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

```
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

```
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

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

- $\ast$ If `l` $<$ 10 a leak occurs: this execution must be stopped
- $\ast$ 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;
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

```
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

```
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

```
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

```
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

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

  - If `h != 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

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

- 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

```
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

`x := 1;`
`if`$_{\mathbf{p_1}}$ `(h == 0) then x := 0 else ()`
`deref`$_{\mathbf{p_2}}$ `(x);`

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

The dependencies are already there, we're just tracking them.

Dynamic Dependencies

$$\texttt{x := 1;}$$
$$\texttt{if}_{\mathbf{p_1}}\ \texttt{(h == 0) then x := 0 else ()}$$
$$\texttt{deref}_{\mathbf{p_2}}\ \texttt{(x);}$$

- Solution: Relate the security level of data to syntactic program points
  - The information at the conditional $\mathbf{p_1}$ is High ($\mathbf{p_1} \mapsto \mathrm{High}$)
  - Since x is assigned under this conditional, then dereferenced, $\mathbf{p_2}$ depends on $\mathbf{p_1}$ ($\mathbf{p_2} \mapsto \mathbf{p_1}$)
  - Since $\mathbf{p_2} \mapsto \mathbf{p_1}$ and $\mathbf{p_1} \mapsto \mathrm{High}$, transitively $\mathbf{p_2} \mapsto \mathrm{High}$ Hence the value read from x must be 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
  - $\mathtt{if}_{\mathbf{p}}\ \mathtt{e}\ \mathtt{then}\ \mathtt{e}\ \mathtt{else}\ \mathtt{e} \mid \mathtt{e}\ (\mathtt{e})_{\mathbf{p}} \mid \mathtt{deref}_{\mathbf{p}}\ \mathtt{e}$
- Values are labeled with a set of program points, P, and a security level L
  - $\langle \mathrm{v}^{\mathrm{L}}, \mathrm{P} \rangle$, for example $\langle 5^{\mathtt{Low}}, \{\mathbf{p_1}, \mathbf{p_3}\} \rangle$
- A dependency cache maps program points to sets of program points, $\{\overline{\mathbf{p} \mapsto \mathrm{P}}\}$
- A direct flow cache maps program points to security levels, $\{\overline{\mathbf{p} \mapsto \mathrm{L}}\}$
- Run-time information flow monitoring defined via operational semantics

Dynamically Capturing Dependencies

