alpha, \gamma, A \rangle \), \( A \) and \( C \) are complete lattices.

Closures: \( \langle uco(C), \sqsubseteq \rangle \) set of all possible abstract domains,

\( A_1 \sqsubseteq A_2 \) if \( A_1 \) is more concrete than \( A_2 \).
**Approximating Interpretation**

$G$ is a sound approximation of $F$ if

$$\alpha \circ F \circ \gamma \subseteq G$$
**Soundness and Completeness**

\[ \text{WhichChess} : \text{Img} \longrightarrow \wp(\text{Chess}) \] returns the type of chess on the chessboard.

\[ \rho : \text{Img} \longrightarrow \text{Img} \] such that: \[ \rho \begin{pmatrix} \includegraphics[width=0.1\textwidth]{chess1} \end{pmatrix} = \begin{pmatrix} \includegraphics[width=0.1\textwidth]{chess2} \end{pmatrix} \]

\[ \eta : \wp(\text{Chess}) \longrightarrow [0, 12] \] counts an *upper bound* to the number of different types of chess

\[ \eta \left( \text{WhichChess} \left( \rho \begin{pmatrix} \includegraphics[width=0.1\textwidth]{chess1} \end{pmatrix} \right) \right) = \eta \left( \text{WhichChess} \begin{pmatrix} \includegraphics[width=0.1\textwidth]{chess2} \end{pmatrix} \right) = 12 \geq \eta \left( \text{WhichChess} \begin{pmatrix} \includegraphics[width=0.1\textwidth]{chess3} \end{pmatrix} \right) = 7 \]
Obscurity as Incompleteness

The attack strategy is a temporal formula to check against an abstraction

The attacker is an abstract interpreter

Failing precision means failing completeness

Obfuscating is making abstract interpreters and strategies incomplete!!

\[
[P] = [\tau(P)] \\
\rho([P]) = [P]^\rho
\]

\(\tau\) obfuscates \(P\) if \([P]^\rho \sqsubseteq [\tau(P)]^\rho\)

\([P]^\rho \sqsubseteq [\tau(P)]^\rho \iff \rho([\tau(P)]) \sqsubseteq [\tau(P)]^\rho\)
Obscurity as incompleteness

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

\[ P : x = a \times b \]

Sign is an obvious abstraction of \( \varphi(\mathbb{Z}) \):
**Obscurity as Incompleteness**

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

\[ P : x = a \times b \]

*Sign* is an abstraction of \( \varphi(\mathbb{Z}) \):

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

\[ \ldots \quad 0^- \quad 0^+ \quad \ldots \]

\[ \{-1, -3, -4\} \quad \ldots \quad \{2, 3, 5\} \]

\[ \ldots \quad 0 \quad 1 \quad \ldots \]

\[ \varnothing \]
OBSCURITY AS INCOMPLETENESS

Failing precision means failing completeness!

Obfuscating programs is making abstract interpreters incomplete

\[
\begin{align*}
\text{\texttt{x} = 0;}
\end{align*}
\]

\[
P : \quad \texttt{x = a \star b} \quad \rightarrow \quad \tau(P) : \quad \text{if } b \leq 0 \text{ then } \{ a = -a; \ b = -b \}; \\
\quad \text{while } b \neq 0 \{ \texttt{x} = a + \texttt{x}; \ b = b - 1 \}
\]

\[\text{Sign} \text{ is complete for } P:\]
\[\checkmark \quad \llbracket P \rrbracket^{\text{Sign}} = \lambda a, b. \text{ Sign}(a \star b)\]

\[\text{Sign} \text{ is incomplete for } \tau(P):\]
\[\checkmark \quad \llbracket \tau(P) \rrbracket^{\text{Sign}} = \lambda a, b. \begin{cases} 
0 & \text{if } a = 0 \lor b = 0 \\
?= \wp(\mathbb{Z}) & \text{otherwise}
\end{cases}\]
\[ P : \ x := a \ast b \]
\[ Q : \ x := a \ast (b - 2) + a + a \]

\[
\begin{align*}
\llbracket P \rrbracket^{\text{Sign}}(\{a \mapsto +, \ b \mapsto +\}) &= \{x, \mapsto +, \ a \mapsto +, \ b \mapsto +\} \\
\llbracket Q \rrbracket^{\text{Sign}}(\{a \mapsto +, \ b \mapsto +\}) &= \{x, \mapsto \mathbb{Z}, \ a \mapsto +, \ b \mapsto +\}.
\end{align*}
\]
EXPOLITING INCOMPLETENESS

Maximize $[P]^\rho$ incompleteness!

