Dynamic Dependency Monitoring to Secure Information Flow

Mark Thober

Joint work with Paritosh Shroff and Scott F. Smith

Department of Computer Science

Johns Hopkins University
Motivation

- Information security is a critical requirement of software systems
  - Personal information, trade secrets, national security, etc.
- Static information flow systems are well-studied
- Run-time information flow systems have been considered highly impractical and sometimes impossible

**Goal:** A sound run-time information flow tracking system
Background: Secure Information Flow

- Objective: ensure confidential data is not exposed to unauthorized users

- Direct and indirect flows
  - Direct:
    \[
    x := h + 1;
    \]
  - Indirect:
    \[
    x := 1;
    \]
    \[
    \text{if } (h == 0) \text{ then } x := 0 \text{ else } ();
    \]
    The value of \( x \) encapsulates information about \( h \)

- We do not address termination, timing, or other covert channels
Why Dynamic Information Flow Security?

• Greater precision
Why Dynamic Information Flow Security?

- Greater precision
  - Reject insecure *executions*, not whole programs

```plaintext
x := 0;
if (l < 10) then x := h else ();
output(deref (x));
```
Why Dynamic Information Flow Security?

• Greater precision
  – Reject insecure *executions*, not whole programs

```plaintext
x := 0;
if (l < 10) then x := h else ();
output(deref (x));
```

* If \( l < 10 \) a leak occurs: this execution must be stopped
Why Dynamic Information Flow Security?

• Greater precision
  – Reject insecure executions, not whole programs

```plaintext
x := 0;
if (l < 10) then x := h else ();
output(deref (x));
```

* If \( l < 10 \) a leak occurs: this execution must be stopped
* If \( l \geq 10 \), no leak occurs: the execution may safely proceed
Why Dynamic Information Flow Security?

• Greater precision
  – Reject insecure *executions*, not whole programs
  – Flow- and path-sensitivity

\[
x := 0; \ y := 0;
\text{if } (l < 0) \text{ then } y := h \text{ else } ();\n\text{if } (l > 0) \text{ then } x := \text{deref} (y) \text{ else } ();\n\text{output} (\text{deref} (x));
\]

* No execution path exists where \( h \) flows into \( x \)
Why Dynamic Information Flow Security?

- Greater precision
  - Reject insecure executions, not whole programs
  - Flow- and path-sensitivity

```plaintext
x := 0; y := 0;
if (l < 0) then y := h else ();
if (l > 0) then x := deref(y) else ();
output(deref(x));
```

* No execution path exists where \( h \) flows into \( x \)
Why Dynamic Information Flow Security?

- Greater precision
  - Reject insecure *executions*, not whole programs
  - Flow- and path-sensitivity

```plaintext
x := 0; y := 0;
if (l < 0) then y := h else ();
if (l > 0) then x := deref (y) else ();
output(deref (x));
```

* No execution path exists where \( h \) flows into \( x \)
Why Dynamic Information Flow Security?

- Greater precision
- Dynamic Data Policies
Why Dynamic Information Flow Security?

- Greater precision

- Dynamic Data Policies
  - Static analyses can only approximate the security level; policy is part of the code
  - Dynamic tracking permits the policy to be a property of the data
  - Programs are easily used in different security domains, as the policy is not tied to the code
Why Dynamic Information Flow Security?

- Greater precision

- Dynamic Data Policies

- Dynamic Languages (e.g. Perl, Javascript)
  - Fundamentally dynamic operations cannot ever be tracked by any static system and so the dynamic approach is the only alternative
Direct Flows are Easy to Track

- All direct flow paths will be taken at run-time

- Simple run-time labeling can account for these flows

\[ x := h + 1; \]

- If \( h \) is labeled high, then the + operation passes along this label, which is further propagated to the location \( x \)
Challenge: Indirect Flows

- Run-time execution only takes one path
- But indirect flows arise due to branching, requiring analysis of all paths

```plaintext
x := 1;
if (h == 0) then x := 0 else ();
output(deref(x));
```
Challenge: Indirect Flows

- Run-time execution only takes one path
- But indirect flows arise due to branching, requiring analysis of all paths

```cpp
x := 1;
if (h == 0) then x := 0 else ();
output(deref(x));
```

