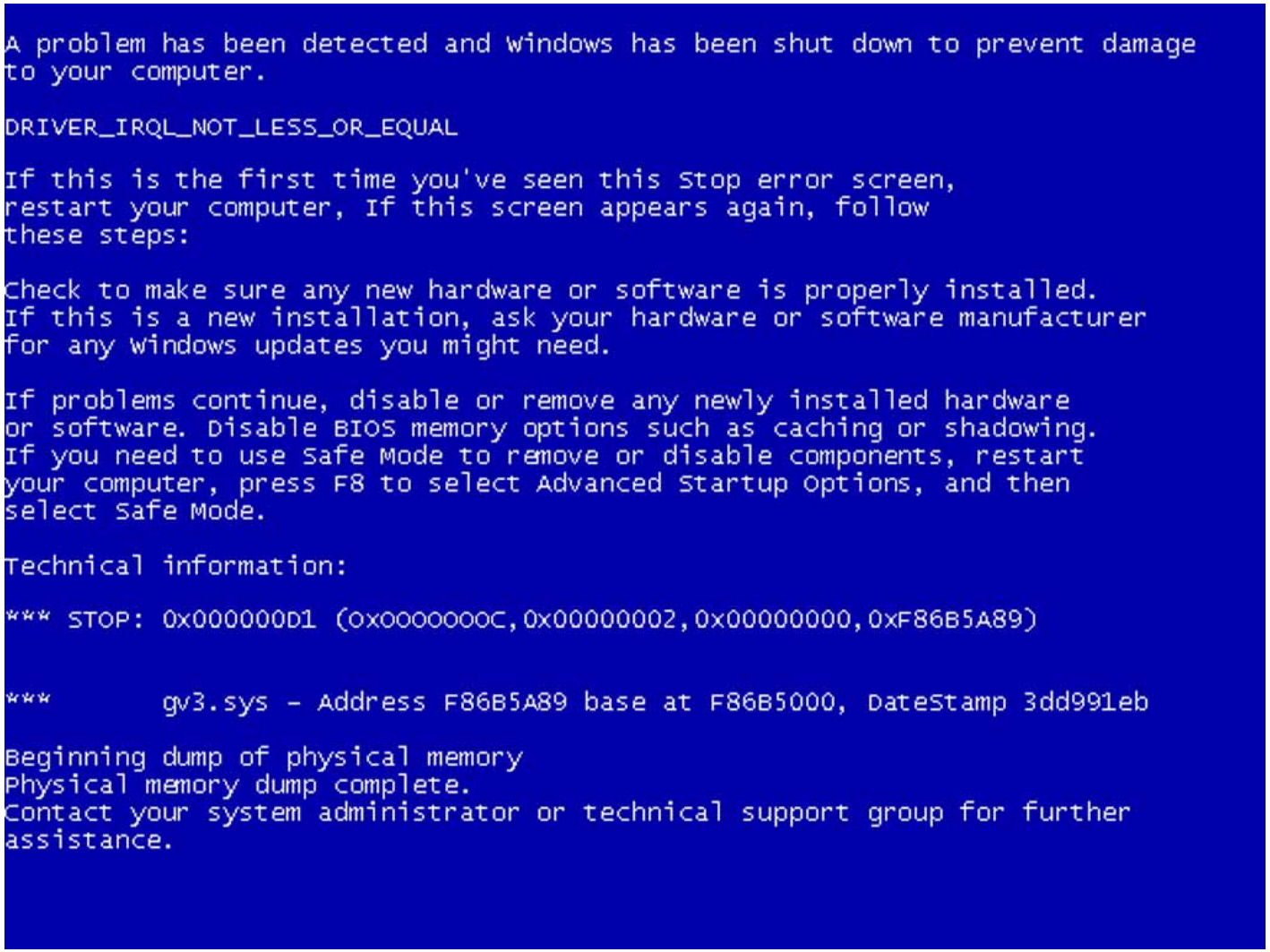
Testing 1…2…3...

Gail Kaiser, Columbia University
kaiser@cs.columbia.edu
April 18, 2013

Why do we test programs?



NEW YORK (CNNMoney)

When it comes to lethal bugs, the computer glitch that set fire to $440 million of Knight Capital Group's funds last Wednesday ranks right up there with the tsetse fly.

Comair cancels all 1,100 flights, cites problems with its computer

Prius hybrids dogged by software

Report: Internal computer woes reportedly cause autos to stall or shut down at highway speeds.

PATRIOT MISSILE DEFENSE

Software Problem Led to System Failure at Dhahran, Saudi Arabia

American Airlines grounds flights after computer outage

Segway recalls 23,500 scooters

Company says software glitch may cause wheels to reverse, causing injuries to the rider.

September 14 2006: 9:56 AM EDT

The day a software bug almost killed the Spirit rover

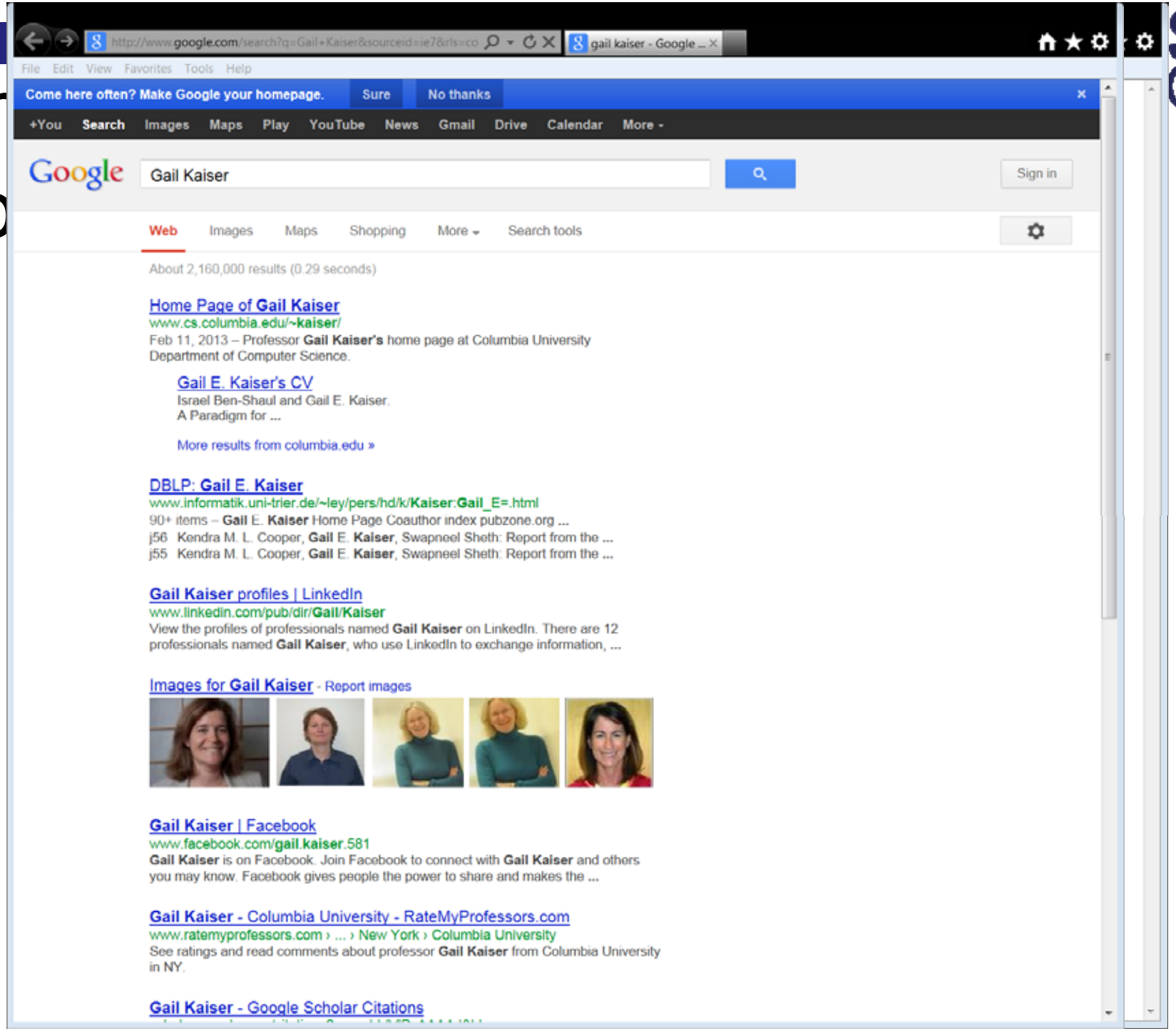
The Spirit rover's Mars mission almost ended before it really got going due to a DOS-related software bug, which wasn't caught due to a rushed development schedule

How do we know whether a test passes or fails?

- 2 + 2 = 4

```
> Add(2,2)
> qwertyuiop
```

Th
so


gail kaiser - Google ...
File Edit View Favorites Tools Help
Come here often? Make Google your homepage.
Sure
No thanks
+You Search Images Maps Play YouTube News Gmail Drive Calendar More
Google
Gail Kaiser
Sign in
Web Images Maps Shopping More Search tools
About 2,160,000 results (0.29 seconds)
Home Page of Gail Kaiser
www.cs.columbia.edu/~kaiser/
Feb 11, 2013 – Professor Gail Kaiser's home page at Columbia University Department of Computer Science.
Gail E. Kaiser's CV
Israel Ben-Shaul and Gail E. Kaiser. A Paradigm for ...
More results from columbia.edu »
DBLP: Gail E. Kaiser
www.informatik.uni-trier.de/~ley/pers/hd/k/Kaiser:Gail_E=.html
90+ items – Gail E. Kaiser Home Page Coauthor index pubzone.org ...
j56 Kendra M. L. Cooper, Gail E. Kaiser, Swapneel Sheth: Report from the ...
j55 Kendra M. L. Cooper, Gail E. Kaiser, Swapneel Sheth: Report from the ...
Gail Kaiser profiles | LinkedIn
www.linkedin.com/pub/dir/Gail/Kaiser
View the profiles of professionals named Gail Kaiser on LinkedIn. There are 12 professionals named Gail Kaiser, who use LinkedIn to exchange information, ...
Images for Gail Kaiser - Report images
Gail Kaiser | Facebook
www.facebook.com/gail.kaiser.581
Gail Kaiser is on Facebook. Join Facebook to connect with Gail Kaiser and others you may know. Facebook gives people the power to share and makes the ...
Gail Kaiser - Columbia University - RateMyProfessors.com
www.ratemyprofessors.com › ... › New York › Columbia University
See ratings and read comments about professor Gail Kaiser from Columbia University in NY.
Gail Kaiser - Google Scholar Citations

Why is testing hard?