The abstraction is the specification of the attacker

- **Profiling**: Abstract memory keeping only (partial) resource usage
- **Tracing**: Abstraction of traces (e.g., by trace compression)
- **Slicing**: Abstraction of traces (relative to variables)
- **Monitoring**: Abstraction of trace semantics ([Cousot&Cousot POPL02])
- **Decompilation**: Abstracts syntactic structures (e.g., reducible loops)
- **Disassembly**: Abstracts binary structures (e.g., recursive traversal)

Each abstraction is incomplete for a concrete enough trace semantics

Maximize incompleteness by code transformation: **Obfuscation**

Exploit incompleteness for hiding information: **Steganography**
Interpreter-based Obfuscation
Whole-program view of Obfuscation

A major conflict makes program obfuscation a subtle problem in programming:

*Good programs are well-structured and have concise invariants*

This is a key to

- understanding a program, and
- adapting it to new purposes.

Good structure and short invariants are necessity in order to develop, debug and perfect a program $P$ in the first place.

However, instead an obfuscated program should not be well-structured and easy to understand.

This suggests (among other things):

- obfuscation by making the program’s control/data flow hard to understand
ABOUT $P' = \text{[spec]}(\text{interp}, P)$

1. Program $P'$ inherits the *algorithm* of program $P$.
2. Program $P'$ inherits the *programming style* of $\text{interp}$.

$arrow$

1: A correct interpreter $\text{interp}$ must faithfully execute the operations specified by program $P$. Usually: specialized program $P'$ performs *the same computations in the same order* as those performed by $P$.

Most interpreters do not devise new computational approaches!

$arrow$

2: $P'$ consists of specialized code from the interpreter $\text{interp}$:

$P' = \text{the operations of } \text{interp} \text{ that depend on its dynamic input}$

(all others will be “specialized away”).
Idea

Build a *general-purpose program transformer* by programming a self-interpreter in a style to give the desired transformation.

**Claim:** $[[P]] = [[P']]$, by simple equational reasoning:

\[
[[P]](d) = [[\text{interp}}](P, d) \quad \text{definition of self-interpreter}
\]

\[
= [[[[\text{spec}}](\text{interp}, P)]](d) \quad \text{definition of specializer}
\]

\[
= [[P']](d) \quad \text{definition of } P'
\]

Therefore the function

\[
P \mapsto [[\text{spec}}](\text{interp}, P)
\]

is a *semantics-preserving program transformer*!!

We need to *change the interpretation*: $\text{interp} \sim \text{interp}^+$
Flattening
CODE FLATTENING

Idea: “scramble” or “distort” the control flow of input program $P$, without changing its whole-program semantics


**Example of Flattening**

The following flattened program $P'$ has

- only one loop (regardless of how many loops $P$ has), and
- an explicit program counter $pc$

Original program $P$:

1. `input x;`
2. `y := 2;`
3. `while x > 0 do`
   4. `y := y + 2;`
   5. `x := x - 1`
   `endw`
6. `output y;`
7. `end`

Flattened equivalent program $P'$:

1. `input x;`
2. `pc := 2;`
3. `while pc < 6 do`
4. `case pc of`
   2: `y := 2; pc := 3;`
   3: `if x > 0 then pc := 4 else pc := 6;`
   4: `y := y + 2; pc := 5;`
   5: `x := x - 1; pc := 3;`
   `endw`
6. `output y;`
7. `end`
**Structure of a simple self-interpreter**

input $P, d$;  
Program to be interpreted, and *its* data

$pc := 2$;  
Initialise program counter and store

$store := \{in \mapsto d, \, out \mapsto 0, \, x_1 \mapsto 0, \ldots\}$;

**while** $pc < \text{length}(P)$ **do**

$instruction := \text{lookup}(P, pc)$;  
Find the $pc$-th instruction

**case** $instruction$ **of**

*skip*: $pc := pc + 1$;

$x := e$ : $store := store[x \mapsto \text{eval}(e, store)]; \; pc := pc + 1$;

... endw;

**output** $store[out]$;

$\text{eval}(e, store) = \text{case } e \text{ of}$  
Function to evaluate expressions

*constant*: $e$

*variable*: $\text{store}(e)$

$e_1 + e_2$ : $\text{eval}(e_1, store) + \text{eval}(e_2, store)$

$e_1 - e_2$ : $\text{eval}(e_1, store) - \text{eval}(e_2, store)$

$e_1 * e_2$ : $\text{eval}(e_1, store) * \text{eval}(e_2, store)$  
...  

