Statistical Debugging

Introduction

1. Statistical debugging harnesses the power of the userbase to catch bugs that slipped through the in-house testing process
   - Collect data from user runs and transmits data back
   - Developers can analyze collected data
     - Learn what causes crashes and where to focus efforts

Statistical Debugging Motivation

1. Bugs will escape in-house testing and analysis tools
   - Dynamic analysis (testing) is unsound
   - Static analysis is incomplete
   - Limited resources (time, money, people)
2. Software ships with unknown (and even known) bugs

An Idea: Statistical Debugging

1. Software asking for permission to send usage statistics and crash reports to the developers of the software is for statistical debugging purposes
   - Essential element of statistical debugging
2. Monitor deployed code in two stages
   - Online: Collect information from user runs
   - Offline: Analyze information to find bugs
3. Effectively a "black box" for software

Benefits of Statistical Debugging

1. Actual runs are a vast resource!
   - Crowdsource-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

1. How do we get the data?
2. What do we do with it?

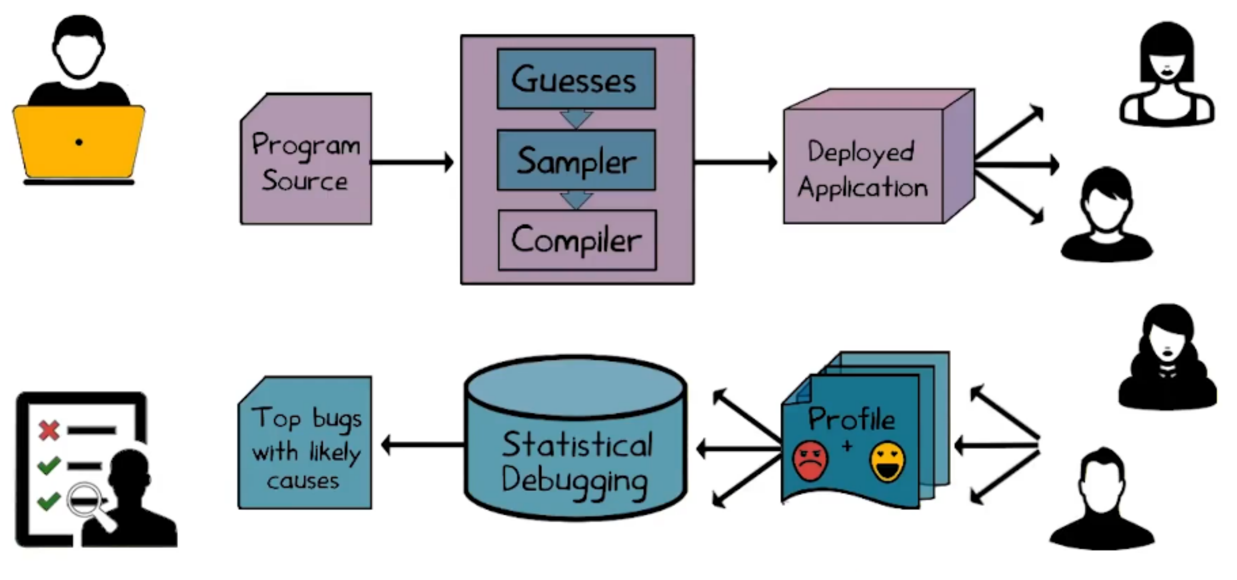
Practical Challenges

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
     - Could be compromised if we transmit sensitive data

The Approach

1. Guess behaviors that are "potentially interesting"
    - Interesting if it points us to a bug
    - Don't know, must guess
    - Compile-time instrumentation of program
2. Collect sparse, fair subset of these behaviors
    - Generic sampling framework
    - Feedback profile + outcome label (success vs. failure) for each run
3. Analyze behavioral changes in successful vs. failing runs to find bugs
    - Statistical debugging

Overall Architecture



Overall Architecture

Model Behavior

1. Assume any interesting behavior is expressible as a predicate P on a program state at a particular program point
    - Observation of behavior = observing P
2. Instrument the program to observe each predicate
3. Which predicates should we observe?

