Statistical Debugging

Mayur Naik
Motivation

• Bugs will escape in-house testing and analysis tools
  – Dynamic analysis (i.e. testing) is unsound
  – Static analysis is incomplete
  – Limited resources (time, money, people)

• Software ships with unknown (and even known) bugs
An Idea: Statistical Debugging

• Monitor Deployed Code
  – **Online**: Collect information from user runs
  – **Offline**: Analyze information to find bugs

• Effectively a “black box” for software
Benefits of Statistical Debugging

Actual runs are a vast resource!

• Crowdsourcing-based Testing
  – Number of real runs >> number of testing runs

• Reality-directed Debugging
  – Real-world runs are the ones that matter most
Two Key Questions

• How do we get the data?

• What do we do with it?
1. **Complex systems**
   - Millions of lines of code
   - Mix of controlled and uncontrolled code

2. **Remote monitoring constraints**
   - Limited disk space, network bandwidth, power, etc.

3. **Incomplete information**
   - Limit performance overhead
   - Privacy and security
The Approach

• Guess behaviors that are “potentially interesting”
  – Compile-time instrument of program

• Collect sparse, fair subset of these behaviors
  – Generic sampling framework
  – Feedback profile + outcome label (success vs. failure) for each run

• Analyze behavioral changes in successful vs. failing runs to find bugs
  – Statistical debugging
Overall Architecture

- Program Source
- Sampler
- Compiler
- Deployed Application
- Top bugs with likely causes
- Statistical Debugging
- Profile + 😊 / 😡
A Model of Program Behaviors

• Assume any interesting behavior is expressible as a predicate $P$ on a program state at a particular program point.

Observation of program behavior = observing $P$

• Instrument the program to observe each predicate

• Which predicates should we observe?
Branches Are Interesting

```c
++branch_17[p!=0];
if (p) ...
else ...
```

Track predicates:

<table>
<thead>
<tr>
<th>p == 0</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>p != 0</td>
<td>0</td>
</tr>
</tbody>
</table>
Return Values Are Interesting

```
n = fopen(...);
++call_41[(n==0)+(n>=0)];
```

<table>
<thead>
<tr>
<th>Track predicates</th>
<th>call_41:</th>
</tr>
</thead>
<tbody>
<tr>
<td>n &lt; 0</td>
<td>23</td>
</tr>
<tr>
<td>n &gt; 0</td>
<td>0</td>
</tr>
<tr>
<td>n == 0</td>
<td>90</td>
</tr>
</tbody>
</table>
What Other Behaviors Are Interesting?

- Depends on the problem you wish to solve!

- Examples:
  - Number of times each loop runs
  - Scalar relationships between variables (e.g. $i < j$, $i > 42$)
  - Pointer relationships (e.g. $p == q$, $p != \text{null}$)
QUIZ: Identify the Predicates

List all predicates tracked for this program, assuming only branches are potentially interesting:

```c
void main() {
    int z;
    for (int i = 0; i < 3; i++) {
        char c = getc();
        if (c == 'a')
            z = 0;
        else
            z = 1;
        assert(z == 1);
    }
}
```
QUIZ: Identify the Predicates

List all predicates tracked for this program, assuming only branches are potentially interesting:

- \( c == 'a' \)
- \( c != 'a' \)
- \( i < 3 \)
- \( i >= 3 \)

```c
void main() {
    int z;
    for (int i = 0; i < 3; i++) {
        char c = getc();
        if (c == 'a')
            z = 0;
        else
            z = 1;
        assert(z == 1);
    }
}
```
Summarization and Reporting

**branch_17:**

- \( p = 0 \):
  - 63
- \( p \neq 0 \):
  - 0

**call_41:**

- \( n < 0 \):
  - 23
- \( n > 0 \):
  - 0
- \( n = 0 \):
  - 90

\( p = 0 \):
- 63
- 0

\( p \neq 0 \):
- 0

\( n < 0 \):
- 23

\( n > 0 \):
- 0

\( n = 0 \):
- 90
Summarization and Reporting

• Feedback report per run is:
  – Vector of predicate states:
    – $*, 0, 1*$
  – Success/failure outcome label

• No time dimension, for good or ill

<table>
<thead>
<tr>
<th>$p == 0$</th>
<th>$p != 0$</th>
<th>$n &lt; 0$</th>
<th>$n &gt; 0$</th>
<th>$n == 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>90</td>
</tr>
</tbody>
</table>

outcome: $S/F$
QUIZ: Abstracting Predicate Counts

- Never observed
- 0 False at least once, never true
- 1 True at least once, never false
- * Both true and false at least once

