Cooperative Debugging with Five Hundred Million Test Cases

Ben Liblit, University of Wisconsin–Madison

Do You Recognize This ~~Soldier~~ Tester?

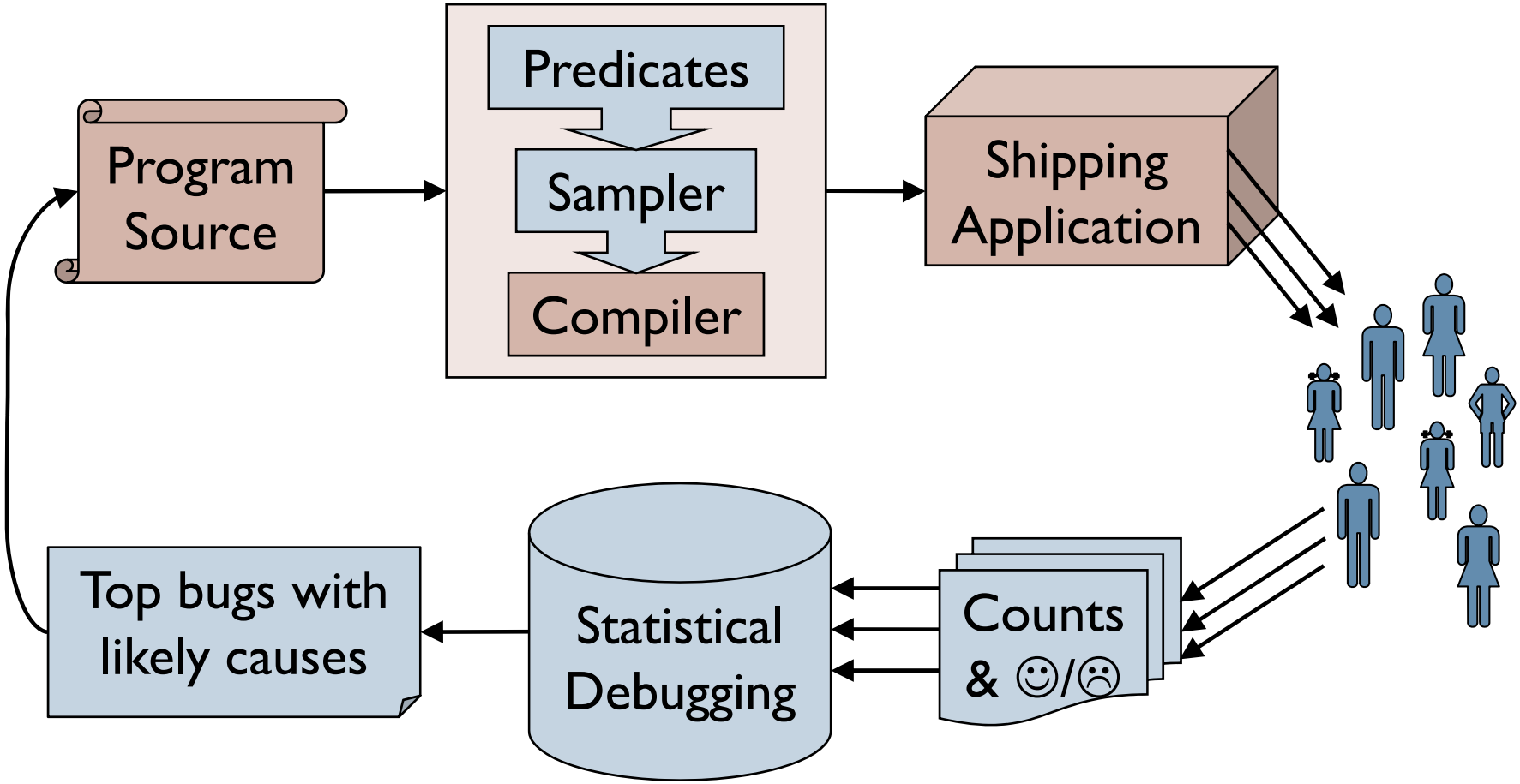
Testing as Approximation of Reality

- Microsoft's Watson error reporting system
  - Crash reports from 500,000 separate programs
  - x% of software errors cause 50% of user crashes
  - Care to guess what x is?
- 1% of software errors cause 50% of user crashes
- Small mismatch ➔ big problems (sometimes)
- Big mismatch ➔ small problems? (sometimes!)
  - Perfection is usually not an economically viable option

Goal: Measure Reality

- Software quality as an *empirical* science
  - Observed trends rather than logical proofs
  - Biologists do pretty well, even without source code!
- Users vastly outnumber testers
  - 500,000,000 Halo 2 games in 19 months
  - Real-world executions are most important
- Post-deployment bug hunting
  - Collect feedback data & mine for bug causes
  - This talk: late-stage analyses for higher-level failure models

Cooperative Bug Isolation Architecture



Delta Latent Dirichlet Allocation (ΔLDA)

$$p(z_k = i \mid \mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) \propto \left(\frac{n^i_{-k,j_k} + \beta^i_{j_k}}{n^i_{-k,*} + \sum_{j'}^{W} \beta^i_{j'}} \right) \left(\frac{n^{d_k}_{-k,i} + \alpha_i^{o_k}}{n^i_{-k,*} + \sum_{i'}^{N_u+N_b} \alpha_{i'}^{o_k}} \right)$$

$$p(\mathbf{w}, \mathbf{z}) = p(\mathbf{w} \mid \mathbf{z}) p(\mathbf{z})$$

$$p(\mathbf{w} \mid \mathbf{z}) = \prod_i^N \int p(\phi_i \mid \beta) \prod_j^W \phi_{ij}^{n^i_j} d\phi_i$$

$$p(\mathbf{z}) = \prod_d^D \int p(\phi_d \mid \alpha) \prod_i^N \phi_{di}^{n^d_i} d\phi_d$$