Branches Are Interesting

1. Branches are interesting because we can see if particular branches have a higher probability of being associated with failing code
    - Add an array before a branch and count which branch is taken each time
        - Array has two elements; true or false
        - ++branch_17[p!=0]
    - Add for if, while, switch, etc.

Return Values Are Interesting

1. Keep track of integer return values' relationship to 0
    - Such return values often convey information about the success or failure of an operation that the call function performed
        - 0 = success
        - Nonzero = failure

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

What Other Behaviors Are Interesting?

1. Depends on the problem you wish to solve
    - Number of times each loop runs
        - Debug performance issues
    - Scalar relationships between variables
        - Array index out of bounds errors
    - Pointer relationships
        - p == q, p != null

Identify the Predicates

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

```
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);
    }
}
```

2. Predicates:
    - c == 'a'
    - c != 'a'
    - i < 3
    - i >= 3

Summarization and Reporting 1

1. Report back data from branch_17 and call_41
    - p == 0: 63
    - p != 0: 0
    - n < 0: 23
    - n > 0: 0
    - n == 0: 90

Summarization and Reporting 2

1. Feedback report per run is:
    - Vector of predicate states: -, 0, 1, *

  - $*$ means predicate was never observed
  - 0 means the predicate was observed at least once to be false and never true
  - 1 means the predicate was observed at least once to be true and never false
  - $*$ means the predicate was observed to be true at least once and false at least once
- Success/failure outcome label
- No time dimension, for good or ill
  - Reduces complexity but less debugging information

Abstracting Predicate Counts

1. Apply the abstraction:
   - p == 0: 1
   - p != 0: 0
   - n < 0: *
   - n > 0: 0
   - n == 0: *

Populate the Predicates

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

```
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);
    }
}
```

	"bba"	"bbb"
c == 'a'	*	0
c != 'a'	*	1
i < 3	1	*
i >= 3	0	*
Outcome Label (S/F)	F	S

The Need for Sampling

1. Tracking all predicates is expensive
2. Decide to examine or ignore each instrumented stie:
   - Randomly
   - Independently
   - Dynamically
3. Why?
   - Fairness
   - We need an accurate picture of rare events

A Naive Sampling Approach

1. Toss a coin at each instrumentation site

- Maybe 1 in 100 trials
- This is too slow
  - Add a random number generation and conditional check at every branch
  - It'd be faster to track all predicates

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

Some Other Problematic Approaches

1. Sample every kth site
   - Violates independence
   - Might miss predicates "out of phase"
2. Periodic hardware timer or interrupt
   - Might miss rare events
   - Not portable across hardware

Amortized Coin Tossing

1. Observation: Samples are rare (e.g., 1/100)
2. 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
3. Example with sampling rate 1/5:
   - 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, . . .
   - Next sample after: 5, 3, 4 sites

An Efficient Approach

```
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];
}
```

1. Can increase the size of the code by twofold

Feedback Reports with Sampling

1. Feedback report per run is:
   - Vector of sampled predicate states (-, 0, 1, *)
   - Success/failure outcome label
2. Certain of what we did observe
   - But may miss some events

3. Given enough runs, samples approximate reality
    - Common events seen most often
    - Rare events seen at a proportionate rate

Uncertainty Due to Sampling

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

P	-	0	1	*
-	X			
0	X	X		
1	X		X	
	X	X	X	X

Overall Architecture Revisited

1. Observational data are collected in a feedback report and transmitted back
    - What do we do with this data?
    - Introduce statistical techniques to determine most significant bugs

Finding Causes of Bugs 1

1. We gather information about many predicates
    - 300K for bc ("bench calculator" program on Unix)
2. Most of these are not predictive of anything
3. How do we find the few useful predicates?