<table>
<thead>
<tr>
<th>Condition</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p == 0$</td>
<td>63</td>
</tr>
<tr>
<td>$p != 0$</td>
<td>0</td>
</tr>
<tr>
<td>$n &lt; 0$</td>
<td>23</td>
</tr>
<tr>
<td>$n &gt; 0$</td>
<td>0</td>
</tr>
<tr>
<td>$n == 0$</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total</th>
<th>1</th>
</tr>
</thead>
</table>
QUIZ: Abstracting Predicate Counts

- Never observed
- 0 False at least once, never true
- 1 True at least once, never false
- * Both true and false at least once

<table>
<thead>
<tr>
<th>Condition</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p == 0</code></td>
<td>63</td>
</tr>
<tr>
<td><code>p != 0</code></td>
<td>0</td>
</tr>
<tr>
<td><code>n &lt; 0</code></td>
<td>23</td>
</tr>
<tr>
<td><code>n &gt; 0</code></td>
<td>0</td>
</tr>
<tr>
<td><code>n == 0</code></td>
<td>90</td>
</tr>
</tbody>
</table>

- 1 Never observed
- 0 False at least once, never true
- * Both true and false at least once
# QUIZ: Populate the Predicates

Populate the predicate vectors and outcome labels for the two runs:

<table>
<thead>
<tr>
<th>Outcome label (S/F)</th>
<th>“bba”</th>
<th>“bbb”</th>
</tr>
</thead>
<tbody>
<tr>
<td>c == ‘a’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c != ‘a’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i &lt; 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i &gt;= 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
void main() {
    int z;
    for (int i = 0; i < 3; i++) {
        char c = getc();
        if (c == ‘a’)
            z = 0;
        else
            z = 1;
    }
    assert(z == 1);
}
```
void main() {
  int z;
  for (int i = 0; i < 3; i++) {
    char c = getc();
    if (c == 'a')
      z = 0;
    else
      z = 1;
    assert(z == 1);
  }
}

Populate the predicate vectors and outcome labels for the two runs:

<table>
<thead>
<tr>
<th></th>
<th>“bba”</th>
<th>“bbb”</th>
</tr>
</thead>
<tbody>
<tr>
<td>c == ‘a’</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>c != ‘a’</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>i &lt; 3</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>i &gt;= 3</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>Outcome label</td>
<td>F</td>
<td>S</td>
</tr>
</tbody>
</table>

void main() {
  int z;
  for (int i = 0; i < 3; i++) {
    char c = getc();
    if (c == 'a')
      z = 0;
    else
      z = 1;
    assert(z == 1);
  }
}
The Need for Sampling

• Tracking all predicates is expensive

• Decide to examine or ignore each instrumented site:
  – Randomly
  – Independently
  – Dynamically

• Why?
  – Fairness
  – We need an accurate picture of rare events
A Naive Sampling Approach

- Toss a coin at each instrumentation site  **Too slow!**

```c
++count_42[p != NULL];
p = p->next;

++count_43[i < max];
total += sizes[i];
```

```c
if (rand(100) == 0)
    ++count_42[p != NULL];
p = p->next;

if (rand(100) == 0)
    ++count_43[i < max];
total += sizes[i];
```
Some Other Problematic Approaches

• **Sample every** $k^{\text{th}}$ **site**
  – Violates independence
  – Might miss predicates “out of phase”

• **Periodic hardware timer or interrupt**
  – Might miss rare events
  – Not portable
Amortized Coin Tossing

- Observation: Samples are rare (e.g. 1/100)
- Idea: Amortize sampling cost by predicting time until next sample
- Implement as countdown values selected from geometric distribution
- Models how many tails (0) before next head (1) for biased coin toss
- Example with sampling rate 1/5:

  0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, ...

Next sample after: 5 3 4 sites
An Efficient Approach

