Research Topics in SE & Distributed Systems

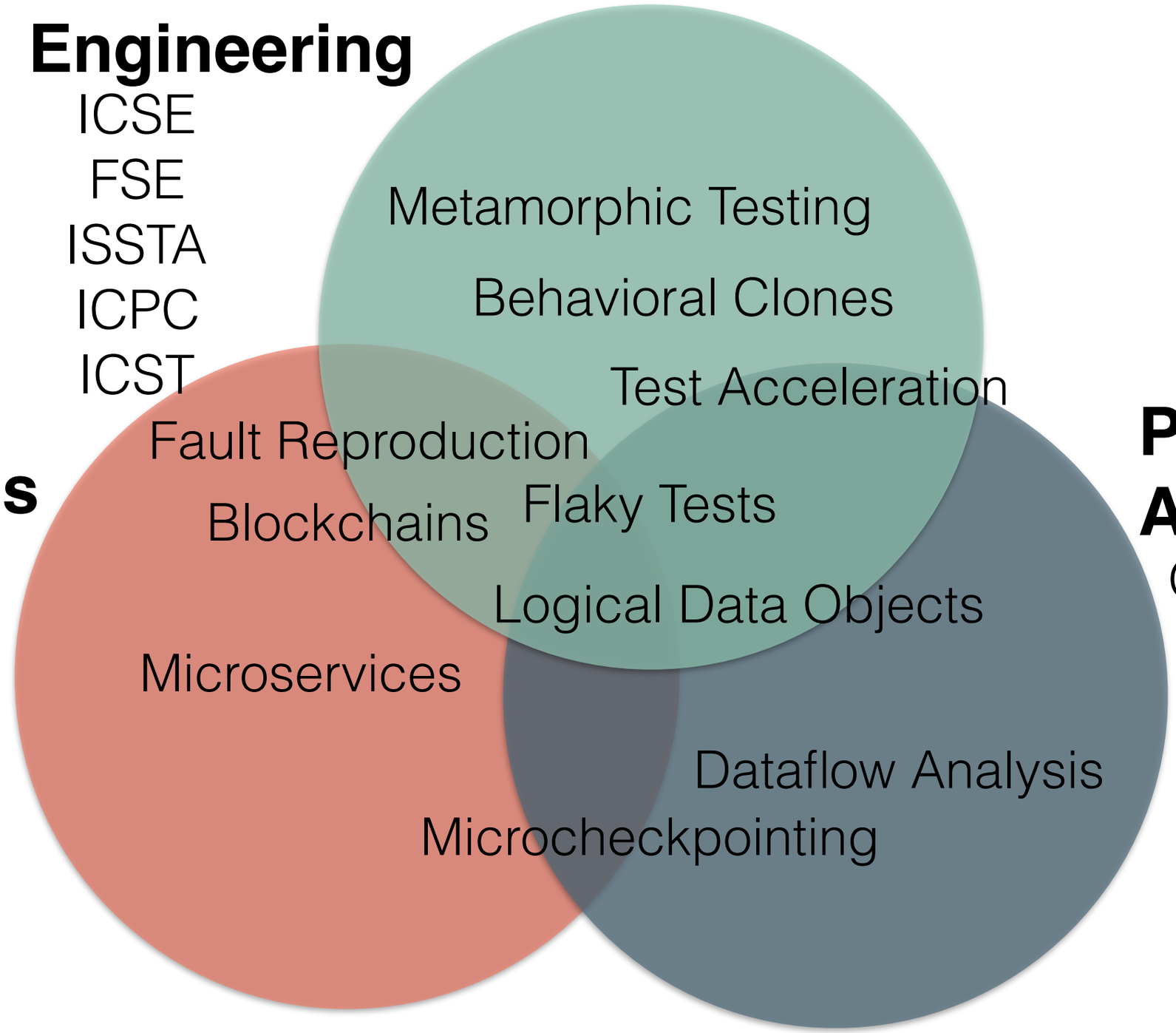
CS 475, Spring 2018
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