$\text{target} := \text{spec}[\text{int}, \text{source}]$
Why?
The attacker is an abstract interpreter extracting the CFG from $\mathcal{P}$

- Forgets the computed memory $M$: $C = \lambda \sigma. M$
- Forgets the branch computation when involving the $\mathit{pc}$: $\eta$
- Fixpoint Graph semantics: $[\mathcal{P}]_G = \mathit{lfp}(G_{\mathcal{P}})$

Theorem

$C([\mathcal{P}]_G) = [\mathcal{P}]_G^{C, \eta}$ iff $\mathit{pc}$ is not a program variable

Completeness!!

Flattening is distorting an interpreter making an abstract interpreter extracting the CFG incomplete
A Theory?
Simplifying abstractions

\[ R \overset{\text{def}}{=} R_{\mathcal{F}}^{\mathcal{F}} = \lambda \rho. \rho \sqcap \mathcal{M}(f(\eta)) \]

\[ sq(X) = \{ x^2 \mid x \in X \} \]
Simplifying abstractions

\[ R \overset{\text{def}}{=} R_{f, \eta}^{\overline{f}} = \lambda \rho. \rho \sqcap \mathcal{M}(f(\eta)) \]

\[ sq(X) = \{ x^2 \mid x \in X \} \]
Simplifying abstractions

\[ R \overset{\text{def}}{=} R_{f, \eta} = \lambda \rho. \rho \sqcap M(f(\eta)) \]
Simplifying abstractions

\[ R \overset{\text{def}}{=} R_{f,\eta} = \lambda \rho. \rho \sqcap \mathcal{M}(f(\eta)) \]

\[ R^+ = \lambda \rho. \biguplus \{ \delta \mid R(\delta) = R(\rho) \} = \lambda \rho. \biguplus \{ \delta \mid \delta \sqcap \mathcal{M}(f(\eta)) = \rho \sqcap \mathcal{M}(f(\eta)) \} \]

\[ \lambda \rho. \mathcal{M}(\text{M} \text{ir}(\rho \sqcap \mathcal{M}(f(\eta))) \setminus \mathcal{M}(f(\eta))) \]
Example

\[ sq(X) = \left\{ x^2 \mid x \in X \right\} \]
Example

$$sq(X) = \left\{ x^2 \mid x \in X \right\}$$

$$Mirr(Int) = \{ [-\infty, b] \mid b \in \mathbb{Z} \} \cup \{ [a, +\infty] \mid a \in \mathbb{Z} \}$$
Example

\[ sq(X) = \{ x^2 \mid x \in X \} \]

\[ Mirr(\text{Int}) = \{ [-\infty, b] \mid b \in \mathbb{Z} \} \cup \{ [a, +\infty) \mid a \in \mathbb{Z} \} \]

\[ \mathcal{M}(sq^2(\text{Int})) = \{ [a^2, b^2] \mid a, b \in \mathbb{Z} \} \cup \{ [a^2, +\infty] \mid a \in \mathbb{Z} \} \]

\[ \text{Int}' = \mathcal{M}(\text{Mirr} (\text{Int}) \setminus \mathcal{M}(sq^2(\text{Int}))) \]
Example

\[ sq(X) = \left\{ x^2 \mid x \in X \right\} \]

\[ \text{Mirr}(\text{Int}) = \{ [-\infty, b] \mid b \in \mathbb{Z} \} \cup \{ [a, +\infty] \mid a \in \mathbb{Z} \} \]

\[ \mathcal{M}(sq^2(\text{Int})) = \{ [a^2, b^2] \mid a, b \in \mathbb{Z} \} \cup \{ [a^2, +\infty] \mid a \in \mathbb{Z} \} \]

\[ \text{Int}' = \mathcal{M}(\text{Mirr}(\text{Int}) \setminus \mathcal{M}(sq^2(\text{Int}))) = \{ [a, b] \mid a, b \in \mathbb{Z}, \exists c. a = c^2 \lor \exists d. b = d^2 \} \cup \{ [-\infty, b] \mid b \in \mathbb{Z} \} \cup \{ [a, +\infty] \mid a \in \mathbb{Z}, \exists c. a = c^2 \} \]
Absolute Incomplete Compressor