- If \( h == 0 \), we can capture the indirect flow since the assignment occurs under a high guard
Challenge: Indirect Flows

- Run-time execution only takes one path

- But indirect flows arise due to branching, requiring analysis of all paths

```plaintext
x := 1;
if (h == 0) then x := 0 else ();
output(deref(x));
```

- If \( h \neq 0 \), we cannot dynamically capture the indirect flow since no assignment occurs under \( h \), yet a leak still occurs

- *Implicit indirect flow* leaks occur due to paths not taken
Challenge: Indirect Flows

• Run-time execution only takes one path

• But indirect flows arise due to branching, requiring analysis of all paths

```
x := 1;
if (h == 0) then x := 0 else ();
output(deref (x));
```

If we could indicate that the data in \(x\) *always* depends on \(h\), we could soundly track information flows at run-time.
Dynamic Dependencies

\begin{verbatim}
x := 1;
if (h == 0) then x := 0 else ()
deref (x);
\end{verbatim}

- How do we indicate that the data in \(x\) depends on \(h\)?
  - Really, the data in \(x\) depends on the data in the conditional branch

- How can we capture this dependency in runs where \(x\) is not assigned under \(h\)?
Dynamic Dependencies

```plaintext
x := 1;
if (h == 0) then x := 0 else ()

deref (x);
```

- Observe: Assignments are manifested at dereference
  - We can leverage this to track flows in both runs
Dynamic Dependencies

x := 1;
if (h == 0) then x := 0 else ()

deref (x);

• Solution: Relate the security level of data to syntactic program points
Dynamic Dependencies

```plaintext
x := 1;
if p_1 (h == 0) then x := 0 else ()
deref p_2 (x);
```

- Solution: Relate the security level of data to syntactic program points

The dependencies are already there, we’re just tracking them.
Dynamic Dependencies

\[
x := 1; \\
\text{if}_{p_1} (h == 0) \text{ then } x := 0 \text{ else } () \\
deref_{p_2} (x);
\]

- Solution: Relate the security level of data to syntactic program points
  - The information at the conditional \( p_1 \) is \text{High} (\( p_1 \leftrightarrow \text{High} \))
  - Since \( x \) is assigned under this conditional, then dereferenced, \( p_2 \) depends on \( p_1 \) (\( p_2 \leftrightarrow p_1 \))
  - Since \( p_2 \leftrightarrow p_1 \) and \( p_1 \leftrightarrow \text{High} \), transitively \( p_2 \leftrightarrow \text{High} \)
  Hence the value read from \( x \) must be \text{High}
Capturing Dependencies

• Maintain a cache of program point dependencies that persists across runs (2 options)

  1. Build the cache dynamically, as the program runs
     – Precise, but leaks are possible in early runs, before all the dependencies are observed

  2. Pre-compute a cache statically
     – Sound, but static analysis will be conservative

• Maintain a cache of security labels that is local to the current run
Language

- Higher-order λ-calculus with mutable state
  - With let-expressions, conditionals, binary operations, ints and bools
- Conditionals, application sites, and dereference points are marked with program point identifiers
  - $\text{if}_p \ e \ \text{then} \ e \ \text{else} \ e \ | \ e(p(e)) \ | \ \text{deref}_p \ e$
- Values are labeled with a set of program points, $P$, and a security level $L$
  - $\langle v^L, P \rangle$, for example $\langle 5^{\text{Low}}, \{p_1, p_3\} \rangle$
- A dependency cache maps program points to sets of program points, $\{p \mapsto P\}$
- A direct flow cache maps program points to security levels, $\{p \mapsto L\}$
- Run-time information flow monitoring defined via operational semantics
Dynamically Capturing Dependencies

\[
x := 1;
\text{if}_{p_1} (h == 0) \text{ then } x := 0 \text{ else } ()
deref_{p_2} (x);
\]

<table>
<thead>
<tr>
<th>Run 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value of ( h )</td>
<td>( 0^{\text{High}} )</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{}</td>
</tr>
<tr>
<td>heap</td>
<td>{}</td>
</tr>
<tr>
<td>final value</td>
<td></td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

\[
x := 1; \\
\text{if } p_1(h == 0) \text{ then } x := 0 \text{ else } () \\
deref_{p_2}(x);
\]

