CERIAS

the center for education and research in information assurance and security

Towards Dynamically Handling Implicit Information Flow

Bin Xin and Xiangyu Zhang

What is Implicit Information Flow (IIF)?

- Information flow caused by execution omission.

```
1:  X = …
2:  Y = …
3:  if ( f(X) )  then
4:       Y = Y + 1
5:  …= Y
```

If f(x) = False,

the branch at 3 is not taken, Y at 5 has the same value as Y at 2.

There is information flow from X to Y – Implicit Information Flow

Why IIF is Important

- Information protection

```
1:  X = getpassword( )
2:  Y = 0
3:  if ( X != “xin&zhang” )  then
4:       Y = Y + 1
5:  sendpacket(Y)
```

Y==0 implies password is “xin&zhang”

- debugging

```
1:  X = … /*buggy*/
2:  Y = …
3:  if ( f(X) )  then
4:       Y = Y + 1
5:  …= Y
```

Dependence backtrace misses the root cause

Static Solution Is Too Conservative

- Static solution consider all possible paths

```
1:  X = …
2:  Y = …
3:  if ( f(X) )  then
        …
        if (…) then
40:         Y = Y + 1
        …
50:  …= Y
```

Static analysis considers there is always information flow from X to Y.

- Points-to analysis

```
1:  X = …
2:  Y = …
3:  if ( f(X) )  then
4:       *p = Y + 1
5:  …= Y
```

If p may points to Y, static analysis considers there is information flow from X to Y.

Existing Dynamic Solutions Fail

- Dynamic analysis is typically designed to focus on dynamic information collected from executed statements, and statements whose execution is omitted do not produce any dynamic information, detection of IIF becomes very challenging.
  - It is a long standing open problem in dynamic information flow and debugging

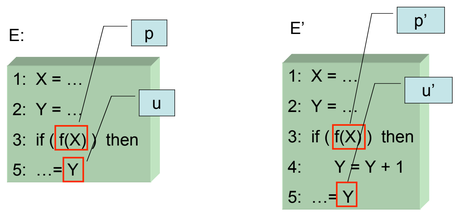
Our Dynamic Solution

- We design a dynamic method that forces the execution of the omitted code by switching outcomes of relevant predicates such that implicit dependences are exposed.
  - *Explicit* dependence is a dependence that can be observed during the execution including data dependence and control dependence.

Implicit Dependence

- DEFINITION *Given an execution E, a predicate p, and a use u s.t. there is no explicit dependence path between p and u, let E' be the reexecution of the same program with the same input as E except the branch outcome of p' being switched, p' and u' be the execution points in E' that match p and u in E, respectively, u* **implicitly** *depends on p, iff*
  - *(i) u' is not found in E', or,*
  - *(ii) there is an explicit dependence path between p' and u'.*

An Example

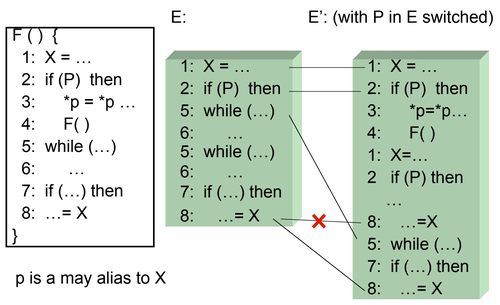


- The explicit dependence path 5-4-3 in E' implies 5 implicitly depends 3 in E.

Two Challenges

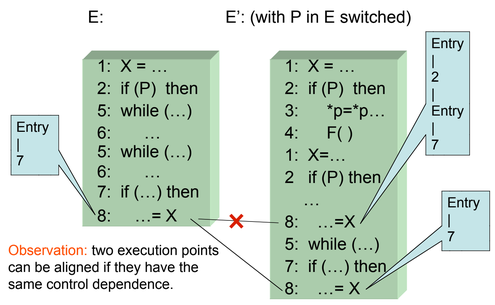
- How to align points in two executions?
  - it remains the same problem even though a thread can be started instead at the moment of the predicate execution.
- How to reduce the number of predicates that are needed to verify?
  - it could potentially be all the executed predicates.

Challenge One – Execution Alignment



p is a may alias to X

Efficient Execution Alignment



Observation: two execution points can be aligned if they have the same control dependence.