If $\mathcal{R}^+(\rho)$ is not T then
\[ \mathcal{R}^+(\rho) \] is absolute incomplete compressor, i.e.,
\[ \mathcal{R}^+(\rho) \circ f \circ \mathcal{R}^+(\rho) \neq f \circ \mathcal{R}^+(\rho) \]

\[ S = \rho_S(\varphi(Z)) \]

\[ \rho_b = \{ \emptyset, [0, +\infty], [0, 9], [-9, 0], [0] \} \]

\[ sq^S(X) = \rho_S(sq(X)) \]

\[ S' = \mathcal{M}(\operatorname{Mirr}(\rho_b) \cap \mathcal{M}(sq^b(\rho_b))) \setminus \mathcal{M}(sq^b(\rho_b)) \]

\[ S' \circ sq^S \circ S' \neq sq^S \circ S' \]
Slicing
Program Slicing Obfuscation

For a variable v and a statement (program point) s (final use of v), the slice S of program P with respect to the slicing criterion \langle s,v \rangle is any executable program such that S can be obtained by deleting zero or more statements from P and if P halts on input I then the value of v at the statement s, each time is reached in P, is the same in P and in S.

P_1 \begin{align*}
1. & x := 0; \\
2. & i := 1; \\
3. & \text{while } i > 0 \text{ do } i := i + 1; \\
4. & y := x;
\end{align*}

P_2 \begin{align*}
1. & x := 0; \\
4. & y := x;
\end{align*}
Data Dependency (Slicing) Obfuscation

Word Count program
which takes a block of text and outputs the number of
lines (nl), words (nw) and characters (nc):

```c
original() {
    int c, nl = 0, nw = 0, nc =0, in;
in = F;
while ((c = getchar()) != EOF) {
    nc ++;
    if (c == ' ' || c == '
' || c == 't') in = F;
    else if (in == F) {in = T; nw ++; }
    if (c == 'n') nl ++;
}
out(nl, nw, nc); }
```
Data Dependency (Slicing) Obfuscation

Word Count program
which takes a block of text and outputs the number of
lines (nl), words (nw) and characters (nc):
Slicing criterion: nl

original() {
    int c, nl = 0, nw = 0, nc =0, in;
in = F;
while ((c = getchar()) ! = EOF) {
    nc ++;
    if (c == ' ' || c == '
' || c == 't') in = F;
    else if (in == F) {in = T; nw ++; }
if (c == '\n') nl ++;
}
out(nl, nw, nc); }

Word Count program
which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):

Slicing criterion: nw

original() {
  int c, nl = 0, nw = 0, nc =0, in;
in = F;
while ((c = getchar()) ! = EOF) {
  nc ++;
  if (c == ' ' || c == '
' || c == '\t') in = F;
  else if (in == F) {in = T; nw ++; }
  if (c == '\n') nl ++;
}
out(nl, nw, nc); }
Data Dependency (Slicing) Obfuscation

Word Count program
which takes a block of text and outputs the number of
lines (nl), words (nw) and characters (nc):

```c
obfuscated() { 
    int c, nl = 0, nw = 0, nc = 0, in;
in = F;
while ((c = getchar()) != EOF) {
    nc ++;
    if (c == ' ' || c == '\n' || c == '\t') in = F;
    else if (in == F) {in = T; nw ++; }
    if (c == '\n') {if (nw <= nc) nl ++; }
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw - nl; }
}
out(nl, nw, nc); }
```
Data Dependency (Slicing) Obfuscation

Word Count program
which takes a block of text and outputs the number of
lines (nl), words (nw) and characters (nc):

```c
obfuscated() { 
    int c, nl = 0, nw = 0, nc = 0, in;
    in = F;
    while ((c = getchar()) != EOF) {
        nc ++;
        if (c == ' ' || c == '
' || c == '	') in = F;
        else if (in == F) {in = T; nw ++; }
        if (c == '
') {
            if (nw <= nc) nl ++;
        }
        if (nl > nc) nw = nc + nl;
        else {
            if (nw > nc) nc = nw - nl;
        }
    }
    out(nl, nw, nc); }
```

Always true

Always false

Word Count program which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):
Data Dependency (Slicing) Obfuscation