Broad Research Areas

Software Engineering

ICSE
FSE
ISSTA
ICPC
ICST

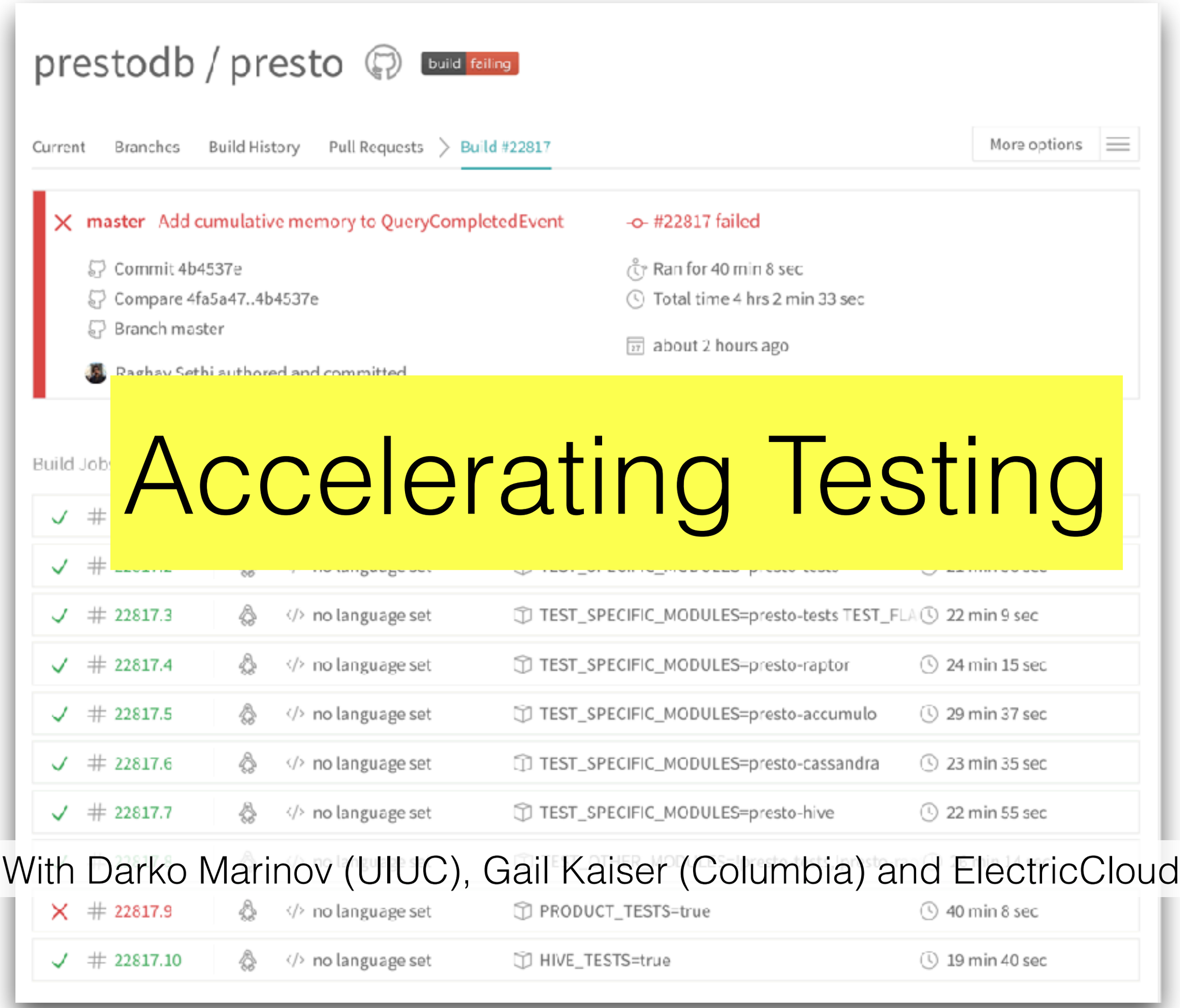


Systems

OSDI
EuroSys

Program Analysis

OOPSLA
ECOOP



Accelerating Testing

With Darko Marinov (UIUC), Gail Kaiser (Columbia) and ElectricCloud, Inc

Packages

All
org.apache.commons.io
org.apache.commons.io.comparator
org.apache.commons.io.filefilter
org.apache.commons.io.input
org.apache.commons.io.monitor
org.apache.commons.io.output
org.apache.commons.io.serialization

Coverage Report - org.apache.commons.io

Package	# Classes	Line Coverage		Branch Coverage		Complexity
org.apache.commons.io	25	90%	1954/2169	89%	1052/1175	3.027
org.apache.commons.io.comparator	10	99%	140/141	93%	45/48	2
org.apache.commons.io.filefilter	25	98%	544/555	90%	252/278	2.215
org.apache.commons.io.input	31	85%	1048/1220	88%	543/611	2.714
org.apache.commons.io.monitor	5	87%	202/231	84%	101/120	1.929
org.apache.commons.io.output	22	86%	647/748	78%	130/166	1.714
org.apache.commons.io.serialization	5	100%	56/56	100%	20/20	1.6

org.apache.commons

Classes

ByteOrderMark *(100%)*
Charsets *(94%)*
CopyUtils *(81%)*
DirectoryWalker *(98%)*
EndianUtils *(100%)*
FileCleaner *(14%)*
FileCleaningTracker *(98%)*
FileDeleteStrategy *(89%)*
FileExistsException *(33%)*
FileSystemUtils *(88%)*
FileUtils *(88%)*
FilenameUtils *(97%)*
HexDump *(94%)*
IOCase *(96%)*
IOExceptionWithCause *(50%)*
IOUtils *(89%)*
Java7Support *(57%)*
LineIterator *(97%)*
TaggedIOException *(80%)*
ThreadMonitor *(100%)*

