Cooperative Bug Isolation

Lecture 9
CS195

Software Release Cycle

1. Coders & testers in tight feedback loop
   - Detailed monitoring, high repeatability
   - Testing approximates reality

2. Testers & management declare "Ship it!"
   - Perfection is not an option

3. Everyone goes on vacation
   - What could possibly go wrong?

4. Upon return, hide from tech support
   - Much can go wrong
   - Users define reality
     - And it's different from what happens in testing

Our Goal: Measure Reality

- We measure bridges, airplanes, cars...
  - Where is the flight data recorder for software?

- Users are a vast, untapped resource
  - 11,000 new Kazoo users during this lecture
  - Users know what matters most

- In this talk: post-deployment bug hunting

Real Engineering Constraints

- Millions of lines of code
- Limited performance overhead
- Limited disk, network bandwidth
- Loose semantics of buggy programs
- Mix of controlled, uncontrolled code
- Threads
- Privacy and security
- Incomplete & inconsistent information

The Approach

1. Guess "potentially interesting" behaviors
   - Compile-time instrumentation

2. Collect sparse, fair subset of complete info
   - Generic sampling transformation
   - Feedback profile - outcome label

3. Find behavioral changes in good/bad runs
   - Statistical debugging
**Bug Isolation Architecture**

**Guessing What Is Interesting**

**Branches Are Interesting**

```c
if (p) ...
else ...
```

**Branch Predicate Counts**

```
++branch_17(!p);
if (p) ...
else ...
```

- Syntax yields instrumentation site
- Site yields predicates on program behavior
- Predicates are folded down into counts

**Returned Values Are Interesting**

```
n = fprintf(...);
```

**Returned Value Predicate Counts**

```
n = fprintf(...);
++call_41((n==0)+(n>=0));
```

- Syntax yields instrumentation site
- Site yields predicates on program behavior
- Predicates are folded down into counts
Pair Relationships Are Interesting

```c
int i, j, k;
...
i = ...;
```

Pair Relationship Predicate Counts

```c
int i, j, k;
...
i = ...;
++pair_6[(i==j)+(i>=j)];
++pair_7[(i==k)+(i>=k)];
++pair_8[(i==5)+(i>=5)];
```

Summarization and Reporting

- Feedback report is:
  - Vector of predicate counters
  - Success/failure outcome label
- No time dimension, for good or ill
- Still quite a lot to measure
  - What about performance?

Fair Sampling Transformation

Amortized Coin Tossing

- Randomized global countdown
  - Small countdown ⇒ upcoming sample
- Selected from geometric distribution
  - Inter-arrival time for biased coin toss
  - How many tails before next head?

Sampling the Bernoulli Way

- Decide to examine or ignore each site...
  - Randomly
  - Independently
  - Dynamically
  - Cannot use clock interrupt: no context
  - Cannot be periodic: unfair
  - Cannot toss coin at each site: too slow
Geometric Distribution

\[ \text{next} = \left\lfloor \frac{\log(\text{rand}(0,1))}{\log(1 - \frac{1}{D})} \right\rfloor + 1 \]

- \( D \) = mean of distribution
- = expected sample density

Weighing Acyclic Regions

- Each acyclic region:
  - Finite number of paths
  - Finite max number of instrumentation sites

Optimizations I

- Identify and ignore "weightless" functions
- Identify and ignore "weightless" cycles

- Cache global countdown in local variable
  - Global \( \rightarrow \) local at func entry \& after each call
  - Local \( \rightarrow \) global at func exit \& before each call

Optimizations II

- Avoid cloning
  - Instrumentation-free prefix or suffix
  - Weightless or singleton regions
- Static branch prediction at region heads
- Partition sites among several binaries
- Many additional possibilities...

What Does This Give Us?

- Absolutely certain of what we do see
  - Subset of dynamic behavior
  - Success/failure label for entire run
- Uncertain of what we don’t see
- Given enough runs, samples = reality
  - Common events seen most often
  - Rare events seen at proportionate rate
Playing the Numbers Game

Case Study: bc Crashing Bug

- Hunt for intermittent crash in bc-1.06
  - Stack traces suggest heap corruption
- 2729 runs with 9MB random inputs
  - Simulate a large, if confused, user base

Regularized Logistic Regression

- S-shaped cousin to linear regression
- Predict success/failure as function of counters
- Penalty factor forces most coefficients to zero
  - Large coefficient ⇒ highly predictive of failure

Top-Ranked Predictors

```c
void more_arrays ()
{
    #1: index > scale
    #2: index > use_math
    #3: index > opterr
    #4: index > next_func
    #5: index > i_base

    /* Copy the old arrays. */
    for (index = 1; index < old_count; index++)
        arrays[index] = old_ary[index];

    /* Initialize the new elements. */
    for (index = 1; index < v_count; index++)
        arrays[index] = NULL;
}
```

Bug Found: Buffer Overrun

```c
void more_arrays ()
{
    /* Copy the old arrays. */
    for (index = 1; index < old_count; index++)
        arrays[index] = old_ary[index];

    /* Initialize the new elements. */
    for (index = 1; index < v_count; index++)
        arrays[index] = NULL;
}
```
Limitations of Logistic Regression

- Linearly-weighted combination of features
  - What does this mean?
- Many correlated features
  - Unpredictable sharing of weights
- Assumes single mode of failure
  - Do you really have just one bug?