- The correct answer may not be known for all inputs – how do we detect an error?
- Even when the correct answer *could* be known for all inputs, it is not possible to check all of them in advance – how do we detect errors after release?
- Users will inevitably detect errors that the developers did not – how do we reproduce those errors?

Problem 1: No test oracle

- Conventional software testing checks whether each output is correct for the set of test inputs.
- But for some software, it is not known what the correct output should be for some inputs. How can we construct and execute test cases that will find coding errors even when we do not know whether the output is correct?
- This dilemma arises frequently for machine learning, simulation and optimization applications, often *"Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known."* [Weyuker, 1982]

Problem 2: Testing after release

- Conventional software testing checks whether each output is correct for the set of test inputs.
- But for most software, the development-lab testing process can not cover all inputs and/or internal states that can arise after deployment. How can we construct and execute test cases that operate in the states that occur during user operation, to continue to find coding errors without impacting the user?
- This dilemma arises frequently for continuously executing on-line applications, where users and/or interacting external software may provide unexpected inputs.

Problem 3: Reproducing errors

- Conventional software testing checks whether each output is correct for the set of test inputs.
- But for some (most?) software, even with rigorous pre and post deployment testing, users will inevitably notice errors that were not detected by the developer's test cases. How can we construct and execute new test cases that reproduce these errors?
- This dilemma arises frequently for software with complex multi-part external dependencies (e.g., from users or network).

Overview

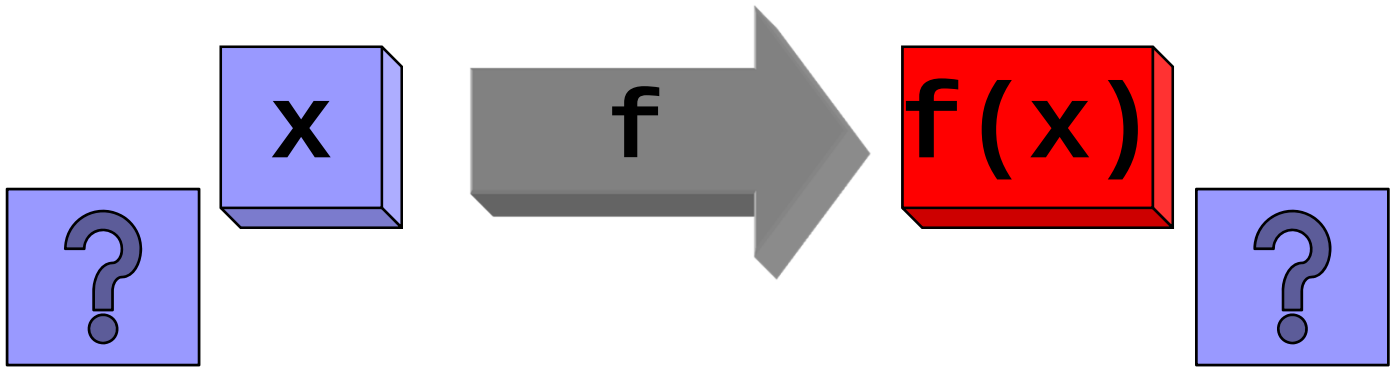
- Problem 1 – testing non-testable programs
- Problem 2 – testing deployed programs
- Problem 3 – reproducing failures in deployed programs

Problem 1: No test oracle

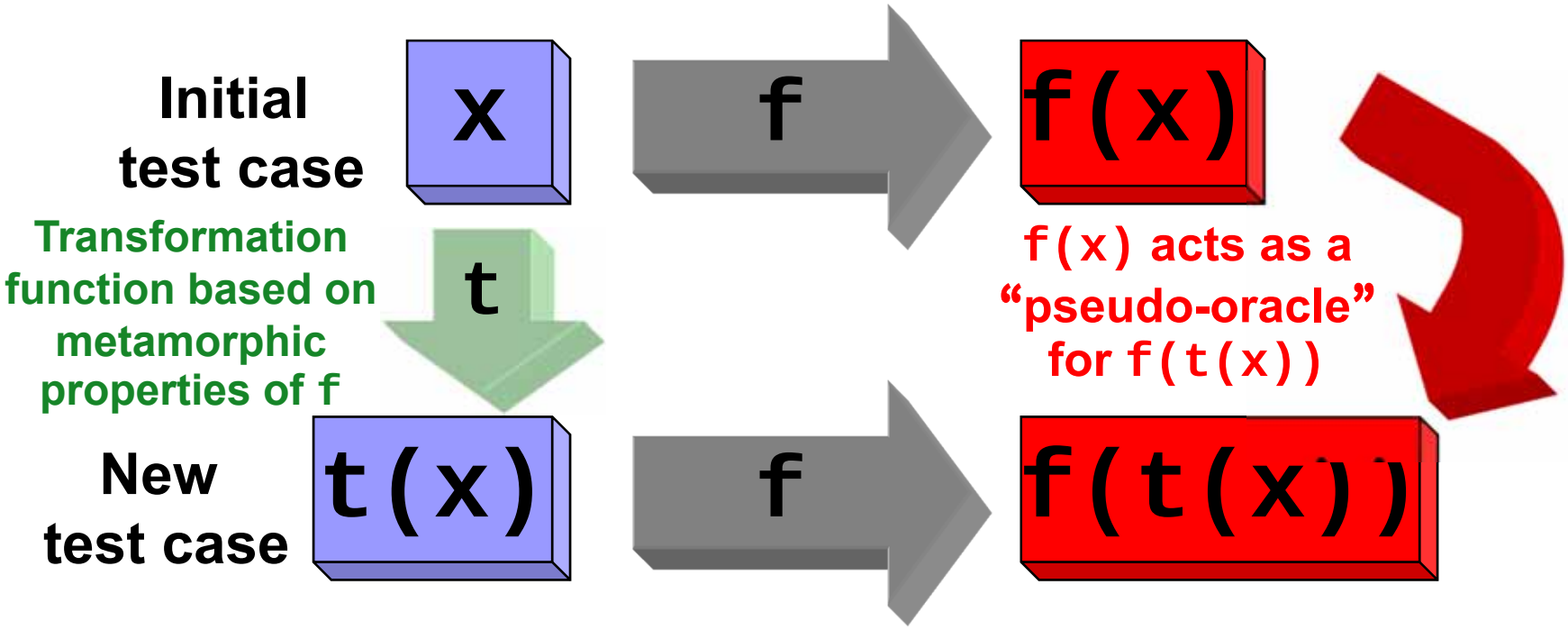
- Conventional software testing checks whether each output is correct for the set of test inputs.
- But for some software, it is not known what the correct output should be for some inputs. How can we construct and execute test cases that will find coding errors even when we do not know whether the output is correct?
- Test oracles may exist for only a limited subset of the input domain and/or may be impractical to apply (e.g., cyberphysical systems).
- Obvious errors (e.g., crashes) can be detected with various testing techniques.
- However, it may be difficult to detect subtle computational defects for arbitrary inputs without true test oracles.

Traditional Approaches

- Pseudo oracles
  - Create two independent implementations of the same program, compare the results
- Formal specifications
  - A complete specification is essentially a test oracle (if practically executable within a reasonable time period)
  - An algorithm may not be a complete specification
- Embedded assertion and invariant checking
  - Limited to checking simple conditions



Metamorphic Testing



- If new test case output `f(t(x))` is as expected, it is not necessarily correct
- However, if `f(t(x))` is not as expected, either `f(x)` or `f(t(x))` – or both! – is wrong

