Explaining Program Failures via Postmortem Static Analysis

Roman Manevich
Tel Aviv University

Manu Sridharan
UC Berkeley

Stephen Adams, Manuvir Das, Zhe Yang
Center for Software Excellence
Microsoft Corporation

Motivation

- Programs are shipped with bugs
- Crash reports ease bug fixing
  - Automated, sent over network
  - Give type of failure and stack trace
- But, problems remain
  - No execution trace provided
  - Reconstructing trace is time-consuming

An Example Crash

```
foo(rec *x, rec *z)
{
 q = z->f;
  *p = u;
  if (b)
    y = z;
  else
    y = x->f;
  *y = …;
}
```

Does dereference of z matter?

What does p point to?

Lots to keep track of!

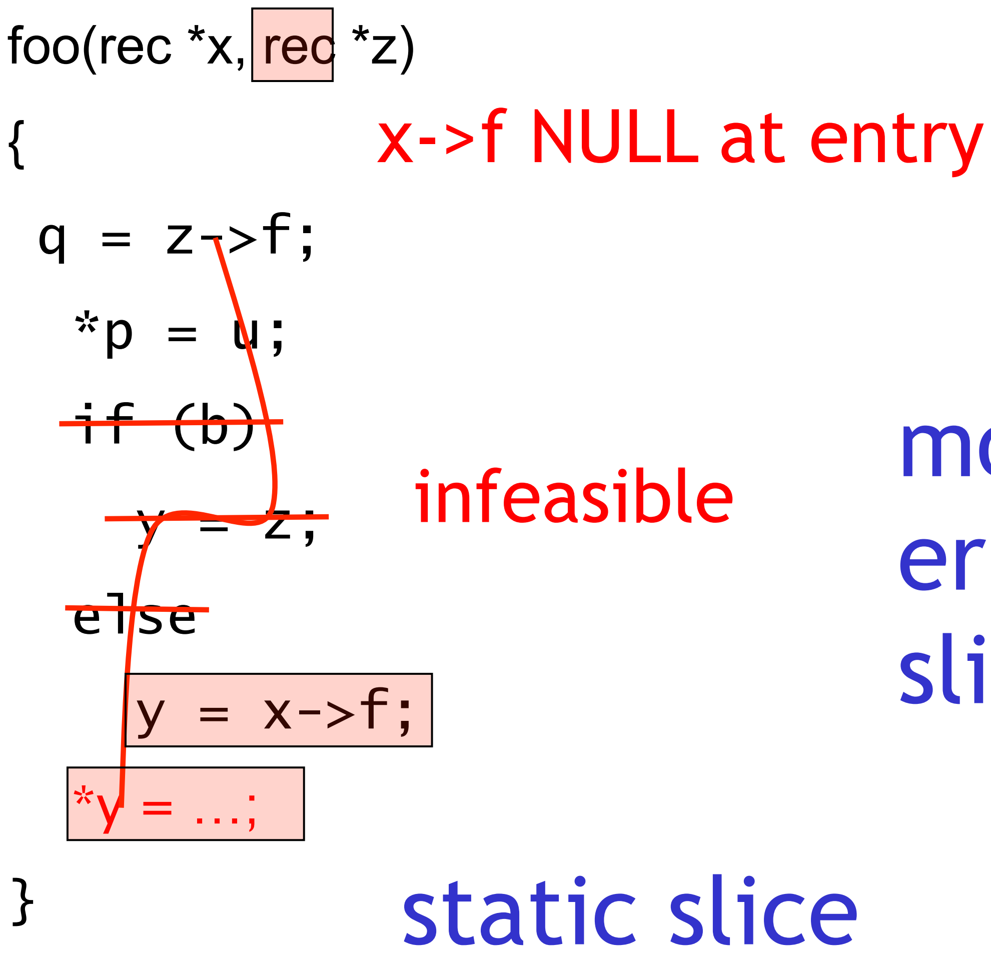
Which branch? Both?

