Information Flow Control for Concurrent Programs via Program Slicing

Dennis Giffhorn

Universität Karlsruhe (TH), Germany

Context

Slicing-based program security, focusing on Java

- Program dependence analysis for full Java
  (Hammer/Snelting: *An improved slicer for Java*, PASTE'04)
- Slicing of concurrent programs
  (Krinke: *Context-sensitive slicing of concurrent programs*, ESEC/FSE'03)
- Connection between IFC and slicing
  (Snelting et al.: *Efficient path conditions in dependence graphs for software safety analysis*, TOSEM'06)
- Slicing-based IFC algorithm for sequential Java
  (Hammer/Krinke: *Intransitive noninterference in dependence graphs*, ISOLA'06)

Information Flow Control

Does a program leak confidential information?

- Data is classified with security levels *high* and *low*
- Scenario: Attacker wants to gain information about high input data and can observe parts of program behaviour
  - Low-classified data at certain program points (e.g. input, output)
  - The relative order of low-observable events
  - Termination behaviour

  $\rightarrow$ *low-observable behaviour*
- Noninterference (idea):
  Two inputs that only differ on high input have to cause the same low-observable behaviour
  $\Rightarrow$ Attacker cannot draw conclusions about high input

Sequential Programs

- Sufficient to control *explicit* and *implicit flow*

```
l1 = pin;
if (pin > 12345)
   l2 = 1;
print(l1+l2);
```

- Explicit flow from `pin` to `l1`
- Implicit flow from `pin` to `l2`
- Attacker can see the output ⇒ `pin` must not contain high data

Sequential Programs

- Sufficient to control *explicit* and *implicit flow*

```
l1 = pin;
if (pin > 12345)
   l2 = 1;
print(l1+l2);
```

- Explicit flow from `pin` to `l1`
- Implicit flow from `pin` to `l2`
- Attacker can see the output ⇒ `pin` must not contain high data

Sequential Programs

- Sufficient to control *explicit* and *implicit flow*

```
l1 = pin;
if (pin > 12345)
   l2 = 1;
print(l1+l2);
```

- Explicit flow from `pin` to `l1`
- Implicit flow from `pin` to `l2`
- Attacker can see the output ⇒ `pin` must not contain high data

Probabilistic Channels

- Not sufficient for concurrent programs
- Problem: High data may influence interleaving

```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```

- The larger the value of h,
  - the larger the probability that l1 = 2 when printed
  - the larger the probability that l1 is printed after l2

⇒ probabilistic channels

Probabilistic Noninterference

- A program input can cause a set of possible low-observable behaviours
- Each one has a certain probability (scheduler-dependent)
- Probabilistic noninterference (idea):
  Two inputs that only differ on high input have to cause the **same possible low-observable behaviours** with the **same probabilities**
  $\Rightarrow$ Attacker cannot draw conclusions about high input

Probabilistic Channels

```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```

Necessary condition:

- A *data conflict* between two concurrent accesses to a shared variable, where at least one of which is a write
- An *order conflict* between two concurrent low-observable events

Probabilistic Channels

```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```

Necessary condition:

- A *data conflict* between two concurrent accesses to a shared variable, where at least one of which is a write
- An *order conflict* between two concurrent low-observable events

Probabilistic Channels

```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```

Necessary condition:

- A *data conflict* between two concurrent accesses to a shared variable, where at least one of which is a write
- An *order conflict* between two concurrent low-observable events

