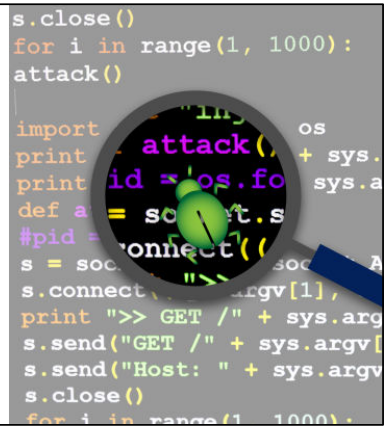
CIS 547
Software Analysis

Statistical Debugging

Mayur Naik

Penn Engineering



Despite our best efforts, even after extensive testing, we as software developers rarely put out bug-free code. In this lesson, we will look at a type of debugging technique used throughout the software industry: statistical debugging. Statistical debugging harnesses the power of the user base to catch the bugs that slip through the in-house testing process.

Based on dynamic analysis, this debugging technique collects data about the program's behavior in user runs, and transmits this data back to a centralized server, where the developers of the program can analyze the collected data to deduce what program behaviors are predictors of program crashes, and focus bug triaging and bug fixing efforts accordingly.

In this lesson, we will look at the process that enables to leverage this source of real-world testing data.

LESSON

Introduction

SEGMENT

Why Statistical Debugging?

Motivation

- Bugs will escape in-house testing and analysis tools
  - Dynamic analysis (i.e. testing) is unsound
  - Static analysis is incomplete
  - There are limited resources (time, money, people)
- Software ships with unknown (and even known) bugs

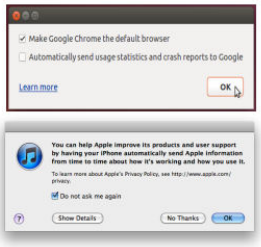
We live in an imperfect world with imperfect software. Despite extensive resources devoted to testing and debugging software, you can safely bet that bugs will escape in-house testing and quality assurance.

There are theoretical and practical reasons for this situation. Dynamic analysis (that is, testing) of computer software is unsound in that it may miss real bugs. On the other hand, static analysis of source code is incomplete in that it may report false bugs. And software development is bounded by constraints on resources: time, money, and people.

Sometimes that means we don't catch all the bugs in our software, and users end up finding bugs. And sometimes it means we have to ship software with known bugs because we just can't address them all.

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



Statistical debugging is an idea that is becoming increasingly common in practice to address this problem.

When you've installed software such as Chrome or even an operating system such as OS X or Windows, you've likely seen messages like these -- messages asking for your permission to send usage and crash reports to the company developing the software. Reports like these are an essential element of statistical debugging.

The idea is to use data from actual users' runs of deployed code. This process has two stages: one online and the other offline. In the online stage, information is collected from the user's runs: information about the machine's state during the execution of the software as well as whether the run ends in success or failure. This information is then transmitted back to the development company. In the offline stage, the company analyzes the data transmitted by many different users to find bugs in the program.

Effectively, these runs are to software what "black boxes" or flight recorders are to airplanes: they record the data prior to a crash which can be analyzed later to determine the cause of the crash.

Benefits of Statistical Debugging

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

The potential advantages of this method of "post-deployment" bug hunting are enticing.

Actual runs by users are a vast resource for two key reasons.

First, they allow to effectively crowdsource testing to the user population, which not only saves the resources for testing software in-house, but also the real runs can greatly outnumber the test runs that can be performed in-house.

For reference, over 60 million licenses for Office XP were sold in its first year after release, which amounted to nearly 2 licenses per second. And the music-sharing software Kazaa was downloaded over 1.9 million times in a single week, amounting to 3 downloads per second. Imagine all of these users sending regular reports on usage statistics and crashes.

Secondly, actual runs allow debugging to be reality-directed. When allocating resources for testing and debugging before release, one is left to guess which features are most critical to test and which bugs are most critical to fix.

Post-deployment statistics provide data on which parts of the code and which bugs are most important to the users: the needs of the userbase can be used to define the allocation of resources.

Two Key Questions

- How do we get the data?
- What do we do with it?

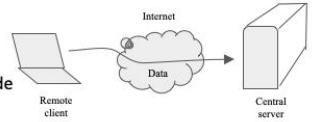
There are two key questions that must be answered to effectively implement the idea of statistical debugging:

How can we get the data from the users in an efficient way?

Once we have the data, how can we analyze it in a way that helps us with our goal of finding and debugging the bugs that most affect users?

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
   - Limits performance overhead
   - Privacy and security

Answering these two questions requires addressing considerable practical challenges. Let's look at each of these challenges one at a time.

First, the systems we are debugging are complex. They are typically millions of lines of code. They comprise many components, some of which we can monitor, but others that we cannot. Additionally, the code may be executed across several threads that interact in complex and unexpected ways. Despite these challenges, we must be able to collect data that helps us to efficiently determine which portions of the code and what elements of the program state are relevant to debugging.

Second, remote monitoring is constrained by a limited amount of disk space to store user data, for example in the case where we are monitoring an app on the users' smartphone, as well as a limit on the bandwidth of the network through which users will need to send their data, the power that is consumed while storing and sending the data, etc. There might also be a cost to users for sending us lots of data over the network. So we must be smart in deciding what data to store and transmit.

Finally, we must live with the fact that we will have incomplete information to analyze. While it is possible in principle to store and send complete information about a program run, in practice this imposes severe performance overhead that makes the program unusable. Another reason why we are limited to incomplete information is that we must safeguard users' privacy and security, which may be compromised if we transmit information over the network that includes sensitive data, such as passwords

or credit card information.

SEGMENT

The Statistical Debugging Approach

The Approach

- Guess which behaviors are "potentially interesting"
  - Compile-time instrumentation 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 versus failing runs to find bugs
  - Statistical debugging

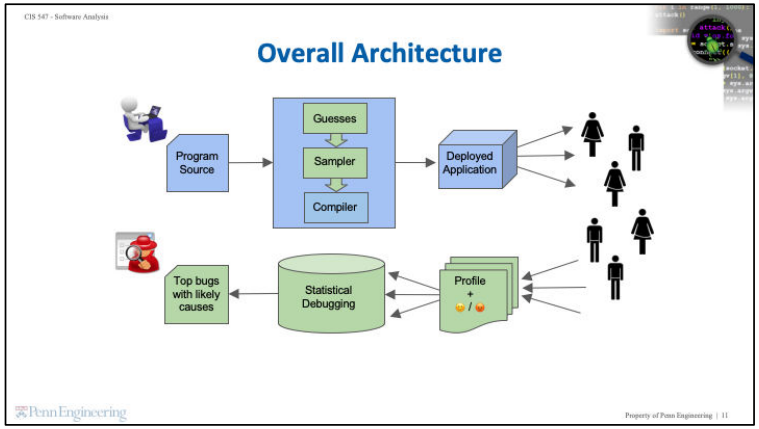
The strategy we will use to overcome these challenges consists of three steps.