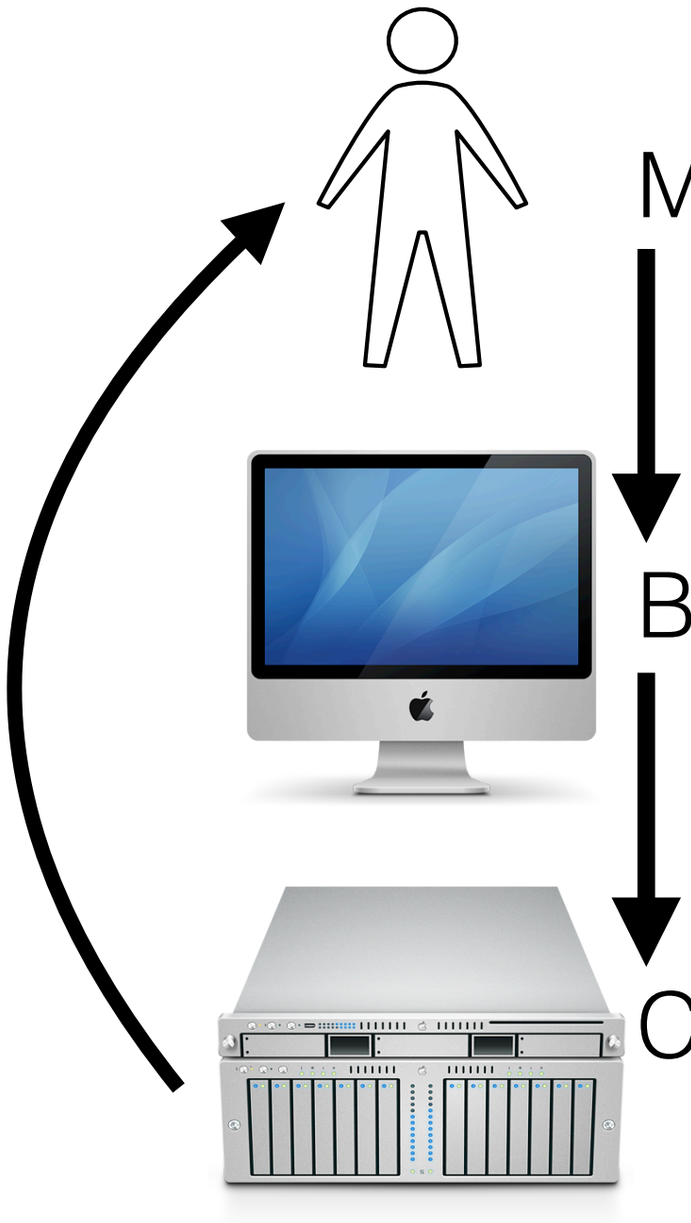
Classes in this Package	Line Coverage		Branch Coverage		Complexity
ByteOrderMark	100%	46/46	100%	24/24	3.125
Charsets	94%	16/17	100%	4/4	1.667
CopyUtils	81%	40/49	100%	4/4	1.154
DirectoryWalker					1.85
EndianUtils					1.85
FileCleaner					1.097
FileCleaningTracker					1
FileDeleteStrategy					2
FileExistsException	33%	2/6	N/A	N/A	2
FileSystemUtils	88%	119/135	90%	59/65	2
FileUtils	89%	623/699	81%	337/416	2.143
FileUtils$1	0%	0/2	N/A	N/A	2.143
FilenameUtils	97%	314/322	97%	291/300	1
HexDump	94%	35/37	100%	24/24	5.231
IOCase	96%	31/32	100%	28/28	3.898
IOExceptionWithCause	50%	2/4	N/A	N/A	3.898
IOUtils	89%	396/444	88%	197/222	6.2
Java7Support	57%	36/63	50%	1/2	4.5
LineIterator	97%	38/39	100%	16/16	2.333
TaggedIOException	80%	8/10	87%	7/8	1
ThreadMonitor	100%	26/26	83%	5/6	2.202
					5.333
					3.25
					1.8
					1.667

Report generated by Cobertura 2.1.1 on 4/21/16 8:51 PM.

