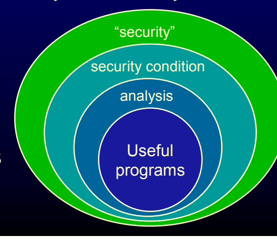
Observational Determinism for Concurrent Program Security

Steve Zdancewic   Andrew Myers
Computer Security Foundations Workshop 2003

Security conditions vs. analyses

- The security-typing game:
  1. An intuitive semantic security condition that guarantees behavior of program is secure
  2. A static program analysis (type system) that ensures the program obeys the security condition
- Useful if:
  - Security condition corresponds to desired security
  - Analysis permits interesting programs



Information flow in concurrent programs

- Various approaches have been tried (e.g., [AR80, SV98, HR98, SS00, S01, SM02, HY02])
- Problems:
  - Some analyses allow arguably insecure programs
  - Most analyses are highly restrictive
- This paper:
  - A more intuitively secure notion of security
  - A more permissive static analysis

Noninterference

- Definitions:

$\langle M,e \rangle$: a configuration (memory M, program e)

$\langle M,e \rangle \Downarrow T$: configuration $\langle M,e \rangle$ executes with result T

$T_1 \approx_L T_2$: 'low observer' at L can't distinguish results

$\langle M_1,e_1 \rangle \approx_L \langle M_2,e_2 \rangle$: can't distinguish inputs

- Noninterference:

$$\langle M_i,e_i \rangle \Downarrow T_i \;\Rightarrow T_1 \approx_L T_2$$

Nondeterminism

- Noninterference:

$$\langle M_1,e_1 \rangle \approx_L \langle M_2,e_2 \rangle \;\&\; \langle M_i, e_i \rangle \Downarrow T_i \;\Rightarrow\; T_1 \approx_L T_2$$

- Scheduler nondeterminism is critical to concurrency:

$$e_1 \mid e_2 \begin{array}{l} \nearrow e_1' \mid e_2 \\ \searrow e_1 \mid e_2' \end{array}$$

- But breaks noninterference:

$$\langle M,e \rangle \Downarrow T_1 \quad \langle M,e \rangle \Downarrow T_2 \quad T_1 \neq T_2$$

- Possibilistic generalizations [Suth86, McCu87, McLe90]: lift to sets of outcomes:

$$\{T \mid \langle M_1,e_1 \rangle \Downarrow T\} \approx_L \{T \mid \langle M_2,e_2 \rangle \Downarrow T\}$$

Possibilistic problems

```
l := false | l := true | l := h
```

- Random scheduling: 2/3 probability leak
- Sequential scheduling: ~1 probability leak
- High information communicated via scheduler
- Possibilistically "secure"

```
h = false ➔ {l = false, l = true}
h = true ➔ {l = false, l = true}
```

- Information "leaked" only if attacker is certain
- Nondeterminism doesn't work against the attacker!

Timing channels

- Time taken by program can reveal sensitive information
- Can be converted into storage channels
- Random scheduling: possibilistically "secure"
- One solution: consider time observable
- Problem: rejects secure sequential programs

```
x := true;
```

Effect: `l := h`

```
sleep(50);          if h then sleep(100)
l := x                   else skip;
                    x := false
```


Problems to solve

1. Possibilistic security: insecure
   - Need a stronger security condition that's not…
2. Ruling out all timing channels: too restrictive
   - Need a weaker security condition (& analysis)

1. A "new" security condition

- Observational determinism [McLe92, Rosc95]:

$$\langle M_i, e_i\rangle \Downarrow T_i \quad \Rightarrow \quad T_1 \approx_L T_2$$

- Any observable difference between outputs permits a refinement attack
- System may still be nondeterministic - depends on choice of T, $\approx_L$

2. Avoiding restrictiveness

- Idea: distinguish between internal and external timing channels
  - Internal: affect program data
  - External: affect only timing of external interactions

```
x := true;
```

```
sleep(50);          if h then sleep(100)
l := x                   else skip;
                    x := false
```

external only

Controlling internal channels

- Insight: Internal timing channels require races
- Write-write race:

```
l := false | l := true | l := h
```