In the first step, we will guess behaviors that are potentially interesting. A behavior is interesting for our purpose if it points us to a bug in the program being deployed. If we knew exactly which behaviors are interesting, there would be nothing left to do. The point is that we do not know exactly which behaviors are interesting. Therefore, we are going to start by guessing a large number of potentially interesting behaviors. We will achieve this by compile-time instrumentation of the program before deploying it. Instrumentation is the process of modifying the program's source code or binary code with probes to monitor the potentially interesting behaviors.

In the second step, we will collect a sparse and fair subset of observations of these behaviors. We will design a generic framework for sampling whether or not to observe some aspect of the program's state so that we do not overwhelm the users' devices by storing and sending large amounts of data. The user's report will then consist of a vector of these observations compressed to a very simple representation, called a feedback profile, plus a bit indicating whether the user's run ended in success or failure, called the outcome label. This will be the entirety of the data we receive in the user report.

In the third and final step, we will analyze the collected data to find differences between behaviors of successful runs versus failing runs that help localize the causes of the failing runs. This is the step that gives the approach the name "statistical

debugging".



Here is the overall architecture of the approach we just outlined.

The source code of the program, after being written by the developer, is instrumented with statements to monitor behaviors that we have guessed are potentially interesting.

For reasons we just outlined, such as performance overhead and network limits, we will not collect these behaviors in every run. To allow skipping behaviors in a fair and efficient manner, the sampler injects code that at run time will probabilistically decide which behaviors to collect.

The instrumented code is then compiled and deployed to the user base. During user runs of the program, behavior feedback profiles are created along with bits indicating the success or failure outcome of the run, and these profiles are transmitted back to the developer's database.

The developer can then perform statistical analysis on this data to determine which bugs are the most important and what their likely causes are.

Notice that, unlike the other software analysis techniques in this course, this approach does not require a formal specification in order to find bugs: users merely need to indicate whether their run succeeded or failed. Such an indication can even come automatically instead of from the user. For instance, if the run violates an assertion or suffers a segmentation fault, the run can automatically be labeled as a failing run. As

another example, if a run produces an output that the user deems incorrect, the user can simply label the run as failing.

LESSON

How to Get Data for Debugging

SEGMENT

Step 1: A Model of Program Behaviors

A Model of Program Behaviors

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

  Observation of program behavior = observing P

- Instrument the program to observe each predicate
- A Question: Which predicates should we observe?

Let's start moving into the first step of the process: deciding which program behaviors we will observe and collect from users.

Our model for an interesting behavior will be a boolean predicate that is either true or false based on the program's state at a given point in the program.

An observation of such a behavior thus consists of an observation of whether the corresponding predicate is true or false. We will instrument the program to observe each such predicate and record their truth values for reporting later. The question remains, however: which predicates should we choose to observe?

Branches Are Interesting

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

Track predicates:

branch_17:

`p == 0`	63
`p != 0`	0

The first interesting program behavior we'll keep track of is branch conditions. This will allow us to see if particular branches in the code are correlated with higher probabilities of program failure.

In particular, we will observe the truth value of the boolean expression in each branch condition in the program.

Suppose that this branch condition p appears at program point numbered 17. The instrumentation process will add a line of code that increments the count in the appropriate cell of a two-cell array called branch_17.

The 0th cell of this array will count the number of times the expression p was observed to be false (in this case 63), and the next cell will count the number of times the expression p was observed to be true (in this case 0).

We will maintain a unique such array for each branch condition in the program.

Note that branch conditions occur not only in if statements, such as in this shown example, but also in while statements, switch statements, and so on.

Return Values Are Interesting

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

call_41:

Track predicates:		
	n < 0	23
	n > 0	0
	n == 0	90

Another program behavior we'll keep track of is the return value from function calls that return an integer data type.

In particular, we will keep track of the return value's relationship to 0. This information is potentially useful because such return values often convey information about the success or failure of an operation that the called function performed. In particular, a return value of 0 may be used to indicate a successful operation, and a non-zero return value may be used to indicate a failure of the operation. Programmers often presume that the operation will succeed, and neglect to check for the case in which the operation fails. Tracking the relationship of the return value to 0 will help us to pinpoint the cause of runs that fail due to such programmer errors.

Let's look at an example. Suppose this call to the fopen function appears at program point 41. The instrumentation process will add a line of code that increments the count in the appropriate cell of a three-cell array called call_41.

The 0th cell of this array will count the number of times n was observed to be less than 0 (in this case 23), the next cell will count the number of times n was observed to be greater than 0 (in this case 0), and the last cell will count the number of times n was observed to be equal to 0 (in this case 90).

We will maintain a unique such array for each integer-returning function call in the program.

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 ! = null)

What other behaviors might we want to keep track of?

As you might have guessed, the answer to this question depends on the context of the problem you're solving.

Other examples of behaviors we might wish to keep track of are:

The number of times each loop runs. This can help us with debugging performance issues, as programs spend most of their time in loops.

Scalar relationships between variables, such as whether one variable is greater than another (this might help us detect array index out-of-bounds errors).

And pointer relationships, such as whether two pointers are equal or whether a pointer is not null.

QUIZ: Identify the Predicates

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);
  }
}
```


{QUIZ SLIDE}

Let's look at the following program. To check your understanding, please type in these boxes which predicates we will keep track of, assuming that we are presently only interested in branch conditions. (So, for example, you can ignore the predicate z == 1, since it does not indicate a branch condition.)

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
```

```
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);
  }
}
```


{SOLUTION SLIDE}

Here are the four branch condition predicates we can extract from the program.

One branch condition is at the if-statement, which gives us two predicates: c == 'a' and its negation c != 'a'.

Another branch condition that might have been less obvious is the one that occurs in the for-loop.

At the end of each iteration of the for-loop, we check whether i is less than 3 to determine whether to iterate through the loop again. So this gives us the predicate i < 3 and its negation i >= 3.