```
x := 1;
$\texttt{if}_{\mathbf{p_1}}$ (h == 0) then x := 0 else ()
$\texttt{deref}_{\mathbf{p_2}}$ (x);
```

	Run 1
value of h	$0^{\mathtt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\}$
heap	$\{\}$
final value	

Dynamically Capturing Dependencies

$\texttt{x := 1;}$
$\texttt{if}_{\mathbf{p_1}}\ \texttt{(h == 0) then x := 0 else ()}$
$\texttt{deref}_{\mathbf{p_2}}\ \texttt{(x);}$

	Run 1
value of $\texttt{h}$	$0^{\texttt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\}$
heap	$\{\texttt{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$
final value	

Dynamically Capturing Dependencies

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p_1}}\ \texttt{(h == 0)}\ \texttt{then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p_2}}\texttt{(x);}$

	Run 1
value of h	$0^{\texttt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \texttt{High}\}$
heap	$\{\texttt{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$
final value	

Dynamically Capturing Dependencies

$\mathtt{x\ :=\ 1;}$

$\mathtt{if}_{\mathbf{p_1}}\ \mathtt{(h\ ==\ 0)\ then\ x\ :=\ 0\ else\ ()}$

$\mathtt{deref}_{\mathbf{p_2}}\mathtt{(x);}$

	Run 1
value of $\mathtt{h}$	$0^{\mathtt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \mathtt{High}\}$
heap	$\{\mathtt{x} \mapsto \langle 0^{\mathtt{Low}}, \{\mathbf{p_1}\}\rangle\}$
final value	

Dynamically Capturing Dependencies

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

	Run 1
value of h	$0^{\texttt{High}}$
dependency cache	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$
direct flow cache	$\{\mathbf{p}_1 \mapsto \texttt{High}\}$
heap	$\{\texttt{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p}_1\}\rangle\}$
final value	$\langle 0^{\texttt{Low}}, \{\mathbf{p}_2\}\rangle$

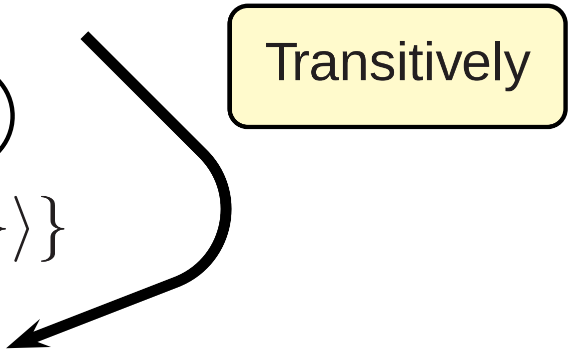
Dynamically Capturing Dependencies

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p_1}}\ \texttt{(h == 0) then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p_2}}\,\texttt{(x);}$

	Run 1
value of `h`	$0^{\texttt{High}}$
dependency cache	$\boxed{\{\mathbf{p_2} \mapsto \mathbf{p_1}\}}$
direct flow cache	$\boxed{\{\mathbf{p_1} \mapsto \texttt{High}\}}$
heap	$\{\mathrm{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p_1}\}\rangle\}$
final value	$0^{\texttt{High}}$

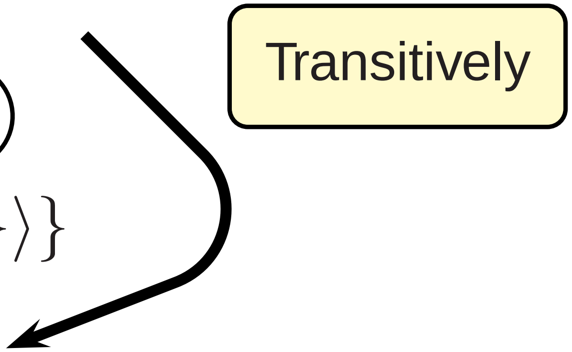


Transitively

Dynamically Capturing Dependencies

$$\begin{array}{l}\texttt{x := 1;}\\ \texttt{if}_{\mathbf{p_1}}\texttt{ (h == 0) then x := 0 else ()}\\ \texttt{deref}_{\mathbf{p_2}}\texttt{ (x);}\end{array}$$

	Run 1
value of h	$0^{\texttt{High}}$
dependency cache	$\boxed{\{\mathbf{p_2} \mapsto \mathbf{p_1}\}}$
direct flow cache	$\boxed{\{\mathbf{p_1} \mapsto \texttt{High}\}}$
heap	$\{\texttt{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p_1}\}\rangle\}$
final value	$0^{\texttt{High}}$



Transitively

Leak Detected!

Dynamically Capturing Dependencies

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p_1}}\texttt{ (h == 0) then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p_2}}\texttt{(x);}$

	Run 1	*Run 2*
value of $\texttt{h}$	$0^{\texttt{High}}$	$1^{\texttt{High}}$
dependency cache	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$	$\xrightarrow{\text{Keep}} \{\mathbf{p_2} \mapsto \mathbf{p_1}\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \texttt{High}\}$	$\xrightarrow{\text{Clear}} \{\}$
heap	$\{\texttt{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p_1}\}\rangle\}$	$\{\}$
final value	$0^{\texttt{High}}$	

Dynamically Capturing Dependencies

```
x := 1;
if_p1 (h == 0) then x := 0 else ()
deref_p2 (x);
```

	Run 1	*Run 2*
value of h	$0^{\mathtt{High}}$	$1^{\mathtt{High}}$
dependency cache	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \mathtt{High}\}$	$\{\}$
heap	$\{\mathtt{x} \mapsto \langle 0^{\mathtt{Low}}, \{\mathbf{p_1}\}\rangle\}$	$\{\mathtt{x} \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$
final value	$0^{\mathtt{High}}$	

Dynamically Capturing Dependencies

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p_1}}\ \texttt{(h == 0) then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p_2}}\texttt{(x);}$

	Run 1	*Run 2*
value of h	$0^{\texttt{High}}$	$1^{\texttt{High}}$
dependency cache	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \texttt{High}\}$	$\{\mathbf{p_1} \mapsto \texttt{High}\}$
heap	$\{\mathrm{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p_1}\}\rangle\}$	$\{\mathrm{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$
final value	$0^{\texttt{High}}$	

Dynamically Capturing Dependencies