Delta Latent Dirichlet Allocation (ΔLDA)

$$p(z_k = i \mid \mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) \propto \left(\frac{n^i_{-k,j_k} + \beta^i_{j_k}}{n^i_{-k,*} + \sum_{j'}^{W} \beta^i_{j'}} \right) \left(\frac{n^{d_k}_{-k,i} + \alpha_i^{o_k}}{n^i_{-k,*} + \sum_{i'}^{N_u+N_b} \alpha_{i'}^{o_k}} \right)$$

$$p(\mathbf{w}, \mathbf{z}) = p(\mathbf{w} \mid \mathbf{z}) p(\mathbf{z})$$

$$p(\mathbf{w} \mid \mathbf{z}) = \prod_i^N \int p(\phi_i \mid \beta) \prod_j^W \phi_{ij}^{n^i_j} d\phi_i$$

$$p(\mathbf{z}) = \prod_d^D \int p(\phi_d \mid \alpha) \prod_i^N \phi_{di}^{n^d_i} d\phi_d$$

Delta Latent Dirichlet Allocation (ΔLDA)

$$p(z_k = i \mid \mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) \propto \left(\frac{n^i_{-k,j_k} + \beta^i_{j_k}}{n^i_{-k,*} + \sum_{j'}^{W} \beta^i_{j'}} \right) \left(\frac{n^{d_k}_{-k,i} + \alpha_i^{o_k}}{n^i_{-k,*} + \sum_{i'}^{N_u+N_b} \alpha_{i'}^{o_k}} \right)$$

Let's not talk about this right now, OK?

$$p(\mathbf{w}, \mathbf{z}) = p(\mathbf{w} \mid \mathbf{z}) p(\mathbf{z})$$

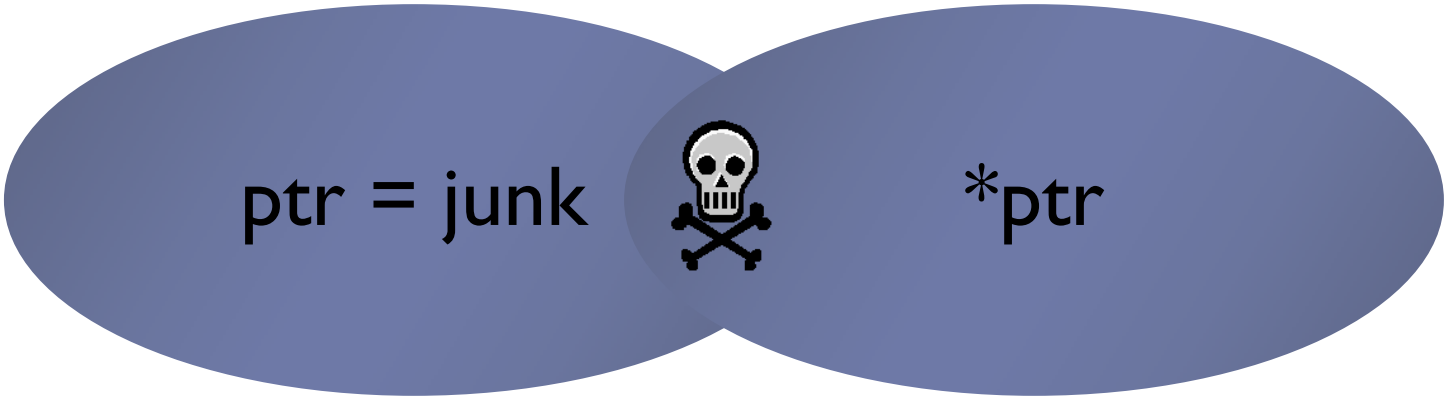
$$p(\mathbf{w} \mid \mathbf{z}) = \prod_i^N \int p(\phi_i \mid \beta) \prod_j^W \phi_{ij}^{n^i_j} d\phi_i$$

$$p(\mathbf{z}) = \prod_d^D \int p(\phi_d \mid \alpha) \prod_i^N \phi_{di}^{n^d_i} d\phi_d$$

Limitations of Simple Predicates

- Each predicate partitions runs into 2 sets:
  - Runs where it was true
  - Runs where it was false
- Can accurately predict bugs that match this partition
- Unfortunately, some bugs are more complex
  - Complex border between good & bad
  - Requires richer language of predicates

Motivation: Bad Pointer Errors



- In function `exif_mnote_data_canon_load`:
  -
```
for (i = 0; i < c; i++) {
    …
    n->count = i + 1;
    …
    if (o + s > buf_size) return;
    …
    n->entries[i].data = malloc(s);
    …
}
```
- Crash on later use of `n->entries[i].data`

Motivation: Bad Pointer Errors

Kinds of Predicate	Best Predicate	Score
Simple Only	new `len` == old `len`	0.71
Simple & Compound	`o + s > buf_size` ∧ `offset < len`	0.94

- In function `exif_mnote_data_canon_load`:
  -
```
for (i = 0; i < c; i++) {
    …
    n->count = i + 1;
    …
    if (o + s > buf_size) return;
    …
    n->entries[i].data = malloc(s);
    …
}
```
- Crash on later use of `n->entries[i].data`

Great! So What's the Problem?

- Too many compound predicates
  - 2^{2^N} functions of N simple predicates
  - N^2 conjunctions & disjunctions of two variables
  - $N \sim 100$ even for small applications
- Incomplete information due to sampling
- Predicates at different locations

Conservative Definition

- A conjunction $C = p_1 \wedge p_2$ is true in a run iff:
  - p_1 is true at least once and
  - p_2 is true at least once
- Disjunction is defined similarly
- Disadvantage:
  - C may be true even if p_1, p_2 never true simultaneously
- Advantages:
  - Monitoring phase does not change
  - $p_1 \wedge p_2$ is just another predicate, inferred offline

Three-Valued Truth Tables

- For each predicate & run, three possibilities:
  1. True at least once
  2. False at least once, but never true
  3. Never observed

Conjunction: $p_1 \wedge p_2$

p_2 \ p_1	T	F	?
T	T	F	?
F	F	F	**F**
?	?	**F**	?