SEGMENT

Step 2: Summarization and Reporting

Summarization and Reporting

branch_17:

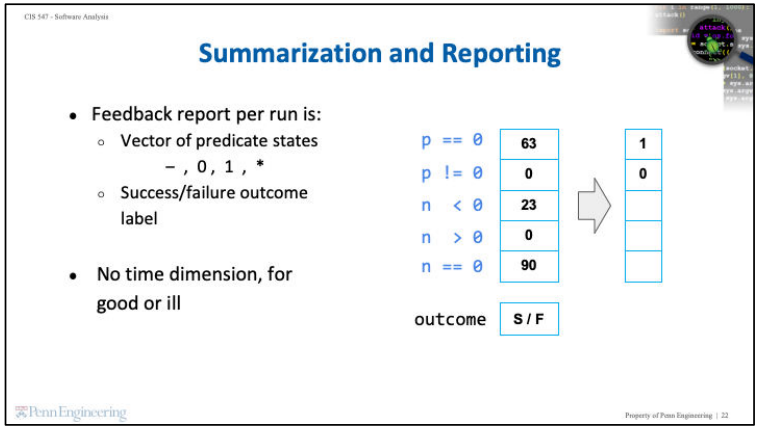
p == 0	63
p != 0	0

call_41:

n < 0	23
n > 0	0
n == 0	90

p == 0	63
p != 0	0
n < 0	23
n > 0	0
n == 0	90

When a run finishes, we combine the arrays of predicate counts that we were tracking at different instrumentation sites in the program into a single array, the feedback profile, in preparation for reporting it. Notice that this feedback profile consists of data from two instrumentation sites, branch_17 and call_41, but five predicates.



We report the feedback profile along with the outcome label indicating whether the run succeeded or failed. It is up to you how you wish to define the state of each predicate in the feedback profile. One possibility is to report the counts itself. To keep things simple, however, we will abstract these counts to one of only four states: -, 0, 1, and *:

The - state means that the predicate was never observed (neither as true nor as false) during the run.
The state 0 means that the predicate was observed at least once to be false, and never observed to be true.
The state 1 means that the predicate was observed at least once to be true, and never observed to be false.
Finally, the * state means that the predicate was observed at least once to be false and at least once to be true.

Let's apply this state abstraction to these predicate counts. Since the predicates at each site are related to each other, remember that we must look at the counts of all the predicates at each site before deciding the state of each predicate at that site.

Let's look at the first site consisting of these two predicates (gesture at p==0 and p!=0). We infer that the state of predicate p == 0 is 1 because p==0 was observed at least once to be true (in fact 63 times), and never observed to be false. On the other hand, the state of predicate p != 0 is 0 because it was observed at least once to be false (in fact 63 times), and never observed to be true.

Using the same procedure, we will fill in the states of these three predicates in a quiz shortly. (gesture at n < 0, n > 0, and n == 0).

Notice that since there are only four possible abstract states, only two bits suffice for reporting the state of each predicate in the feedback profile, compared to say 32 or 64 bits for a count.

Also, notice that we are abstracting away the time dimension in our report. Like all abstraction schemes, this has its benefits (such as reducing complexity) and drawbacks (such as losing potentially useful debugging information).

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

p == 0	63	1
p != 0	0	0
n < 0	23	
n > 0	0	
n == 0	90	

{QUIZ SLIDE}

Now go ahead and complete this table to abstract the predicate counts for n < 0, n > 0, n == 0.

You can use the box on the left if you need to remind yourself what each of the four abstract states means.

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

p == 0	63	1
p != 0	0	0
n < 0	23	*
n > 0	0	0
n == 0	90	*

{SOLUTION SLIDE}

We infer that the state of predicate n < 0 is * because it was observed at least once to be true (in fact 23 times) and at least once to be false (in fact 90 times).

The reasoning is similar for the predicate n == 0: it was observed to be true 90 times and false 23 times.

On the other hand, the state of predicate n > 0 is 0 because it was observed at least once to be false (in fact 90 + 23 times), but never observed to be true.

QUIZ: Populate the Predicates

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

	"bba"	"bbb"
c == 'a'		
c != 'a'		
i < 3		
i >= 3		
Outcome label (S/F)		

```
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 SLIDE}

Next, let's see how these predicates would be reported in a feedback profile in two different runs of this program: one in which the three characters read by getc() are "bba" and another in which the three characters read by getc() are "bbb". For each predicate, enter -, 0, 1, or * as appropriate.

Use the assertion outcome as the outcome label for the run. That is, if the assertion never fails in a run, then use the letter S as the outcome label of that run. If the assertion ever fails, then use the letter F as the outcome label of that run. Also, in this case, assume that the program terminates as soon as the assertion fails, instead of continuing to run.

QUIZ: Populate the Predicates

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

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