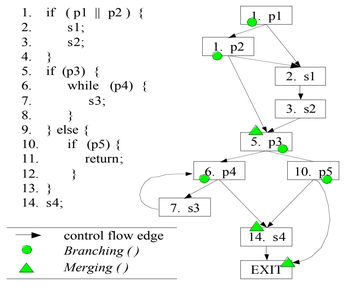
Detection Of Dynamic Control Dependence

- Existing dynamic control dependence (DCD) detection can not meet our goal
  - Offline: the control flow trace is first collected and then processed to compute DCD
    - Expensive (both time and space)
  - Online: if a statement s has multiple static control depending predicates $p_1, p_2, \ldots,$ at the moment s is executed, the latest p_x is the dynamic CD.
    - Not efficient
    - Can not handle interprocedural DCD.

Our Approach

- Observation: DCD at runtime has a stack-like structure.
  - An entry is pushed onto the control dependence stack if a branching point (predicates, switch statements, etc.) executes.
  - The current entry is popped if the post-dominator of the branching point executes.
- Advantages:
  - Multiple static control depending predicates are no longer a problem, becomes much more efficient.
  - Naturally handle interprocedural DCD even in the presence of irregular control flow caused by longjmp, setjmp.

An Example



An Example (continued)

Trace	Instrumentation	Control Dependence Stack
$p_1@1_1$	push <$p_1@1_1$,5>	[<$p_1@1_1$,5>]
$p_2@1_1$	replace top w/ <$p_2@1_1$,5>	[<$p_2@1_1$,5>]
2_1	-	same as above
3_1	-	same as above
5_1	pop push <5_1,*EXIT*>	[<5_1,*EXIT*>]
6_1	push <6_1,*EXIT*>	[<6_1,*EXIT*>|<5_1,*EXIT*>]
7_1	-	same as above
6_2	replace top w/ <6_2, *EXIT*>	[<6_2,*EXIT*>|<5_1,*EXIT*>]
14_1	pop	[<5_1,*EXIT*>]
$EXIT_1$	pop	[]

Evaluation

Benchmark	Base(s)	DCD(s)	Overhead	Old(s)	Improvement
008.espresso	1.35	5.03	3.73x	14	2.78x
124.m88ksim	0.18	0.64	3.55x	1.98	3.09x
129.compress	115	255	2.22x	657	2.58x
132.ijpeg	40	73	1.83x	160	2.19x
164.gzip	3.7	12.6	3.41x	37	2.94x
175.vpr	24	81	3.37x	-	-
181.mcf	90	127	1.41x	196	1.54x
197.parser	23	52	2.26x	175	3.37x
256.bzip2	36	71	1.97x	128	1.80x
300.twolf	39	79	2.02x	-	-
Average	-	-	2.57x	-	2.54x

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 (seg fault or wrong output value), a dynamic slice is computed. Confidence analysis (our PLDI 2006 work) is applied to produce a pruned slice.
  - Only the predicates in the slices are tested for implicit dependences.

Evaluation

Benchmark	Error	# of verification	# of expanded edges
flex	V_1-F_9	5	5
	V_2-F_{14}	4	1
	V_3-F_{10}	1	1
	V_4-F_6	6	5
	V_5-F_6	2	2
grep	V_4-F_2	313	62
gzip	V_2-F_3	1	1
sed	V_3-F_2	36	2
	V_3-F_3	115	1

On Going Work

- Forward scenario
  - Goal: a low overhead dynamic information flow system that handles implicit dependence.
  - Sketched solution
    - Taint the execution with security labels.
    - If a predicate is tainted, both branches will be taken by starting two threads (could be on two cores). The two threads share the same security label space but separated memory space.
    - A DCD stack is maintained to synchronize the two threads at the end of both branches.

On Going Work

- The soundness of predicate switching
  - Switching one predicate may not suffice

```
1:  X = … /* buggy */
2:  Y = …
3:  if ( X > 10 )  then
4:       if (X > 100) then
5:            Y = Y + 1
5:  …= Y
```

Author Information

- *Bin Xin*: Ph.D. student, Department of Computer Science, Purdue University, West Lafayette, IN 47906; E-mail: xinb@cs.purdue.edu.
- *Xiangyu Zhang*: Assistant Professor, Department of Computer Science, Purdue University, West Lafayette, IN 47906; E-mail: xyzhang@cs.purdue.edu.

PURDUE UNIVERSITY

CERIAS

Discovery Park e-Enterprise Center