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
<td>(0^{\text{High}})</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{}</td>
</tr>
<tr>
<td>heap</td>
<td>{x \mapsto (1^{\text{Low}}, {})}</td>
</tr>
<tr>
<td>final value</td>
<td></td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

\[
x := 1; \\
\text{if}_p (h == 0) \text{ then } x := 0 \text{ else } () \\
\text{deref}_p (x);
\]

<table>
<thead>
<tr>
<th>value of h</th>
<th>0^{\text{High}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>{}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{p_1 \leftrightarrow \text{High}}</td>
</tr>
<tr>
<td>heap</td>
<td>{x \leftrightarrow \langle 1^{\text{Low}}, {}\rangle}</td>
</tr>
<tr>
<td>final value</td>
<td></td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

\[
x := 1;
\]
\[
\text{if } p_1 (h == 0) \text{ then } x := 0 \text{ else } ()
\]
\[
deref_{p_2} (x);
\]

<table>
<thead>
<tr>
<th>value of ( h )</th>
<th>( 0^{\text{High}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>{}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{\textit{p}_1 \mapsto \text{High}}</td>
</tr>
<tr>
<td>heap</td>
<td>{x \mapsto \langle 0^{\text{Low}}, {\textit{p}_1} \rangle}</td>
</tr>
<tr>
<td>final value</td>
<td></td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

\[
x := 1;
\text{if}_{p_1}(h == 0) \text{ then } x := 0 \text{ else } ()
\]
\[
deref_{p_2}(x);
\]

<table>
<thead>
<tr>
<th>Run 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of ( h )</td>
</tr>
<tr>
<td>dependency cache</td>
</tr>
<tr>
<td>direct flow cache</td>
</tr>
<tr>
<td>heap</td>
</tr>
<tr>
<td>final value</td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

```plaintext
x := 1;
if \( p_1 \) (h == 0) then x := 0 else ()
deref_{p_2} (x);
```

<table>
<thead>
<tr>
<th>value of h</th>
<th>0^{High}</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>( {p_2 \leftarrow p_1} )</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>( {p_1 \leftarrow \text{High}} )</td>
</tr>
<tr>
<td>heap</td>
<td>( {x \leftarrow \langle 0^{\text{Low}}, {p_1}\rangle} )</td>
</tr>
<tr>
<td>final value</td>
<td>0^{High}</td>
</tr>
</tbody>
</table>

Transitively
Dynamically Capturing Dependencies

```plaintext
x := 1;
if_{p_1} (h == 0) then x := 0 else ()
deref_{p_2} (x);
```

---

**Run 1**

<table>
<thead>
<tr>
<th>value of h</th>
<th>0\text{High}</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>{p_2 \mapsto p_1}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{p_1 \mapsto \text{High}}</td>
</tr>
<tr>
<td>heap</td>
<td>{x \mapsto \langle 0^\text{Low}, {p_1}\rangle}</td>
</tr>
<tr>
<td>final value</td>
<td>0\text{High}</td>
</tr>
</tbody>
</table>

---

**Leak Detected!**
Dynamically Capturing Dependencies

```plaintext
x := 1;
if \text{p}_1 (h == 0) \text{ then } x := 0 \text{ else } ()
\text{deref}_{\text{p}_2} (x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of (h)</td>
<td>0^{\text{High}}</td>
<td>1^{\text{High}}</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{ \text{p}_2 \mapsto \text{p}_1 }</td>
<td>{ \text{p}_2 \mapsto \text{p}_1 }</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{ \text{p}_1 \mapsto \text{High} }</td>
<td>{}</td>
</tr>
<tr>
<td>heap</td>
<td>{ x \mapsto \langle 0^{\text{Low}}, { \text{p}_1 } \rangle }</td>
<td>{}</td>
</tr>
<tr>
<td>final value</td>
<td>0^{\text{High}}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