```
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);
  }
}
```


{SOLUTION SLIDE}

Let's start first with the "bba" run.

On the first iteration of the for-loop, we first check the predicate c == 'a', which is false. Then we increment i and check whether i < 3. That predicate is true, since i = 1.

On the second iteration, we check c == 'a', which is again false, we increment i, and then we check whether i < 3, which is again true, since i = 2.

On the third iteration, we check c == 'a', which is now true. Then z is assigned the value of 0, and we fail the assert statement, ending the program run.

Since we observed c == 'a' to be both true and false in this run, we assign these two predicates the value * (gesture to two * in bba column). Since we always observed i < 3 to be true and never false, we assign predicate i < 3 the value 1 and predicate i >= 3 the value 0 (gesture to 1 and 0 in bba column). And we enter F for the outcome label since the assertion failed in this run (gesture to F in bba column).

Now let's look at the "bbb" run.

The first two iterations of the for-loop will be the same. Predicate c == 'a' will be false and predicate i < 3 will be true for both of these iterations.

On the third iteration, we check c == ‘a’, which is false. Then we reach the end of the loop body, increment i to 3, and observe that i < 3 is now false. The loop therefore ends, and so does the program.

Since we observed c == ‘a’ to be always false and never true in this run, we assign predicate c == ‘a’ the value 0 and predicate c != ‘a’ the value 1 (gesture to 0 and 1 in bbb column). Since we observed i < 3 to be true and false during the run, we assign these two predicates the value * (gesture to two * in bbb column). And we enter S for the outcome label since the assertion did not fail in this run (gesture to S in bbb column).

SEGMENT

Step 3: Sampling Program Predicates

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

`p == 0`	1
`p != 0`	0
`n < 0`	*
`n > 0`	0
`n == 0`	*

As you can imagine, a complex piece of software could contain hundreds of thousands of predicates that could be potentially interesting for us to track. Keeping track of all of these predicates could cause significant slowdown in the user's experience, and it would also be difficult for all that data to be sent back from the user on a regular basis.

Therefore, we will only observe and collect a tiny fraction of the predicates in a given user's run. We will do this by randomly choosing whether or not to examine each instrumentation site independent of whether we've sampled the other sites. Once we decide to examine a site, we will observe the values of all predicates at that site. For instance, if we decide to sample the first site here, we will observe the values of both predicates p==0 and p!=0.

Moreover, we'll sample the sites to examine dynamically -- that is, during the run of the program -- instead of statically (that is, hardcoding which sites will be sampled).

We need to use a random, independent, and dynamic sampling procedure because it allows us to collect data on each predicate fairly across all users. This will keep our data from inaccurately representing the rarity of events, which will be important to our statistical analysis of the data later on.

A Naive Sampling Approach

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

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


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

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

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


A dynamic sampling system must define a trigger mechanism that signals when a sample is to be taken. Let's look at one candidate mechanism, which is to toss a coin at each instrumentation site.

Suppose we want to sample at a sampling rate of 1%, that is, we want to observe on average only 1 site out of every 100 sites in a run.

We might use some sort of random number generator that produces an integer between 0 and 99 at each site, and then only increment the count of the appropriate predicate at that site if the random number generator produces say 0.

Unfortunately, this approach ends up being inappropriate for our needs, since it will cause a significant loss in performance to add a random number generation and an integer comparison at each instrumentation site. In fact, it is slower than unconditionally tracking all predicates, whose overhead we set out to reduce in the first place.

Some Other Problematic Approaches

- Sample every k^{th} site
  - Violates independence
  - Might miss predicates "out of phase"
- Periodic hardware timer or interrupt
  - Might miss rare events
  - Not portable

Since tossing a coin at each instrumentation site was too slow, what are some other approaches we could try to implement a sampling procedure?

One idea might be to sample every kth site: perhaps every 100th site if we wanted a 1% sampling rate. While this eliminates the overhead required for generating a random number, it has the flaw that our observations are no longer independent.

For example, if it were the case that every other iteration of a loop had a site with a predicate indicating incorrect program behavior, and we only sampled every 100th site, then it we would never sample that predicate if the program always enters the loop after having observed an odd number of sites.

Another idea is to use a periodic hardware timer or interrupt to decide when to sample a site. While this approach may suffice when hunting for large performance bottlenecks, however, it may systematically miss rare events. Also, this approach is not portable across systems using different hardware.

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, 1, …

Next sample after: 5 3 4 sites

Tossing a coin at each instrumentation site preserved all the properties we wanted out of a sampling procedure, so let's not give up on it quite yet.

In general, our sampling rates are going to be low: perhaps only 1% of the sites reached in a run will be sampled. Since we'll have such a small number of observations, a promising strategy is to "amortize" the sampling cost by predicting the time until the next sample.

This strategy can be implemented as countdown values selected from a geometric distribution which gives a random number of "tails" (that is, 0s) with the same distribution between occurrences of "heads" (that is, 1s). This distribution of runs of tails is close enough to the distribution we'd observe if we had instead flipped a coin with a 1% chance of heads many times in a row.

This gives us a way of knowing ahead of time how many sites we will need to skip before making our next sample. For example, with a sampling rate of 1/5, three numbers drawn from the geometric distribution might be 5, 3, and 4, indicating that our first sampled site will happen after skipping 5 sites; our next sample will happen after skipping 3 more sites, and our next sample will happen after skipping 4 more.

An Efficient Approach