Automated Security Testing

With Luís Pina (GMU), Christian Hammer (Padderborn)

Simplified Software Lifecycle



Make changes to code

Build & test

Commit

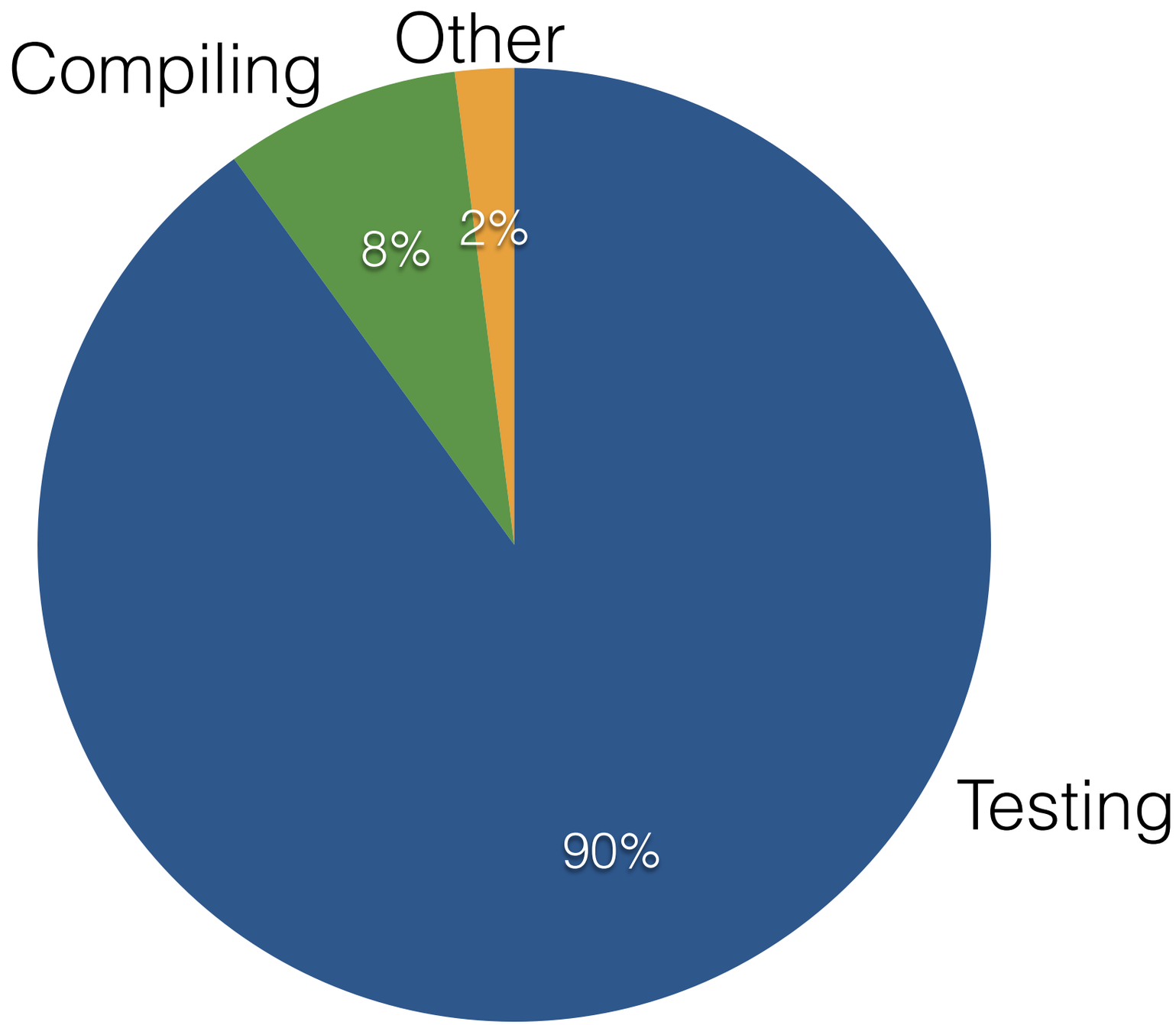
How long is too long of a build?

1 day?

6 hours?

10 minutes?

Testing Dominates Build Times



Projects taking > 1 hour to build on GitHub using Maven

JUnit Test Execution

Overhead of restarting the JVM?