Disjunction: $p_1 \vee p_2$

p_2 \ p_1	T	F	?
T	T	T	**T**
F	T	F	?
?	**T**	?	?

Mixed Compound & Simple Predicates

- Compute score of each conjunction & disjunction
  - $C = p_1 \wedge p_2$
  - $D = p_1 \vee p_2$
- Compare to scores of constituent simple predicates
  - Keep if higher score: better partition between good & bad
  - Discard if lower score: needless complexity
- Integrates easily into iterative ranking & elimination

Still Too Many

- Complexity: $N^2 \cdot R$
  - N = number of simple predicates
  - R = number of runs being analyzed
  - 20 minutes for N ~ 500, R ~ 5,000
- Idea: early pruning optimization
- Compute score upper bound and discard if too low
  - "Too low" = lower than constituent simple predicates
- Reduces $O(R)$ to $O(1)$ per complex predicate

Upper Bound On Score

- ↑ Harmonic mean

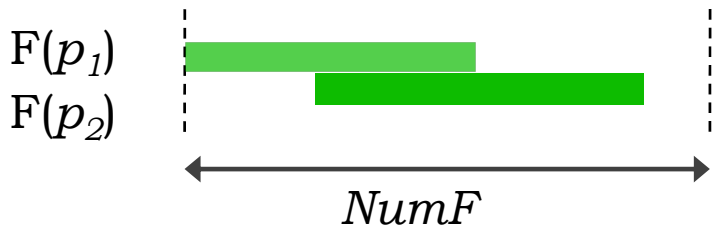
$$\uparrow Increase(C) = \frac{\uparrow F(C)}{\uparrow F(C) + \downarrow S(C)} - \frac{\downarrow F(C\ \text{obs})}{\downarrow F(C\ \text{obs}) + \uparrow S(C\ \text{obs})}$$

$$\uparrow Sensitivity(C) = \frac{\uparrow \log F(C)}{\log NumF}$$

- Upper Bound on C = $p_1 \wedge p_2$
  - Find ↑F(C), ↓S(C), ↓F(C obs) and ↑S(C obs)
  - In terms of corresponding counts for p_1, p_2

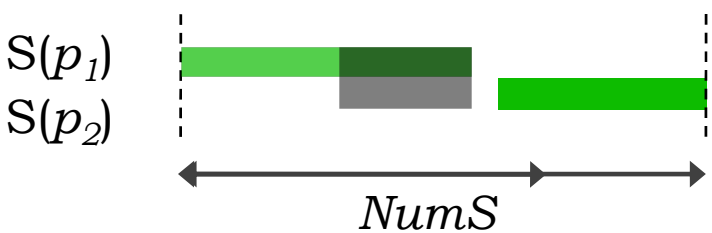
↑F(C) and ↓S(C) for conjunction

- ↑F(C): true runs completely overlap



Min(F(p_1), F(p_2))

- ↓S(C): true runs are disjoint

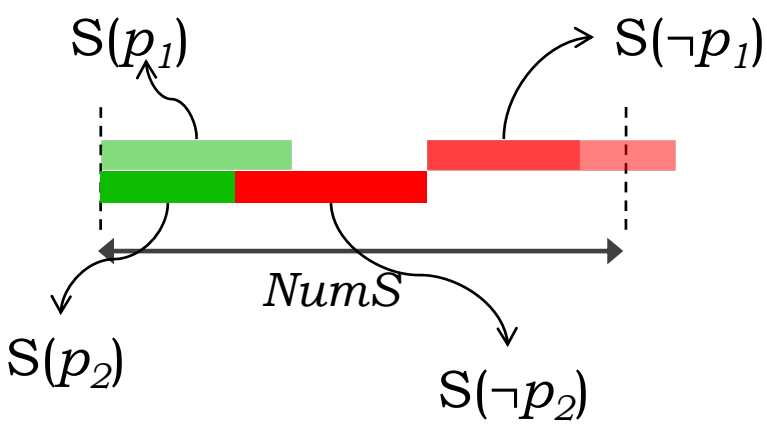


either 0 or
S(p_1)+S(p_2) - *NumS*

(whichever is larger)

↑S(C obs)

