Towards Dynamically Handling Implicit Information Flow

Bin Xin and Xiangyu Zhang

What is Implicit Information Flow (IIF)?
• Information flow caused by execution omission.

Why IIF is Important
• Information protection
• Debugging

Static Solution is Too Conservative
• Static solution considers all possible paths

Implicit Dependence
• Definition: Given an execution E, a predicate p, and a use u. If there is no explicit dependence path between p and u in E and there is the execution of the same program with the same input as E except the branch outcome of p being switched, p and u are the execution points in E that match p and u respectively. Implicitly depends on:
  - p is not found in E, or
  - if there is an explicit dependence path between p and u.

Detection Of Dynamic Control Dependence
• Existing dynamic control dependence (DCD) detection can not meet our goal
  - Online: if a statement s has multiple static control depending predicates p1, p2, …, at the moment s is executed, the latest px is processed to compute DCD
  • Not efficient

An Example
• The explicit dependence path 5-4-3 in E implies p implicitly depends 3 in E.

Our Dynamic Solution
• We design a dynamic method that forces the execution of the omitted code by switching outcomes to relevant predicates such that implicit dependences are exposed.

Observation: DCD at runtime has a stack-like structure.
• How to reduce the number of predicates that are needed to verify?
  – It could potentially be all the executed predicates.

Two Challenges
• How to align points in two executions?
  – It remains the same problem even though a thread can be started instead at the moment of the predicate execution.
• How to reduce the number of predicates?
  – It could potentially be all the executed predicates.

Our Approach
• Observation: DCD at runtime has a stack-like structure.
  – As an entry is pushed onto the control dependence stack if a branching point produces new statements, it is executed.
  – As an exit is pushed onto the control dependence stack if the branching point executes.
• Advantage:
  - Multiple static control dependent predicates are no longer a problem, becomes much more efficient.
  - Reducing the number of executional DCD even in the presence of irregular control flow caused by branching point.

Challenge Two – Reducing the Number of Verification
• Two scenarios:
  - Backward scenario: debugging
  - Forward scenario: dynamic control dependence.
• Our solution for the backward scenario
  - Given a program failure (seq fault or wrong output value), a dynamic slice is computed. Confident analysis (our PLDI 2006 work) is applied to produce a pruned slice.
  - Only the predicates in the slices are tested for implicit dependences.

Author Information
• Bin Xin: Ph.D. student, Department of Computer Science, Purdue University, West Lafayette, IN 47906; E-mail: binx@cs.purdue.edu
• Xiangyu Zhang: Assistant Professor, Department of Computer Science, Purdue University, West Lafayette, IN 47906; E-mail: xzhang@cs.purdue.edu