Bug Isolation via Remote Program Sampling

Ben Liblit, Alex Aiken, Alice Zheng, Michael Jordan
University of California, Berkeley

Nicholas Riley
Pablo group meeting
23 April 2003

The problem

- Applications have bugs
- Testing resources limited
- Existing prerelease testing flawed
- Once released, automated testing limited
- No information gathered from successful executions

User feedback today

- Small test group
- Application released to general public
- Many more users than developers, QA, technical support representatives
- Most feedback one-on-one
- Most users connected to Internet

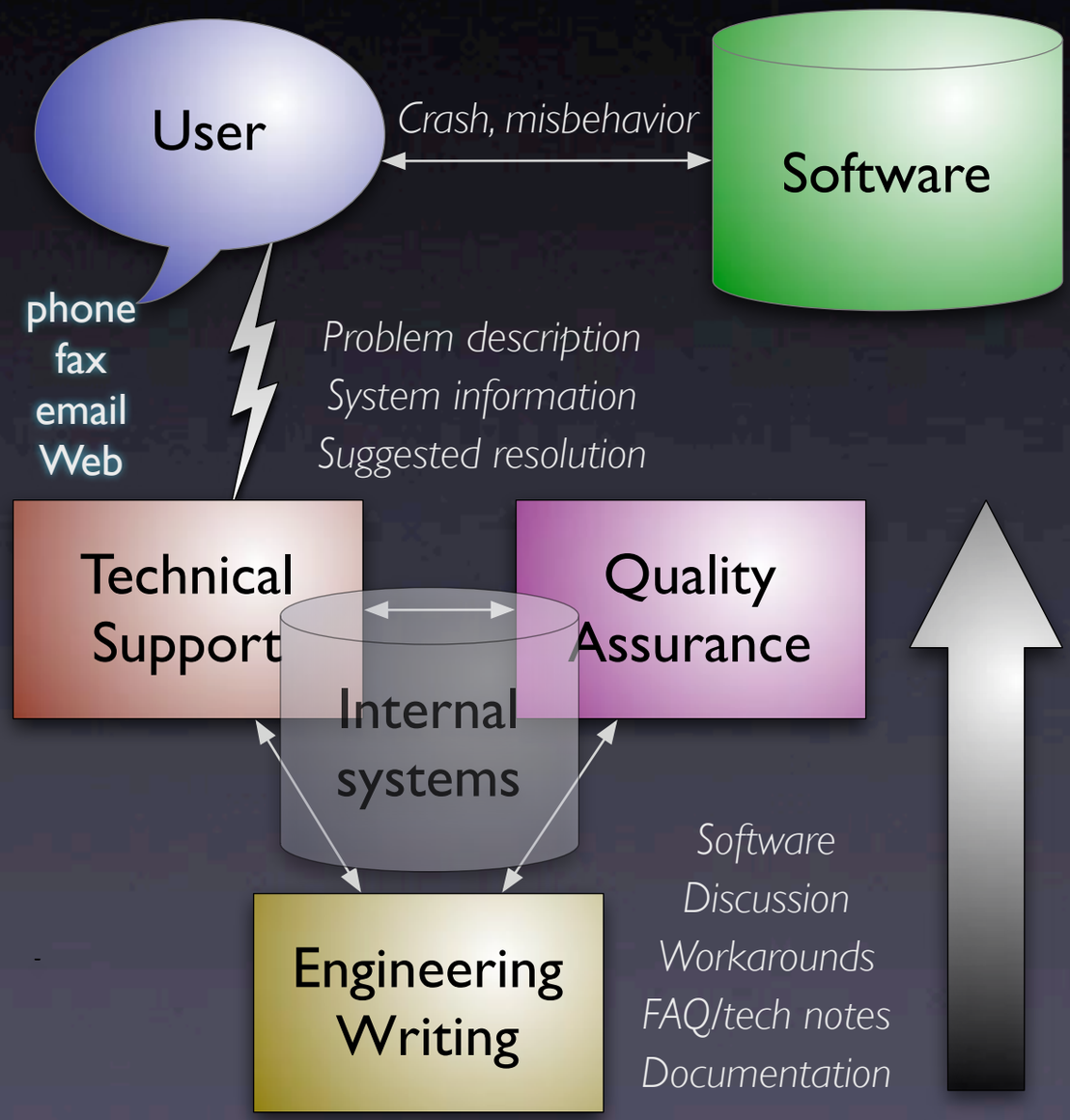
Prerelease testing: *not representative*