Finding Causes of Bugs 2

1. How likely is failure when predicate P is observed to be true?
    - F(P) = # failing runs where P is observed to be true
    - S(P) = # successful runs where P is observed to be true
    - Failure(P) = F(P) / (F(P) + S(P))
2. Tracking failure is not enough
    - Failure(f == NULL) = 1.0
    - Failure(x == 0) = 1.0
    - Predicate x == 0 is an innocent bystander
        - Program is already doomed
        - Need a different statistic that considers the context in which failures occur

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

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

Tracking Context

1. What is the background chance of failure, regardless of P's value?
    - F(P observed) = # failing runs observing P

- S(P observed) = # successful runs observing P
- Context(P) = F(P observed) / (F(P observed) + S(P observed))

A Useful Measure: Increase()

1. Does P being true increase chance of failure over the background rate?
   - Increase(P) = Failure(P) - Context(P)
2. A form of likelihood ratio testing
   - Increase(P) near 1 => High correlation with failing runs
   - Increase(P) near -1 => High correlation with successful runs

Increase() Works

1. If we have 1 failing run and 2 successful runs, compute Failure, Context, and Increase for f == NULL
   - Failure(f == NULL) = 1.00
   - Context(f == NULL) = 0.33
   - Increase(f == NULL) = 0.67
2. If we have 1 failing run and 2 successful runs, compute Failure, Context, and Increase for x == 0
   - Failure(f == NULL) = 1.00
   - Context(f == NULL) = 1.00
   - Increase(f == NULL) = 0.00

Computing Increase()

	"bba"	"bbb"	Failure	Context	Increase
c == 'a'	*	0	1.0	0.5	0.5
c != 'a'	*	1	0.5	0.5	0.0
i < 3	1	*	0.5	0.5	0.0
i >= 3	0	*	0.0	0.5	-0.5
Outcome Label (S/F)	F	S			

Isolating the Bug 1

1. Increase metric tends to localize bugs at the point where the condition that causes the bug first becomes true rather than the crash point
   - Desirable feature of the metric

A First Algorithm

1. Discard predicates having Increase(P) <= 0
   - e.g., bystander predicates, predicates correlated with success
   - Exact value is sensitive to few observations
     - Compute a confidence interval for Increase
   - Use lower bound of 95% confidence interval
     - Discard high increase scores but low confidence due to few observations
2. Sort remaining predicates by Increase(P)
   - Again, use 95% lower bound
   - Likely causes with determinacy metrics

Isolating the Bug 2

1. In step 1 of the algorithm, we discard the predicates: c != 'a', i < 3, and i >= 3 due to their increase scores

2. In step 2, the algorithm outputs the predicate c == 'a'

Isolating a Single Bug in bc

1. The following predicates are true for the below program
    - indx > scale
    - indx > use_math
    - indx > opterr
    - indx > next_func
    - indx > i_base
    - This indicates that the program fails when indx grows large and likely overruns the bounds of arrays
        - Writes nulls to unintended memory locations