```
x := 1;
if$_{p_1}$ (h == 0) then x := 0 else ()
deref$_{p_2}$ (x);
```

	Run 1	*Run 2*
value of h	$0^{\mathtt{High}}$	$1^{\mathtt{High}}$
dependency cache	$\{p_2 \mapsto p_1\}$	$\{p_2 \mapsto p_1\}$
direct flow cache	$\{p_1 \mapsto \mathtt{High}\}$	$\{p_1 \mapsto \mathtt{High}\}$
heap	$\{x \mapsto \langle 0^{\mathtt{Low}}, \{p_1\}\rangle\}$	$\{x \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$
final value	$0^{\mathtt{High}}$	

Dynamically Capturing Dependencies

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p}_1}\ \texttt{(h == 0) then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p}_2}\ \texttt{(x);}$

	Run 1	*Run 2*
value of h	$0^{\texttt{High}}$	$1^{\texttt{High}}$
dependency cache	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$
direct flow cache	$\{\mathbf{p}_1 \mapsto \texttt{High}\}$	$\{\mathbf{p}_1 \mapsto \texttt{High}\}$
heap	$\{\mathrm{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p}_1\}\rangle\}$	$\{\mathrm{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$
final value	$0^{\texttt{High}}$	$\langle 1^{\texttt{Low}}, \{\mathbf{p}_2\}\rangle$

Dynamically Capturing Dependencies

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p}_1}\ \texttt{(h == 0) then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p}_2}\ \texttt{(x);}$

	Run 1	*Run 2*
value of h	$0^{\texttt{High}}$	$1^{\texttt{High}}$
dependency cache	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$
direct flow cache	[illegible]	$\{\mathbf{p}_1 \mapsto \texttt{High}\}$
heap	$\{\texttt{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p}_1\}\rangle\}$	$\{\texttt{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$
final value	$0^{\texttt{High}}$	$1^{\texttt{High}}$

Transitively

Dynamically Capturing Dependencies

$\texttt{x := 1;}$
$\texttt{if}_{\mathbf{p}_1}\ \texttt{(h == 0) then x := 0 else ()}$
$\texttt{deref}_{\mathbf{p}_2}\texttt{(x);}$

	Run 1	*Run 2*
value of h	$0^{\texttt{High}}$	$1^{\texttt{High}}$
dependency cache	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$	$\boxed{\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}}$
direct flow cache	$\}$	$\boxed{\{\mathbf{p}_1 \mapsto \texttt{High}\}}$
heap	$\{\mathrm{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p}_1\}\rangle\}$	$\{\mathrm{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$
final value	$0^{\texttt{High}}$	$1^{\texttt{High}}$

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;
if_p1 (h == 0) then x := 0 else ()
deref_p2 (x);
```

	Run 1a
value of `h`	$1^{\mathtt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\}$
heap	$\{\}$
final value	

In Reverse Order

```
x := 1;
if_p1 (h == 0) then x := 0 else ()
deref_p2 (x);
```

	Run 1a
value of h	$1^{\mathtt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\mathbf{p}_1 \mapsto \mathtt{High}\}$
heap	$\{\mathtt{x} \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$
final value	$\langle 1^{\mathtt{Low}}, \{\mathbf{p}_2\}\rangle$

In Reverse Order

$\mathtt{x\ :=\ 1;}$
$\mathtt{if}_{\mathbf{p_1}}\ \mathtt{(h\ ==\ 0)\ then\ x\ :=\ 0\ else\ ()}$
$\mathtt{deref}_{\mathbf{p_2}}\mathtt{(x);}$

	Run 1a
value of h	$1^{\mathtt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \mathtt{High}\}$
heap	$\{\mathtt{x} \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$
final value	$1^{\mathtt{Low}}$

In Reverse Order

$\mathtt{x\ :=\ 1;}$
$\mathtt{if}_{\mathbf{p_1}}\ \mathtt{(h\ ==\ 0)\ then\ x\ :=\ 0\ else\ ()}$
$\mathtt{deref}_{\mathbf{p_2}}\mathtt{(x);}$

	Run 1a
value of $\mathtt{h}$	$1^{\mathtt{High}}$
dependency cache	$\{\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \mathtt{High}\}$
heap	$\{\mathtt{x} \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$
final value	$1^{\mathtt{Low}}$

Leak NOT detected!

In Reverse Order

$\mathtt{x\ :=\ 1;}$
$\mathtt{if}_{\mathbf{p_1}}\ \mathtt{(h\ ==\ 0)\ then\ x\ :=\ 0\ else\ ()}$
$\mathtt{deref}_{\mathbf{p_2}}\mathtt{(x);}$

	Run 1a	*Run 2a*
value of $\mathtt{h}$	$1^{\mathtt{High}}$	$0^{\mathtt{High}}$
dependency cache	$\{\}$	$\{\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \mathtt{High}\}$	$\{\}$
heap	$\{\mathtt{x} \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$	$\{\}$
final value	$1^{\mathtt{Low}}$	

In Reverse Order

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p_1}}\ \texttt{(h == 0) then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p_2}}\texttt{(x);}$