Metamorphic Testing Approach

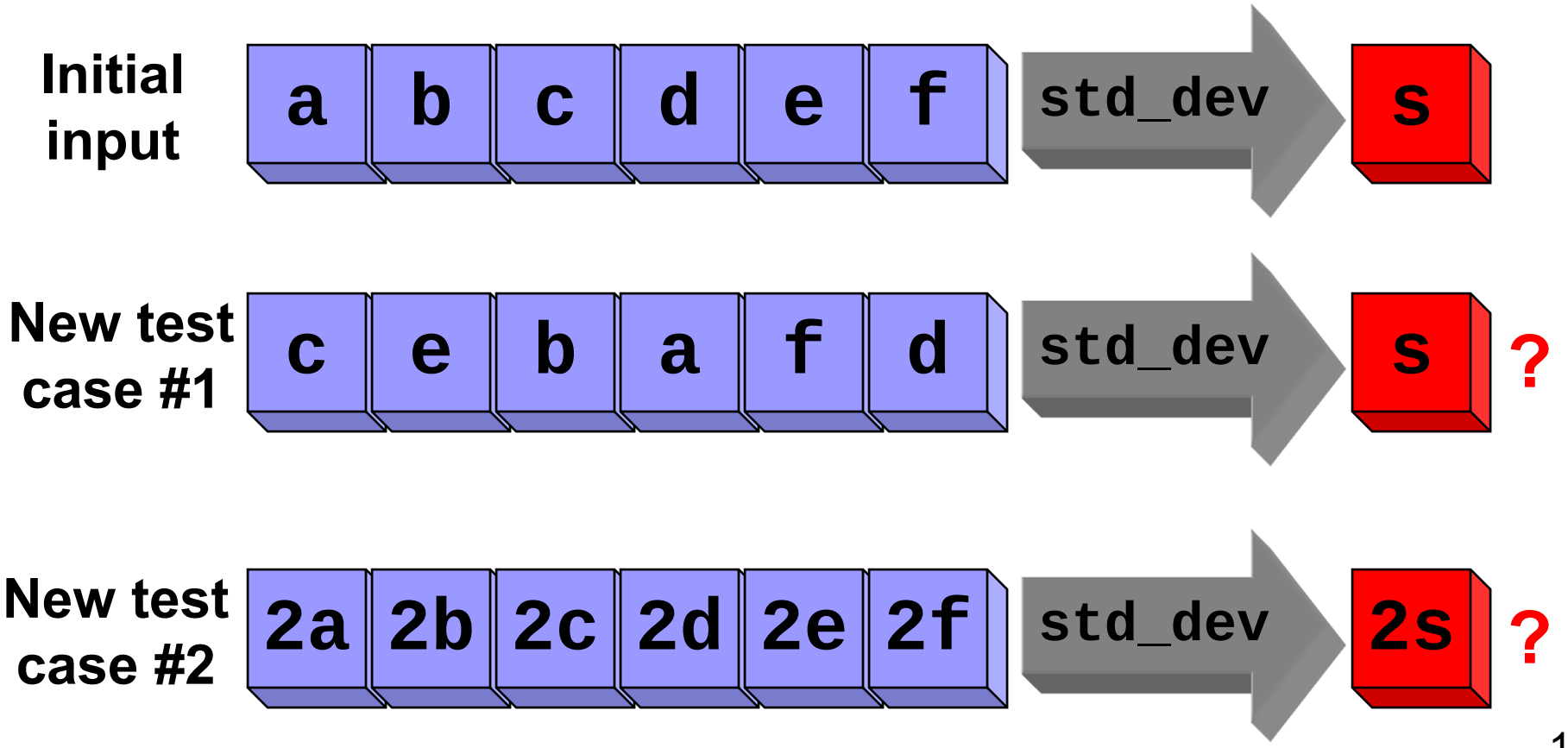
- Many “non-testable” programs have properties such that certain changes to the input yield predictable changes to the output
- That is, when we cannot know the relationship between an input and its output, it still may be possible to know relationships amongst a set of inputs and the set of their corresponding outputs
- Test the programs by determining whether these “metamorphic properties” [TY Chen, 1998] hold as the program runs
- If the properties do not hold, then a defect (or an anomaly) has been revealed

Metamorphic Runtime Checking

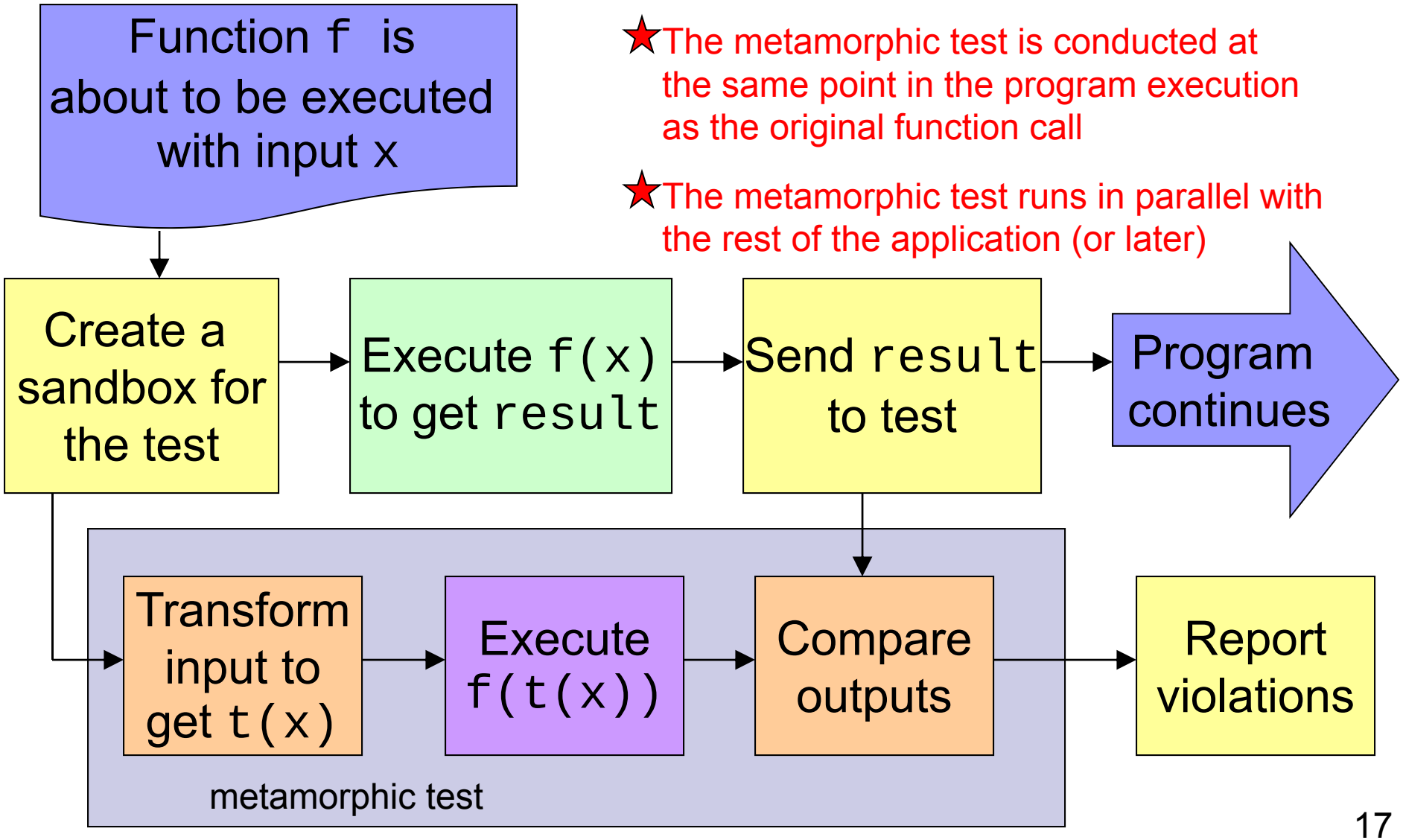
- Most research only considers metamorphic properties of the entire application or of individual functions in isolation
- We consider the metamorphic properties of individual functions and check those properties as the entire program is running
- System testing approach in which functions' metamorphic properties are specified with code annotations
- When an instrumented function is executed, a metamorphic test is conducted at that point, using the current state and current function input (cloned into a sandbox)

Example

- Consider a function to determine the standard deviation of a set of numbers



Model of Execution



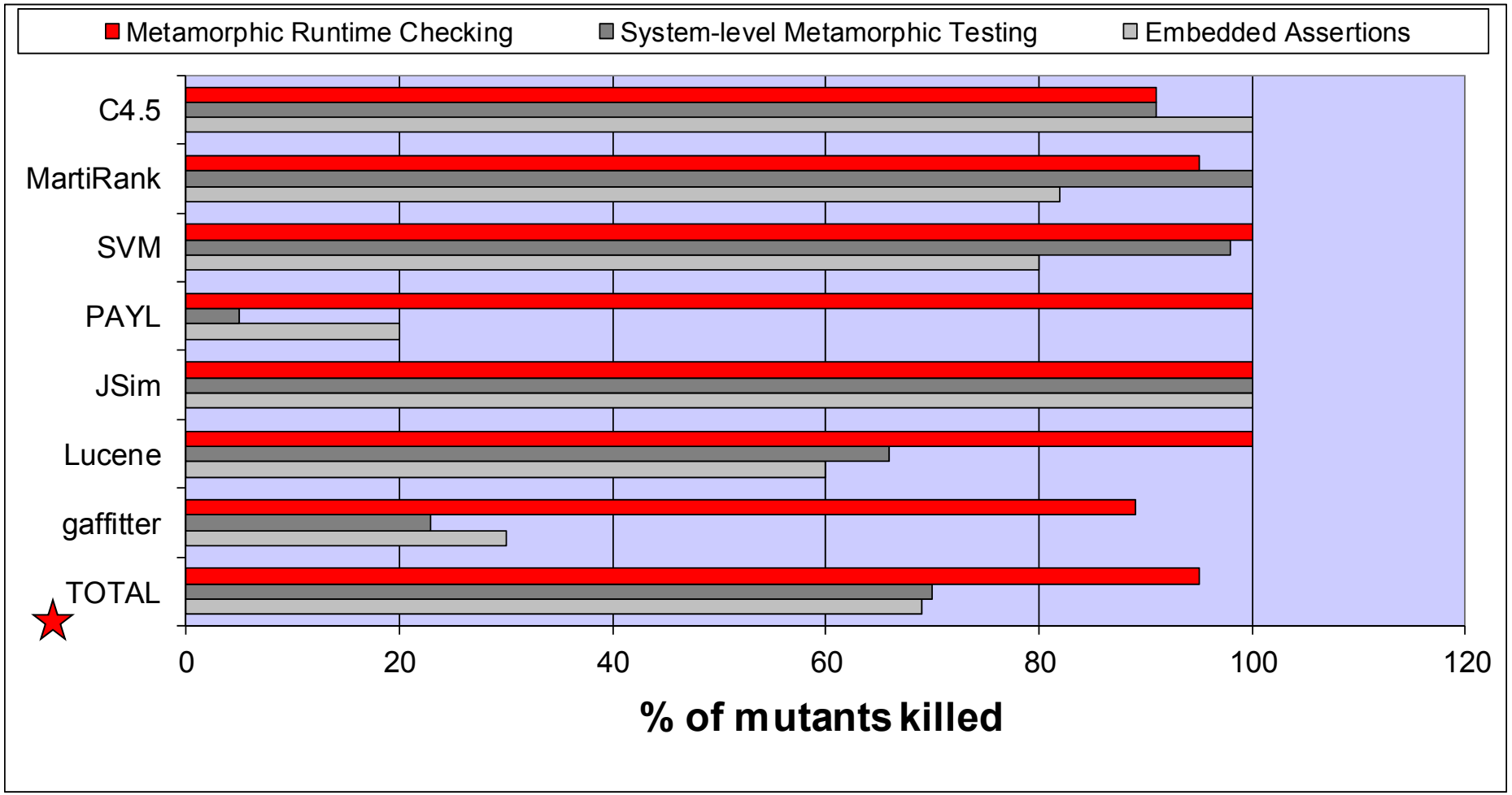
Effectiveness Case Studies

- Comparison:
  - Metamorphic Runtime Checking
    - Using metamorphic properties of individual functions
  - System-level Metamorphic Testing
    - Using metamorphic properties of the entire application
  - Embedded Assertions
    - Using Daikon-detected program invariants [Ernst]
- Mutation testing used to seed defects
  - Comparison operators were reversed
  - Math operators were changed
  - Off-by-one errors were introduced
- For each program, we created multiple versions, each with exactly one mutation
- We ignored mutants that yielded outputs that were obviously wrong, caused crashes, etc.
- Goal is to measure how many mutants were “killed”