Probabilistic Channels

```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```

Necessary condition:

- A *data conflict* between two concurrent accesses to a shared variable, where at least one of which is a write
- An *order conflict* between two concurrent low-observable events

Observational Determinism

- Low-observable behaviour does not depend on a conflict
  $\rightarrow$ no probabilistic channels
- Such a program is *observational deterministic*
  - The same input produces always the same low-observable behaviour
  - It is sufficient to check implicit and explicit flow
- Security for concurrent programs:
  Observational determinism + sane implicit and explicit flow
- Generalization of prob. NI:
  Only one possible low-observable behaviour with probability = 1

IFC for Concurrent Programs via Program Slicing

A three-phase approach

1. Annotate the program
2. Check observational determinism
3. Check explicit and implicit flow
    - Algorithm of Hammer/Krinke.
    - Developed for full sequential Java

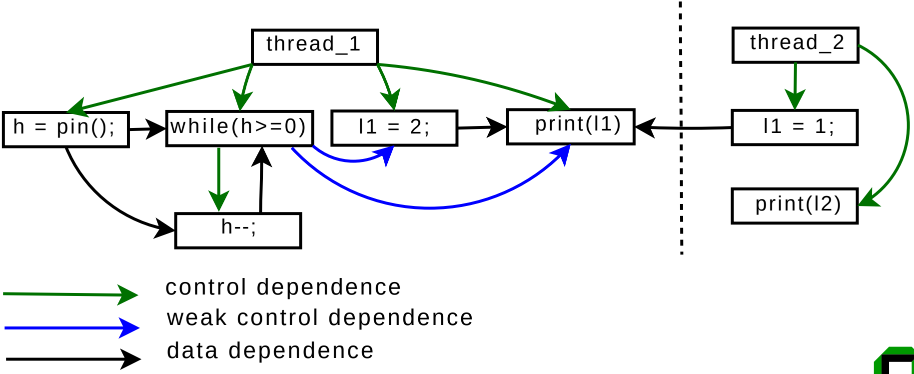
Program Dependence Graph

```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```

- Statements are the nodes
- The definition of a variable defined at v reaches its use at w (not redefined inbetween) (Data Dependence).
- v controls the execution of w (termination-insensitive) (Control Dependence)
- v controls the execution of w (termination-sensitive) (Weak Control Dependence)

Program Dependence Graph

```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```



Program Annotation

```
h = readPIN();     ||    l1 := 1;
while (h != 0)     ||    print(l2);
   h--;            ||
l1 := 2;           ||
print(l1);         ||
```

Phase 1: Annotate program

- Annotation mechanism similar to Hammer/Krinke
- Sources of information are annotated with a *providing level*
- Observable statements are annotated with a *requiring level*
  - `h = readPIN()` gets providing level high
  - both print-statements get requiring level low

Conflict Dependence

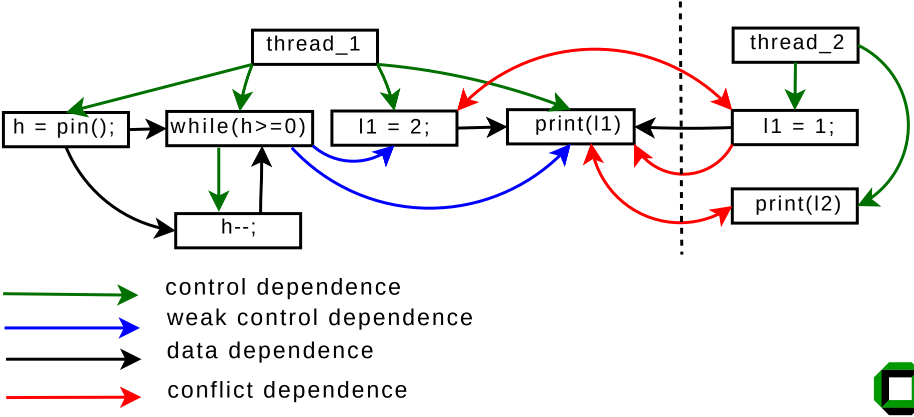
```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```

Augment the PDG with *conflict dependences*

- representing a *data conflict* or an *order conflict*
- Data conflicts: Directed from the write-access to the other one
- Order conflicts: Between two concurrent low-observable events, bidirected