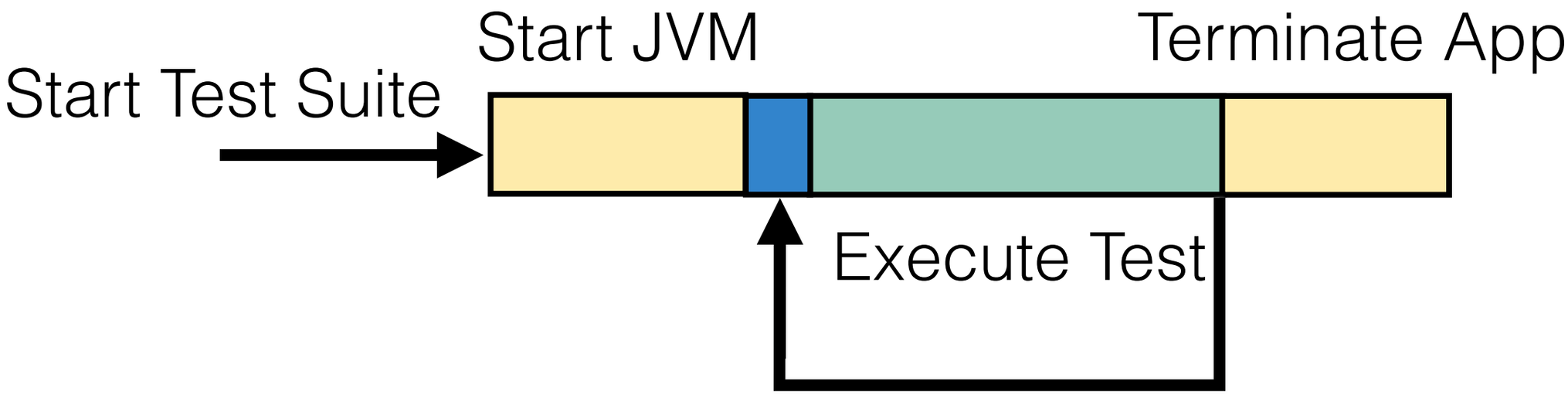
Unit tests as fast as 3-5 ms

JVM startup time is fairly constant (1.4 sec)

Up to 4,153%, avg 618%

***From our study of 20 popular FOSS apps**

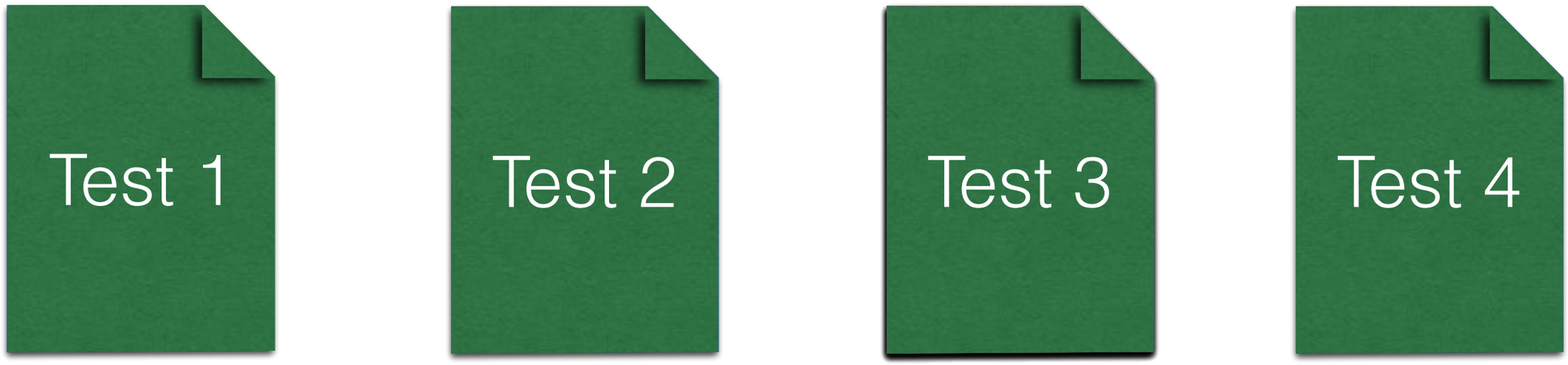
Accelerating Test Execution



Efficiently resetting state between tests reduces time by 60% on average!

