Interactive, imperative programs

- Why interactive?
  - Interactive programs: allow user input and output at runtime.
  - Programs whose security we care about are invariably interactive.
    - E.g., web servers, communication systems, etc.

- Why imperative?
  - How most real systems are built:
    - Traditional control-flow structures.
    - Mature compilers and analyses.

Models of interactivity

- Two major models:
  - Interactive state-based (trace-based) systems.
  - Process algebras.

- We reuse some important ideas:
  - Input and output as fundamental operations.
  - Traces to encode runtime observations.
  - Explicit modeling of agents.

Informal preview

- Take a straightforward sequential language:
  
  \[-\text{skip} \mid x := e \mid \text{if } e \text{ then } c_0 \text{ else } c_1 \mid \text{while } e \text{ do } c\]

- Assume programs can interact with channels, which have high or low confidentiality levels.

- Add commands for interaction with channels:
  
  \[-\text{input } x \text{ from } \tau\]
  \[-\text{output } e \text{ to } \tau\]

- Goal: define semantic security conditions for such a language.

Imperative meets interactive

- Imperative usually implies “batch-job”:
  - “Inputs” are initial variable values.
  - “Outputs” are final (and sometimes interim) values.
  - Security conditions seek to protect confidential information stored in program variables.

- Interactive programs are more realistic.
  - Want to capture dependencies between program outputs and subsequent user input.
  - Input/output operators are a useful abstraction.
    - Don’t want to assume observable runtime memory.

Our contributions

- A semantic definition of noninterference for interactive programs.

- Generalizations to deal with probability and nondeterminism.

- Proof that VSI type system (with minor modifications) soundly enforces our new conditions.
Our system model

- Users interact with programs via channels.
  - Input and output events occur on channels.
  - Channels/users labeled H (high) or L (low).
- High users interact with high channel; low users interact with low channel.

User interaction model

Low users can’t observe high inputs or outputs.

High users can’t input on low channel.

Users and channels: assumptions

- Inputs are blocking.
- Users cannot directly observe values of variables.
- Users observe only the sequence of events occurring on the channels they observe.
  - We ignore timing channels in this work.
- Users eventually supply inputs when prompted.
  - Our definitions still valid without this assumption.
What is a secure system?

• Noninterference: low users must not be able to infer anything about high behavior, given low observations.
• In general, we assume that users may:
  – Know text of programs.
  – Be “logically omniscient.”
• Let’s look at some examples...

Insecure interactive programs

• A direct flow:
  input $x$ from $H$;
  output $x$ to $L$

• An implicit flow:
  input $x$ from $H$;
  if $(x=0)$ then
    output 0 to $L$
  else
    output 1 to $L$

Secure interactive programs

• Programs with no high inputs are secure:
  output $x$ to $L$
• Care about high inputs, not contents of memory.
• If programs run multiple times with same memory, can:
  – Require programs to “zero out” memory before each execution.
  – Model program sequence as a single program.

One-time pad 2.0

• What if we tell high users the one-time pad?
  while (true) do
    $x:=0$ [0.5] $x:=1$;
    output $x$ to $H$;
    input $y$ from $H$;
    output $(x \text{ XOR } y)$ to $L$
• Is this program still secure?
  – Note that low user still can’t infer value of $y$.

Why v2.0 isn’t secure

• Suppose a high user wants to transmit bit $z$:
  while (true) do
    $x:=0$ [0.5] $x:=1$;
    output $x$ to $H$;
    input $y$ from $H$;
    output $(x \text{ XOR } y)$ to $L$

• High user can transmit value $z$ directly.
  – Even though value of $y$ remains secret.
• Thus low users can learn about behavior of high users.
**User strategies**

• How to formalize behavior in our model?
  – In one-time pad v2.0, confidential user can transmit arbitrary bit strings by selecting inputs based on outputs already received.
  – This suggests that we should protect the function from inputs and outputs seen thus far to future inputs.
• Following Wittbold and Johnson [1990] we call this function a *user strategy*.
  – Strategies are more general than inputs.
  – Like processes, they describe user behavior.