Ranked Predicate Selection

- Consider each predicate \( P \) one at a time
  - Include inferred predicates (e.g., \( \leq, \neq, \geq \))
- How likely is failure when \( P \) is true?
  - (technically, when \( P \) is observed to be true)
- Multiple bugs yield multiple bad predicates

Some Definitions

\[
F(P) = \# \text{failing runs with } |P| > 0
\]
\[
S(P) = \# \text{successful runs with } |P| > 0
\]
\[
Bad(P) = \frac{F(P)}{S(P) + F(P)}
\]

Are We Done? Not Exactly!

```java
if (f == NULL) {
    Bad(f == NULL) = 1.0;
    x = 0;
    *f;
}
```

- Predicate \( (x == 0) \) is innocent bystander
  - Program is already doomed

Multi-Valued Logic

- Identify unlucky sites on the doomed path
  
  \[
  Context(P) = \frac{F(P \lor \neg P)}{S(P \lor \neg P) + F(P \lor \neg P)}
  \]

- Captures risk of failure from reaching site, regardless of predicate truth/falsehood

Getting to the Heart of the Matter

- Look for increase in failure odds
  
  \[
  Increase(P) = Bad(P) - Context(P)
  \]

- Correspondence to likelihood ratio testing
Filtering & Ranking Algorithm

1. Discard predicates having $\text{Increase}(p) < 0$
   - E.g. dead, invariant, bystander predicates
   - Exact value is sensitive to small $R(p)$
   - Use lower bound of 95% confidence interval

2. Sort remaining predicates by $\text{Increase}(p)$
   - Again, actually use 95% lower bound
   - Likely causes with determinacy metrics

---

Case Study: Moss

- Reintroduce nine historic Moss bugs
  - High- and low-level errors
  - Includes wrong-output bugs
- Instrument with everything we’ve got
  - Branches, returns, scalar pairs, the works
- 32,000 randomized runs at 1/100 sampling

---

Effectiveness of Filtering

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Total</th>
<th>Retained</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>branches</td>
<td>4170</td>
<td>18</td>
<td>0.4%</td>
</tr>
<tr>
<td>returns</td>
<td>2964</td>
<td>11</td>
<td>0.4%</td>
</tr>
<tr>
<td>scalar-pairs</td>
<td>195,864</td>
<td>2682</td>
<td>1.4%</td>
</tr>
</tbody>
</table>

---

Sample Report

---

Effectiveness of Ranking

- Five bugs: captured by branches, returns
  - Short lists, easy to scan
- Two bugs: captured by scalar pairs results
  - Much redundancy
- Two bugs: never cause a failure
  - No failure, no problem
- One surprise bug, revealed by returns!

---

Back to the Big Picture
**Wider Deployment in Progress**

- Downloadable Open Source applications
  - 1.8 million lines of source code across six applications
  - Evolution, Gvim, The GDWM, Gnus, Neatlib, Rhythmbox
  - Shared libraries, plug-ins, multithreading, ...
  - Recently solved our first few bug kills

- Discussion with commercial testing groups
  - Spontaneous contacts, enthusiastic responses

**Current and Future Work**

- Non-uniform sampling
- Improved statistical models
  - Redundancy as clustering problem
- Use of program structure in analysis
- Many opportunities for collaboration
  - Statistics, machine learning
  - Security, databases, visualization, ...

**Alternate Applications for Reality**

- Interactive hypothesis testing
- Classical feedback-directed optimizations
- Code, feature, component coverage
- Identify features used jointly, exclusively
  - Related to mutual information
- Algorithm tuning and specialization
  - E.g. how big should this hash table be?

**Principal Contributions**

- Fair but fast random sampling
  - Feedback from end user community
- Reality-directed bug isolation
  - Explicit specification not required
  - Can find both high- and low-level errors
  - Tolerant of mislabeling, other noise
- Statistical lens for understanding software
  - Which known techniques are valid here?
  - What new techniques are needed?

**Join the Cause!**

The Cooperative Bug Isolation Project
http://www.cs.berkeley.edu/~liblit/sampler/