Applications Investigated

- Machine Learning
  - **Support Vector Machines** (SVM): vector-based classifier
  - **C4.5**: decision tree classifier
  - **MartiRank**: ranking application
  - **PAYL**: anomaly-based intrusion detection system
- Discrete Event Simulation
  - **JSim**: used in simulating hospital ER
- Optimization
  - **gaffitter**: genetic algorithm approach to bin-packing problem
- Information Retrieval
  - **Lucene**: Apache framework's text search engine

Effectiveness Results



Contributions and Future Work

- Improved the way that metamorphic testing is conducted in practice
  - Classified types of metamorphic properties *[SEKE'08]*
  - Automated the metamorphic testing process *[ISSTA'09]*
- Demonstrated ability to detect real defects in machine learning and simulation applications
  - *[ICST'09; QSIC'09; SEHC'11]*
- Increased the effectiveness of metamorphic testing
  - Developed new technique: Metamorphic Runtime Checking

➢ Open problem: Where do the metamorphic properties come from?

Overview

- Problem 1 – testing non-testable programs
- Problem 2 – testing deployed programs
- Problem 3 – recording deployed programs

Problem 2: Testing after release

- Conventional software testing checks whether each output is correct for the set of test inputs.
- But for most software, the development-lab testing process can not cover all inputs and/or internal states that can arise after deployment. How can we construct and execute test cases that operate in the states that occur during user operation, to continue to find coding errors without impacting the user?
- Re-running the development-lab test suite in each deployment environment helps with configuration options but does not address real-world usage patterns

Traditional Approaches

- Self-checking software is an old idea [Yau, 1975]
- Continuous testing, perpetual testing, software tomography, cooperative bug isolation
  - Carefully managed acceptance testing, monitoring, analysis, profiling across distributed deployment sites
- Embedded assertion and invariant checking
  - Limited to checking conditions with no side-effects

In Vivo Testing Approach

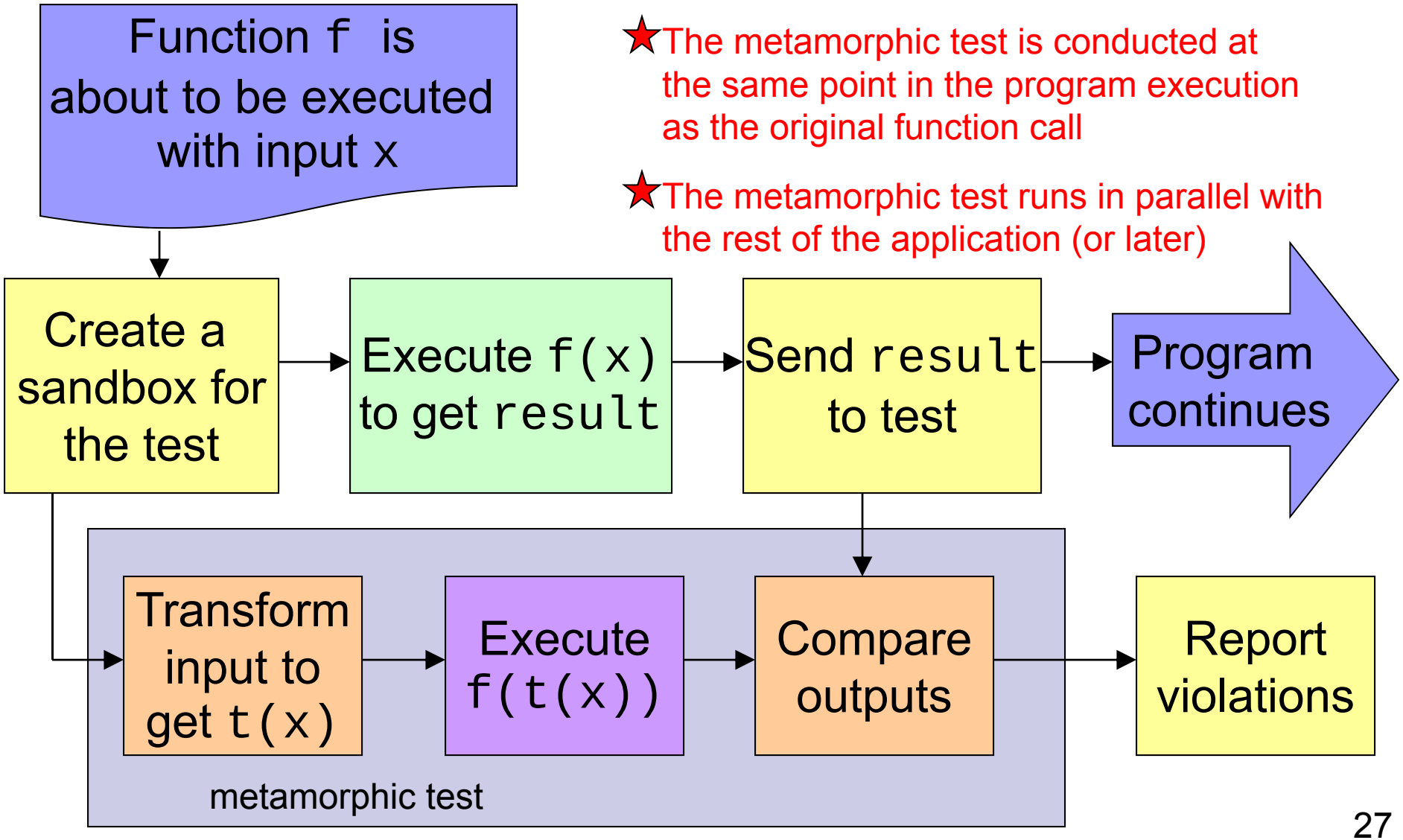
- Continually test applications executing in the field (*in vivo*) as opposed to only testing in the development environment (*in vitro*)
- Conduct unit-level tests in the context of the full running application
- Do so with side-effects but without affecting the system's users
  - Clone to a sandbox (or run "later")
  - Minimal run-time performance overhead

In Vivo Testing

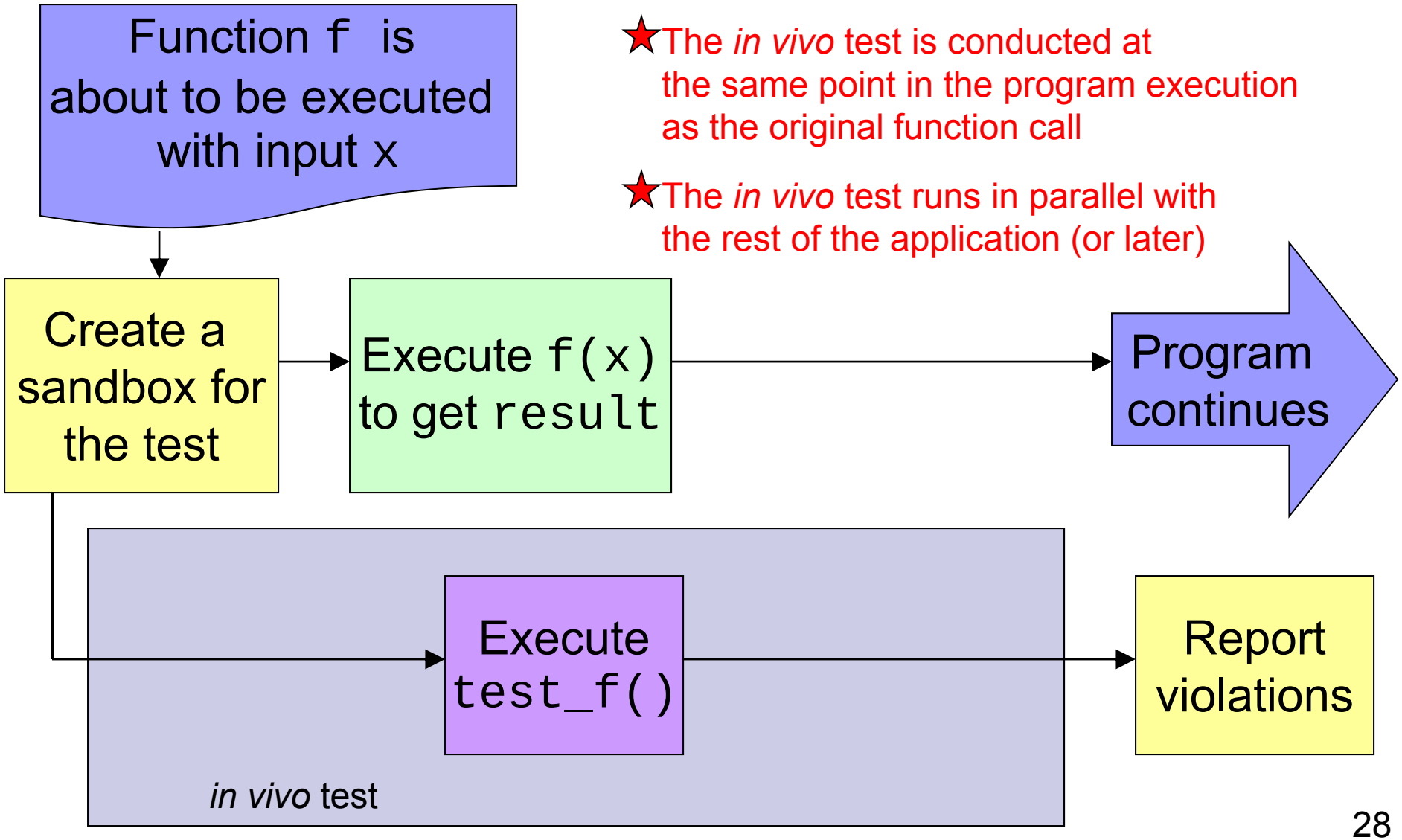
```
int main ( ) {
    ...
    ...
    ...
    foo(x);
```

```
    test_foo(x);
}
```