“VMVM” transferred to industrial partner ElectricCloud

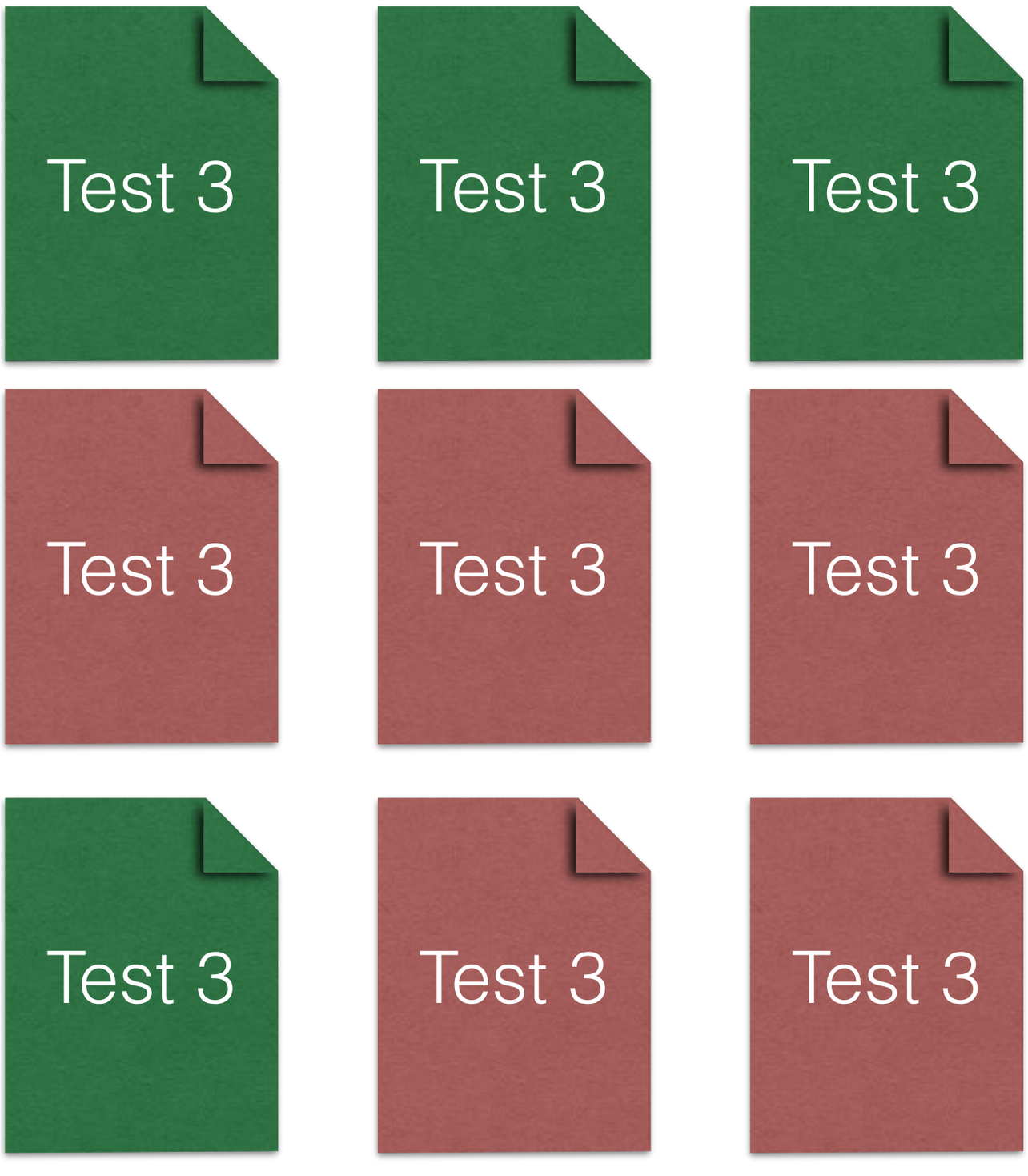
Flaky Tests



Flaky Tests

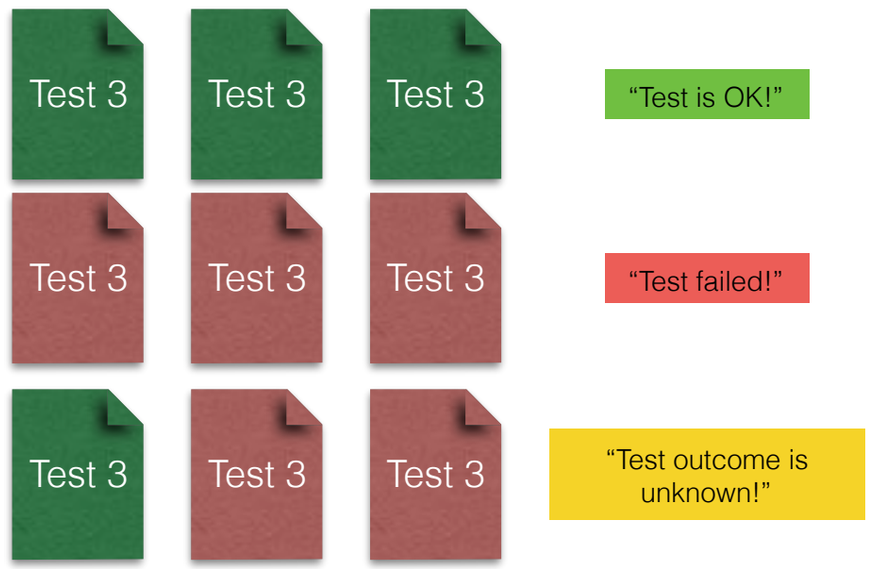
- Test might pass or fail given the SAME code
- Google: 16% of tests are “flaky” in some way
- How do you handle these flaky tests?
  - Typical fix: if you think something is flaky, run it again and again - outcome is only decided from the complete status

Flaky Tests


Test 3
Test 3
Test 3
Test 3
Test 3
Test 3
Test 3
Test 3
Test 3
“Test is OK!”
“Test failed!”
“Test outcome is unknown!”

Flaky Tests

- What happened to *accelerating* testing?
- Now tests need to be run three times!
- Can we identify with certainty that a test is a false alarm without re-running?
  - Previous approaches were slower than re-running!



DeFlaker: Automatically Detecting Flaky Tests

Jonathan Bell, Owolabi Legunsen, Michael Hilton,
Lamyaa Eloussi, Tifany Yung and Darko Marinov
George Mason University, University of Illinois at Urbana-Champaign