- Read-write race:

```
sleep(50);          if h then sleep(100)
l := x                   else skip;
                    x := false
```

- Race = two memory accesses to same location, at least one a write, that can occur in either order
- Observational determinism ⇒ rule out races
- Nondeterminism ok at different locations

Limiting observational power

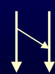
- Idea: capture invisibility of external timing channels in relation $T_1 \approx_L T_2$
- Result of concurrent computation is trace T of memory states $[M_1, M_2, M_3,…]$
- Projection of T onto location l is
  $T(l) = [M_1(l), M_2(l), …]$
- Traces are indistinguishable if they look the same at every memory location
  - Can't time updates
  - Can't time execution

$$T_1(l) = [v_1, v_1, v_2, v_3, v_3, v_4, \ldots]$$
$$\approx_L$$
$$T_2(l) = [v_1, v_2', v_2', v_3', v_3']$$

Synchronization

- Races considered harmful!
- Unsynchronized writes to shared memory unsafe
  - ⇒ need synchronization and communication mechanisms
- Our choice: message passing (blocking snd/rcv)

```
snd(c,v)    x := rcv(c);
```



- Supports non-block-structured communication
- Shared memory, but restricted to prevent unsynchronized communication

λ^{par}_{sec} A secure concurrent language

- Variant of the Join calculus [Fournet et al.]
  - Explicit message passing
  - High-level abstraction for synchronization
  - Similar to Milner's π-calculus
- Explicit state using ML-style references
  Use an alias analysis that prevents races
- Linear/Nonlinear channels
  - Adapted from linear continuations [ZM'02]
  - Regulates communication & synchronization between threads
  - See also [Honda & Yoshida '02]

λ^{par}_{sec} Details

```
J ::= f(x,y)   nonlinear channels
      f(x)     linear channels
      (J|J)    join patterns

P ::= let x = ref v in P      ref creation
   |  set v := v     in P     ref assignment
   |  let J ⊳ P      in P     chan. defn.
   |  let J ⊸ P      in P     lin. chan. defn.
   |  if v then P else P      conditional
   |  v(v,l)                  msg. send
   |  l(v)                    lin. msg. send
   |  (P|P)                   parallel comp.
   |  0                       inactive proc.
```


λ^{par}_{sec} Example

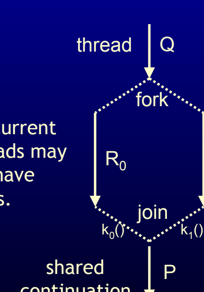
```
let double(c) ⊳ c() | c() in
let d() ⊳ P in
  P() | d()
```


λ^{par}_{sec} Synchronization

```
let in1(x) | in2(y) ⊳
  let z = x + y in out(z)
in
  out(7)
```


Linear channels

Concurrent threads may not have races.



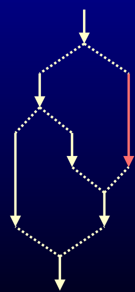

```
let k₀() | k₁() ⊸ P in
  Q;( (R₀; k₀())
    | (R₁; k₁())  )
```

Linear synchronization and race-freedom imply that that P learns no information about how the scheduler executed R_0 and R_1.

Linear message passing

The type system also permits rich kinds of synchronization behavior.

Previous type systems reject processes with this synchronization structure.
[Honda et al. '02]
[Pottier '02]
[Sabelfeld '01]



Depends on high-security data.

External channels

- Memory locations are externally observable
- Can encode external I/O channels
- Limited observational power
  $\Rightarrow$ external I/O channels can't be timed against each other

Shared memory vs message-passing

- Shared-memory programming model:
  - Common shared memory locations used for mutation, communication
  - Synchronization: locks/semaphores, condition variables
- Locks don't help!

```
l := false | l := true | l := h
```

- Shared-memory model is fundamentally uncongenial to information flow analysis

Compositionality

- Connecting secure programs with communication channels isn't secure in general
- Composition is in the language
  - Channels must agree on security labels
  - Composition must not introduce races

Future work

- Need a good race freedom analysis
  - Ideally, compositional (but what annotations?)
- Application to practical language (Jif?)
- Handle lock/semaphore synchronization