Metamorphic Model of Execution



In Vivo Model of Execution



Effectiveness Case Studies

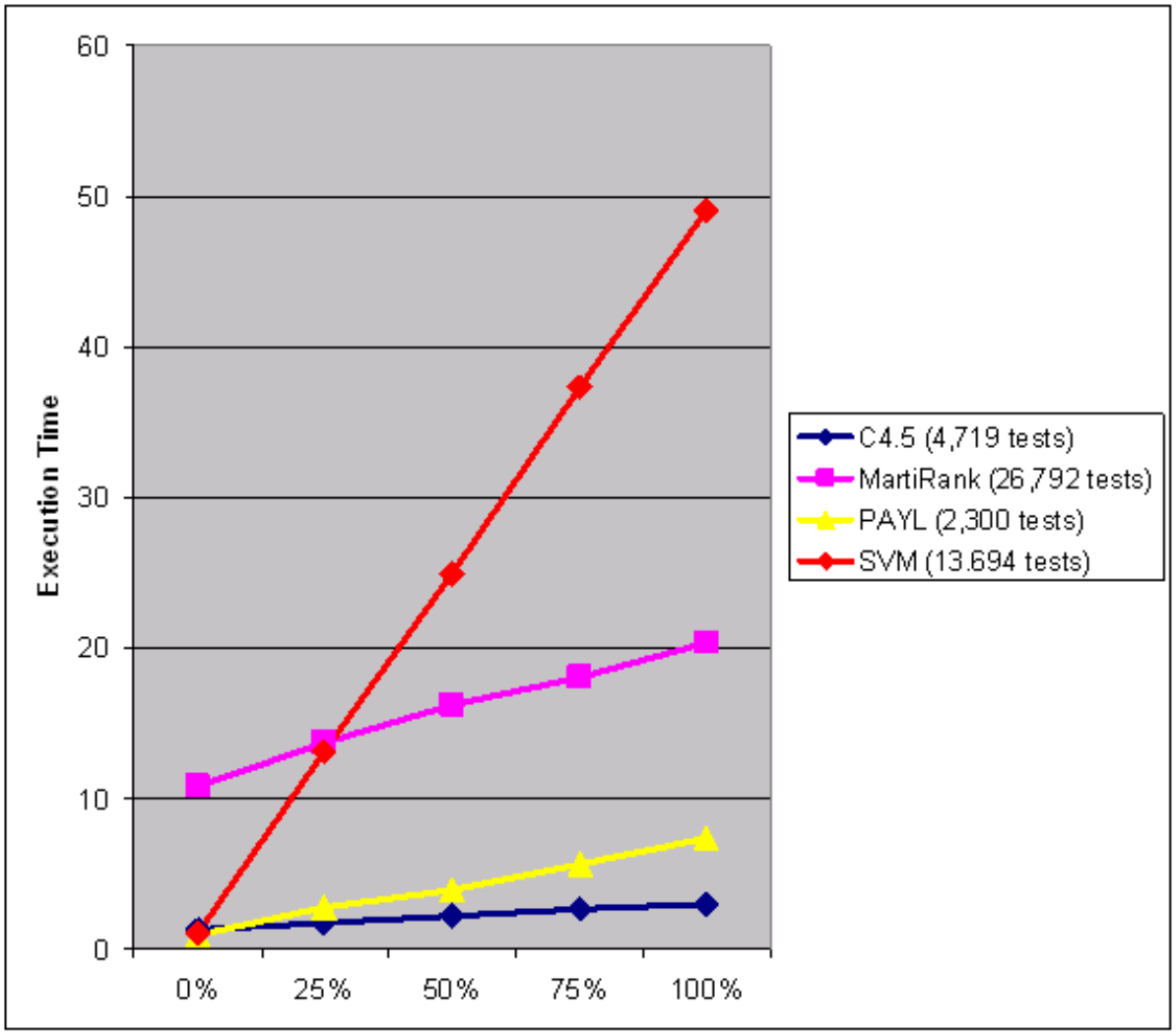
- Two open source caching systems had known defects found by users but no corresponding unit tests
  - OpenSymphony OSCache 2.1.1
  - Apache JCS 1.3
- An undergraduate student created unit tests for the methods that contained the defects
  - These tests passed in “development environment”
- Student then converted the unit tests to *in vivo* tests
  - Driver simulated usage in a “deployment environment”

- *In Vivo* testing revealed the defects, even though unit testing did not
  - Some defects only appeared in certain states, *e.g.*, when the cache was at full capacity

Performance Evaluation

- Each instrumented method has a set probability ρ with which its test(s) will run
- To avoid bottlenecks, can also configure:
  - Maximum allowed performance overhead
  - Maximum number of simultaneous tests
- Config also specifies what actions to take when a test fails
- Applications investigated
  - **Support Vector Machines** (SVM): vector-based classifier
  - **C4.5**: decision tree classifier
  - **MartiRank**: ranking application
  - **PAYL**: anomaly-based intrusion detection system

Performance Results



Contributions and Future Work

- A concrete approach to self-checking software
  - Automated the *in vivo* testing process *[ICST'09]*
- Steps towards overhead reduction
  - Distributed management: Each of the N members of an application community perform 1/Nth the testing *[ICST'08]*
  - Automatically detect previously tested application states *[AST'10]*
- Has found real-world defects not found during pre-deployment testing

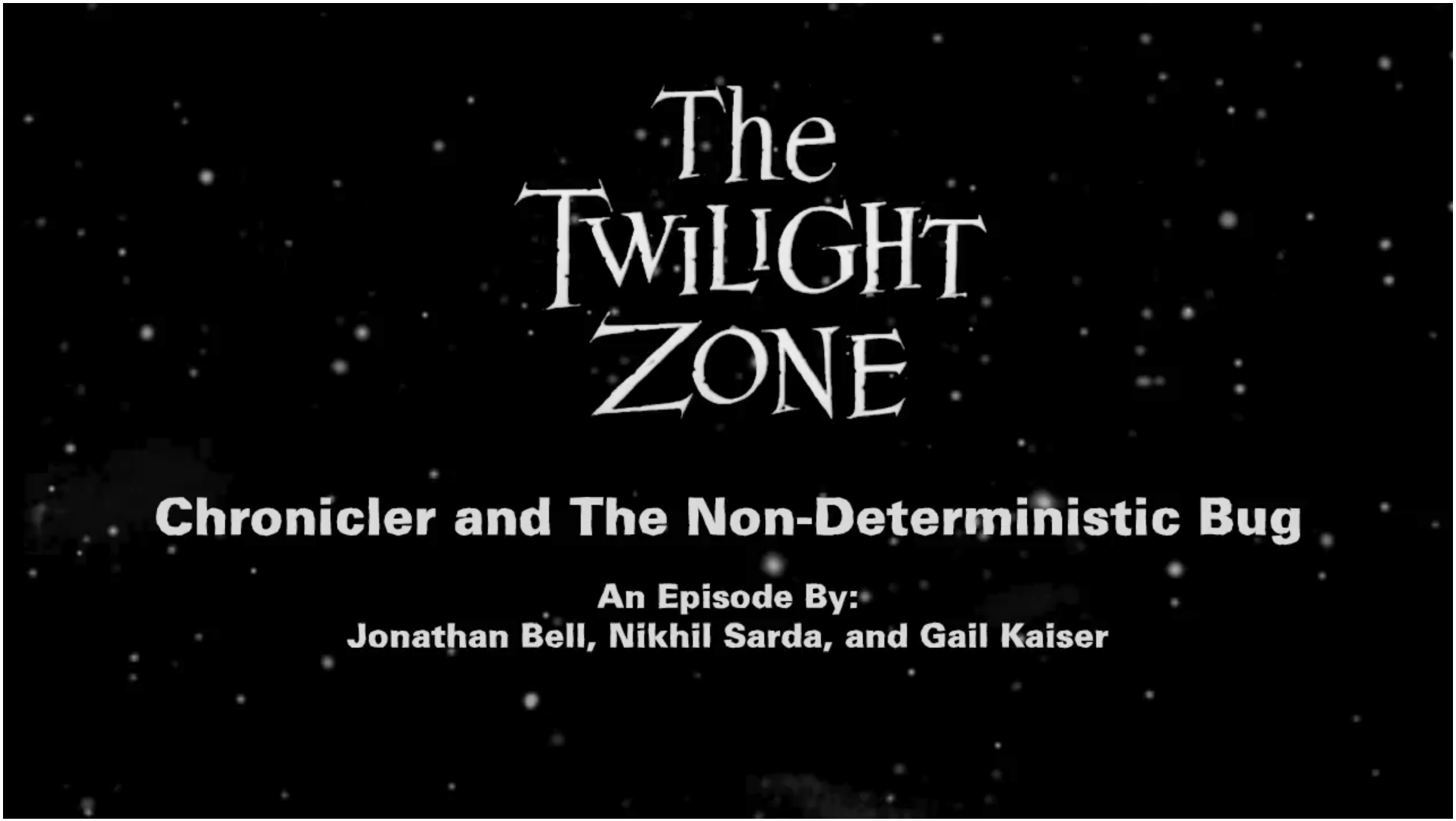
- Open problem: Where do the *in vivo* tests come from?