```
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];
}
```


We will utilize this knowledge of how many sites we'll skip before our next sample by making two copies of each loop-free section in the program we are instrumenting.

For example, if we have a section of code that we know contains just two sites of interest, then we'll surround this code by an if-else block. In one block, we'll have the bare code with sites not instrumented at all. In the other block, we'll have the same code with instrumentation sites.

If, upon reaching the if-statement, there are two or more sites remaining to be skipped before the next sample occurs, the program will run the uninstrumented code (which avoids the checks associated with performance decreases) (gesture to the checks in the then part).

On the other hand, if there are fewer than two sites to be skipped before the next sample, the instrumented code will be executed, with the countdown variable being decremented at each instrumentation site until it reaches zero. At that point, the site will be sampled, the countdown variable will be reset according to the geometric distribution, and the program will continue executing.

This gives us a performance advantage while still being able to choose sites in a random, independent, and dynamic manner. This does come at the cost of upto a two-fold increase in code size, since we need two copies of each section of code: one "fast" version without instrumentation and one "slow" version with instrumentation.

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

With sampling in place, let's see what the feedback report looks like.

As before, it consists of a vector of predicate states, each either -, 0, 1, or *, but this time each predicate's state is sampled as opposed to exact. And of course we have the success or failure outcome label of the run, as before.

While sampling introduces noise in the vector of predicate states, we can still be certain of what we did observe, although we may miss some events. We'll make this statement precise in the following quiz.

More significantly, given enough runs, the impact of noise diminishes further as samples begin to approximate reality. Common events are seen most often despite sampling, and even rare events will be seen at a 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).

P	–	0	1	*
–				
0				
1	✅		✅	
*				

{QUIZ SLIDE}

We haven't yet precisely outlined the possible states that might be recorded for a predicate P due to sampling. Let's do this in the form of a quiz.

Suppose the first column of this table shows the actual state of predicate P in a run without sampling. Check all possible states that we might record for predicate P due to sampling.

As a hint, let's determine the possible states that might be recorded when the actual state of predicate P is 1. In other words, P is a predicate that is always true in the run we are looking at.

Clearly, we might get lucky and end up observing P at least once, in which case we will find that its value is true and therefore we will record its state as 1. But we might get unlucky and never observe P, in which case we will record its state as dash.

This case suggests how noise, or uncertainty, creeps into the feedback report due to sampling. But as we mentioned earlier, we can still be certain of what we did observe. For instance, if the recorded state is 1, we can be certain that predicate P was true at least once in the run.

Finally, note that we will never record the state as 0 or star, as this would imply that predicate P was false at least once in the run without sampling, which is impossible since we assumed that the state of predicate P in the run without sampling is 1 (gesture to 1 in first column).

Go ahead and check the boxes in the remaining rows corresponding to what states we might see after sampling if the actual state of P is the one on the far left of the 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).

P	–	0	1	*
–	✅			
0	✅	✅		
1	✅		✅	
*	✅	✅	✅	✅

{SOLUTION SLIDE}

Let's review the solution.

If P's actual state is -, then we never observe P during the run. Because sampling only removes observations, after sampling we'll still have no observations of P. Therefore P's state after sampling must still be -.

If P's actual state is 0, then (like the case for the state 1) after sampling, we have two possibilities. Either we kept some observation of P in which P was observed to be false, or we lost all observations of P. In the former case, P's state will be recorded as 0. In the latter case, P's state will be recorded as -.

If P's actual state is *, then there is at least one observation of P where P is true and at least one observation where it is false. After sampling, there are four possibilities.

We might keep a true observation and a false observation (in which case the sampled state would still be *).
We might keep a true observation and lose all false observations (in which case the sampled state would be 1).
We might keep a false observation and lose all true observations (in which case the sampled state would be 0).
And we might lose all observations (in which case the sampled state would be -).

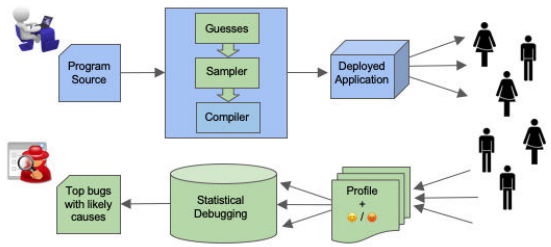
LESSON

What to Do with Data to Debug

SEGMENT

Step 4: Finding Causes of Bugs

Overall Architecture Revisited



Let's look again at our roadmap for this debugging process. So far, we have focused on the first half of the process, how to get data potentially useful for debugging from our users. We instrument our program's code to observe predicates, and we intelligently sample observations of these predicates in order to keep performance overhead minimal for users. The observational data plus a success/failure outcome label are then collected in a feedback report and transmitted back to us.

Now let's look at the second half of the process: what we need to do with this collected data to effectively debug our software. Here we will introduce the statistical techniques to isolate the most important bugs (in terms of numbers of affected users) and the likely causes of these bugs.

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?

We've started out by gathering information on a large number of predicates, as we do not know ahead of time which predicates will be predictive of bugs. For instance, by instrumenting branches, function calls, and scalar variable relationships, we obtain nearly 300,000 predicates for bc, the bench calculator program on UNIX which comprises roughly 300 thousand lines of code.

Clearly, most of these predicates are not predictive of anything, since the program being deployed is expected to be mostly correct. It likely has far fewer number of bugs than the number of predicates, and therefore we expect only a few of these predicates to be predictive of bugs.

The question then is: how do we find those few useful predicates that will point us to those bugs?

Finding Causes of Bugs

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

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

Example: F(P) = 20, S(P) = 30 ⇒ Failure(P) = 20/50 = 0.4

Let's introduce a statistical measure that will help us determine which predicates are predictors of failure.

Let F(P) be the number of failing runs in which P is observed to be true. (That is, P has a state of 1 or *.) And let S(P) be the number of successful runs in which P is observed to be true.

Then Failure(P) is the ratio of F(P) to the sum of F(P) and S(P).

For example, if the predicate P were observed to be true in 20 failed runs and 30 successful runs, then Failure(P) would equal 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

However, it is not enough simply to track which predicates have the highest Failure scores. To see why, let's look at the following code.

In this code, we would track the predicates f == NULL and x == 0 because the former is a branch condition and the latter is an integer return value of a function. Notice that in all runs of this program in which f is NULL, the program will fail because it attempts to dereference a null pointer. Therefore, the Failure value of the predicate f == NULL would be 1.

However, notice that the predicate x == 0 would also have a Failure score of 1, since whenever this call of foo is completed, we will then proceed to dereference a null pointer.

This predicate is an "innocent bystander," since the program was already doomed by the time we observed the predicate.

Therefore, we need a different statistic that takes into account the context in which predicates are observed so that we don't falsely consider predicates as being predictors of program failure.

Tracking Context

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

F(P observed) = # failing runs observing P

S(P observed) = # successful runs observing P

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

Example: F(P observed) = 40, S(P observed) = 80 ⇒
Context(P) = 40/120 = 0.333…

This is the motivation for the Context statistic, which measures the background chance of program failure regardless of what we observe the value of a predicate to be.

Let F(P observed) be the number of failing runs in which P was observed to be either true or false (that is, in which P had a state of 0, 1, or * but not -), and similarly let S(P observed) be the number of successful runs in which P was observed to be either true or false.

Then Context(P) is defined to be the ratio of F(P observed) to the sum of F(P observed) and S(P observed).

For example, if the predicate P were observed in 40 failed runs and 80 successful runs, then Context(P) would equal 40/120 = one-third.

A Useful Measure: Increase()

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

`Increase(P) = Failure(P) - Context(P)`

- A form of likelihood ratio testing
- `Increase(P) ≈ 1` ⇒ High correlation with failing runs
- `Increase(P) ≈ -1` ⇒ High correlation with successful runs

With the definitions of Failure and Context in place, we can form a useful metric of how well a predicate predicts the failure of a run.

Define Increase(P) to be the difference of Failure(P) and Context(P). We can think of Increase(P) as being a form of likelihood ratio testing.

Increase(P) can range from negative 1 to positive 1.

If P's Failure score is close to 1, then either there were no successful runs where P was true, or there must have been many more failing runs in which P was true than successful runs; and if additionally P's Context score is close to 0, then there must have also been many successful runs in which P was false. So if P's Increase score is near 1, it means that P is highly correlated with the failure of a run.

On the other hand, if P's Failure score is close to 0, then either there were no failing runs where P was true, or there must have been many more successful runs in which P was true than failing runs; and if additionally P's Context score is close to 1, then there must have also been many failing runs in which P was false. So if P's Increase score is near -1, it means that P is highly correlated with the success of a run.

Additional Explanation:

P is an invariant (true in all runs) ⇒ Increase(P) = 0

If P is an invariant -- that is, P is true for all runs of a program -- then Increase(P) would equal zero (assuming it were perfectly sampled; in practice, it will likely be very close to but not exactly zero). This means P has no correlation with the success or failure of a run.

P is dead code (not observed in any run) ⇒ Increase(P) undef.

And if P is never observed in any run -- meaning P is effectively dead code -- then Failure(P) and Context(P) are both undefined, so Increase(P) is likewise undefined.

Increase() Works