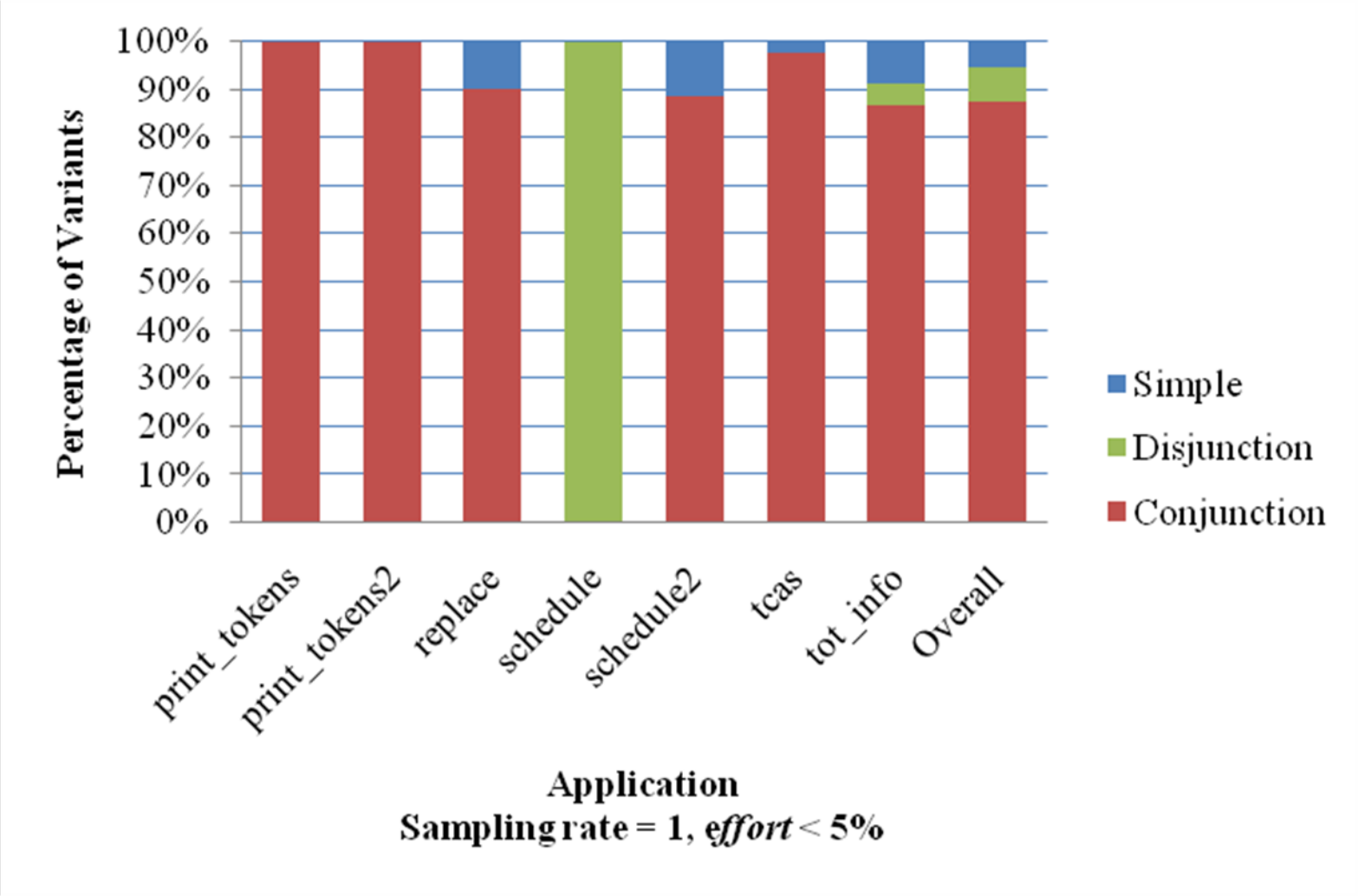
- Maximize two cases
- C = true
  - True runs of p_1, p_2 overlap
- C = false
  - False runs of p_1, p_2 are disjoint



Min(S(p_1), S(p_2)) + S(¬p_1) + S(¬p_2)
or
NumS

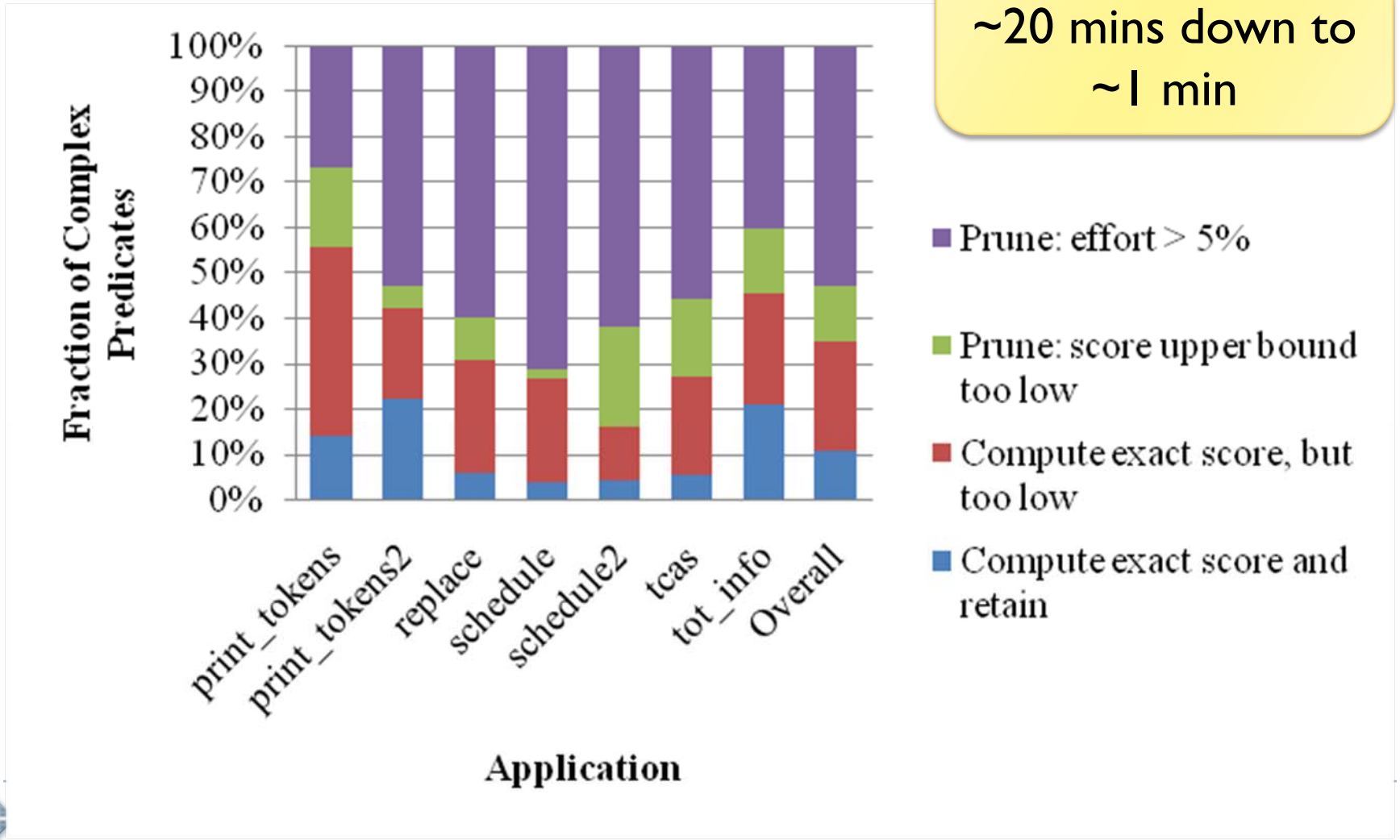
(whichever is smaller)

Evaluation:
What kind of predicate has the top score?