NULL pointer dereference

Tool Support Needed

- Input: crash report
  - Program point of failure
  - Type of failure, eg. NULL dereference
- Output: error traces
  - Paths to point of failure that cause error

Static slicing?



```
foo(rec *x, rec *z)
{
 q = z->f;
  *p = u;
  ~~if (b)~~
   ~~y = z;~~
  ~~else~~
   y = x->f;
  *y = ...;
}
```

x->f NULL at entry

infeasible

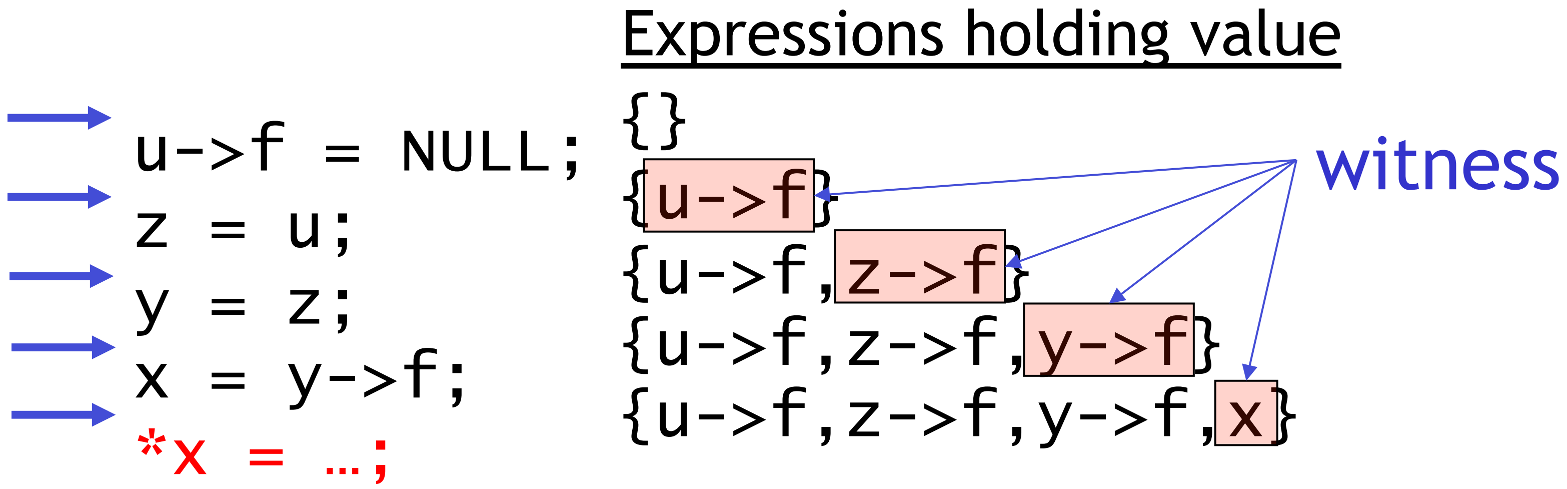
static slice

more informative error-specific slice

Postmortem Symbolic Evaluation

- Dataflow analysis to find traces
  - Track value backwards from error
  - Maintain flow information on each path
  - Use error type to filter traces
- Borrow techniques from ESP [DLS02]
  - For scalability, precision, soundness

Tracking Flow: The Witness



```
u->f = NULL;
z = u;
y = z;
x = y->f;
*x = …;
```

Expressions holding value

```
{}
{u->f}
{u->f,z->f}
{u->f,z->f,y->f}
{u->f,z->f,y->f,x}
```

witness

- Expression from which value is copied
  - Specific to path
- Single witness per point on path
  - Demand analysis

Computing The Witness

	Witness	Witness
u->f = NULL;	done	<z->f>
*p = u;	<u->f> <?>	<z->f>
y = z;	<z->f>	<z->f>
x = y->f;	<y->f>	<y->f>
*x = …;	<x>	<x>
	p == &z	p != &z