- Code coverage
- Users affected
- Context
- Scale (especially in interactive applications)
- Knowledge in test group

Feedback routes

- One-on-one feedback
- Community feedback
- Automated feedback

One-on-one feedback



Community feedback

User User … User



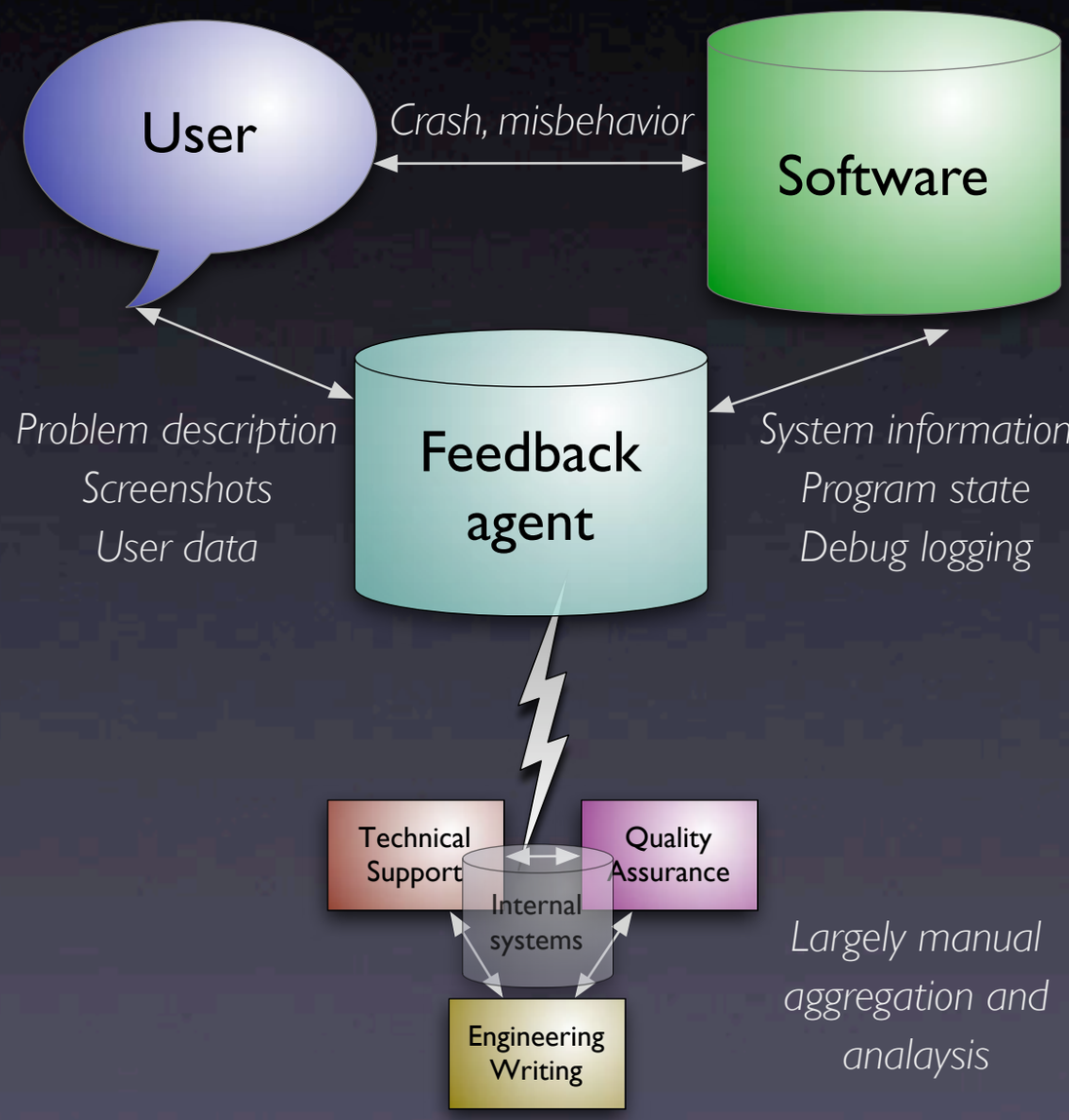
Mailing lists, news, forums, Wikis *Real-time chat* *Public bug database*

Technical Support

Quality Assurance

Engineering Writing

Automated feedback



Feedback provided

- Application crashes when I do *X*
- Application misbehaves when I do *X*
- I want feature *X*
- How do I do *X* with the application?
- It's too difficult/confusing/takes too long to do *X*

Feedback desired

- Application crashes/misbehaves when I do *X*
  - What was the application's state at the time *X* happened?
  - What events led to *X*?
  - Is *X* reproducible?
  - Does *X* depend on details of the user's system configuration?
  - How many users experience *X*?