Evaluation: Effectiveness of Pruning

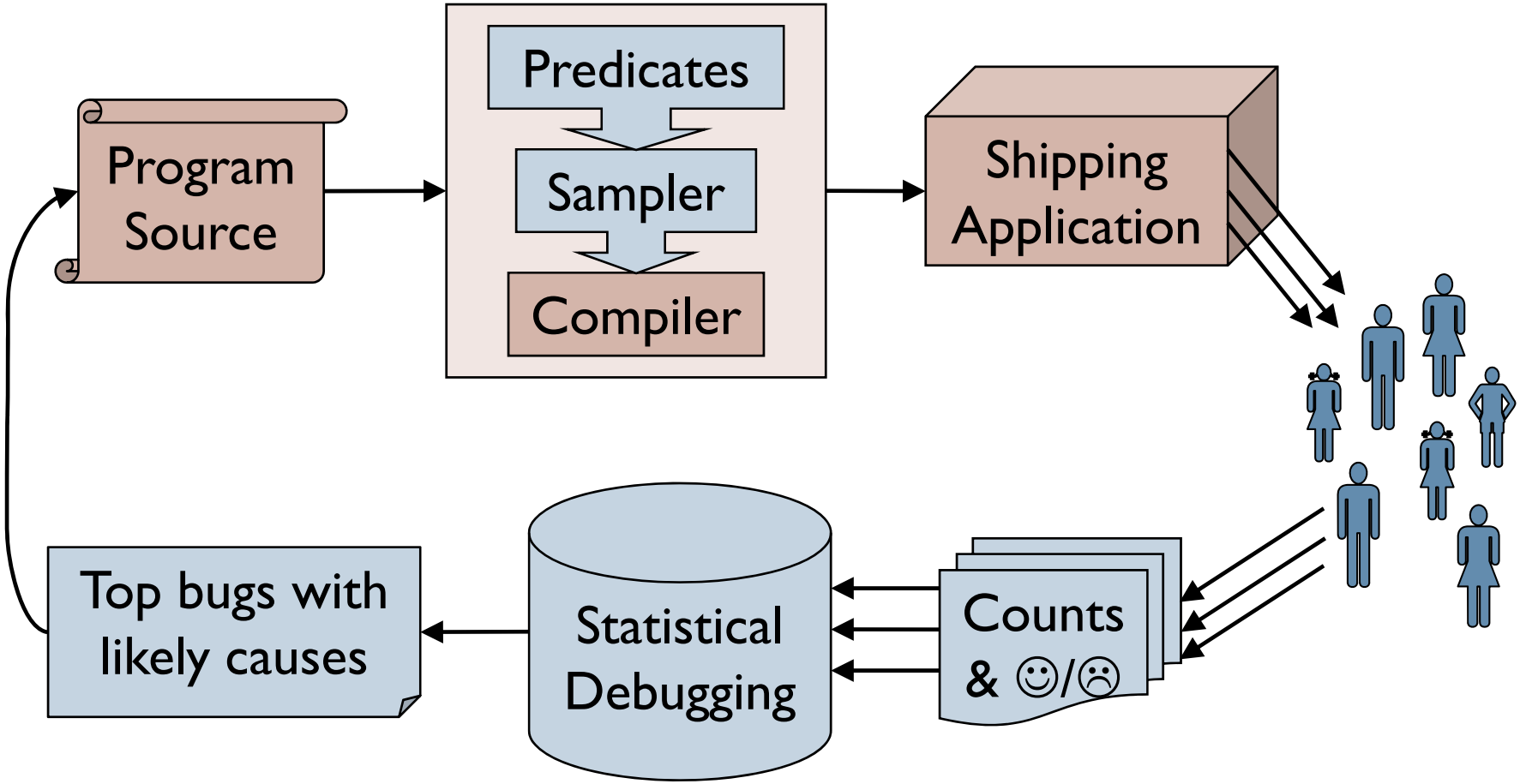
Analysis time: from ~20 mins down to ~1 min



Practical Experiences With CBI

- Bug predictor is often the smoking gun, but not always
- "Redundant" predicates actually carry clues
  - Especially when spread across source code
- Bidirectional thinking can be very tricky
  - Debuggers only train us to think backwards

Putting Predictors in Context



Goal: Find Minimal Failure Path

- Explore paths subject to constraints
  - Dynamic info (bug predictors, failure stack)
  - Static info (control flow, dataflow)
  - Interactive guidance from user
- Want short, feasible path that exhibits bug
  - Undecidable ☹
  - But still a very interesting problem!

A Debugging Scenario

```
int **a;

void main()
{
  ...
  process_input(a);
  ...
}

void clear_array(int **a)
{
  for (...)
    a[i] = NULL;
}
```

```
void process_input(int **a)
{
  cin >> input;
  switch (input) {
    case 'e':
        clear_array(a);
        break;
    case 'p':
        ...
    ...
  }
  ...
  a[i][j]++;
}
```

A Debugging Scenario

```
int **a;

void main()
{
  ...
  process_input(a);
  ...
}

void clear_array(int **a)
{
  for (...)
    a[i] = NULL;
}
```

```
void process_input(int **a)
{
  cin >> input;
  switch (input) {
    case 'e':
        clear_array(a);
        break;
    case 'p':
        ...
    ...
  }
  ...
  a[i][j]++;
}
```