```plaintext
x := 1;
if \(p_1 (h == 0)\) then x := 0 else ()
deref_{p_2} (x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of (h)</td>
<td>0^{High}</td>
<td>1^{High}</td>
</tr>
<tr>
<td>dependency cache</td>
<td>({p_2 \mapsto p_1})</td>
<td>({p_2 \mapsto p_1})</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>({p_1 \mapsto \text{High}})</td>
<td>({})</td>
</tr>
<tr>
<td>heap</td>
<td>({x \mapsto \langle 0^{\text{Low}}, {p_1}\rangle})</td>
<td>({x \mapsto \langle 1^{\text{Low}}, {}\rangle})</td>
</tr>
<tr>
<td>final value</td>
<td>0^{High}</td>
<td></td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

```plaintext
x := 1;
if p₁ (h == 0) then x := 0 else ()
deref₂ (x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
<td>0^{High}</td>
<td>1^{High}</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{p₂ ↦ p₁}</td>
<td>{p₂ ↦ p₁}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{p₁ ↦ \text{High}}</td>
<td>{p₁ ↦ \text{High}}</td>
</tr>
<tr>
<td>heap</td>
<td>{x ↦ \langle 0^{Low}, {p₁}⟩}</td>
<td>{x ↦ \langle 1^{Low}, {}}⟩</td>
</tr>
<tr>
<td>final value</td>
<td>0^{High}</td>
<td></td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

```plaintext
x := 1;
if \( p_1(h == 0) \) then x := 0 else ()
deref_{p_2}(x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
<td>( 0^{\text{High}} )</td>
<td>( 1^{\text{High}} )</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{ ( p_2 \mapsto p_1 ) }</td>
<td>{ ( p_2 \mapsto p_1 ) }</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{ ( p_1 \mapsto \text{High} ) }</td>
<td>{ ( p_1 \mapsto \text{High} ) }</td>
</tr>
<tr>
<td>heap</td>
<td>{ x \mapsto \langle 0^{\text{Low}}, { p_1 } \rangle }</td>
<td>{ x \mapsto \langle 1^{\text{Low}}, { } \rangle }</td>
</tr>
<tr>
<td>final value</td>
<td>( 0^{\text{High}} )</td>
<td></td>
</tr>
</tbody>
</table>

Heap:
- \( x \mapsto \langle 0^{\text{Low}}, \{ p_1 \} \rangle \)
- \( x \mapsto \langle 1^{\text{Low}}, \{ \} \rangle \)
## Dynamically Capturing Dependencies

```plaintext
x := 1;
if \( p_1(h == 0) \) then x := 0 else ()
deref_{p_2}(x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of ( h )</td>
<td>( 0^{\text{High}} )</td>
<td>( 1^{\text{High}} )</td>
</tr>
<tr>
<td>dependency cache</td>
<td>( { p_2 \mapsto p_1 } )</td>
<td>( { p_2 \mapsto p_1 } )</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>( { p_1 \mapsto \text{High} } )</td>
<td>( { p_1 \mapsto \text{High} } )</td>
</tr>
<tr>
<td>heap</td>
<td>( { x \mapsto \langle 0^{\text{Low}}, { p_1 } \rangle } )</td>
<td>( { x \mapsto \langle 1^{\text{Low}}, { } \rangle } )</td>
</tr>
<tr>
<td>final value</td>
<td>( 0^{\text{High}} )</td>
<td>( \langle 1^{\text{Low}}, { p_2 } \rangle )</td>
</tr>
</tbody>
</table>
Dynamically Capturing Dependencies

```plaintext
x := 1;
if p₁ (h == 0) then x := 0 else ()
deref p₂ (x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
<td>$0^{\text{High}}$</td>
<td>$1^{\text{High}}$</td>
</tr>
<tr>
<td>dependency cache</td>
<td>${ p₂ \mapsto p₁ }$</td>
<td>${ p₂ \mapsto p₁ }$</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>${ x \mapsto \langle 0^{\text{Low}}, { p₁ } \rangle }$</td>
<td>${ p₁ \mapsto \text{High} }$</td>
</tr>
<tr>
<td>heap</td>
<td>${ x \mapsto \langle 0^{\text{Low}}, { p₁ } \rangle }$</td>
<td>${ x \mapsto \langle 1^{\text{Low}}, { } \rangle }$</td>
</tr>
<tr>
<td>final value</td>
<td>$0^{\text{High}}$</td>
<td>$1^{\text{High}}$</td>
</tr>
</tbody>
</table>