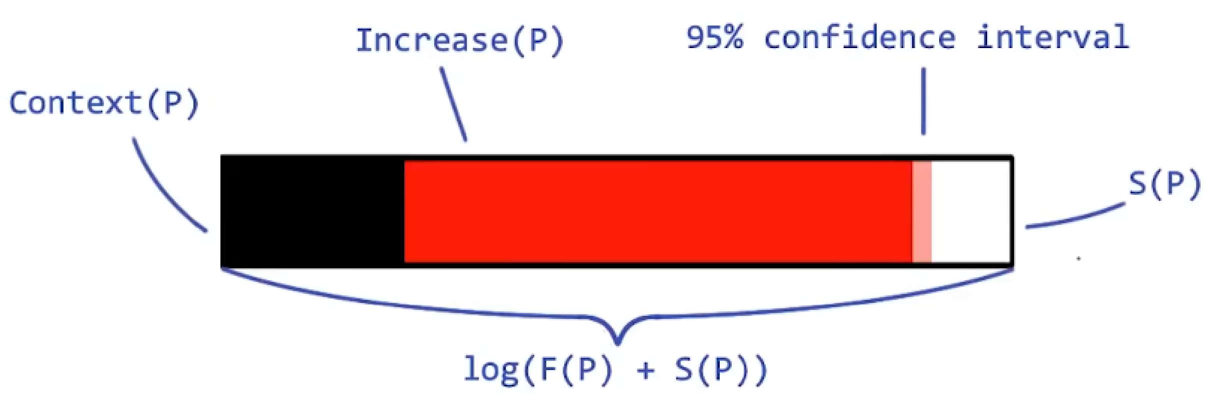
```
void more_arrays()
{
    /* Copy the old arrays. */
    for (indx = 1; indx < old_count; indx++)
        arrays[indx] = old_ary[indx];
    /* Initialize the new elements. */
    for (; indx < v_count; indx++)
        arrays[indx] = NULL;
}
```

It Works

1. Works at least for programs with a single bug
2. Real programs typically have multiple, unknown bugs
3. Redundancy in the predicate list is a major problem

Using the Information

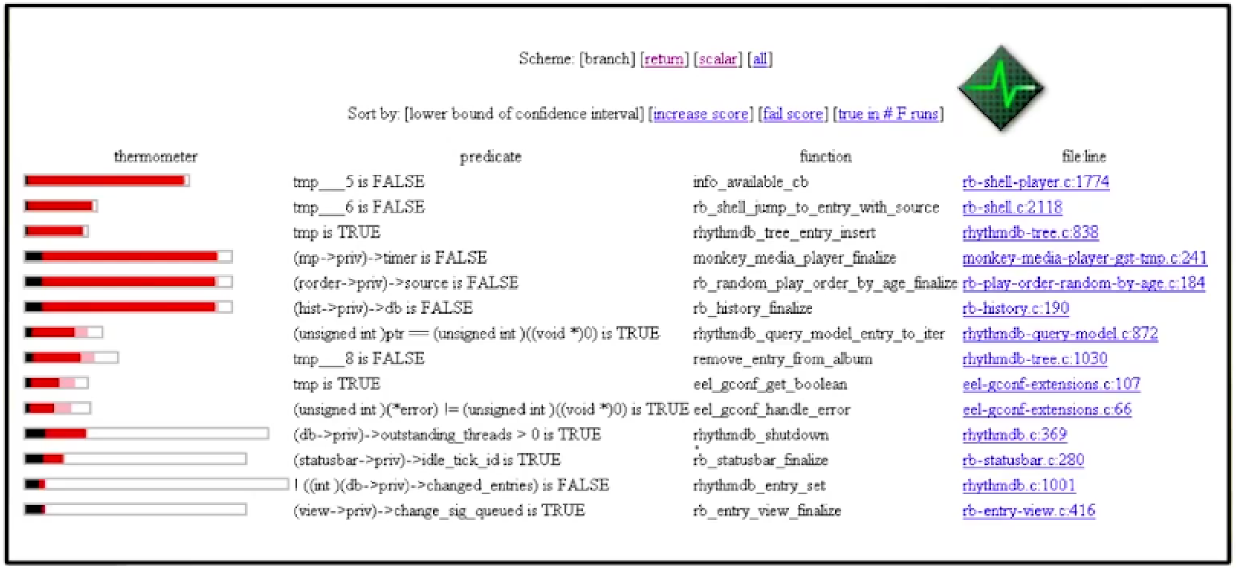
1. Multiple useful metrics: Increase(P), Failure(P), F(P), S(P)
2. Organize all metrics in compact visual (bug thermometer)



Thermometer

Sample Report

1. More red in the thermometer indicates that predicate is predictive of program failure
    - Shorter bar indicates that it wasn't observed in as many runs of the program


