Scalable Statistical Bug Isolation

Ben Liblit *Alex Aiken*
Alice Zheng *Mike Jordan*
Mayur Naik
UC Berkeley / Stanford

Motivation: Users Matter

- Imperfect world with imperfect software
  - Ship with known bugs
  - Users find new bugs
  - Bug fixing is a matter of triage
- Important bugs happen often, to many users
- Can users help us find and fix bugs?
  - Learn a little bit from each of many runs

Users as Debuggers

- Users run instrumented programs
- Instrumentation must not disturb individual users
  - ⇒ Sparse sampling: spread costs wide and thin
- Feedback reports sent to central server
- Collected data is statistically analyzed to identify causes of bugs

Instrumentation

- Predicates tested at particular program points
- Feedback report R sent to server
  - Run succeeded/failed
  - Bit vector of predicates P.

    $R(P) = 1 \Leftrightarrow$ P was true at least once in R
- B denotes a bug, β a bug profile

Instrumentation

- If $R(P)=1 \implies R \in \beta$, then P is called a predictor of B
- Statistical debugging selects a small subset S of the the set of all predicates such that S has predictors of all bugs.
- Predictors in S are ranked from most to least important

Instrumentation

- Impossible to gather complete execution traces of every run
- Instead, use a combination of sparse random sampling and client-side summarization of the data
- Each time instrumentation code is reached, a coin flip decides whether the instrumentation is executed or not

Instrumentation Schemes

- Branches
  - Track two predicates, for both branches
- Returns
  - Six predicates for return value: <0, >0 $=0$, ≤ 0, ≥ 0
- Scalar-pairs
  - Relationship between a variable and another var or a constant

Instrumentation

- Observation is a single dynamic check of all predicates at a single instrumentation site
- P observed
- P observed to be true
- E.g., sampling a negative value
  - <0, >0 $=0$, ≤ 0, ≥ 0 are observed
  - Only three of them are true