Transitively
Dynamically Capturing Dependencies

```plaintext
x := 1;
if \( p_1 (h == 0) \) then x := 0 else ()
deref_{p_2}(x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>value of h</strong></td>
<td>( 0^{High} )</td>
<td>( 1^{High} )</td>
</tr>
<tr>
<td><strong>dependency cache</strong></td>
<td>{p_2 \mapsto p_1}</td>
<td>{p_2 \mapsto p_1}</td>
</tr>
<tr>
<td><strong>direct flow cache</strong></td>
<td>{x \mapsto \langle 0^{Low}, {p_1} \rangle}</td>
<td>{p_1 \mapsto \text{High}}</td>
</tr>
<tr>
<td><strong>heap</strong></td>
<td>( 0^{High} )</td>
<td>{x \mapsto \langle 1^{Low}, {} \rangle}</td>
</tr>
<tr>
<td><strong>final value</strong></td>
<td>( 0^{High} )</td>
<td>( 1^{High} )</td>
</tr>
</tbody>
</table>

Transitively

Leak Detected!
What if the Order of Execution is Reversed?

- Executing the `then` branch before the `else` branch allowed us to catch both leaks

- What happens if we execute the `else` branch first?
In Reverse Order

\[ x := 1; \]
\[ \text{if}_{p_1}(h == 0) \text{ then } x := 0 \text{ else } () \]
\[ \text{deref}_{p_2}(x); \]

<table>
<thead>
<tr>
<th>Run 1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
</tr>
<tr>
<td>dependency cache</td>
</tr>
<tr>
<td>direct flow cache</td>
</tr>
<tr>
<td>heap</td>
</tr>
<tr>
<td>final value</td>
</tr>
</tbody>
</table>
In Reverse Order

\[ x := 1; \]
\[ \text{if}_{p_1} (h == 0) \text{ then } x := 0 \text{ else } () \]
\[ \text{deref}_{p_2} (x); \]

\[ \begin{array}{|c|}
\hline
\text{value of } h & 1^{\text{High}} \\
\text{dependency cache} & \{\} \\
\text{direct flow cache} & \{p_1 \mapsto \text{High}\} \\
\text{heap} & \{x \mapsto \langle 1^{\text{Low}}, \{\}\rangle\} \\
\text{final value} & \langle 1^{\text{Low}}, \{p_2\}\rangle \\
\hline
\end{array} \]
In Reverse Order