Feedback desired

- I want feature *X*
  - How many users want *X*?
- How do I do *X* with the application?
  - Where did user interface/documentation fail?
- It's too difficult/confusing/takes too long to do *X*
  - Not addressed

Proposed approach

- Extend automated feedback collection
- Obtain more information using sampling
- Gather information always, not just for failures
- Analyze data automatically

What we want

1. Runtime evaluation of assertions
2. Identification of causes separating normal from abnormal execution behavior
3. Automatic isolation of nondeterministic bugs

What we don't want

- Slower program execution
- Excessive network usage
- Other scaling problems
- Too much information
  - Excessive manual triage
- Privacy violations or security issues

Problems of scale

- Microsoft had accumulated 1 billion crash reports as of a few weeks ago
- Bugs in Mozilla's browser component as of April 23, 2003:

Untriaged	3749
Unresolved	12664
Assigned	4591
Total	141148

Sampling

- Randomized sampling
- Precompute interval until next sample, countdown cached in local variable
- Propagate instrumentation ‘weight’ in CFG

  → Gives minimum safe interval per region
- Perform interprocedural analysis to determine weightless functions

Sampling overhead

- CCured generates many assertions, each with minimal overhead
- Measurements with varying density of instrumentation (Table 2, reproduced at right)
- Static selection reduces overhead to <5% in 94% of cases, worst-case 12%

benchmark	always	10^{-2}	10^{-3}	10^{-4}	10^{-6}
bh	2.81	1.31	1.10	1.07	1.07
bisort	1.08	1.07	1.05	1.05	1.04
em3d	2.14	1.12	1.04	1.02	1.04
health	1.02	1.03	1.02	1.02	1.02
mst	1.25	1.06	1.04	1.03	1.04
perimeter	1.08	1.19	1.13	1.13	1.12
power	1.36	1.07	1.05	1.04	1.04
treeadd	1.13	1.09	1.09	1.09	1.11
tsp	1.05	1.17	1.16	1.15	1.14
compress	2.01	1.21	1.14	1.14	1.14
go	1.17	1.46	1.26	1.22	1.22
ijpeg	2.46	1.17	1.05	1.04	1.03
li	1.58	1.24	1.18	1.16	1.16