A Debugging Scenario

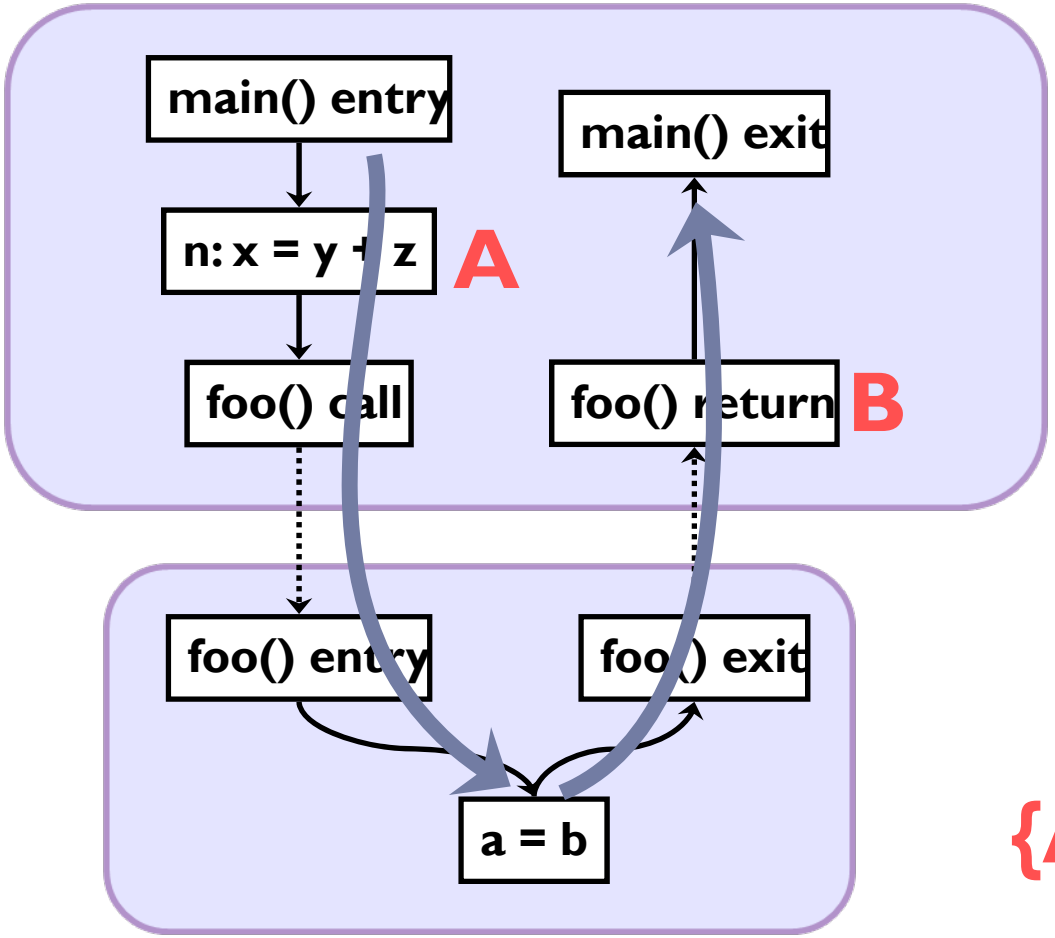
```
int **a;

void main()
{
  ...
  process_input(a);
  ...
}

void clear_array(int **a)
{
  for (...)
    a[i] = NULL;
}
```

```
void process_input(int **a)
{
  cin >> input;
  switch (input) {
    case 'e':
        clear_array(a);
        break;
    case 'p':  return
        ...
    ...
  }
  ...
  a[i][j]++;
}
```

Weighted Pushdown Systems

- PDS: finite automaton with stack
  - Describes control-feasible paths, including call/return
- WPDS: track dataflow "payload" along each path
  - Weight as transfer function on dataflow facts
- Instantiate WPDS by defining:
  - Initial weight associated with each PDS transition
  - Binary *extend* operator ($\otimes$) for concatenating paths
  - Binary *combine* operator ($\oplus$) for joining paths

Weight as Set of Bug Predictors



$$\{A\} \otimes \{B\} = \{A, B\}$$

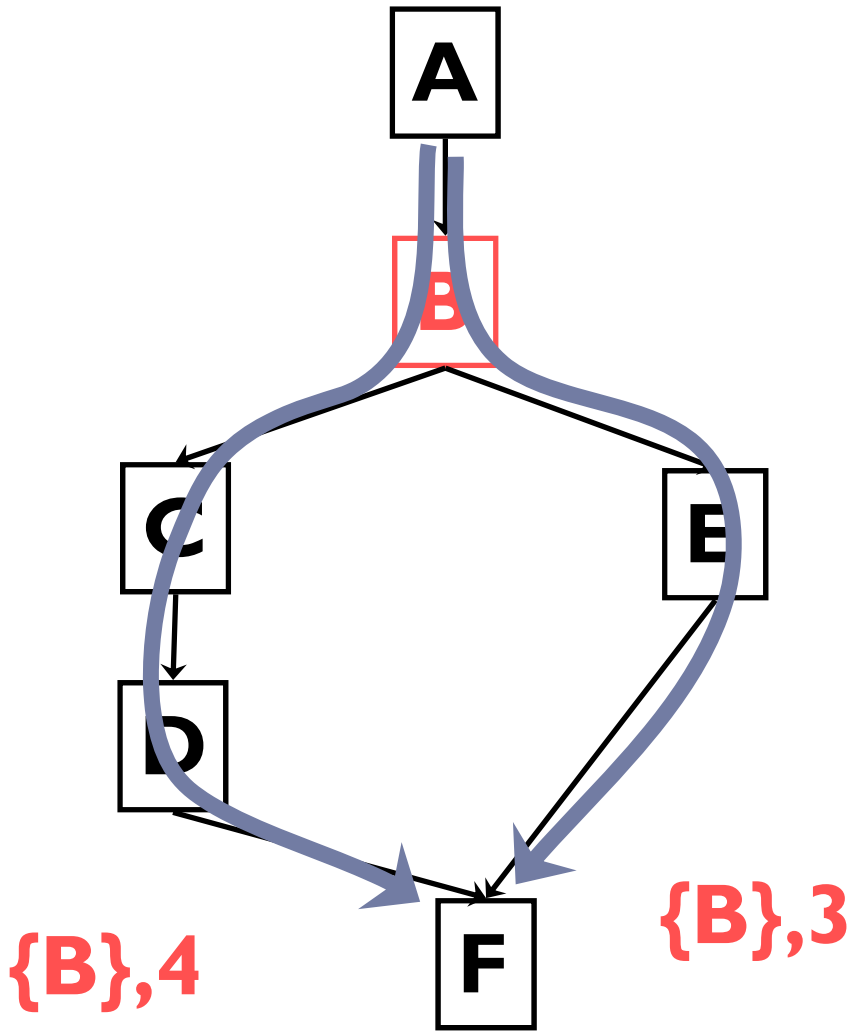
Weight as Set of Bug Predictors

- Path weight is set of predictors touched
- Singleton set at each bug predictor
  - Use "redundant" predictors suppressed earlier
  - Empty set at all other CFG nodes
- Path extension is set union
- Path merging: select path with biggest set?