```plaintext
x := 1;
if \text{if}_{p_1}(h == 0) \text{ then } x := 0 \text{ else } ()
deref_{p_2}(x);
```

<table>
<thead>
<tr>
<th>value of ( h )</th>
<th>( 1^{\text{High}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>{ }</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{ p_1 \mapsto \text{High} }</td>
</tr>
<tr>
<td>heap</td>
<td>{ x \mapsto \langle 1^{\text{Low}}, { } \rangle }</td>
</tr>
<tr>
<td>final value</td>
<td>( 1^{\text{Low}} )</td>
</tr>
</tbody>
</table>
In Reverse Order

\[\begin{align*}
x &:= 1; \\
\text{if}_{p_1}(h == 0) \text{ then } x &:= 0 \text{ else } () \\
deref_{p_2}(x);
\end{align*}\]

| **Run 1a** | \\
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value of (h)</td>
<td>(1^{\text{High}})</td>
</tr>
<tr>
<td>dependency cache</td>
<td>({})</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>({p_1 \mapsto \text{High}})</td>
</tr>
<tr>
<td>heap</td>
<td>({x \mapsto \langle 1^{\text{Low}}, {}\rangle})</td>
</tr>
<tr>
<td>final value</td>
<td>(1^{\text{Low}})</td>
</tr>
</tbody>
</table>

Leak NOT detected!
In Reverse Order

\[ x := 1; \]
\[ \text{if}_{p_1} (h == 0) \text{ then } x := 0 \text{ else } () \]
\[ \text{deref}_{p_2} (x); \]

<table>
<thead>
<tr>
<th></th>
<th>Run 1a</th>
<th>Run 2a</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
<td>(1^{\text{High}})</td>
<td>(0^{\text{High}})</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{p_1 \mapsto \text{High}}</td>
<td>{}</td>
</tr>
<tr>
<td>heap</td>
<td>{x \mapsto \langle 1^{\text{Low}}, {}\rangle}</td>
<td>{}</td>
</tr>
<tr>
<td>final value</td>
<td>(1^{\text{Low}})</td>
<td></td>
</tr>
</tbody>
</table>
In Reverse Order

\[
x := 1; \\
\text{if}_{p_1}(h == 0) \text{ then } x := 0 \text{ else } () \\
deref_{p_2}(x);
\]

<table>
<thead>
<tr>
<th>value of h</th>
<th>Run 1a</th>
<th>Run 2a</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>(1^\text{High})</td>
<td>(0^\text{High})</td>
</tr>
<tr>
<td>dependency cache</td>
<td>({})</td>
<td>({p_2 \mapsto p_1})</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>({p_1 \mapsto \text{High}})</td>
<td>({p_1 \mapsto \text{High}})</td>
</tr>
<tr>
<td>heap</td>
<td>({x \mapsto \langle 1^\text{Low}, {}\rangle})</td>
<td>({x \mapsto \langle 0^\text{Low}, {p_1}\rangle})</td>
</tr>
<tr>
<td>final value</td>
<td>(1^\text{Low})</td>
<td>(\langle 0^\text{Low}, {p_2}\rangle)</td>
</tr>
</tbody>
</table>
In Reverse Order

\begin{align*}
x & := 1; \\
\text{if } p_1(h == 0) \text{ then } x := 0 \text{ else } () \\
deref_{p_2}(x);
\end{align*}

<table>
<thead>
<tr>
<th></th>
<th>Run 1a</th>
<th>Run 2a</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of (h)</td>
<td>(1^{\text{High}})</td>
<td>(0^{\text{High}})</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{}</td>
<td>{(p_2 \mapsto p_1}}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{(p_1 \mapsto \text{High}}}</td>
<td>{(p_1 \mapsto \text{High}}}</td>
</tr>
<tr>
<td>heap</td>
<td>{x \mapsto \langle 1^{\text{Low}}, {}\rangle}</td>
<td>{x \mapsto \langle 0^{\text{Low}}, {p_1}\rangle}</td>
</tr>
<tr>
<td>final value</td>
<td>(1^{\text{Low}})</td>
<td>(0^{\text{High}})</td>
</tr>
</tbody>
</table>
In Reverse Order

\[
x := 1;
\]
if \(p_1 (h == 0)\) then \(x := 0\) else ()

deref_{p_2} (x);

<table>
<thead>
<tr>
<th></th>
<th>Run 1a</th>
<th>Run 2a</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of (h)</td>
<td>(1^{\text{High}})</td>
<td>(0^{\text{High}})</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{}</td>
<td>({p_2 \mapsto p_1})</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>({p_1 \mapsto \text{High}})</td>
<td>({p_1 \mapsto \text{High}})</td>
</tr>
<tr>
<td>heap</td>
<td>({x \mapsto \langle 1^{\text{Low}}, {}\rangle})</td>
<td>({x \mapsto \langle 0^{\text{Low}}, {p_1}\rangle})</td>
</tr>
<tr>
<td>final value</td>
<td>(1^{\text{Low}})</td>
<td>(0^{\text{High}})</td>
</tr>
</tbody>
</table>

Leak Detected!
Important Observations

• The size of the dependency cache is important
  – Missing dependencies may permit leaks

• The ordering of executions matters
  – Only certain trouble-some orderings cause leaks, where the
    branch without the assignment is executed first
Partial Dynamic Noninterference

**Theorem 1.** If $r_1$ and $r_2$ are two runs of a program that both terminate and differ only in high inputs, and both runs result in values labeled low, then these values are identical.

**Proof.** By bisimulation of the low computation.
What if we want Full Noninterference?

- In many cases, no leaks can be tolerated!
Use a Fixed Point Dependency Cache

• A cache that contains all the program dependencies, and will never grow during computation

• How to find a Fixed Point Dependency Cache?
  – Through testing
    ♦ Will be more precise, but it is undecidable in general to always be sure all dependencies are captured
  – With a static analysis
    ♦ Will contain all dependencies, but be conservative
Use a Fixed Point Dependency Cache

• A cache that contains all the program dependencies, and will never grow during computation

• How to find a Fixed Point Dependency Cache?
  – Through testing
    ✴ Will be more precise, but it is undecidable in general to always be sure all dependencies are captured
  – With a static analysis
    ✴ Will contain all dependencies, but be conservative

• Still better than a completely static system, due to run-time precision, dynamic policies, etc.
Use a Fixed Point Dependency Cache

- A cache that contains all the program dependencies, and will never grow during computation

- How to find a Fixed Point Dependency Cache?
  - Through testing
    * Will be more precise, but it is undecidable in general to always be sure all dependencies are captured
  - With a static analysis
    * Will contain all dependencies, but be conservative

- Still better than a completely static system, due to run-time precision, dynamic policies, etc.

See paper for details.
AllLeaksDetectedwithFullCache

\[
x := 1;
\]

\[
\text{if } p_1(h == 0) \text{ then } x := 0 \text{ else } ()
\]

deref_{p_2}(x);

<table>
<thead>
<tr>
<th></th>
<th>Run 1b</th>
<th>Run 2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
<td>$1^{\text{High}}$</td>
<td>$0^{\text{High}}$</td>
</tr>
<tr>
<td>dependency cache</td>
<td>${p_2 \leftrightarrow p_1}$</td>
<td>${p_2 \leftrightarrow p_1}$</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>${p_1 \leftrightarrow \text{High}}$</td>
<td>${p_1 \leftrightarrow \text{High}}$</td>
</tr>
<tr>
<td>heap</td>
<td>${x \leftrightarrow \langle 1^{\text{Low}}, {}\rangle}$</td>
<td>${x \leftrightarrow \langle 0^{\text{Low}}, {p_1}\rangle}$</td>
</tr>
<tr>
<td>final value</td>
<td>$1^{\text{High}}$</td>
<td>$0^{\text{High}}$</td>
</tr>
</tbody>
</table>
AllLeaks Detected withFull Cache

