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.
    \[ R(P) = 1 \iff P \text{ was true at least once in } R \]
• B denotes a bug, \( \beta \) 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 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: $<0, >0 =0, \leq 0,$ $\geq 0$

• 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
  - $<0$, $>0 =0$, $\leq0$, $\geq0$ 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 isolating causes of individual bugs
- 99% of predicates are not predictive of anything
- The remaining predicates containing some bug, sub-bug and super-bug predictors
Bug Isolation Architecture

Program Source → Sampler → Compiler → Shipping Application

Guesses

Top bugs with likely causes

Statistical Debugging

Profile & 🧐/.ma

Customers
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

\begin{verbatim}
f = ...               (a)
if (f==NULL) {
    x = 0;          (b)
    *f              (c)
}                   (d)
\end{verbatim}

- **Predicate** $f == \text{null}$ **is correlated with failure**
- **Deterministic/non-deterministic**
- **Consider**

```c
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:
  \[
  \text{Failure}(P) \equiv \Pr(\text{Crash} | \text{P observed true})
  \]
- Estimate Failure(P) as:

\[
\text{Failure}(P) = \frac{F(P)}{S(P) + F(P)}
\]
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

```c
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)

\[
\text{f} = \ldots \quad (a) \\
\text{if} \ (\text{f}==\text{NULL}) \ {\{ \quad (b) \\
\quad \text{x} = 0; \quad (c) \\
\quad \text{*f} \quad (d) \\
\}}
\]

- 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):

\[
Context(P) = \frac{F(P_{oberved})}{S(P_{oberved}) + F(P_{oberved})}
\]

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

\[
Increase(P) \equiv Failure(P) - Context(P)
\]
Example, again

\[ f = \ldots \] (a)
\[ \text{if} \ (f==\text{NULL}) \ {\} \] (b)
  \[ x = 0; \] (c)
  \[ *f \] (d)

- \text{Failure}(x==0) = \text{Context}(x==0) = 1.0
- \text{Increase}(x==0) = 0
- \text{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 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)

- Highly non-deterministic
- Super-bug predictors
Ranking Predicates by Increase(P)

(b) Sort descending by Increase(P)

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
<th>F + S</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.935 ± 0.019</td>
<td>0</td>
<td>23</td>
<td>23</td>
<td>(**(fi + i)) \rightarrow this.last_token &lt; filebase</td>
</tr>
<tr>
<td>0.065</td>
<td></td>
<td>0.935 ± 0.020</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>(**(fi + i)) \rightarrow other.last_line == last</td>
</tr>
<tr>
<td>0.071</td>
<td></td>
<td>0.929 ± 0.020</td>
<td>0</td>
<td>18</td>
<td>18</td>
<td>(**(fi + i)) \rightarrow other.last_line == filebase</td>
</tr>
<tr>
<td>0.073</td>
<td></td>
<td>0.927 ± 0.020</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>(**(fi + i)) \rightarrow other.last_line == yy_n_chars</td>
</tr>
<tr>
<td>0.071</td>
<td></td>
<td>0.929 ± 0.028</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td>bytes &lt;= filebase</td>
</tr>
<tr>
<td>0.075</td>
<td></td>
<td>0.925 ± 0.022</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>(**(fi + i)) \rightarrow other.first_line == 2</td>
</tr>
<tr>
<td>0.076</td>
<td></td>
<td>0.924 ± 0.022</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>(**(fi + i)) \rightarrow this.first_line &lt; nid</td>
</tr>
<tr>
<td>0.077</td>
<td></td>
<td>0.923 ± 0.023</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>(**(fi + i)) \rightarrow other.last_line == yy_init</td>
</tr>
</tbody>
</table>

2732 additional predictors follow

- Number of failing runs is small
- Sub-bug predictors
Ranking by Harmonic Mean

- Balance between sensitivity and specificity

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
<th>F + S</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.176</td>
<td>0.824±0.009</td>
<td>0</td>
<td>1585</td>
<td>1585</td>
<td>files[fileindex].language &gt; 16</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824±0.009</td>
<td>0</td>
<td>1584</td>
<td>1584</td>
<td>strcmp &gt; 0</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824±0.009</td>
<td>0</td>
<td>1580</td>
<td>1580</td>
<td>strcmp == 0</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824±0.009</td>
<td>0</td>
<td>1577</td>
<td>1577</td>
<td>files[fileindex].language == 17</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824±0.009</td>
<td>0</td>
<td>1576</td>
<td>1576</td>
<td>tmp == 0 is TRUE</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824±0.009</td>
<td>0</td>
<td>1573</td>
<td>1573</td>
<td>strcmp &gt; 0</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.883±0.012</td>
<td>1</td>
<td>774</td>
<td>775</td>
<td>(**(fi + i)))-&gt;this.last_line == 1</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.883±0.012</td>
<td>1</td>
<td>776</td>
<td>777</td>
<td>(**(fi + i)))-&gt;other.last_line == yynlen</td>
</tr>
</tbody>
</table>

2732 additional predictors follow
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>
<thead>
<tr>
<th>Lines of Code</th>
<th>Successful</th>
<th>Failing</th>
<th>Sites</th>
<th>Initial</th>
<th>Increase &gt; 0</th>
<th>Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOSS</td>
<td>6001</td>
<td>26,299</td>
<td>5598</td>
<td>35,223</td>
<td>202,998</td>
<td>2740</td>
</tr>
<tr>
<td>cCrypt</td>
<td>5276</td>
<td>20,684</td>
<td>10,316</td>
<td>9948</td>
<td>58,720</td>
<td>50</td>
</tr>
<tr>
<td>BC</td>
<td>14,288</td>
<td>23,198</td>
<td>7802</td>
<td>50,171</td>
<td>298,482</td>
<td>147</td>
</tr>
<tr>
<td>Rhythmbox</td>
<td>56,484</td>
<td>12,530</td>
<td>19,431</td>
<td>14,5176</td>
<td>857,384</td>
<td>537</td>
</tr>
<tr>
<td>Exif</td>
<td>10,588</td>
<td>30,789</td>
<td>2211</td>
<td>27,380</td>
<td>156,476</td>
<td>272</td>
</tr>
</tbody>
</table>
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!