```
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
```


Let's revisit our earlier example and work out the Increase values.

Recall that this example has two predicates f == NULL and x == 0, and only f == NULL is a predictor of run failure whereas x == 0 is an innocent bystander. But these two predicates are indistinguishable based on their Failure values alone, since they both are observed to be true on all failing runs of the program.

Now, assuming one (failing) run where f == NULL was observed to be true and two (successful) runs where f == NULL was observed to be false, we can compute Context(f == NULL) to be 1/3 or 0.33, which means that Increase(f == NULL) will be 2/3, or 0.67. This indicates that when f is NULL, there is a much higher chance of run failure than usual.

On the other hand, in the two successful runs in which f != NULL, we never observe the predicate x == 0. Since there is just one failing run in which the predicate x==0 is observed and no successful run in which the predicate is observed, the context of predicate x == 0 is 1. This means that Increase(x == 0) is 0, indicating that when x is 0, there is little -- if any -- change in the probability of run failure. So the predicate x == 0 is not "blamed" for causing the program to fail.

QUIZ: Computing Increase()

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

{QUIZ SLIDE}

Now let's look at the predicates from our previous example program. Previously you recorded the state vectors for these four predicates over two different runs of the program.

As an example, we've filled in the row for the predicate c != 'a'. There is one failing run and one successful run in which this predicate was true. Therefore its Failure score is 1/2 or 0.5. However, since there is one failing run in which it was observed and one successful run in which it was observed, the predicate's Context score is also 1/2. Therefore, its Increase is 0, indicating that the truth of c != 'a' does not have a significant effect on the failure rate of the program beyond its "background" probability of failure.

Now, to check your understanding, compute the Failure, Context, and Increase metrics for the remaining three predicates.

QUIZ: 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			

{SOLUTION SLIDE}

Let's start with the first predicate, c == 'a'. There was one failing run in which we saw this predicate was true, and there were no successful runs in which we saw the predicate was true, so the Failure score is 1. Next, for its Context score, there was one failing run in which the predicate was observed, and there was one successful run in which it was observed, so the Context score is 1/2. Thus the Increase score of this predicate is 1/2, indicating that there is a higher chance of program failure when this predicate is true. (If we were to do more runs of the program, this Increase score would likely become even higher.)

For the third predicate, i < 3, there was one failing run in which the predicate was true and one successful run in which it was true, so its Failure score is 1/2; since these were also the only observations we made of the predicate, the Context score will likewise be 1/2. Thus, just like c != 'a', this predicate's truth is not associated with any significant change in the probability of a run's success or failure.

Now let's look at the fourth predicate, i >= 3. Since there are no failing runs where it is true and 1 successful run in which it is true, its Failure score is 0. However, since it was observed in one successful and one failing run, its Context score is 1/2, so its Increase score is – 1/2. This means that if i >= 3 is true, then the probability of run failure goes down. (This makes sense, because in the code, observing i >= 3 to be true means that the for-loop has ended, leaving no room for violating the assertion and crashing the program.)

SEGMENT

Step 5.1: Isolating a Bug

Isolating the Bug

	Increase
c == 'a'	0.5
c != 'a'	0.0
i < 3	0.0
i >= 3	-0.5

```
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);
  }
}
```


Looking at the Increase scores, we can now see which program behavior is a strong indicator of failure. When the character returned by getc() is a lowercase a, then something happens that tends to cause the program to fail. This allows us to focus our debugging effort on inspecting the code around that program point, rather than the crash point. This is a desirable property of the Increase metric: it tends to localize bugs at the point where the condition that causes the bug first becomes true, rather than at the crash point.

This example suggests an algorithm for automatically identifying predicates that are the best indicators of failure.

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
   - Use lower bound of 95% confidence interval
2. Sort remaining predicates by Increase(P)
   - Again, use 95% lower bound
   - Likely causes with determinacy metrics

The algorithm consists of two steps.

First, it discards all predicates whose Increase score is less than or equal to 0. As we saw in the previous example, these are predicates that are not correlated or inversely correlated with failing runs. These include bystander predicates, predicates that are correlated with successful runs, and so on.

For various reasons such as the use of sampling, the presence of rarely executed parts of the code, or the scarcity of failing runs, the Increase scores for certain predicates may be based on few observations of those predicates. It is therefore important to attach a confidence interval to the Increase scores. Since the Increase metric is a statistic, computing a confidence interval for the underlying parameter is a well-understood problem.

Then, the revised condition for discarding a predicate in this step is if the lower bound of the 95% confidence interval based on the predicate's Increase score is less than or equal to zero. In addition to the predicates discarded by the earlier condition, this revised condition also discards predicates that have high increase scores but very low confidence because of few observations.

In the second step, we simply sort the remaining predicates by their Increase scores, again using the lower bound of the 95% confidence interval based on the Increase scores. We then output the sorted list of predicates along with the Increase scores.

These predicates constitute likely causes of the failing runs, while their Increase scores serve as a determinacy metric; that is, the higher ranked a predicate is in the list, the more deterministic is the failure when that predicate is true.

Isolating the Bug

	Increase
c == 'a'	0.5
c != 'a'	0.0
i < 3	0.0
i >= 3	-0.5

```
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);
  }
}
```


Let's look at the operation of this algorithm on our previous example.

In the first step, the algorithm discards predicates with Increase scores less than or equal to 0. This amounts to discarding predicates c != a, i < 3, and i >= 3. Recall that these predicates indeed are not indicators of this program's failure.

In the second step, the algorithm outputs the only remaining predicate, c == 'a', along with its Increase score of 0.5. Note that there is nothing to sort as only one predicate is retained after the first step.

Let's look at a more realistic example next, where multiple predicates are retained.

Isolating a Single Bug in bc