```plaintext
x := 1;
if p1 (h == 0) then x := 0 else ()
deref_p2 (x);
```

<table>
<thead>
<tr>
<th></th>
<th>Run 1b</th>
<th>Run 2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of h</td>
<td>(1^\text{High})</td>
<td>(0^\text{High})</td>
</tr>
<tr>
<td>dependency cache</td>
<td>{p_2 \mapsto p_1}</td>
<td>{p_2 \mapsto p_1}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{p_1 \mapsto \text{High}}</td>
<td>{p_1 \mapsto \text{High}}</td>
</tr>
<tr>
<td>heap</td>
<td>{x \mapsto \langle1^{\text{Low}}, {}\rangle}</td>
<td>{x \mapsto \langle0^{\text{Low}}, {p_1}\rangle}</td>
</tr>
<tr>
<td>final value</td>
<td>(1^\text{High})</td>
<td>(0^\text{High})</td>
</tr>
</tbody>
</table>

Leak Detected! Leak Detected!
Better Precision at Run-time

\[ \text{Run 1, with Fixed Point Dependency Cache} \]

<table>
<thead>
<tr>
<th>value of ( l, h )</th>
<th>(-1^{\text{Low}}, 1^{\text{High}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>{( p_3 \mapsto {p_1, p_2}, p_4 \mapsto {p_2, p_3}}}</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{( p_1 \mapsto \text{Low}, p_2 \mapsto \text{Low}}}</td>
</tr>
<tr>
<td>heap</td>
<td>{( x \mapsto \langle 0^{\text{Low}}, {}\rangle, y \mapsto \langle 1^{\text{High}}, {p_1}\rangle}}</td>
</tr>
<tr>
<td>final value</td>
<td>(\langle 0^{\text{Low}}, {p_4}\rangle)</td>
</tr>
</tbody>
</table>
Better Precision at Run-time

