Information-Flow Security for Interactive Programs

Kevin O'Neill, Michael Clarkson, Stephen Chong
Cornell University

Computer Security Foundations Workshop
July 6, 2006

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.

Informal preview

- Take a straightforward sequential language:
  - `skip | x:=e | if e then` c_0 `else` c_1 `| while e do c`
- Assume programs can interact with *channels*, which have high or low confidentiality levels.
- Add commands for interaction with channels:
  - `input x from` τ
  - `output e to` τ
- Goal: define semantic security conditions for such a language.

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.

Imperative meets interactive

- Imperative usually implies "batch-job":
  - "Inputs" are initial variable values.
  - "Outputs" are final (& 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.

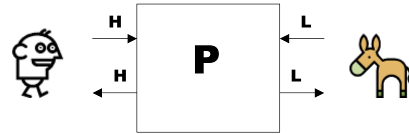
Our contributions

- 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.

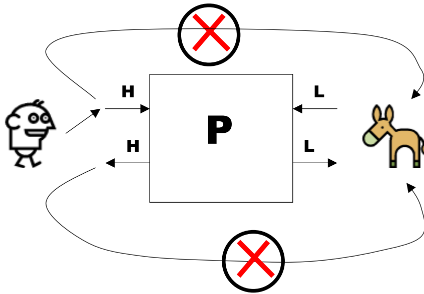
Our system model

- 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

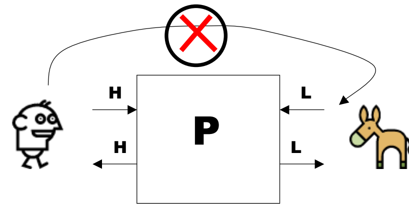


User interaction model



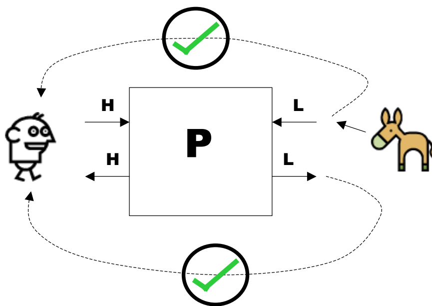
Low users can't observe high inputs or outputs.

User interaction model



High users can't input on low channel.

User interaction model



High users may observe low inputs and outputs directly.

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.

Secure interactive programs

- One-time pad encryption is secure:

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


One-time pad 2.0

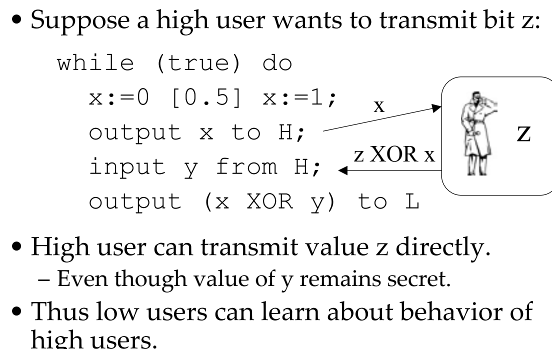
- What if we tell high users the one-time pad?

```
while (true) do
  x:=0 [0.5] x:=1;
  output x to H;
  input y from H;
  output (x 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:



- 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:

```
e ::= n | x | e0 op e1

c ::= skip | x:=e |
      input x from τ |
      output e to τ | c0 ; c1 |
      if e then c0 else c1 |
      while e do c | c0 [p] c1
```


Event traces

- As a program executes, it modifies the values of variables and produces events on channels.
- Event notation:
  - in(τ,v): input of integer v on channel τ.
  - out(τ,v): output of integer v on channel τ.
- A *trace* is a finite sequence of events:

  Example: $t = \langle$ in(H,0), out(L,1), out(H,1) $\rangle$

User strategies, more formally

- Formally, a user strategy for channel τ is a function from traces of events on τ to inputs.
  - Trace restriction: write $t \upharpoonright \tau$ to denote the subsequence of t comprising events on τ.
    - Example:
      $\langle$ in(H,0), out(L,1), out(H,1) $\rangle \upharpoonright$ H = $\langle$ in(H,0), out(H,1) $\rangle$
    - 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* σ:
    - Maps from program variables to integer values.
  - A trace *t:*
    - Of events that have occurred thus far.
  - A *joint strategy* ω:
    - Specifies a user strategy for each channel.
    - A function from channel names τ to user strategies.
- These give us *configurations* (c, σ, t, ω).
  - Which take steps, according to standard operational rules (described in the paper).

Configurations emit traces

- Write $m \rightsquigarrow t$ to mean that configuration m can produce ("emit") trace t as the program executes.
- Example:
  - c = `input x from H; output x to L`
  - σ is some arbitrary state
  - ε is the empty trace
  - strategy ω(H) is to input 1
- Then $(c, \sigma, \varepsilon, \omega)$ emits two nonempty traces:
  - < in(H,1) >
  - < in(H,1), 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 $m \rightsquigarrow t$,
  there exists t' such that $t \upharpoonright L = t' \upharpoonright L$ and $m' \rightsquigarrow t'$.

Probabilistic noninterference

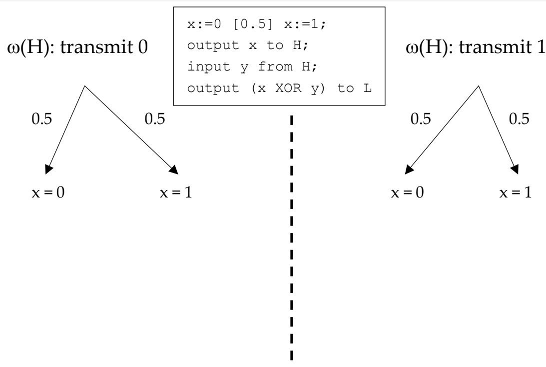
- A configuration m gives us a probability measure μ_m on execution sequences.
  - Details in the paper.
- Let $E_m(t)$ be the event that m emits a trace t' such that $t \upharpoonright L = t' \upharpoonright L$.

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 $\mu_m(E_m(t)) = \mu_{m'}(E_{m'}(t))$.

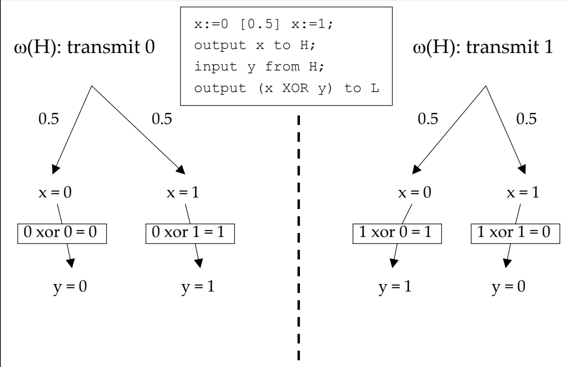
One-time pad v2.0 is not secure

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

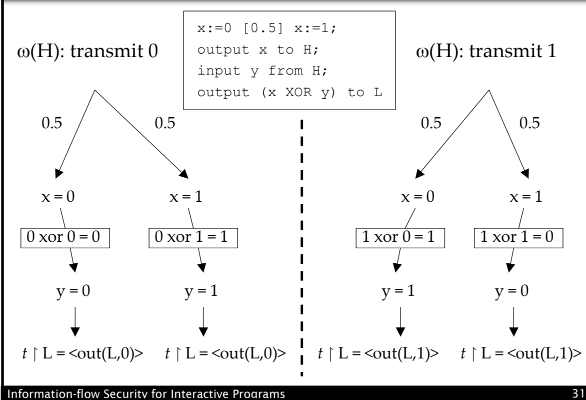

One-time pad v2.0 is not secure



One-time pad v2.0 is not secure



One-time pad v2.0 is not secure



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.