**Recap: what is a secure system?**

• Noninterference: low users must not be able to infer anything about high behavior, given low observations.
• Summing up:
  – “Behavior” = user strategy.
  – “Observations” = sequence of input/output events.
  – “Infer” = determine that one strategy is more likely than another, given observations seen and knowledge of program text.
• Now, let’s get formal.

**The interactive language**

We reason about simple while-programs:

\[
\begin{align*}
    e & ::= n \mid x \mid e_0 \text{ op } e_1 \\
    c & ::= \text{skip} \mid x := e \mid \\
    & \quad \text{input } x \text{ from } \tau \mid \\
    & \quad \text{output } e \text{ to } \tau \mid c_0 \text{ ; } c_1 \mid \\
    & \quad \text{if } e \text{ then } c_0 \text{ else } c_1 \mid \\
    & \quad \text{while } e \text{ do } c \mid c_0 \text{ [p] } c_1
\end{align*}
\]

**Event traces**

• As a program executes, it modifies the values of variables and produces events on channels.
• Event notation:
  – \text{in}(\tau,v): input of integer \(v\) on channel \(\tau\).
  – \text{out}(\tau,v): output of integer \(v\) on channel \(\tau\).
• A *trace* is a finite sequence of events:
  – Example: \(t = <\text{in}(H,0), \text{out}(L,1), \text{out}(H,1)>\)

**User strategies, more formally**

• Formally, a user strategy for channel \(\tau\) is a function from traces of events on \(\tau\) to inputs.
  – Trace restriction: write \(t \upharpoonright \tau\) to denote the subsequence of \(t\) comprising events on \(\tau\).
  – Example: \(<\text{in}(L,0), \text{out}(L,1), \text{out}(H,1) > | H = <\text{in}(L,0), \text{out}(L,1) > \)
  – Call \(t \upharpoonright L\) a “low trace” and \(t \upharpoonright H\) a “high trace.”
  – User strategies: functions from high/low traces to integers.
• We assume strategies are deterministic.
  – Probabilistic generalizations are straightforward.

**Language semantics**

• To model program execution we use:
  – A command \(c\).
  – A state \(\sigma\):
    • Maps from program variables to integer values.
  – A trace \(t\):
    • Of events that have occurred thus far.
  – A joint strategy \(\omega\):
    • Specifies a user strategy for each channel.
    • A function from channel names \(\tau\) to user strategies.
• These give us *configurations* \((c, \sigma, t, \omega)\).
  – Which take steps, according to standard operational rules (described in the paper).
Configurations emit traces

- Write \( m \rightarrow t \) to mean that configuration \( m \) can produce (“emit”) trace \( t \) as the program executes.
- Example:
  - \( c = \text{input } x \text{ from } H; \text{output } x \text{ to } L \)
  - \( \sigma \) is some arbitrary state
  - \( \varepsilon \) is the empty trace
  - strategy \( \omega (H) \) is to input 1
- Then \((c, \sigma, \varepsilon, \omega)\) emits two nonempty traces:
  - \(< \text{in}(H,1) >\)
  - \(< \text{in}(H,1), \text{out}(L,1) >\)

Formalizing noninterference

- Define observations with trace restriction:
  - If \( t \upharpoonright L = t' \upharpoonright L \), traces \( t \) and \( t' \) have the same subsequence of low events.
- Start with a definition for deterministic programs:

  Command \( c \) satisfies noninterference if:
  - For all \( m = (c, \sigma, \varepsilon, \omega) \) and \( m' = (c, \sigma, \varepsilon, \omega') \) such that \( \omega(L) = \omega'(L) \), and for all traces \( t \) such that \( m \rightarrow t \), there exists \( t' \) such that \( t \upharpoonright L = t' \upharpoonright L \) and \( m' \rightarrow t' \).

Probabilistic noninterference

- A configuration \( m \) gives us a probability measure \( \mu_m \) on execution sequences.
  - Details in the paper.
- Let \( \mathbb{E}_m(t) \) be the event that \( m \) emits a trace \( t' \) such that \( t \upharpoonright L = t' \upharpoonright L \).