Conflict Dependence

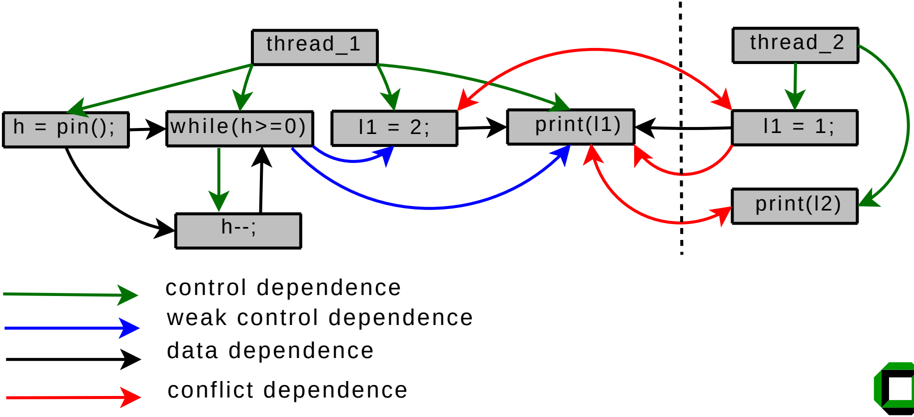
```
h = readPIN();      ||    l1 := 1;
while (h != 0)      ||    print(l2);
   h--;             ||
l1 := 2;            ||
print(l1);          ||
```



Check Observational Determinism

Phase 2: Check observational determinism

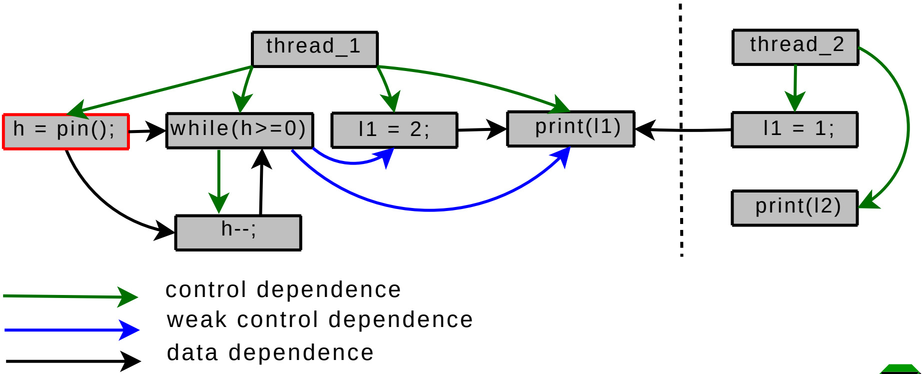
- Compute a slice for the low-class. statements
- If the slice contains a conflict dependence, the program is not obs. det. → reject program



Check Implicit and Explicit Flow

Phase 3: Check implicit and explicit flow

- Basic idea: Compute a slice for the low-class. statements
- If it contains a statement with a provided level of high, reject the program



Related Work

Exploiting observational determinism is not a new idea

- McLean: *Proving noninterference and functional correctness using traces* (JCS 1992)
  - Obs. det. for trace-based specifications
- Roscoe: *CSP and determinism in security modelling* (IEEE SP 1995)
  - Algorithm for obs. det. for CSP calculus
- Zdancevic/Myers: *Observational determinism for concurrent program security* (CSFW 2003)
  - *Low-security observational determinism*
  - Abscence of data conflicts + sane implicit and explicit flow
  - Non-standard type system
- Huisman et al.: *A temporal logic characterization of observational determinism* (CSFW 2006)
  - Termination-sensitive extension
  - Model checking produces counter-examples

Future Work

- PDGs can be computed for mature languages, e.g. C, C++, Java
  → implementation and evaluation
- Obs. det. restricts inter-thread communication of low data
  - Message-passing mechanisms?
  - Declassification?
- Not compositional
  → incremental development of secure systems is complicated