Sampling effectiveness

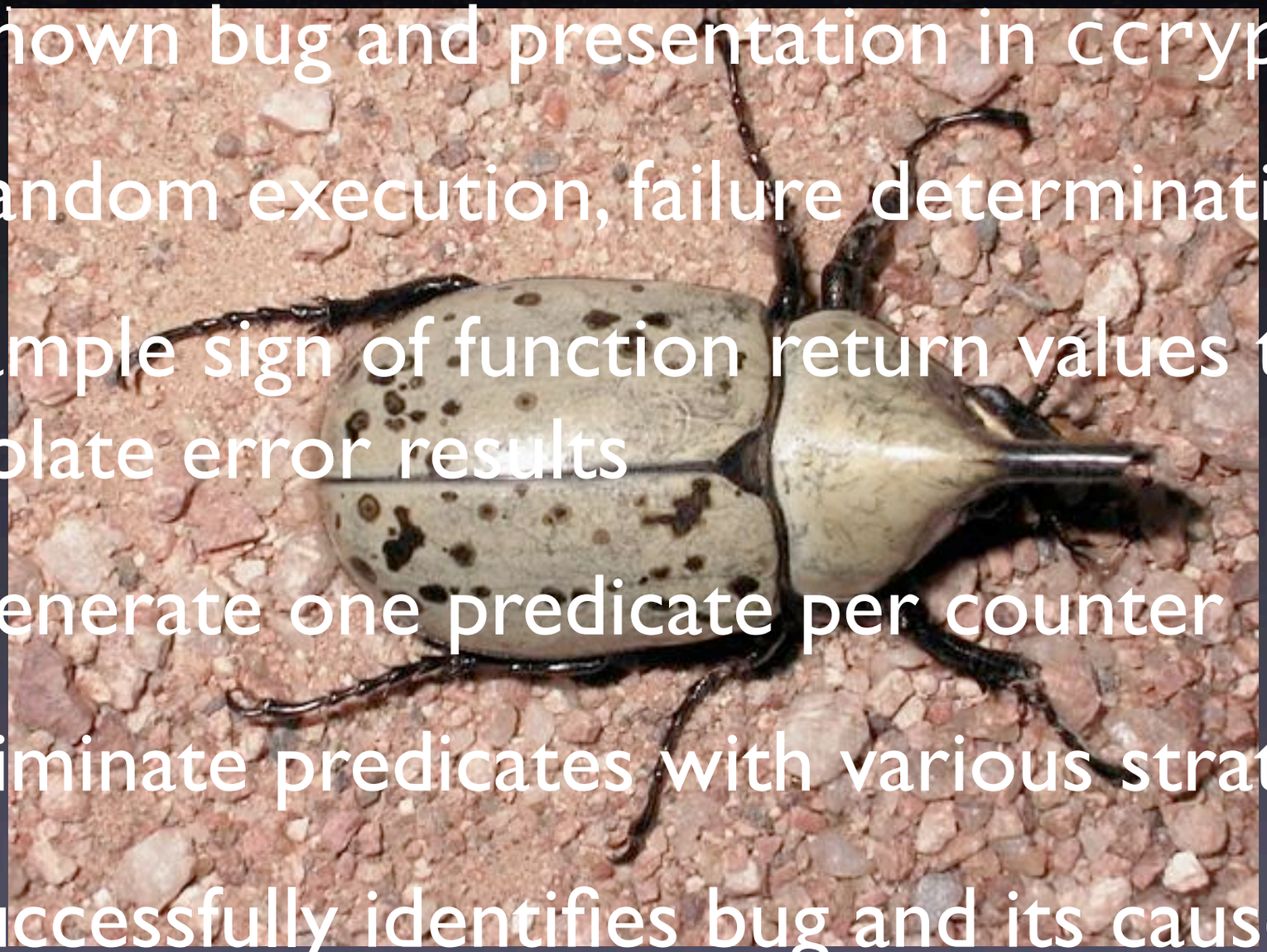
- With 10^{-3} sampling, 10^{-2} frequency of anomaly, for 90% confidence, we need:

$$\log\left(1-0.90\right)/\log\left(1-\frac{1}{100\times1000}\right)=230{,}258\text{ runs}$$

- For useful metrics, need at least thousands of executions per day

Bug isolation

- Known bug and presentation in `ccrypt`
- Random execution, failure determination
- Sample sign of function return values to isolate error results
- Generate one predicate per counter
- Eliminate predicates with various strategies
- Successfully identifies bug and its cause