and Carnegie Mellon University

[To appear at ICSE 2018 in Gothenburg, May 31, 2018]



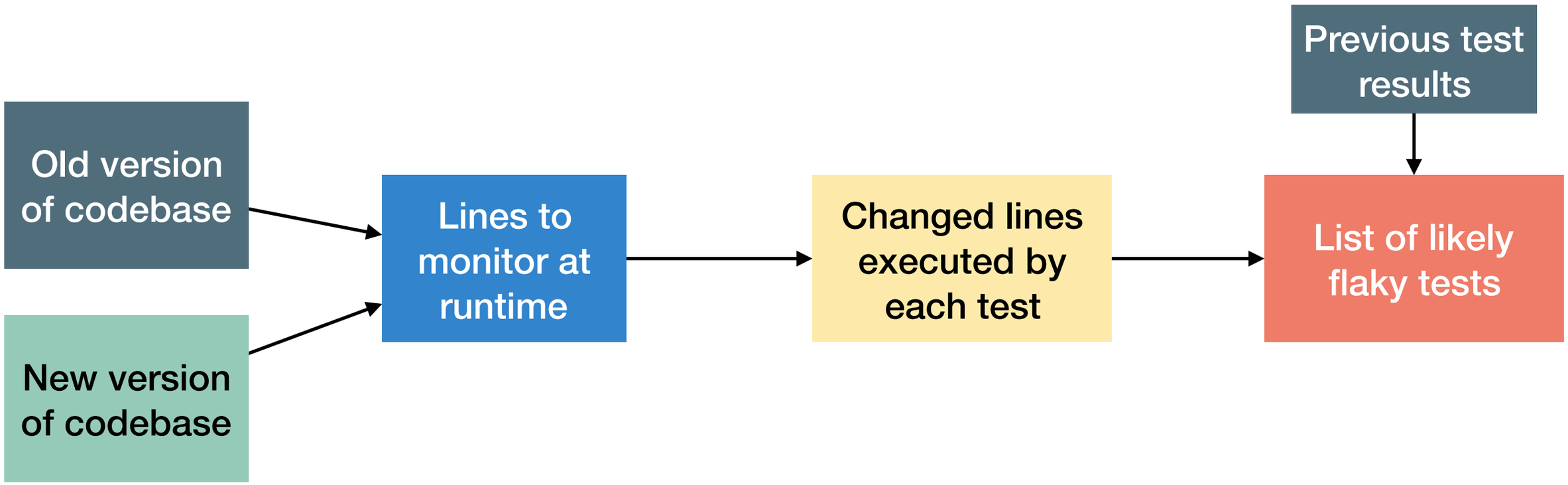


Flaky Tests

- Our key insight: there is lightweight information we can track while a test runs
- “Did this test run any code that changed?”
- Tracking coverage can be slow though! (40-50% overhead!)
- …and we want to make things faster

DeFlaker's Differential Coverage

DeFlaker tracks *differential coverage* — only tracking code that changed since the last execution of the tool



Differential Coverage

Just **syntactic** diff (e.g. from git) is insufficient to notice coverage of all kinds of changes!

```
public class SuperOld {
   public void magic() {
   }
}

public class SuperNew extends SuperOld {
   public void magic() {
      assert(false); // causes test to fail
   }
}

public class App extends ~~SuperOld~~ SuperNew {
}
public class TestApp {
   @Test public void testApp() {
      new App().magic();
   }
}
```

Now calls SuperNew.magic!