```
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
```

```
for (; indx < a_count; indx++)
```


The following code snippet is from bc, the bench calculator program in Unix that we talked about earlier.

Assume that we are tracking scalar relationships between integer-valued variables such as indx and scale, besides other kinds of predicates.

The top-ranked predicates, all comparisons between the indx variable and some other program variable at the highlighted line of code, suggest that this program's failure is strongly correlated with the indx variable growing very large.

Looking at this report, one can infer that the most likely explanation is that when this variable grows very large, it exceeds the bound of the buffer named arrays. This in turn results in NULLs being written to unintended memory locations and crashing the program non-deterministically.

This suggests that variable v_count is likely not the real bound of this buffer. A closer inspection reveals that this is indeed the cause of the bug, and that the bound should be the variable a_count instead of the variable v_count.

This happens to be a classic copy-paste error wherein this code snippet is copied from another location in the same program, and the old variable v_count was not renamed to the new variable a_count.

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

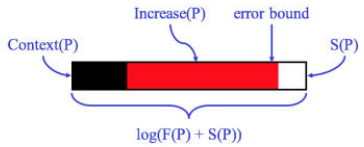
Great, using the Increase score works! Well, at least it works for programs with a single bug.

But real-world programs tend to have more than one bug and, moreover, the effects of these bugs may be interrelated.

In such cases, the algorithm we just saw outputs lots of redundant predicates with high increase values, which is a major problem as it fails to focus the debugging effort.

Using the Information

- Multiple useful metrics: `Increase(P)`, `Failure(P)`, `F(P)`, `S(P)`
- Organize all metrics in compact visual (bug thermometer)



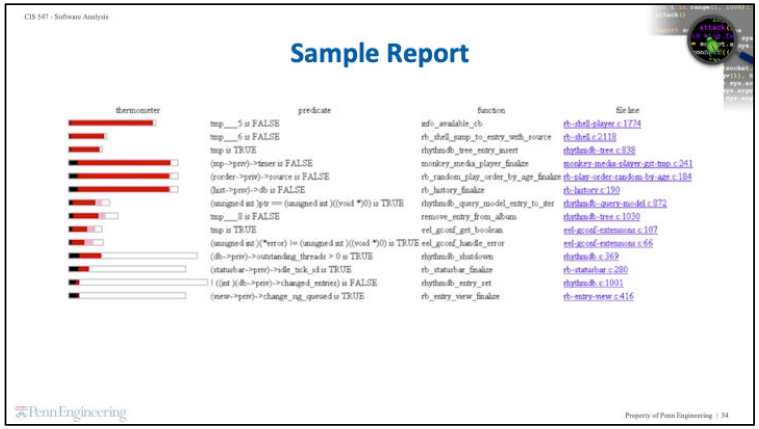
To solve this problem of redundancy, we need to come up with a useful way of organizing the information we have collected.

One way this can be done is by arranging the metrics into a "thermometer."

For each predicate, we have a bar whose length is given by the logarithm of the total number of runs in which the predicate was observed to be true. So small increases in the thermometer size indicate many more runs.

The thermometer has a sequence of bands:

The black band at the left end of the thermometer represents the context score of the predicate.
The red bad represents the increase score of the predicate, more specifically, the lower bound of the Increase score with 95% confidence.
The pink region represents the size of the confidence interval.
And the white band at the right end of the thermometer represents the number of successful runs in which the predicate was observed to be true.



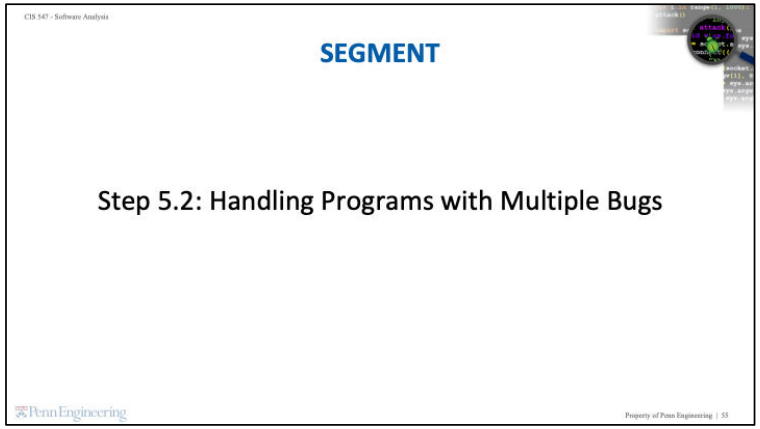
Here's a sample report in which this information has been organized. Recall that these predicates are sorted by Increase scores, which in turn are proportional to the red portions of the thermometers.

The thermometer for the top predicate "tmp___5 is FALSE" is almost entirely red, so this predicate appears to be highly predictive of program failure.

The second predicate from the top, "tmp___6 is FALSE", also has a large proportion of red in its thermometer, so it is also predictive of program failure. But its bar is shorter than the top predicate's, meaning that it was observed to be true in much fewer runs in total.

Near the bottom, we see thermometers with lots of white and very little red, indicating many successful runs where the predicate was true. The increase scores of the predicates near the bottom are close to zero, so these do not rank as high on our list of potential bugs.

The problem of redundancy caused by multiple bugs is evident from this report. In particular, how do we gauge from such a report how many distinct bugs the program contains, and which of these predicates are the best indicators of those bugs?



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.

To summarize, we need an algorithm that will find the best predictor for each bug in the program, without prior knowledge of the number of bugs.

Furthermore, we wish to sort these predictors by the importance of the bugs.

Identifying the best predictor of each bug will help to focus our debugging effort in identifying the bug, whereas sorting these predictors by importance will allow us to prioritize our debugging and patching efforts towards the bugs that affect the most users.

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

In trying to track multiple bugs via statistical debugging, we face new issues that we did not have to worry about when tracking down just one bug.

A single bug may have many different predicates that predict it. On one hand, we only need one predictor in order to start the process of debugging. On the other hand, tracking correlated predictors might reveal extra information useful for the debugging process.

Another issue is that bugs occur on vastly different scales. Some bugs are much more common, and their predictor predicates may overwhelm other predicates that predict less common problems in the program.

An Idea

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

One idea to solve these problems is to approach statistical debugging much like humans tend to fix bugs: we find the first, most important bug and fix it. Then we look at the program to see what the next most important bug is, fix it, and repeat.

We will use this basic procedure to manage our statistical debugging algorithm in a way that prioritizes common bugs but does not lose track of the rarer bugs: these rarer bugs will bubble up to the top as the algorithm simulates fixing the common bugs.

Revised Algorithm

Repeat

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

Until no runs are left

Let's revise our earlier statistical debugging algorithm to now handle programs with multiple bugs without prior knowledge of the number of bugs.

The new algorithm repeats the following steps until it has discarded the entire set of runs of the program.

First, it computes the Failure, Context, Increase, and other metrics for each predicate in our list.

Next, it ranks the predicates according to some ranking scheme based on these metrics. We will describe a suitable ranking scheme shortly.

Third, it takes the top-ranked predicate P and adds it to the list of results. This predicate is deemed the best predictor of the most common bug so far.

Finally, the algorithm removes predicate P from further consideration and discard all successful and failing runs in which P is true. This has the effect of simulating "fixing" the bug predicted by P. Removing these runs reduces the rank of predicates correlated with P, allowing predictors of other bugs to rise to the top. Once we've discarded all runs, then we end the algorithm.

There's one more problem we'll need to overcome: picking the right way of ranking predicates in Step 2. Let's take a look at some different ranking strategies and arrive

at one that works well in practice.

Ranking by Increase(P)

Thermometer	Context	Increase	S	F
	0.065	0.935 ± 0.019	0	23
	0.065	0.935 ± 0.020	0	10
	0.071	0.929 ± 0.020	0	18
	0.073	0.927 ± 0.020	0	10
	0.071	0.929 ± 0.028	0	19
	0.075	0.925 ± 0.022	0	14
	0.076	0.924 ± 0.022	0	12
	0.077	0.923 ± 0.023	0	10