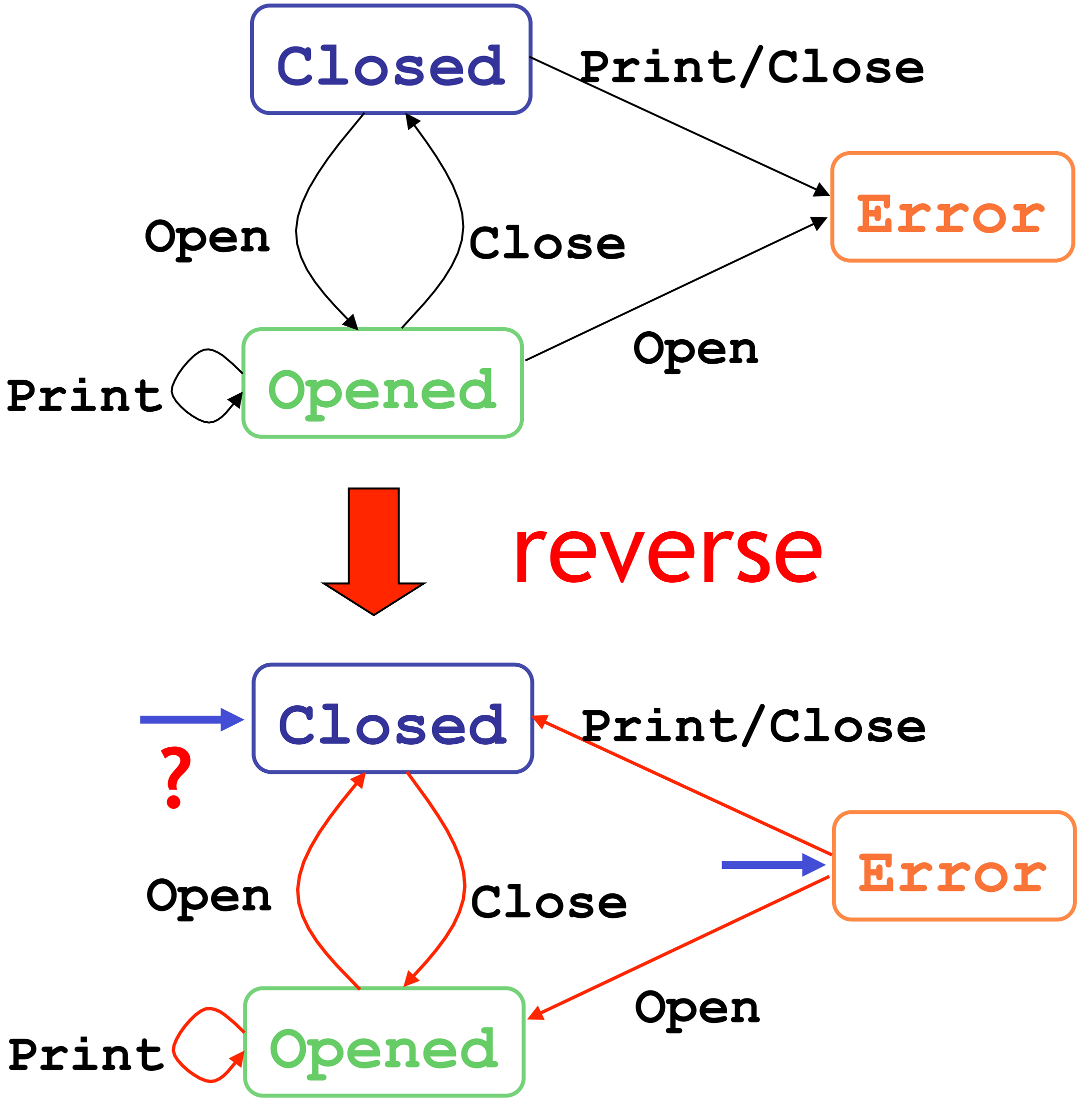
- Substitution like weakest preconditions
- Query aliasing oracle for indirect updates
- Still polynomial time
    - Bound number of witnesses
    - Switch to abstract location when too long

Using The Error Type

- No double deref of NULL on path
  - x = NULL; *x = y; *x = z is infeasible
  - Just check if witness is dereferenced
- In general, handle typestate errors
  - Automaton describes behavior
  - Crash at transition to error state
- Do double derefs generalize?