Scheme: [branch] [return] [scalar] [all]

Sort by: [lower bound of confidence interval] [increase score] [fail score] [true in # F runs]

thermometer	predicate	function	file:line
	tmp___5 is FALSE	info_available_cb	rb-shell-player.c:1774
	tmp___6 is FALSE	rb_shell_jump_to_entry_with_source	rb-shell.c:2118
	tmp is TRUE	rhythmdb_tree_entry_insert	rhythmdb-tree.c:838
	(mp->priv)->timer is FALSE	monkey_media_player_finalize	monkey-media-player-gst-tmp.c:241
	(rorder->priv)->source is FALSE	rb_random_play_order_by_age_finalize	rb-play-order-random-by-age.c:184
	(hist->priv)->db is FALSE	rb_history_finalize	rb-history.c:190
	(unsigned int)ptr == (unsigned int)((void *)0) is TRUE	rhythmdb_query_model_entry_to_iter	rhythmdb-query-model.c:872
	tmp___8 is FALSE	remove_entry_from_album	rhythmdb-tree.c:1030
	tmp is TRUE	eel_gconf_get_boolean	eel-gconf-extensions.c:107
	(unsigned int)(*error) != (unsigned int)((void *)0) is TRUE	eel_gconf_handle_error	eel-gconf-extensions.c:66
	(db->priv)->outstanding_threads > 0 is TRUE	rhythmdb_shutdown	rhythmdb.c:369
	(statusbar->priv)->idle_tick_id is TRUE	rb_statusbar_finalize	rb-statusbar.c:280
	! ((int)(db->priv)->changed_entries) is FALSE	rhythmdb_entry_set	rhythmdb.c:1001
	(view->priv)->change_sig_queued is TRUE	rb_entry_view_finalize	rb-entry-view.c:416

Sample Report

Multiple Bugs: The Goal

1. Find the best predictor for each bug, without prior knowledge of the number of bugs, sorted by the importance of the bugs.
2. Causes new issues
   - A bug may have 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

Another Idea

1. Simulate the way humans fix bugs
2. Find the first (most important) bug
3. Fix it, and repeat

Revised Algorithm

1. Repeat until no runs are left:
   - Step 1: Compute Increase(), 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
2. How do we rank the predicates?

Ranking by Increase(P)

1. Rank predicates by increase metric
   - Problem: High Increase() scores but few failing runs!

- Sub-bug predictors: covering special cases of more general bgus

Ranking by F(P)

1. Rank predicates by number of failed runs
   - Problem: Many failing runs but low Increase() scores!
     - Also true in many successful runs
     - Super-bug predictors: covering several different bugs together

A Helpful Analogy

1. In the language of information retrieval
   - Precision: fraction of retrieved instances that are relevant
   - Recall: fraction of relevant instances that are retrieved
2. In our setting:
   - Retrieved instances ~ predicates reported as bug predictors
   - Relevant instances ~ predicates that are actual bug predictors
3. Trivial to achieve only high precision or only high recall
   - Need both high precision and high recall

Combining Precision and Recall

1. Increase(P) has high precision, low recall
2. F(P) has a high recall, low precision
3. Standard solution: take the harmonic mean of both
   - 2 / (1/Increase(P) + 1/F(P))
   - Rewards high scores in both dimensions

Sorting by the Harmonic Mean

1. It works
   - Top predicates have many failing runs and high Increase() score

Summary

1. Monitoring deployed code to find bugs
2. Observing predicates as model of program behavior
3. Sampling instrumentation framework
4. Metrics to rank predicates by importance
   - Failure(P), Context(P), Increase(P), ...
5. Statistical debugging algorithm to isolate bugs

Conclusion

1. A lot can be learned from actual executions
   - Users are executing them anyway
   - We should capture some of that information
2. Key takeaway
   - 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