Overview

- Problem 1 – testing non-testable programs
- Problem 2 – testing deployed programs
- Problem 3 – recording deployed programs

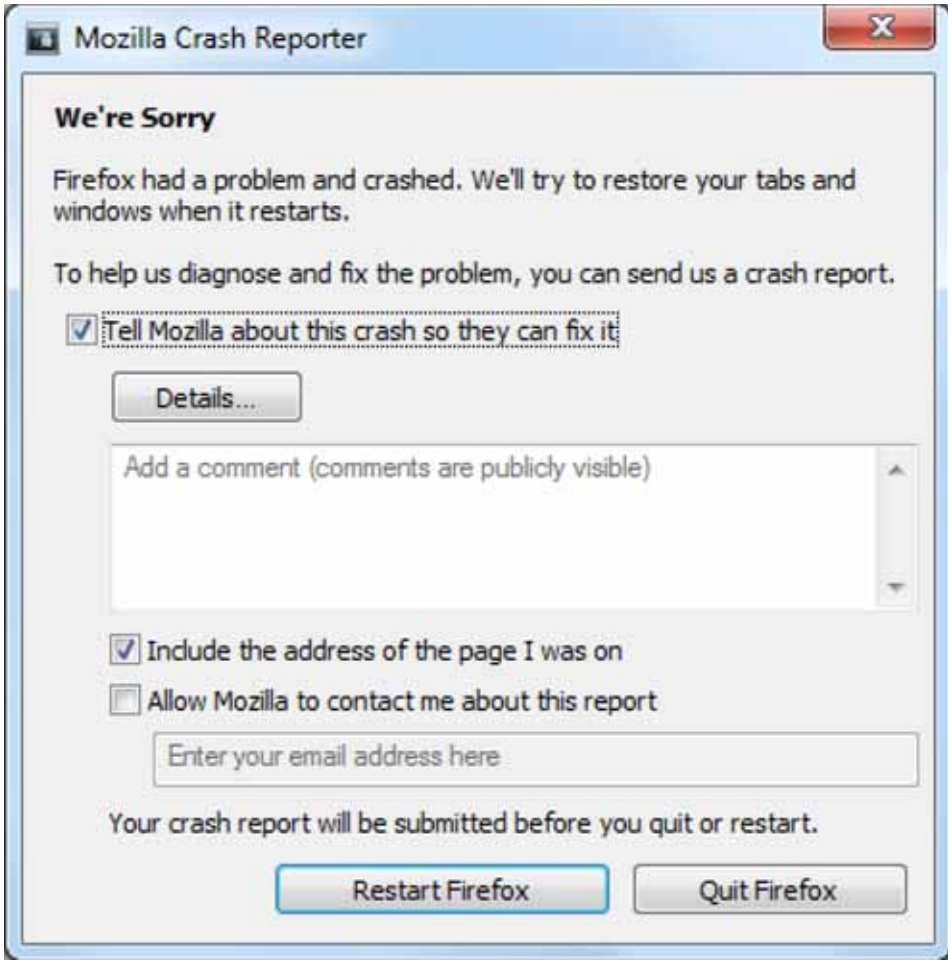
Problem 3: Reproducing errors

- Conventional software testing checks whether each output is correct for the set of test inputs.
- But for some (most?) software, even with rigorous pre and post deployment testing, users will inevitably notice errors that were not detected by the developer's test cases. How can we construct and execute new test cases that reproduce these errors?
- The user may not even know what triggered the bug.


The Twilight Zone
Chronicler and The Non-Deterministic Bug
An Episode By:
Jonathan Bell, Nikhil Sarda, and Gail Kaiser

Traditional Approaches

- Mozilla Crash Reporter

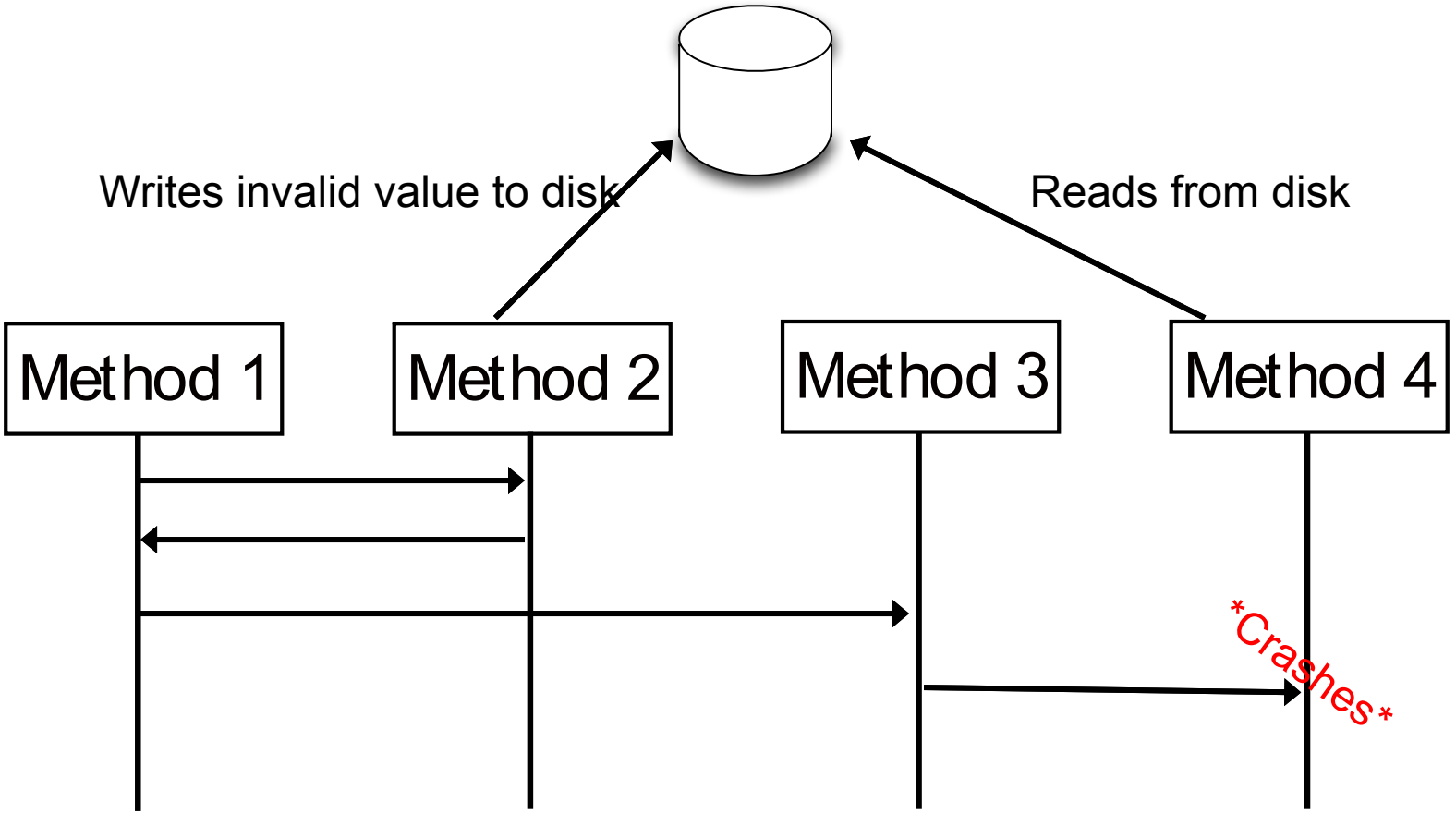


Frame	Module	Signature	Source
0	XUL	js::MaybeGC	js/src/jsgc.cpp:2654
1	XUL	nsJSContext::ScriptEvaluated	dom/base/nsJSEnvironment.cpp:2806
2	XUL	nsCxPusher::~nsCxPusher	content/base/src/nsContentUtils.cpp:3127
3	XUL	nsGlobalWindow::SetNewDocument	dom/base/nsGlobalWindow.cpp:2150
4	XUL	DocumentViewerImpl::InitInternal	layout/base/nsDocumentViewer.cpp:926
5	XUL	DocumentViewerImpl::Init	layout/base/nsDocumentViewer.cpp:676
6	XUL	nsDocShell::SetupNewViewer	docshell/base/nsDocShell.cpp:8023
7	XUL	nsDocShell::Embed	docshell/base/nsDocShell.cpp:6075
8	XUL	nsDocShell::CreateAboutBlankContentViewer	docshell/base/nsDocShell.cpp:6814
9	XUL	nsDocShell::EnsureContentViewer	docshell/base/nsDocShell.cpp:6697
10	XUL	nsDocShell::GetInterface	docshell/base/nsDocShell.cpp:951
11	XUL	nsGetInterface::operator	obj-firefox/x86_64/xpcom/build /nsInterfaceRequestorUtils.cpp:19
12	XUL	nsCOMPtr_base::assign_from_helper	obj-firefox/x86_64/xpcom/build /nsCOMPtr.cpp:110
13	XUL	nsGlobalWindow::GetDocument	
14	XUL	XUL@0x673d6d	
15	XUL	XPCConvert::NativeInterface2JSObject	js/xpconnect/src/XPCConvert.cpp:837
16	XUL	XPCConvert::NativeData2JS	js/xpconnect/src/XPCConvert.cpp:319
17	XUL	XPCWrappedNative::CallMethod	js/xpconnect/src/xpcprivate.h:3312
18	XUL	XPC_WN_GetterSetter	js/xpconnect/src/xpcprivate.h:2823
19	XUL	js::InvokeKernel	js/src/jscntxtinlines.h:372
20	XUL	js::Invoke	js/src/jsinterp.h:119
21	XUL	js::InvokeGetterOrSetter	js/src/jsinterp.cpp:469
22	XUL	js::Shape::get	js/src/jsscopeinlines.h:296