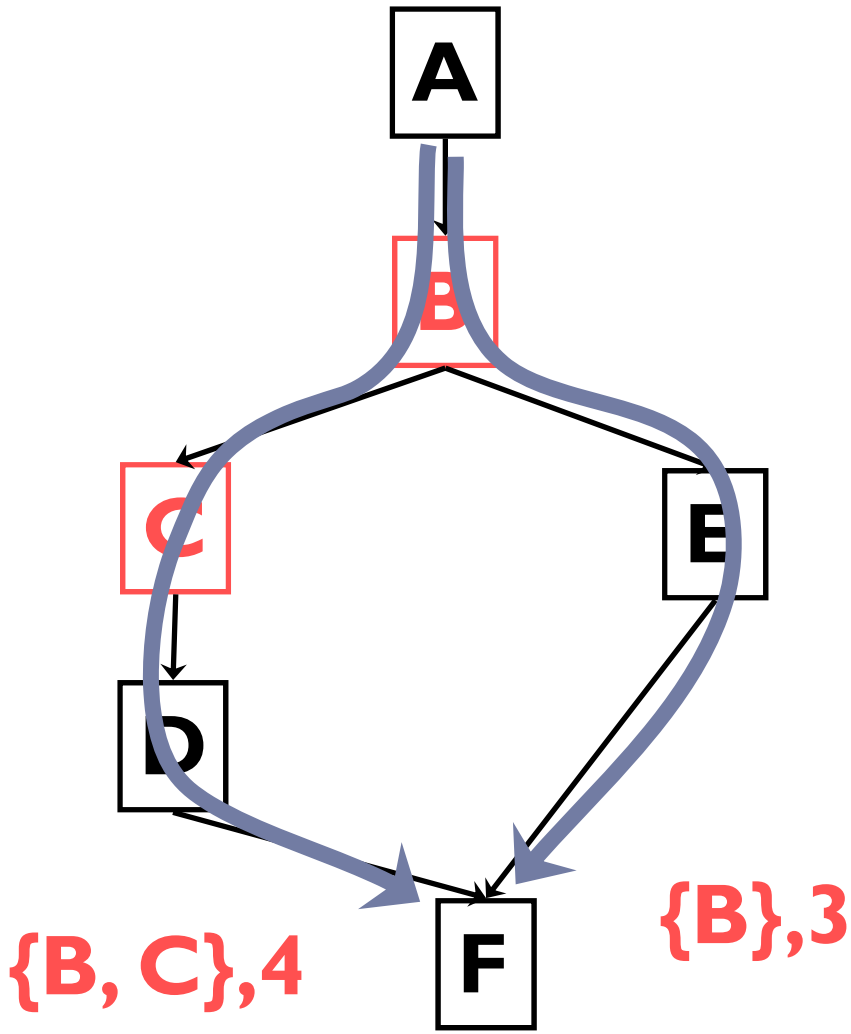
How Good is a Path?

- If two paths touch same bug predictors, which one do we want?
  - Shortest one!
- Need to reflect length in path weights
  - Weight = (set of bug predictors, path length)
  - Extend operator takes union of sets, sum of lengths
  - Initial weights have length 1 for every transition

Path Weight Merging



Path Weight Merging



- One path per set of predictors touched
  - Exponential in # of predictors
  - Near linear in program size

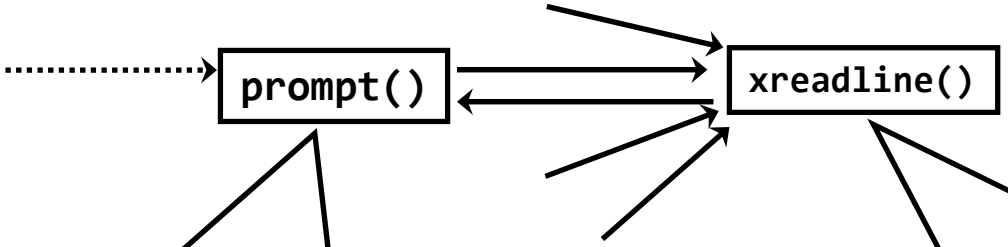
User Guidance & Interactivity

- Ordering constraints: A before B
  - $\{A\} \otimes \{B\} = \{A, B\}$
  - $\{B\} \otimes \{A\} = \perp$
  - Requires rebuild of solution automaton
- Steer path by changing scoring of nodes & paths
  - Assign scores based on statistical metrics
  - Avoid selected nodes (anti-predictors)
  - No rebuild of solution automaton
- Easy to mix in (most) dataflow analyses

Experiments: Siemens Test Suite

- Each program contains a single bug
  - Chose three programs where statistical models "fail"
  - Top bug predictors miss the true bug
- Reconstructed failure paths pass through the buggy lines of code

Experiments: ccrypt




```
int prompt(void) {
    …
    line = xreadline(fin, cmd.name);
    return (!strcmp(line, 'y') ||
            !strcmp(line, "yes");
}
```

```
char *
xreadline(FILE *fin, char *name) {
    int buflen = INITSIZE;

    char *buf = xalloc(buflen, name);
    char *res, *nl;

    res = fgets(buf, INITSIZE, fin);
    if (res == NULL) {
        free(buf);
        return NULL;
    }
    nl = strchr (buf, '\n');
    …
    return buf;
}
```