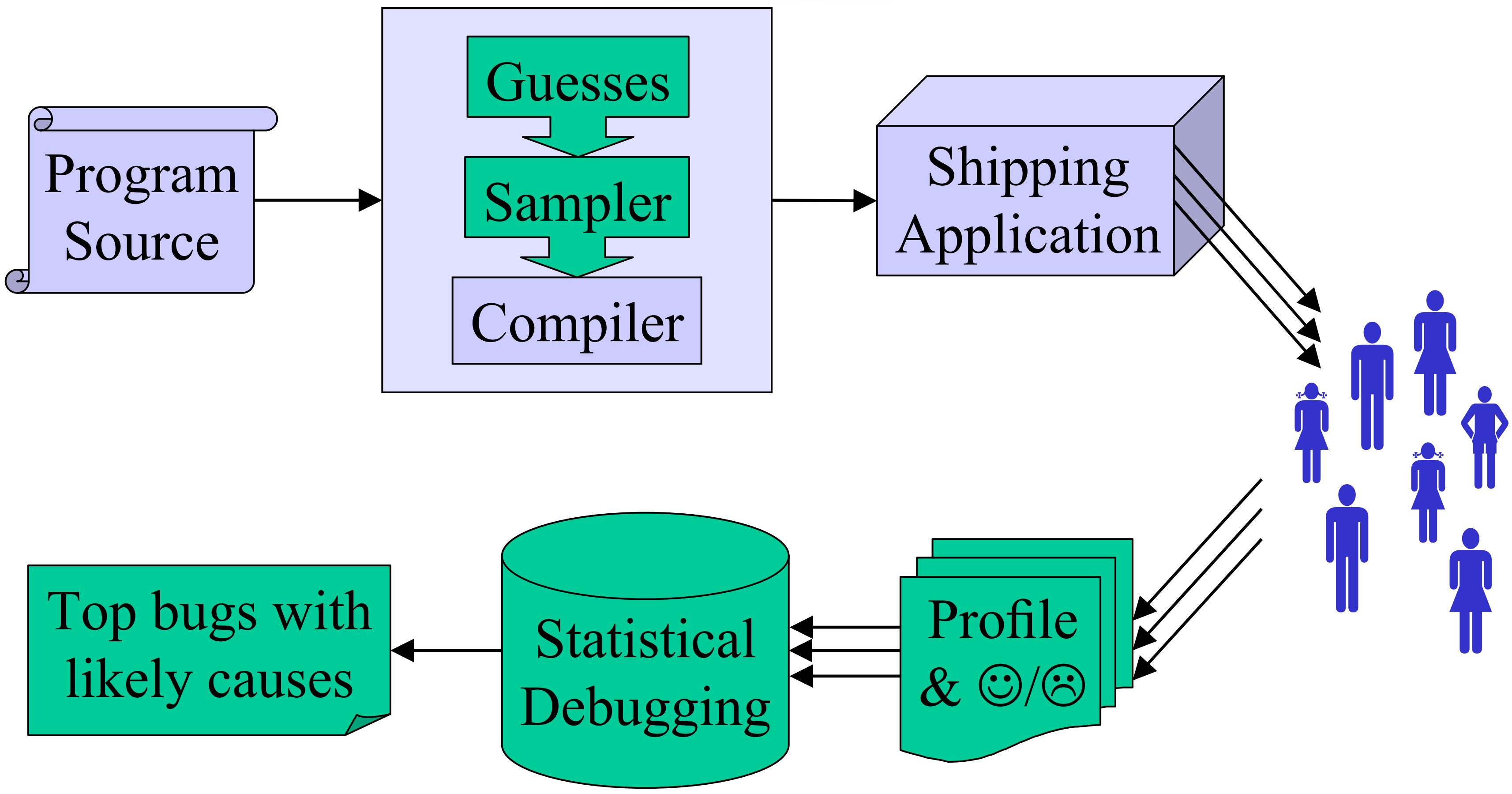
Previous Work

- Authors focused on lightweight instrumentation and sampling
- Two preliminary statistical debugging algorithms
  - Regularized logistic regression
- In practice, there were problems applying the algorithms

Previous work issues

- 100,000+ predictors, many of which redundant
- Prevalence of predicates predicting multiple bugs, which are useless for islating causes of individual bugs
- 99% of predicates are not predictive of anything
- The remaining predicates containg some bug, sub-bug and super-bug predictors

Bug Isolation Architecture



Cause Isolation Algorithm

- Input: set of feedback reports from individual program runs R
- Output: List of most likely predictors of bugs
- Simple idea behind the algorithm:

1. Identify the most important bug B.
    a) Eliminate predicates with no predictive power.
    b) Rank the surviving predicates by importance.
2. Fix B, and repeat.

Example

```
f = …                  (a)
if (f==NULL) {         (b)
     x = 0;            (c)
    *f                 (d)
}
```

- Predicate `f == null` is correlated with failure
- Deterministic/non-deterministic
- Consider

```
if (…) f = … some valid pointer ;
*f;
```

Failure

- Let Crash be an atomic predicate that is true for failing runs and false for successful runs.
- Want to compute:

  Failure(P) ≡ Pr(Crash | P observed true)
- Estimate Failure(P) as:

$$Failure(P) = \frac{F(P)}{S(P)+F(P)}$$

Properties of Failure(P)

- It is not affected by the set of runs in which P is not observed true.
- Runs in which is not observed at all are also do not affect Failure(P).
- Provides a generalized idea for deterministic/nondeterministic bugs.

Back to the Example

```
f = …                (a)
if (f==NULL) {       (b)
     x = 0;          (c)
    *f               (d)
}
```

- We have Failure(f==NULL) = 1.0 at (b)
- What about Failure(x==0)?
- High Failure(P) does not imply P is a cause of a bug.

Solution: Context(P)

```
f = …                (a)
if (f==NULL) {       (b)
     x = 0;          (c)
    *f               (d)
}
```

- Score a predicate by how much difference it makes if it is observed to be true.

  $$\text{Context}(P) \equiv \Pr(\text{Crash} \mid P \text{ observed})$$

Increase(P)

- Estimate Context(P):

$$Context(P) = \frac{F(Pobserved)}{S(Pobserved) + F(Pobserved)}$$

- How much does P being true increase the probability of failure over simply reaching the line where P is sampled?

Increase(P) ≡ Failure(P) – Context(P)

Example, again

```
f = …                (a)
if (f==NULL) {       (b)
     x = 0;          (c)
    *f               (d)
}
```

- Failure(x==0) = Context(x==0) = 1.0
- Increase(x==0) = 0
- Predicate P with Increase(P) < 0 has no predictive power – can be safely discarded

Pruning Predicates

- Attach confidence intervals to scores
- Which predicates are discarded?
  - Unreachable, invariants, control-dependent on a true cause
- Pruning localizes bugs at their cause, not at to crash site

Cause Isolation Algorithm

- Input: set of feedback reports from individual program runs R
- Output: List of most likely predictors of bugs
- Simple idea behind the algorithm:

1. Identify the most important bug B.
   a) Eliminate predicates with no predictive power.
   b) Rank the surviving predicates by importance.
2. Fix B, and repeat.