Automaton Reversal

File I/O



```
print(f,"hi");
close(f);
```

infeasible

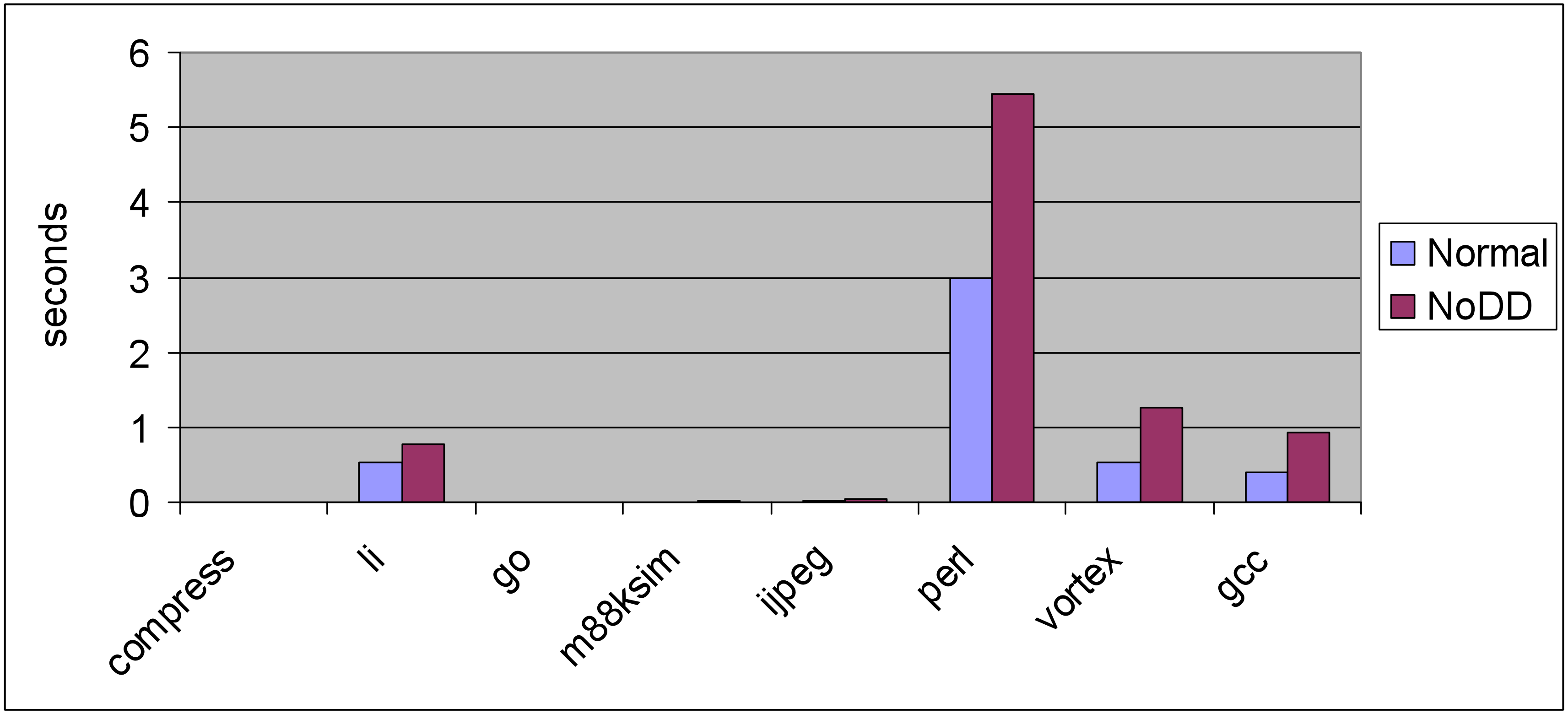
Putting It All Together

- ESP-style dataflow analysis [DLS02]
  - Interprocedural, path-sensitive
  - Engine maintains / presents traces
  - GOLF serves as aliasing oracle [DLFR01]
- Stack trace used if available
  - Restricts traversal up call stack
- Detect simple tests for NULL
  - Eg. `if (p)`
  - If p is witness on true branch, infeasible

Evaluation: Does It Scale?

- Test SPEC95 derefs for NULL deref
  - 2,000 – 140,000 lines of code
  - 100 random derefs per benchmark
  - If no traces for a deref, proven safe
- No stack traces
- Configurations
  - Normal: full analysis
  - NoDD: no filtering using double derefs

Average Query Times



- Most queries fast (usually more than 90%)
- The rest are quite slow (minutes)
    - No useful analysis result, so timeout (15 seconds)

Aliasing

- Imprecise analysis for heap pointers
  - False positives + increased analysis time
- Traces with aliasing inscrutable
  - No explanation for alias
  - Thus far, useless to developers
- Configuration "Unsound"
  - No checking for indirect updates
  - No abstraction for long witnesses

SPEC Number of Error Reports

Bench	Normal	Unsound
compress	0	0
li	25	5
go	1	1
m88ksim	2	2
ijpeg	3	0
perl	44	29
vortex	17	12
gcc	18	2

- Remaining false positives
  - Global flag
  - Use of abstract locs (eg. a[i])

Evaluation: Useful traces?

- PREfix: static bug finding tool [BPS00]
- Checked five real NULL deref errors
- Five successes with “Unsound”
  - Found error-causing traces only
  - Query times under a second
  - Stack traces helpful
  - Four succeeded with “Normal”

Related Work

- Slicing [Tip95]
- Postmortem analysis [LA02]
- Typestate analysis [SY86,SY93]
- Fault localization
- Remote program sampling [LAZJ03]
- Forward analyses (Metal, ESP, model checkers)

Conclusions

- New analysis for diagnosing errors
  - Value traced back from error
  - Witnesses give useful flow information
  - False traces pruned using error type
- Results are promising
- Extensions
  - Integration with Watson
  - Evaluating other typestate errors
  - Presentation of aliases to developer

The End