```c
if (rand(100) == 0)
    ++count_42[p != NULL];
p = p->next;
if (rand(100) == 0)
    ++count_43[i < max];
total += sizes[i];
if (countdown >= 2) {
    countdown -= 2;
p = p->next;
total += sizes[i];
}
else {
    if (countdown-- == 0) {
        ++count_42[p != NULL];
countdown = next();
    }
    p = p->next;
    if (countdown-- == 0) {
        ++count_43[i < max];
countdown = next();
    }
total += sizes[i];
}
```
Feedback Reports with Sampling

- Feedback report per run is:
  - Vector of sampled predicate states (−, 0, 1, *)
  - Success/failure outcome label

- Certain of what we did observe
  - But may miss some events

- Given enough runs, samples ≈ reality
  - Common events seen most often
  - Rare events seen at proportionate rate
QUIZ: Uncertainty Due to Sampling

Check all possible states that a predicate $P$ might take due to sampling. The first column shows the actual state of $P$ (without sampling).

<table>
<thead>
<tr>
<th></th>
<th>–</th>
<th>0</th>
<th>1</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>–</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QUIZ: Uncertainty Due to Sampling

Check all possible states that a predicate $P$ might take due to sampling. The first column shows the actual state of $P$ (without sampling).

<table>
<thead>
<tr>
<th></th>
<th>–</th>
<th>0</th>
<th>1</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
</tbody>
</table>
Overall Architecture Revisited

- Program Source
- Sampler
- Compiler
- Deployed Application
- Statistical Debugging
- Profile

Top bugs with likely causes

😊 / 😡
Finding Causes of Bugs

• We gather information about many predicates
  – ≈ 300K for bc (“bench calculator” program on Unix)

• Most of these are not predictive of anything

• How do we find the few useful predicates?
Finding Causes of Bugs

How likely is failure when predicate $P$ is observed to be true?

$F(P) = \# \text{ failing runs where } P \text{ is observed to be true}$

$S(P) = \# \text{ successful runs where } P \text{ is observed to be true}$

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

Example: $F(P)=20, S(P)=30 \Rightarrow \text{Failure}(P) = \frac{20}{50} = 0.4$
Tracking Failure Is Not Enough

if (f == NULL) {
    x = foo();
    *f;
}

int foo() {
    return 0;
}

Failure(f == NULL) = 1.0
Failure(x == 0) = 1.0

Predicate x == 0 is an innocent bystander

• Program is already doomed
Tracking Context

How likely is failure when predicate P is observed to be true?

\[ F(P \text{ observed}) = \# \text{ failing runs observing } P \]
\[ S(P \text{ observed}) = \# \text{ successful runs observing } P \]

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

Example: \( F(P \text{ observed}) = 40 \), \( S(P \text{ observed}) = 80 \) \( \Rightarrow \)
\[ \text{Context}(P) = \frac{40}{120} = 0.333... \]
A Useful Measure: Increase()

Does $P$ being true increase chance of failure over the background rate?

$$\text{Increase}(P) = \text{Failure}(P) - \text{Context}(P)$$

- A form of likelihood ratio testing

- $\text{Increase}(P) \approx 1 \Rightarrow$ High correlation with failing runs

- $\text{Increase}(P) \approx -1 \Rightarrow$ High correlation with successful runs
Increase() Works

```c
if (f == NULL) {
    x = foo();
    *f;
}

int foo() {
    return 0;
}
```

1 failing run:  f == NULL
2 successful runs:  f != NULL

Failure(f == NULL) = 1.00
Context(f == NULL) = 0.33
Increase(f == NULL) = 0.67

Failure(x == 0) = 1.00
Context(x == 0) = 1.00
Increase(x == 0) = 0.00
### QUIZ: Computing Increase()

<table>
<thead>
<tr>
<th></th>
<th>“bba”</th>
<th>“bbb”</th>
<th>Failure</th>
<th>Context</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c == 'a'</code></td>
<td>*</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>c != 'a'</code></td>
<td>*</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td><code>i &lt; 3</code></td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>i &gt;= 3</code></td>
<td>0</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Outcome label</td>
<td>F</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(S/F)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### QUIZ: Computing Increase()

<table>
<thead>
<tr>
<th>c</th>
<th>“bba”</th>
<th>“bbb”</th>
<th>Failure</th>
<th>Context</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>c == ‘a’</td>
<td>*</td>
<td>0</td>
<td>1.0</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>c != ‘a’</td>
<td>*</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>i &lt; 3</td>
<td>1</td>
<td>*</td>
<td>0.5</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>i &gt;= 3</td>
<td>0</td>
<td>*</td>
<td>0.0</td>
<td>0.5</td>
<td>-0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outcome label</th>
<th>(S/F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>S</td>
</tr>
</tbody>
</table>
# Isolating the Bug

<table>
<thead>
<tr>
<th>Condition</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c == 'a'$</td>
<td>0.5</td>
</tr>
<tr>
<td>$c != 'a'$</td>
<td>0.0</td>
</tr>
<tr>
<td>$i &lt; 3$</td>
<td>0.0</td>
</tr>
<tr>
<td>$i &gt;= 3$</td>
<td>-0.5</td>
</tr>
</tbody>
</table>