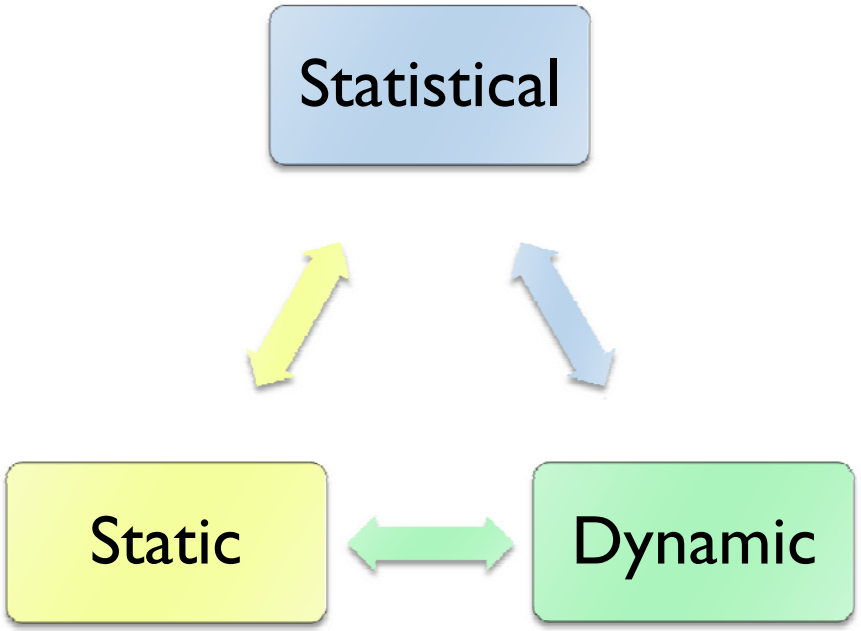
Dataflow isolates call in **`prompt()`** as culprit

Experiments: bc

- Calculator tool with buffer overrun
- Statistical model: two bug predictor lists
  - Suggests two bugs in the program
- But reconstructed failure paths are identical!
  - Correctly reveals that only one bug is present

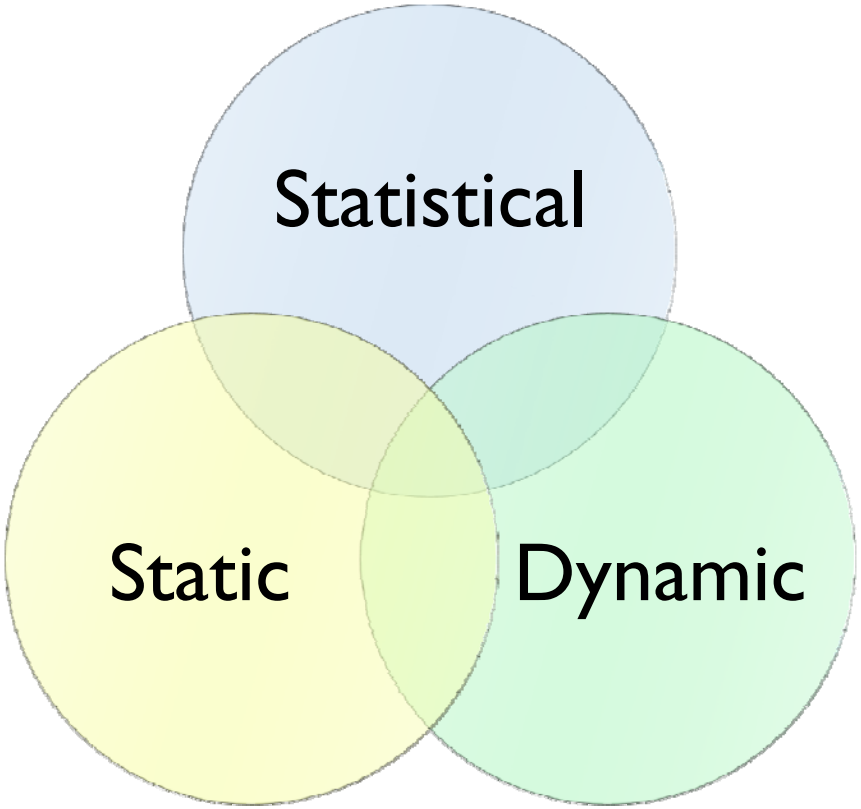
Hybrid Analysis Today

- Focus is on *methods*
  - Algorithms & models
- Three distinct sources of knowledge
  - Three distinct communities
- Use one to focus or approximate another



Hybrid Analysis Future

- Complementary styles
  - Limited questions + guaranteed answers
  - Arbitrary questions + approximate answers
- One common goal
  - Describe the behavior of a complex dynamic system



Static + Statistical = ???

- Static analyses describe what is possible
- Statistical models describe what is likely
  - Likely behaviors can be statically impossible!
- Can we deeply integrate static program structure into statistical models?
  - Challenges of scale and social engineering

Static + Statistical = SRL?

- Machine learning community is (re)discovering structured problems
  - *"Statistical relational learning (SRL) addresses the challenge of applying statistical inference to problems which involve rich collections of objects linked together in complex relational networks."*
- Software is highly structured
  - We have decades of research on extracting that structure
  - We even have source code!
- PL folks: need to get over our fear of real numbers

Lessons Learned

- Can learn a lot from actual executions
  - Users are running buggy code anyway
  - We should capture some of that information
- Great potential in hybrid approaches
  - Dynamic: reality-driven debugging
  - Statistical: best-effort with uncertainty
  - Static: use program structure to fill in the gaps

Vision for Statistical Debugging

- Bug triage that directly reflects reality
  - Learn the most, most quickly, about the bugs that happen most often
- Variability is a benefit rather than a problem
  - Results grow stronger over time
- Find bugs while you sleep!

Join the Cause!

The Cooperative Bug Isolation Project

http://www.cs.wisc.edu/cbi/