Ranking Predicates by F(P)

(a) Sort descending by F(P)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.007 ± 0.012	22554	5045	27599	`files[filesindex].language != 15`
	0.176	0.007 ± 0.012	22566	5045	27611	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.012	22571	5045	27616	`strcmp != 0`
	0.176	0.007 ± 0.013	18894	4251	23145	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.013	18885	4240	23125	`files[filesindex].language != 14`
	0.176	0.008 ± 0.013	17757	4007	21764	`filesindex >= 25`
	0.177	0.008 ± 0.014	16453	3731	20184	`M < M`
	0.176	0.261 ± 0.023	4800	3716	8516	`config.winnowing_window_size != argc`

.... 2732 additional predictors follow ....

- Highly non-deterministic
- Super-bug predictors

Ranking Predicates by Increase(P)

(b) Sort descending by Increase(P)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.065	0.935 ± 0.019	0	23	23	((*(fi + i)))->this.last_token < filesbase
	0.065	0.935 ± 0.020	0	10	10	((*(fi + i)))->other.last_line == last
	0.071	0.929 ± 0.020	0	18	18	((*(fi + i)))->other.last_line == filesbase
	0.073	0.927 ± 0.020	0	10	10	((*(fi + i)))->other.last_line == yy_n_chars
	0.071	0.929 ± 0.028	0	19	19	bytes <= filesbase
	0.075	0.925 ± 0.022	0	14	14	((*(fi + i)))->other.first_line == 2
	0.076	0.924 ± 0.022	0	12	12	((*(fi + i)))->this.first_line < nid
	0.077	0.923 ± 0.023	0	10	10	((*(fi + i)))->other.last_line == yy_init

2732 additional predictors follow

- Number of failing runs is small
- Sub-bug predictors

Ranking by Harmonic Mean

(c) Sort descending by harmonic mean

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.824 ± 0.009	0	1585	1585	`files[filesindex].language > 16`
	0.176	0.824 ± 0.009	0	1584	1584	`strcmp > 0`
	0.176	0.824 ± 0.009	0	1580	1580	`strcmp == 0`
	0.176	0.824 ± 0.009	0	1577	1577	`files[filesindex].language == 17`
	0.176	0.824 ± 0.009	0	1576	1576	`tmp == 0 is TRUE`
	0.176	0.824 ± 0.009	0	1573	1573	`strcmp > 0`
	0.116	0.883 ± 0.012	1	774	775	`((*(fi + i)))->this.last_line == 1`
	0.116	0.883 ± 0.012	1	776	777	`((*(fi + i)))->other.last_line == yyleng`

. . . . 2732 additional predictors follow . . . .

- Balance between sensitivity and specificity

Redundancy Elimination

- Iterative algorithm used to eliminate redundant predicates:
  - Rank predicates by Importance
  - Remove the top-ranked predicate P and discard all runs where R(P) = 1.
  - Repeat these steps until the set of runs is empty or the set of predicates is empty

Experiments

- Algorithm applied to 5 case studies
- Each study has 32,000 random inputs
- Algorithm proved extremely effective in reducing the number of predicates the user has to examine
- Sampling is important

Eliminating Predicates

Table 2: Summary statistics for bug isolation experiments

		Runs			Predicate Counts		
	Lines of Code	Successful	Failing	Sites	Initial	*Increase* > 0	Elimination
MOSS	6001	26,299	5598	35,223	202,998	2740	21
CCRYPT	5276	20,684	10,316	9948	58,720	50	2
BC	14,288	23,198	7802	50,171	298,482	147	2
RHYTHMBOX	56,484	12,530	19,431	14,5176	857,384	537	15
EXIF	10,588	30,789	2211	27,380	156,476	272	3

Related Work

- Earlier work used logistic regression
  - Almost all of the predictors are sub-bug or super-bug
- Podgurski et al. apply statistical methods in classifying failure reports.
  - No sampling used
  - Logistic regression used
- Daikon project
  - Assumes complete monitoring

Conclusions

- Public deployment is challenging
  - Real world code pushes tools to their limits
  - Large user communities take time to build

- However, there is strength in numbers:

      many users
  \+ statistical modeling
  = find bugs while you sleep!