Word Count program
which takes a block of text and outputs the number of
lines (nl), words (nw) and characters (nc):
Slicing criterion: nl

```c
obfuscated() {
    int c, nl = 0, nw = 0, nc = 0, in;
in = F;
while ((c = getchar()) != EOF) {
    nc ++;
    if (c == ' ' || c == '
' || c == '	') in = F;
else if (in == F) {in = T; nw ++; }
if (c == '
') {if (nw <= nc) nl ++;
if (nl > nc) nw = nc + nl;
else {if (nw > nc) nc = nw - nl; }
}
out(nl, nw, nc); }
```
Data Dependency (Slicing) Obfuscation

Word Count program
which takes a block of text and outputs the number of lines (nl), words (nw) and characters (nc):
  Slicing criterion: nw

```
obfuscated() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    nc ++;
    if (c == ' ' || c == '
' || c == 't') in = F;
    else if (in == F) {in = T; nw ++; }
    if (c == '
') {if (nw <= nc) nl ++; }
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw - nl; }
  }
  out(nl, nw, nc); }
```
P_1 = \begin{array}{l}
1. x := 0; \\
2. i := 1; \text{while } i > 0 \text{ do } i := i + 1; \\
4. y := x;
\end{array}

D_{\text{Entry}} = \emptyset \quad s = \langle \sigma, \langle \text{Entry}, 1 \rangle \rangle
Attack: Semantic PDG

\[
\begin{align*}
&1. x := 0; \\
&P_1 \begin{cases} \\
2. i := 1; \quad \text{while } i > 0 \text{ do } i := i + 1; \\
4. y := x;
\end{cases}
\end{align*}
\]

\[
D_1 = [D_x = 1] \quad s = \langle \sigma, \langle 1, 2 \rangle \rangle
\]
5.1 Program slicing

Let us provide a brief overview on program slicing [8] and on the way slices are computed in [9]. Each time a program component and edges denoting control and flow dependence are abstracted in the program slicing; in particular, we first define a semantics similar to Definition 22.

5.2 Semantic

We define now the abstractions characterizing the program semantics that can be abstracted backwards the edges starting from the starting vertex. An edge represents either a control dependence edge or a flow dependence edge. With this representation we have only control and flow dependence edges without any intervening definition of such that 1-attack: Semantic PDG

\[ P_1 \begin{cases} 
  1. x := 0; \\
  2. i := 1; \quad \text{while } i > 0 \text{ do } i := i + 1; \\
  4. y := x; 
\end{cases} \]

\[ D_2 = [D_x = 1, D_i = 2] \quad s = \langle \sigma, \langle 2, 3 \rangle \rangle \]

\[
\text{Entry}
\]

1 2 3 4
We define now the abstractions characterizing the program semantics that can be ab, node
\[ \text{Definition 22} \]
and that 4an assignment5. 4)5
\[ \text{Example 23.} \]

Example 23.

Consider the following programs. Note that the subgraph containing all the vertexes that can reach the final use of the program components and edges denoting dependencies between components. The vertices of the program dependence graph (PDG) represent the assignment statements and control predicates that may be reached more times in the program. The standard approach for characterizing slices is based on the criterion the value of
\[ \text{Definition 22} \]

\[ P_1 \begin{align*} 1. x &:= 0; \\ 2. i &:= 1; \text{ while } i > 0 \text{ do } i := i + 1; \\ 4. y &:= x; \end{align*} \]

\[ D_3 = [D_x = 1, D_i = 2] \quad s_1 = \langle \sigma, \langle 3, 3a \rangle \rangle \text{ and } s_2 = \langle \sigma, \langle 3, 4 \rangle \rangle \]
Attack: Semantic PDG

\[ P_1 \begin{cases} 
1. & x := 0 ; \\
2. & i := 1 ; \text{ while } i > 0 \text{ do } i := i + 1 ; \\
4. & y := x ;
\end{cases} \]

\[ D_{3a} = [D_x = 1, D_i = 3a] \quad s_1 = \langle \sigma, \langle 3a, 3 \rangle \rangle \text{ and } s_2 = \langle \sigma, \langle 3, 4 \rangle \rangle \]
Attack: Semantic PDG

\[ P_1 \begin{cases} 1. x := 0; \\ 2. i := 1; \quad \text{while } i > 0 \text{ do } i := i + 1; \\ 4. y := x; \end{cases} \]

\[ D_{3a} = [D_x = 1, D_i = 3a] \quad s_1 = \langle \sigma, (3, 3a) \rangle \quad \text{and} \quad s_2 = \langle \sigma, (3, 4) \rangle \]
**Attack: Semantic PDG**

\[
P_1 = \begin{cases} 
1. x := 0; \\
2. i := 1; \quad \text{while } i > 0 \text{ do } i := i + 1; \\
3. y := x; \\
4. y := x; 
\end{cases}
\]

\[D_4 = [D_x = 1, D_i = 2, D_y = 4] \quad s_1 = \langle \sigma, \langle 3, 3a \rangle \rangle \text{ and } s_2 = \langle \sigma, \langle 4, \bot \rangle \rangle\]
**Potency of Data obfuscation**