	Run 1a	*Run 2a*
value of $\texttt{h}$	$\texttt{1}^{\texttt{High}}$	$\texttt{0}^{\texttt{High}}$
dependency cache	$\{\}$	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \texttt{High}\}$	$\{\mathbf{p_1} \mapsto \texttt{High}\}$
heap	$\{\texttt{x} \mapsto \langle \texttt{1}^{\texttt{Low}}, \{\}\rangle\}$	$\{\texttt{x} \mapsto \langle \texttt{0}^{\texttt{Low}}, \{\mathbf{p_1}\}\rangle\}$
final value	$\texttt{1}^{\texttt{Low}}$	$\langle \texttt{0}^{\texttt{Low}}, \{\mathbf{p_2}\}\rangle$

In Reverse Order

$$\begin{array}{l} \texttt{x := 1;} \\ \texttt{if}_{\mathbf{p}_1}\,\texttt{(h == 0) then x := 0 else ()} \\ \texttt{deref}_{\mathbf{p}_2}\texttt{(x);} \end{array}$$

	Run 1a	*Run 2a*
value of h	$1^{\mathtt{High}}$	$0^{\mathtt{High}}$
dependency cache	$\{\}$	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$
direct flow cache	$\{\mathbf{p}_1 \mapsto \mathtt{High}\}$	$\{\mathbf{p}_1 \mapsto \mathtt{High}\}$
heap	$\{\mathtt{x} \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$	$\{\mathtt{x} \mapsto \langle 0^{\mathtt{Low}}, \{\mathbf{p}_1\}\rangle\}$
final value	$1^{\mathtt{Low}}$	$0^{\mathtt{High}}$

In Reverse Order

$\texttt{x := 1;}$
$\texttt{if}_{\mathbf{p_1}}\ \texttt{(h == 0) then x := 0 else ()}$
$\texttt{deref}_{\mathbf{p_2}}\texttt{(x);}$

	Run 1a	*Run 2a*
value of h	$1^{\texttt{High}}$	$0^{\texttt{High}}$
dependency cache	$\{\}$	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \texttt{High}\}$	$\{\mathbf{p_1} \mapsto \texttt{High}\}$
heap	$\{\texttt{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$	$\{\texttt{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p_1}\}\rangle\}$
final value	$1^{\texttt{Low}}$	$0^{\texttt{High}}$

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* $\mathtt{r}_1$ *and* $\mathtt{r}_2$ *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.

All Leaks Detected with Full Cache

```
x := 1;
if$_{\mathbf{p_1}}$ (h == 0) then x := 0 else ()
deref$_{\mathbf{p_2}}$ (x);
```

	Run 1b	*Run 2b*
value of $\mathtt{h}$	$1^{\mathtt{High}}$	$0^{\mathtt{High}}$
dependency cache	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$	$\{\mathbf{p_2} \mapsto \mathbf{p_1}\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \mathtt{High}\}$	$\{\mathbf{p_1} \mapsto \mathtt{High}\}$
heap	$\{\mathtt{x} \mapsto \langle 1^{\mathtt{Low}}, \{\}\rangle\}$	$\{\mathtt{x} \mapsto \langle 0^{\mathtt{Low}}, \{\mathbf{p_1}\}\rangle\}$
final value	$1^{\mathtt{High}}$	$0^{\mathtt{High}}$

All Leaks Detected with Full Cache

$\texttt{x := 1;}$

$\texttt{if}_{\mathbf{p}_1}\ \texttt{(h == 0) then x := 0 else ()}$

$\texttt{deref}_{\mathbf{p}_2}\texttt{(x);}$

	Run 1b	*Run 2b*
value of h	$1^{\texttt{High}}$	$0^{\texttt{High}}$
dependency cache	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$	$\{\mathbf{p}_2 \mapsto \mathbf{p}_1\}$
direct flow cache	$\{\mathbf{p}_1 \mapsto \texttt{High}\}$	$\{\mathbf{p}_1 \mapsto \texttt{High}\}$
heap	$\{\texttt{x} \mapsto \langle 1^{\texttt{Low}}, \{\}\rangle\}$	$\{\texttt{x} \mapsto \langle 0^{\texttt{Low}}, \{\mathbf{p}_1\}\rangle\}$
final value	$1^{\texttt{High}}$	$0^{\texttt{High}}$
	Leak Detected!	**Leak Detected!**

Better Precision at Run-time

$\mathtt{x := 0;\ y := 0;}$
$\mathtt{if}_{\mathbf{p}_1}\ \mathtt{(l < 0)\ then\ y := h\ else\ ()}$
$\mathtt{if}_{\mathbf{p}_2}\ \mathtt{(l > 0)\ then\ x := deref}_{\mathbf{p}_3}\ \mathtt{(y)\ else\ ()}$
$\mathtt{deref}_{\mathbf{p}_4}\ \mathtt{(x);}$