```c
void main() {
    int z;
    for (int i = 0; i < 3; i++)
    {
        char c = getc();
        if (c == 'a')
            z = 0;
        else
            z = 1;
        assert(z == 1);
    }
}
```
A First Algorithm

1. Discard predicates having $\text{Increase}(P) \leq 0$
   - e.g. bystander predicates, predicates correlated with success
   - Exact value is sensitive to few observations
   - Use lower bound of 95% confidence interval

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c == 'a'</code></td>
<td>0.5</td>
</tr>
<tr>
<td><code>c != 'a'</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>i &lt; 3</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>i &gt;= 3</code></td>
<td>-0.5</td>
</tr>
</tbody>
</table>

```c
void main() {
    int z;
    for (int i = 0; i < 3; i++)
    {
        char c = getc();
        if (c == 'a')
            z = 0;
        else
            z = 1;
        assert(z == 1);
    }
}
```
Isolating a Single Bug in bc

```c
void more_arrays()
{
    ...
    /* Copy the old arrays. */
    for (indx = 1; indx < old_count; indx++)
        arrays[indx] = old_ary[indx];
    /* Initialize the new elements. */
    for (; indx < v_count; indx++)
        arrays[indx] = NULL;
    ...
}
```

#1: indx > scale
#2: indx > use_math
#3: indx > opterr
#4: indx > next_func
#5: indx > i_base
It Works!

• At least for programs with a single bug

• Real programs typically have multiple, unknown bugs

• Redundancy in the predicate list is a major problem
Using the Information

- Multiple useful metrics: \textit{Increase}(P), \textit{Failure}(P), \textit{F}(P), \textit{S}(P)
- Organize all metrics in compact visual (bug thermometer)
Multiple Bugs: The Goal

Find the best predictor for each bug, without prior knowledge of the number of bugs, sorted by the importance of the bugs.
Multiple Bugs: Some Issues

- A bug may have many redundant predictors
  - Only need one
  - But would like to know correlated predictors

- Bugs occur on vastly different scales
  - Predictors for common bugs may dominate, hiding predictors of less common problems
An Idea

- Simulate the way humans fix bugs
- Find the first (most important) bug
- Fix it, and repeat
Revised Algorithm

Repeat

Step 1: Compute \textit{Increase(\)}, \textit{F(\)}, etc. for all predicates
Step 2: Rank the predicates
Step 3: Add the top-ranked predicate \( P \) to the result list
Step 4: Remove \( P \) and discard all runs where \( P \) is true
  - Simulates fixing the bug corresponding to \( P \)
  - Discard reduces rank of correlated predicates

Until no runs are left
### Ranking by Increase(P)

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.065</td>
<td>0.935 ± 0.019</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>0.065</td>
<td>0.935 ± 0.020</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.071</td>
<td>0.929 ± 0.020</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>0.073</td>
<td>0.927 ± 0.020</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.071</td>
<td>0.929 ± 0.028</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>0.075</td>
<td>0.925 ± 0.022</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>0.076</td>
<td>0.924 ± 0.022</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>0.077</td>
<td>0.923 ± 0.023</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

**Problem:** High Increase() scores but few failing runs!

**Sub-bug predictors:** covering special cases of more general bugs
Problem: Many failing runs but low Increase() scores!

**Super-bug predictors**: covering several different bugs together
A Helpful Analogy

• In the language of information retrieval
  – **Precision** = fraction of retrieved instances that are relevant
  – **Recall** = fraction of relevant instances that are retrieved

• In our setting:
  – Retrieved instances ~ predicates reported as bug predictors
  – Relevant instances ~ predicates that are actual bug predictors

• Trivial to achieve only high precision or only high recall
• Need both high precision and high recall
Combining Precision and Recall

- \textbf{Increase}(P) has high precision, low recall
- \textbf{F}(P) has a high recall, low precision
- Standard solution: take the \textit{harmonic mean} of both
  \[
  \frac{2}{\frac{1}{\text{Increase}(P)} + \frac{1}{\text{F}(P)}}
  \]
- Rewards high scores in both dimensions
## Sorting by the Harmonic Mean

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1585</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1584</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1580</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1577</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1576</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1573</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.883 ± 0.012</td>
<td>1</td>
<td>774</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.883 ± 0.012</td>
<td>1</td>
<td>776</td>
</tr>
</tbody>
</table>

It works!
What Have We Learned?

- Monitoring deployed code to find bugs
- Observing predicates as model of program behavior
- Sampling instrumentation framework
- Metrics to rank predicates by importance: $\text{Failure}(P), \text{Context}(P), \text{Increase}(P), \ldots$
- Statistical debugging algorithm to isolate bugs
Key Takeaways (1 of 2)

• A lot can be learned from actual executions
  – Users are executing them anyway
  – We should capture some of that information
Key Takeaways (2 of 2)

• Crash reporting is a step in the right direction
  – But stack is useful for only about 50% of bugs
  – Doesn’t characterize successful runs
• But this is changing ...