Traditional Approaches

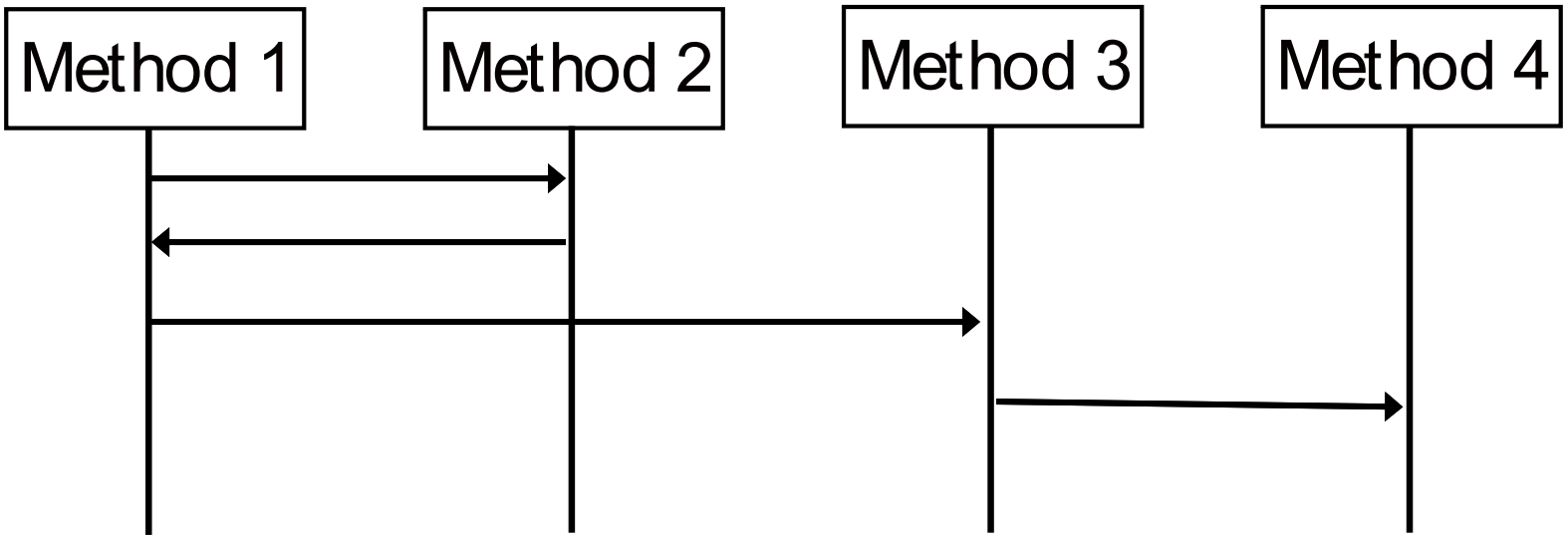
- Log a stack trace at the time of failure
- Log the complete execution
  - This requires logging the result of every branch condition
  - Assures that the execution can be replayed, but has a very high overhead
- Log the complete execution of selected component(s)
  - Only reduces overhead for pinpoint localization
- There are also many systems that focus on replaying thread interleavings

The Problem with Stack Traces



The Problem with Stack Traces

- The stack trace will only show Method 1, 3, and 4 – not 2!
- How does the developer discern that the bug was caused by method 2?

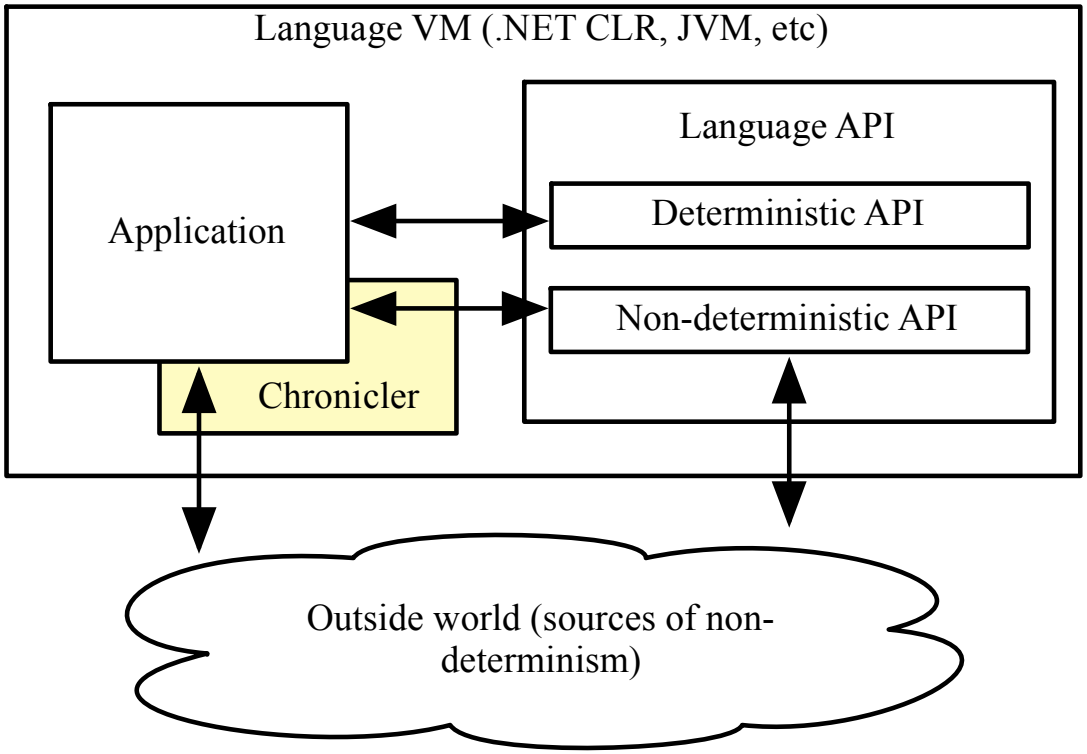


Non-determinism in Software

- Bugs are hard to reproduce because they appear non-deterministically
- Examples:
  - Random numbers, current time/date
  - Asking the user for input
  - Reading data from a shared database or shared files
  - Interactions with external software systems
  - Interactions with devices (gps, etc.)
- Traditional approaches that record this data do so at the system call level – we do so at the API level to improve performance

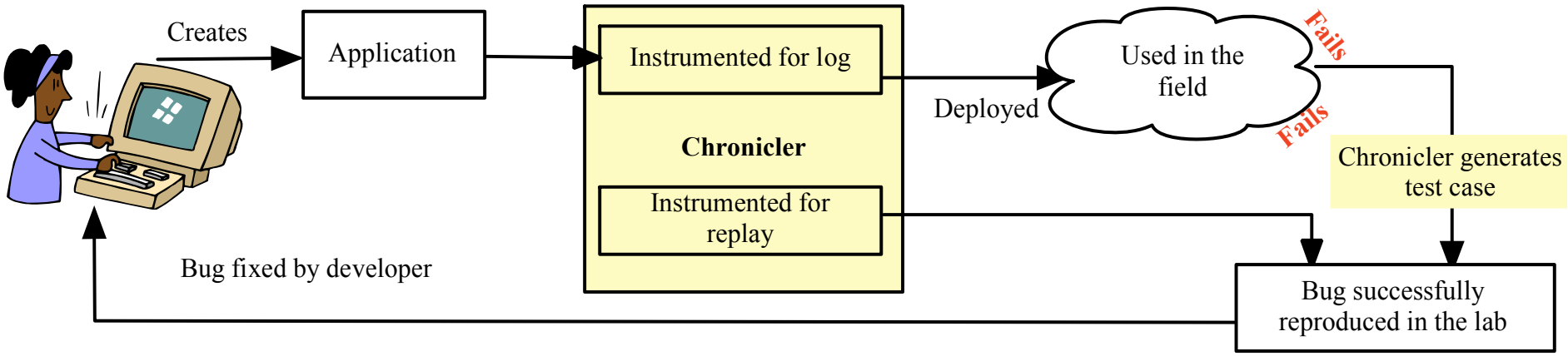
Chronicler Approach

- Chronicler runtime sits between the application and sources of non-determinism at the API level, logging the results
- Many fewer API calls than system calls

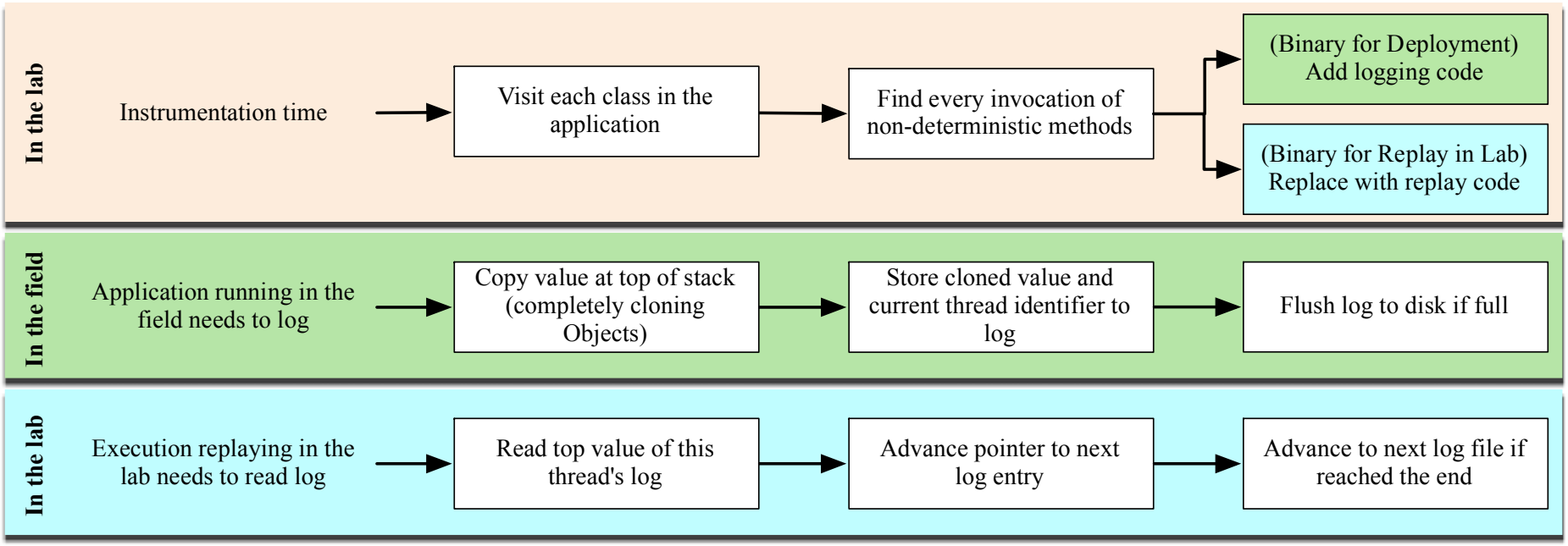


Model of Execution

- Instrument the application to log these non-deterministic inputs and create a replay-capable copy