Predicate elimination

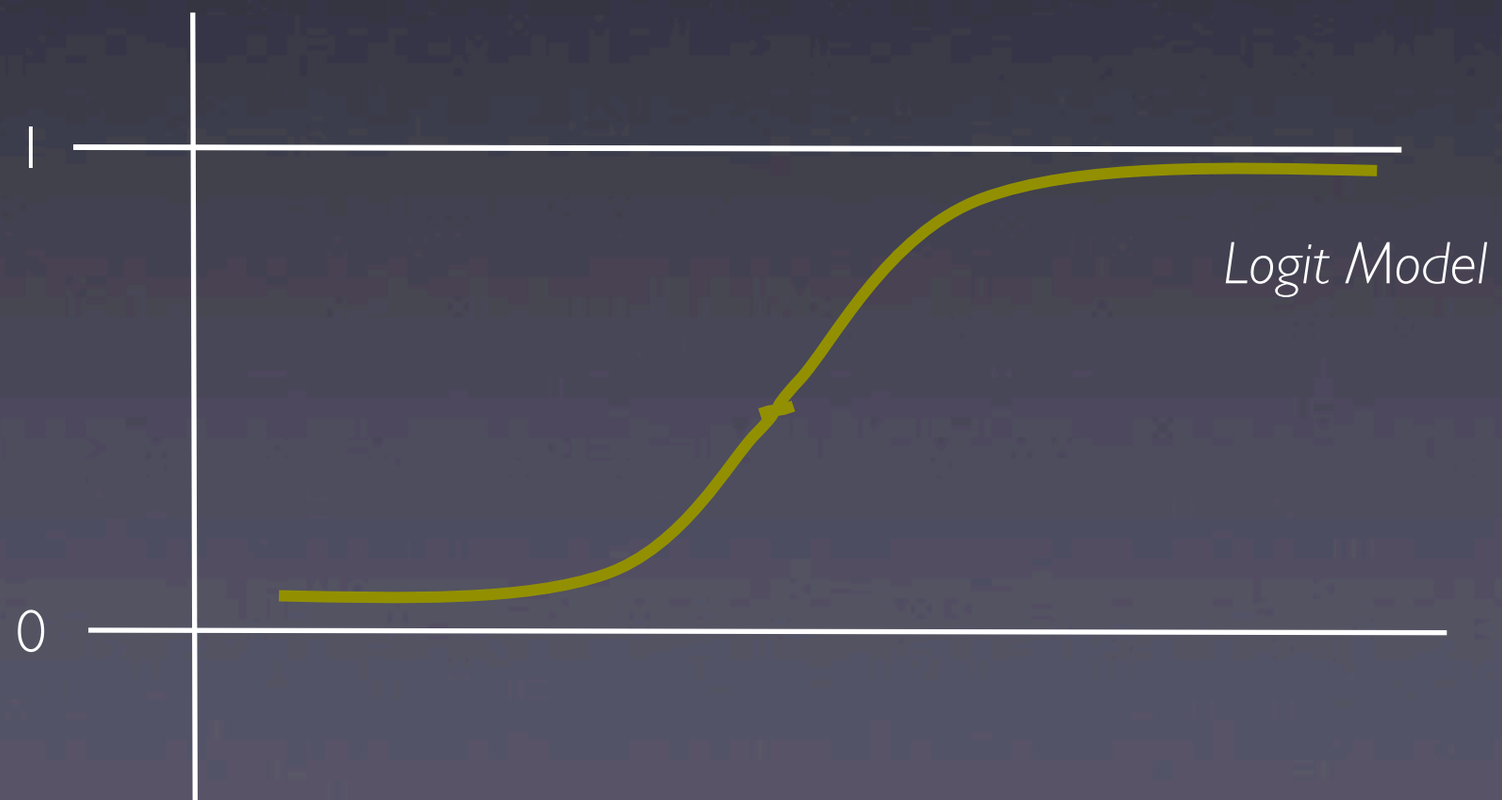
Strategy	Discarded counters	Remaining predicates
1. Universal falsehood	1569	141
2. Lack of failing coverage	1578	132
3. Lack of failing example	1665	45
4. Successful counterexample	139	1571
1 + 4 or 3 + 4	~1968	2

Nondeterminism

- Can’t eliminate by “successful counterexample”.
- Focus on pointer/memory errors in bc
- For each scalar update, compare new value with in-scope same-typed variables
  - Count <, =, >: separate features in model
- Train predictions of success or failure based on features

Binary logistic regression

- S-shaped function
- Dependent variable is "dummy" (0, 1)
- Irrelevant variables can affect model quality



Regularization

- Maximize log likelihood of training set
- Want to ignore most input features
- Penalize log likelihood by regularization term λ to reduce number of nonzero β coefficients

Data analysis

- Training
- Cross-validation ($\lambda = 0.3$)
- Testing

Strategy	Eliminated features	Remaining features
1. Universal falsehood	27242	2908
3. Lack of failing example	642	1571

Performance

- Deterministic bug isolation
  - Hard to optimize instrumented returns
  - Unconditional instrumentation is OK
- Nondeterministic isolation (Figure 4)
  - Unconditional instrumentation: 13%
  - 10^{-3} as in experiment: 0.5% overhead

Security and privacy

- User control
- Anonymity
- Statistical methods
- Accidental data collection
- Spamming
- Trust

Other applications

- Evaluation of arbitrary predicates by statistical sampling
- Lower-overhead dynamic instrumentation
- Isolation of instrumentation or performance problems

Questions

- Generalization of bug isolation?
  - Use existing techniques to detect memory errors
    - Algorithms need to be amenable to working with partial (sampled) information
- Scaling never addressed?

Questions?