Command \( c \) satisfies probabilistic noninterference if:
  - For all \( m = (c, \sigma, \varepsilon, \omega) \) and \( m' = (c, \sigma, \varepsilon, \omega') \) such that \( \omega(L) = \omega'(L) \), and all traces \( t \), we have \( \mu_m(\mathbb{E}_m(t)) = \mu_{m'}(\mathbb{E}_{m'}(t)) \).

One-time pad v2.0 is not secure

while (true) do
  \( x := 0 \) \([0.5]\)
  \( x := 1 \)
  output \( x \) to \( H \);
  input \( y \) from \( H \);
  output \( (x \text{ XOR } y) \) to \( L \)

\( \omega(H) \): transmit 0
\( x = 0 \) \([0.5]\)
\( x = 1 \)
\( y = 0 \)
\( y = 1 \)

\( \omega(H) \): transmit 1
\( x = 0 \) \([0.5]\)
\( x = 1 \)
\( y = 0 \)
\( y = 1 \)
One-time pad v2.0 is not secure

\[ x := 0 \quad \text{or} \quad x := 1; \]
\[ \text{output } x \text{ to } H; \]
\[ \text{input } y \text{ from } H; \]
\[ \text{output } (x \text{ XOR } y) \text{ to } L \]

ω(H): transmit 0
\[ \begin{array}{l}
0.5 \\
\text{x = 0} \\
\text{y = 0} \\
\text{t | L = out(L,0)}
\end{array} \]
\[ \begin{array}{l}
0.5 \\
\text{x = 1} \\
\text{y = 1} \\
\text{t | L = out(L,1)}
\end{array} \]

ω(H): transmit 1
\[ \begin{array}{l}
0.5 \\
\text{x = 0} \\
\text{y = 1} \\
\text{t | L = out(L,1)}
\end{array} \]
\[ \begin{array}{l}
0.5 \\
\text{x = 1} \\
\text{y = 0} \\
\text{t | L = out(L,0)}
\end{array} \]

What I didn’t tell you about

• We also handle nondeterministic choice.
  – Like probabilistic choice, but no numbers.
  • Models underspecified behavior like schedulers.
  – Noninterference under refinement rules out refinement attacks in programs with “compile-time” nondeterminism.
• We prove a result that a variant of VSI type system soundly enforces new conditions.
  – Including probabilistic noninterference.
  – More precise enforcement mechanisms should apply without much extra work.

Summary

• We give novel semantic security conditions for interactive, imperative programs.
• We extend definitions to nondeterministic programs:
  – With an explicit randomization command.
  – With compile-time nondeterminism.
• We present a new soundness result demonstrating feasibility of static enforcement mechanisms for the definitions.

Some related work

• Semantic conditions for interactive systems mostly limited to more abstract systems.
  – Process algebras and related formalisms:
    • Ryan & Schneider, Focardi & Gorrieri, Honda & Yoshida, Pottier, Zdancewic & Myers…
    • Preliminary work suggests our conditions equivalent to (probabilistic) NDC, given reasonable assumptions.
  – State-based and trace-based systems:
    • Goguen & Meseguer, McLean, Gray & Syverson, Mantel, Zakinthinos & Lee, Halpern & O’Neill…
  • Our work synthesizes PL-based work with strategy-based definitions of noninterference for interactive systems.

Why not “bridge the gap”??

• Idea: translate imperative programs to interactive setting, then reason about security:
  – E.g.: Honda & Yoshida; Mantel & Sabelfeld; Focardi, Rossi & Sabelfeld.
  – This kind of work is valuable.
    – Helpful to see connections between different threads of research.
    – Example: can use security checkers for process algebras to verify security of imperative programs.
• But doesn’t solve all our problems.
  – Current translations assume batch-job model.
  – With our system model, no “bridging” is necessary.

Future work

• Concurrent interactive programs:
  – Nondeterminism due to concurrency is tricky to model and to reason about.
  – Can extend ideas for batch-job programs.
• More powerful users/attackers.
  – Low users who see time when events occur.
• More accurate enforcement mechanisms.
  – E.g., relax assumption that high users always provide input.
• Applications to real languages like Jif and Flow Caml.