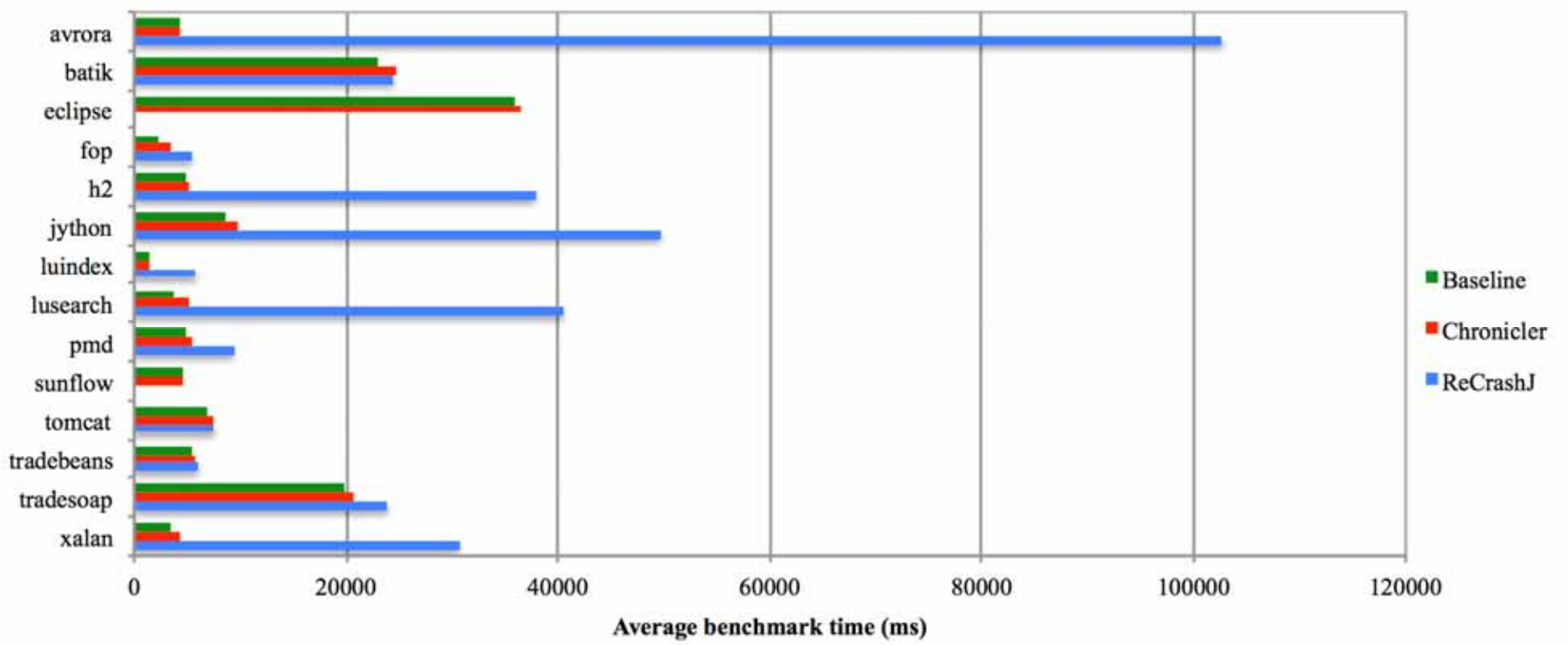
Chronicler Process



Performance Evaluation

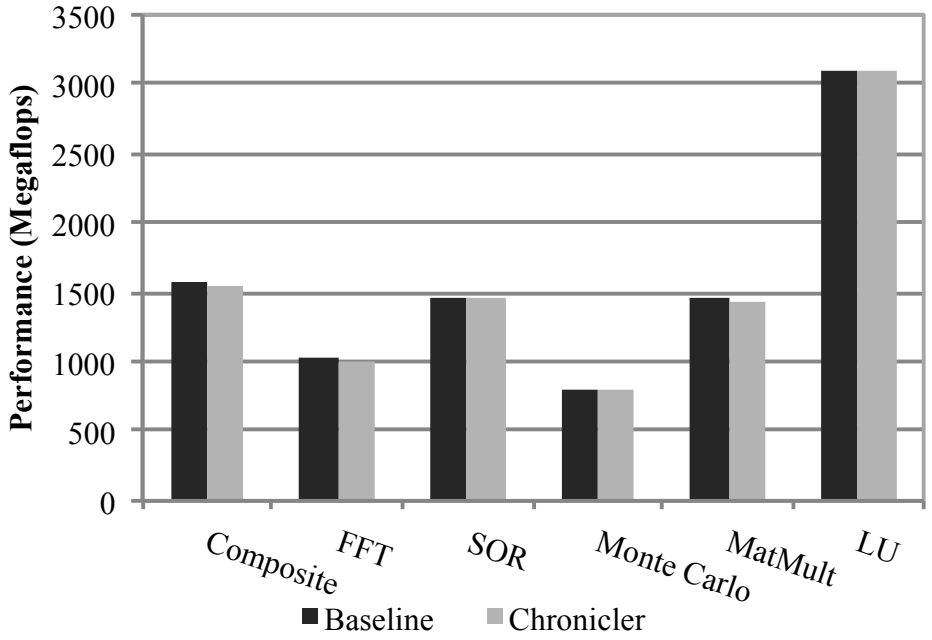
- DaCapo real-world workload benchmark
  - Comparison to RecrashJ [Ernst], which logs partial method arguments
- Computation heavy SciMark 2.0 benchmark
- I/O heavy benchmark – 2MB to 3GB files with random binary data and no linebreaks, using `readLine` to read into a string

Performance Results

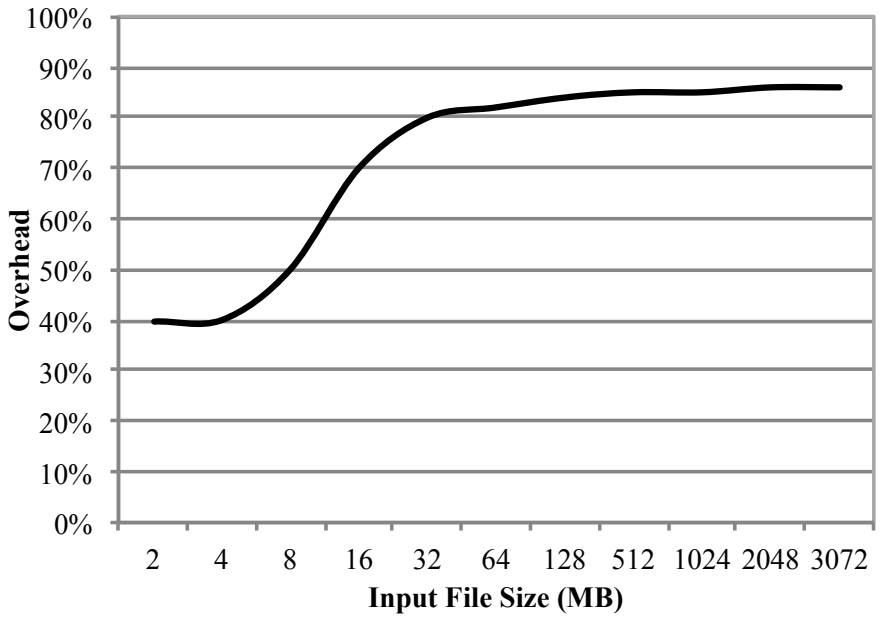


Performance Results

SciMark Benchmark Results (Best Case)



I/O Benchmark Results (Worst Case)



Contributions and Future Work

- A low overhead mechanism for record-and-replay for VM-based languages *[ICSE '13]*
- A concrete, working implementation of Chronicler for Java: ChroniclerJ
  - https://github.com/Programming-Systems-Lab/chroniclerj

- Open problem: How do we maintain privacy of user data?

Collaborators in Software Reliability Research at Columbia

- Software Systems Lab: Roxana Geambasu, Jason Nieh, Junfeng Yang
  - With Geambasu: Replay for sensitive data
  - With Nieh: Mutable replay extension of Chronicler
  - With Yang: Infrastructure for testing distribution
- Institute for Data Sciences and Engineering Cybersecurity Center: Steve Bellovin, Angelos Keromytis, Tal Malkin, Sal Stolfo, et al.
  - With Malkin: Differential privacy for recommender systems
- Computer Architecture and Security Technology Lab: Simha Sethumadhavan
  - Adapting legacy code clones to leverage new microarchitectures
- Bionet Group: Aurel Lazar
  - Mutable record and replay for Drosophila (fruit fly) brain models
- Columbia Center for Computational Learning: cast of thousands
  - With Roger Anderson: Monitoring smart grid, green skyscrapers, electric cars, …

enable (vt): *to make possible, practical, or easy*

PROGRAMMING SYSTEMS LAB
COLUMBIA UNIVERSITY

http://www.psl.cs.columbia.edu/