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

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

One strategy is to rank the predicates by the Increase metric.

While this does give us predicates with high increase scores, it tends to give us predicates that explain relatively few failing runs.

We call these these types of predicates “sub-bug predictors” because each of them tends to predict a special case of a more general bug.

In this sample report sorted by Increase scores, each of these predicates has a high increase score, close to 1, but each of them also explains relatively few failing runs, ranging from only 10 to 23 failing runs out of over 5000 failing runs in total. These small numbers of failing runs strongly suggest that these predicates are sub-bug predictors.

Ranking by F(P)

Thermometer	Context	Increase	S	F
	0.176	0.007 ± 0.012	22554	5045
	0.176	0.007 ± 0.012	22566	5045
	0.176	0.007 ± 0.012	22571	5045
	0.176	0.007 ± 0.013	18894	4251
	0.176	0.007 ± 0.013	18885	4240
	0.176	0.008 ± 0.013	17757	4007
	0.177	0.008 ± 0.014	16453	3731
	0.176	0.261 ± 0.023	4800	3716

Problem: Many failing runs but low Increase() scores!

Super-bug predictors: covering several different bugs together

A natural second strategy then is to rank predicates by the number of failing runs in which they are true.

In contrast to the previous strategy of ranking by the Increase metric, this strategy gives us predicates that ostensibly explain large numbers of failing runs but have low increase scores. That is, these predicates are true in many more successful runs than the many failing runs in which they are true.

We call these types of predicates "super-bug predictors" because they tend to cover several different bugs together.

In this sample report sorted by the number of failing runs, each of these predicates is true in a large number of failing runs, ranging from over 3700 to 5000 each, but these predicates also have very low Increase scores, barely above 0. This is corroborated by the fact that these predicates are also true in many successful runs, ranging from 4800 to over 22,000. These numbers strongly imply that these predicates are super-bug predictors.

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 real bug predictors
- Trivial to achieve only high precision or only high recall
- Need both high precision and high recall

To resolve this problem, let's look at an analogy from the field of information retrieval, which defines two concepts: precision and recall. Precision is the fraction of retrieved instances that are relevant, while recall is the fraction of relevant instances that are retrieved.

Translated into our setting, "retrieved instances" are analogous to the predicates that our algorithm reports as bug predictors, whereas relevant instances are analogous to the predicates that are the actual bug predictors.

Achieving high precision or high recall alone is trivial. For instance, if we do not report any predicates as bug predictors, we trivially have high precision but low recall. Likewise, if we report all predicates as bug predictors, we trivially have high recall but low precision. Our goal is to report exactly those predicates which are actual bug predictors, which amounts to achieving both high precision and high recall.

Combining Precision and Recall

- `Increase(P)` has high precision, low recall
- `F(P)` has a high recall, low precision
- Standard solution: take the **harmonic mean** of both

$$= \frac{2}{1/\text{Increase}(P) + 1/F(P)}$$

- Rewards high scores in both dimensions

Returning to our metrics, Increase(P) has high precision but low recall, whereas F(P), the number of failing runs in which P is true, has high recall but low precision.

The standard solution to achieve both high precision and high recall is to take the harmonic mean of these two metrics. In other words, 2 upon the sum of the reciprocals of Increase(P) and F(P).

In this metric, predicates with high scores in both dimensions will have a harmonic mean close to 1, whereas having a low score in either dimension will greatly reduce the harmonic mean (and therefore, the ranking of the predicate).

Sorting by the Harmonic Mean

Thermometer	Context	Increase	S	F
	0.176	0.824 ± 0.009	0	1585
	0.176	0.824 ± 0.009	0	1584
	0.176	0.824 ± 0.009	0	1580
	0.176	0.824 ± 0.009	0	1577
	0.176	0.824 ± 0.009	0	1576
	0.176	0.824 ± 0.009	0	1573
	0.116	0.883 ± 0.012	1	774
	0.116	0.883 ± 0.012	1	776

It works!

And, as you can see, using the harmonic mean works!

Using it as our ranking criterion in the revised algorithm leads to our top predicates having both a large number of failing runs and a large increase score (so the large increase score was likely not due to a low number of samples). This gives us an excellent place to start in our hunt for bugs.

LESSON

Review

SEGMENT

What Have We Learned and Key Takeaways

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: Failure(P), Context(P), Increase(P), ...
- Statistical debugging algorithm to isolate bugs

As we end this module, let's take a moment to review the concepts you've learned.

The essential idea behind this lesson is that statistical debugging takes advantage of the multitude of users running live tests on software through their everyday use of the software, and we can take advantage of these tests to find and fix the bugs that most affect the users' experience.

We saw how to model program behavior by observing predicates on the program's state at given points in the program, and we used this model to abstract away much of the complexity of program state while maintaining sufficient information to locate sources of bugs.

We saw how to reduce the impact of these observations on users' experience by using a sampling framework that amortizes the cost of randomly deciding whether or not to sample an observation; the errors caused by lost information in these samples are mitigated by the large number of users we collect reports from.

We defined metrics to rank predicates' importance to our debugging efforts. These metrics, such as Failure, Context, and Increase scores, give us different kinds of information about the relationship of predicates' observed values to program success or failure.

Finally, we developed an algorithm to isolate bugs in such a way to prevent correlated predictors of common bugs from drowning out predictors of less common bugs.

We just touched on the basics of statistical debugging in this module. If you'd like to learn more, an in-depth treatment of this topic can be found from the link:

http://research.cs.wisc.edu/cbi/

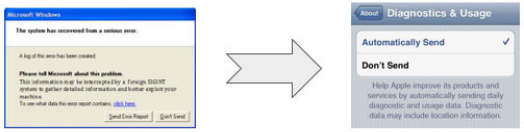
Key Takeaways (1 of 2)

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

The key takeaway from this lesson, though, is that we can learn a lot from the actual executions of software by users. These executions of the code are going to happen anyway, so 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 ...



The common practice of sending crash reports after a failed execution is a step in the right direction, but stack traces have been shown to be useful for only about half the bugs that crop up in the lifetime of a piece of software. And we fail to receive information about successful runs from crash reports.

However, this has been changing. And, as time goes on, you're quite likely to see many more dialogue boxes like this, asking you for permission to continuously monitor executions of the program.