```plaintext
x := 0; y := 0;
if \( p_1 (l < 0) \) then y := h else ()
if \( p_2 (l > 0) \) then x := deref_{p_3} (y) else ()
deref_{p_4} (x);
```

<table>
<thead>
<tr>
<th>Run 1, with Fixed Point Dependency Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of ( l, h )</td>
</tr>
<tr>
<td>dependency cache</td>
</tr>
<tr>
<td>direct flow cache</td>
</tr>
<tr>
<td>heap</td>
</tr>
<tr>
<td>final value</td>
</tr>
</tbody>
</table>
Better Precision at Run-time

\[ x := 0; \ y := 0; \]
\[ \text{if}_{p_1}(l < 0) \ \text{then} \ y := h \ \text{else} \ () \]
\[ \text{if}_{p_2}(l > 0) \ \text{then} \ x := \text{deref}_{p_3}(y) \ \text{else} \ () \]
\[ \text{deref}_{p_4}(x); \]

Run 1, with Fixed Point Dependency Cache

<table>
<thead>
<tr>
<th>value of ( l, h )</th>
<th>(-1^{\text{Low}}, 1^{\text{High}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>{ \text{p}_3 \mapsto { \text{p}_1, \text{p}_2 }, \text{p}_4 \mapsto { \text{p}_2, \text{p}_3 } }</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>{ \text{p}_1 \mapsto \text{Low}, \text{p}_2 \mapsto \text{Low} }</td>
</tr>
<tr>
<td>heap</td>
<td>{ x \mapsto \langle 0^{\text{Low}}, {} \rangle, y \mapsto \langle 1^{\text{High}}, { \text{p}_1 } \rangle }</td>
</tr>
<tr>
<td>final value</td>
<td>( 0^{\text{Low}} )</td>
</tr>
</tbody>
</table>
Better Precision at Run-time

```plaintext
x := 0; y := 0;
if \( p_1 (l < 0) \) then y := h else ()
if \( p_2 (l > 0) \) then x := \text{deref}_{p_3} (y) \) else ()
deref_{p_4} (x);
```

---

### Run 1, with Fixed Point Dependency Cache

<table>
<thead>
<tr>
<th>value of l, h</th>
<th>(-1^{\text{Low}}, 1^{\text{High}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency cache</td>
<td>( {p_3 \mapsto {p_1, p_2}, p_4 \mapsto {p_2, p_3}} )</td>
</tr>
<tr>
<td>direct flow cache</td>
<td>( {p_1 \mapsto \text{Low}, p_2 \mapsto \text{Low}} )</td>
</tr>
<tr>
<td>heap</td>
<td>( {x \mapsto \langle 0^{\text{Low}}, {} \rangle, y \mapsto \langle 1^{\text{High}}, {p_1} \rangle } )</td>
</tr>
<tr>
<td>final value</td>
<td>( 0^{\text{Low}} )</td>
</tr>
</tbody>
</table>

---

No False Positive!
Dynamic Noninterference

**Theorem 2.** If $r_1$ and $r_2$ are two runs of a program that begin with a fixed point of dependencies, both terminate, and differ only in high inputs, and either run results in a value labeled low, then both runs result in low values, and these values are identical.

**Proof.** Follows from Partial Dynamic Noninterference result, and Definition of a Fixed Point Dependency Cache.
Bonus: Static Noninterference

- A sound run-time system is a perfect set-up for Static Noninterference

- Static Noninterference can now be proved directly by Subject Reduction over the labelled semantics, using the Dynamic Noninterference property
Related Work

- Le Guernic et. al.
  - Label tracking in a small imperative language with while-loops, conditionals, and assignment
  - Uses a static analysis at run-time to discover flows in branches not taken

- A few hybrid systems that track direct flows at run-time and use a pre-process analysis for indirect flows
  - Not interprocedural, and no proofs

- Many other works on dynamic aspects of information flow
Future Work

• Improve the current system
  – Interactive IO, exceptions, etc.

• Efficiency
  – Precomputing cache closure, soft-typing, etc.

• Declassification

• Dynamic policy changes

• Run-time auditing

• Other dependency-related problems
  – Slicing, optimization, debugging
Conclusion

- A sound, run-time dependency tracking system for monitoring direct and indirect information flows
  - Dependencies can be captured dynamically or approximated statically
- Provides increased precision and dynamically defined policies
- New proof technique for dynamic (and static) noninterference
- Much more work to be done on dynamic information flow tracking!!