	Run 1, with Fixed Point Dependency Cache
value of `l`, `h`	$-1^{\mathtt{Low}},\ 1^{\mathtt{High}}$
dependency cache	$\{\mathbf{p}_3 \mapsto \{\mathbf{p}_1, \mathbf{p}_2\}, \mathbf{p}_4 \mapsto \{\mathbf{p}_2, \mathbf{p}_3\}\}$
direct flow cache	$\{\mathbf{p}_1 \mapsto \mathtt{Low}, \mathbf{p}_2 \mapsto \mathtt{Low}\}$
heap	$\{\mathtt{x} \mapsto \langle 0^{\mathtt{Low}}, \{\}\rangle, \mathtt{y} \mapsto \langle 1^{\mathtt{High}}, \{\mathbf{p}_1\}\rangle\}$
final value	$\langle 0^{\mathtt{Low}}, \{\mathbf{p}_4\}\rangle$

Better Precision at Run-time

$\texttt{x := 0; y := 0;}$

$\texttt{if}_{p_1}\ \texttt{(l < 0) then y := h else ()}$

$\texttt{if}_{p_2}\ \texttt{(l > 0) then x := deref}_{p_3}\ \texttt{(y) else ()}$

$\texttt{deref}_{p_4}\ \texttt{(x);}$

	Run 1, with Fixed Point Dependency Cache
value of $\texttt{l}$, $\texttt{h}$	$-1^{\texttt{Low}},\ 1^{\texttt{High}}$
dependency cache	$\{p_3 \mapsto \{p_1, p_2\}, p_4 \mapsto \{p_2, p_3\}\}$
direct flow cache	$\{p_1 \mapsto \texttt{Low}, p_2 \mapsto \texttt{Low}\}$
heap	$\{x \mapsto \langle 0^{\texttt{Low}}, \{\}\rangle, y \mapsto \langle 1^{\texttt{High}}, \{p_1\}\rangle\}$
final value	$\langle 0^{\texttt{Low}}, \{p_4\}\rangle$

p_2 is $\texttt{Low}$, p_3 is undefined

Better Precision at Run-time

$\texttt{x := 0; y := 0;}$
$\texttt{if}_{\mathbf{p}_1}\ \texttt{(l < 0) then y := h else ()}$
$\texttt{if}_{\mathbf{p}_2}\ \texttt{(l > 0) then x := deref}_{\mathbf{p}_3}\texttt{(y) else ()}$
$\texttt{deref}_{\mathbf{p}_4}\texttt{(x);}$

	Run 1, with Fixed Point Dependency Cache
value of l, h	$-1^{\texttt{Low}},\ 1^{\texttt{High}}$
dependency cache	$\{\mathbf{p}_3 \mapsto \{\mathbf{p}_1, \mathbf{p}_2\}, \mathbf{p}_4 \mapsto \{\mathbf{p}_2, \mathbf{p}_3\}\}$
direct flow cache	$\{\mathbf{p}_1 \mapsto \texttt{Low}, \mathbf{p}_2 \mapsto \texttt{Low}\}$
heap	$\{\texttt{x} \mapsto \langle 0^{\texttt{Low}}, \{\}\rangle, \texttt{y} \mapsto \langle 1^{\texttt{High}}, \{\mathbf{p}_1\}\rangle\}$
final value	$0^{\texttt{Low}}$

Better Precision at Run-time

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

	Run 1, with Fixed Point Dependency Cache
value of `l`, `h`	$-1^{\mathtt{Low}}$, $1^{\mathtt{High}}$
dependency cache	$\{\mathbf{p_3} \mapsto \{\mathbf{p_1}, \mathbf{p_2}\}, \mathbf{p_4} \mapsto \{\mathbf{p_2}, \mathbf{p_3}\}\}$
direct flow cache	$\{\mathbf{p_1} \mapsto \mathtt{Low}, \mathbf{p_2} \mapsto \mathtt{Low}\}$
heap	$\{\mathtt{x} \mapsto \langle 0^{\mathtt{Low}}, \{\}\rangle, \mathtt{y} \mapsto \langle 1^{\mathtt{High}}, \{\mathbf{p_1}\}\rangle\}$
final value	$0^{\mathtt{Low}}$

No False Positive!

Dynamic Noninterference

Theorem 2. *If* $\mathtt{r}_1$ *and* $\mathtt{r}_2$ *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!!