- If a program P contains *fake dependencies* (i.e., not semantic) the analyses deceived are those analyzing *syntactic dependencies* only!

- The *incomplete backward compressor* of the abstraction of the PDG based on *semantic* dependencies is the PDG based on *syntactic dependencies* only, i.e., *standard slicing*!

- Namely, any obfuscator adding *fake dependencies* deceives slicing algorithms based on the computation of *syntactic dependencies*!!

---

The semantic PDG is complete (the PDG analysis is precise) iff the program does not contain fake dependencies
Concluding
So what?

Any obfuscation technique is an instance of

$$[\text{spec}] (\text{interp}^+, P)$$

for some $\text{interp}^+$ making an abstraction $\alpha$ incomplete!

- Profiling: Abstract memory keeping only (partial) resource usage
- Tracing: Abstraction of traces (e.g., by trace compression)
- Slicing: Abstraction of traces (relative to variables)
- Monitoring: Abstraction of trace semantics ([Cousot&Cousot POPL02])
- Decompilation: Abstracts syntactic structures (e.g., reducible loops)
- Disassembly: Abstracts binary structures (e.g., recursive traversal)

Given an obfuscated code $P$, what is $\alpha$?

Given $\alpha$, can we derive $\text{interp}^+$ systematically?
META-LEVEL DISCUSSION

The Futamura projections are as follow for a distorted interpreter interp⁺:

1. $P' := \llbracket\text{spec}\rrbracket(\text{interp}^+ , P)$  
   Transform program
2. $\text{comp} := \llbracket\text{spec}\rrbracket(\text{spec}, \text{interp}^+)$  
   Generate transformer
3. $\text{cogen} := \llbracket\text{spec}\rrbracket(\text{spec}, \text{spec})$  
   Transformer generator

We have just seen instances of the 1st Futamura projection.

If the set of *incomplete program structures* is Turing complete: Write the distorted interpreter as incomplete structures

If you want to act *locally*: use interference to ensure that incompleteness is propagated

An *obfuscating compiler* can also be generated, by the 2nd Futamura projection; this has been done using the UNMIX partial evaluator.

For example, if $P$ is interp$^{\text{flat}}$, then compiler is a *stand-alone obfuscator*: a “flattening” program transformer.
Immediate Consequences

There are other better ways to obfuscate and to produce a obfuscator:

\[ P' = [\text{comp}](p) \text{ (obfuscate by compiler) and} \]

\[ \text{comp} = [\text{cogen}](\text{interp}^+) \text{ (generate obfuscator).} \]

Future developments will involve gaining a deeper understanding in expected time factors:

1. \( \text{time}_{P'}(d) \) and \( \text{time}_P(d) \), exemplifying the slowdown imposed by the obfuscation;

2. \( \text{time}_{\text{spec}}(\text{interp}^+, p) \) and \( \text{length}(P) \), exemplifying the amount of time required to do the obfuscation by specialization;

3. \( \text{time}_{\text{comp}}(p) \) and \( \text{length}(P) \), exemplifying the time to do the obfuscation by running the generated obfuscator.
By constraining the adversary within a theorem prover we can quantify the security achieved from obfuscation.

- Force the attacker to use automated tools (programs of large size and highly interconnected)
- Design code transformations making tools blind
- Determine lower bounds for proof complexity in obfuscated code
- Measure the degree of noise/slowdown induced in obfuscation
By constraining the adversary within a theorem prover we can quantify the security achieved from obfuscation.

- Force the attacker to use automated tools (programs of large size and highly interconnected)
- Design code transformations making tools blind
- Determine lower bounds for proof complexity in obfuscated code
- Measure the degree of noise/slowdown induced in obfuscation

Measuring Adversary Strength?

Proof complexity

Low  High

Degree of obfuscation

Low  High
Thanks!

Mila

Neil

Isa