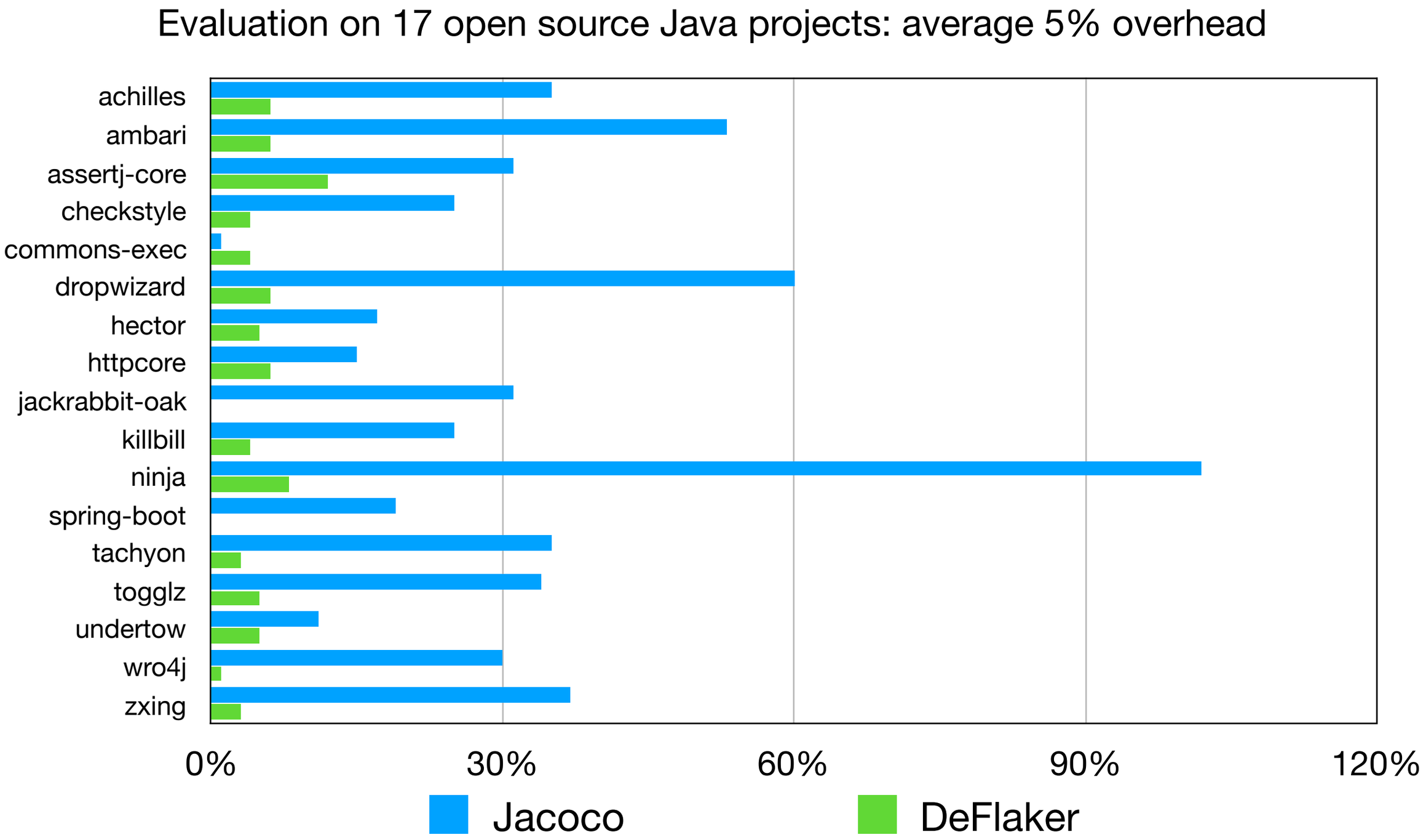
DeFlaker

- Tracks line coverage of all changed statements (in both tests and SUT)
- Identifies non-statement changes in classes by parsing them, tracks with class-level coverage
- Detects flaky test failures “just-in-time” when they fail
- Implemented as a maven extension (3-line addition to pom.xml)

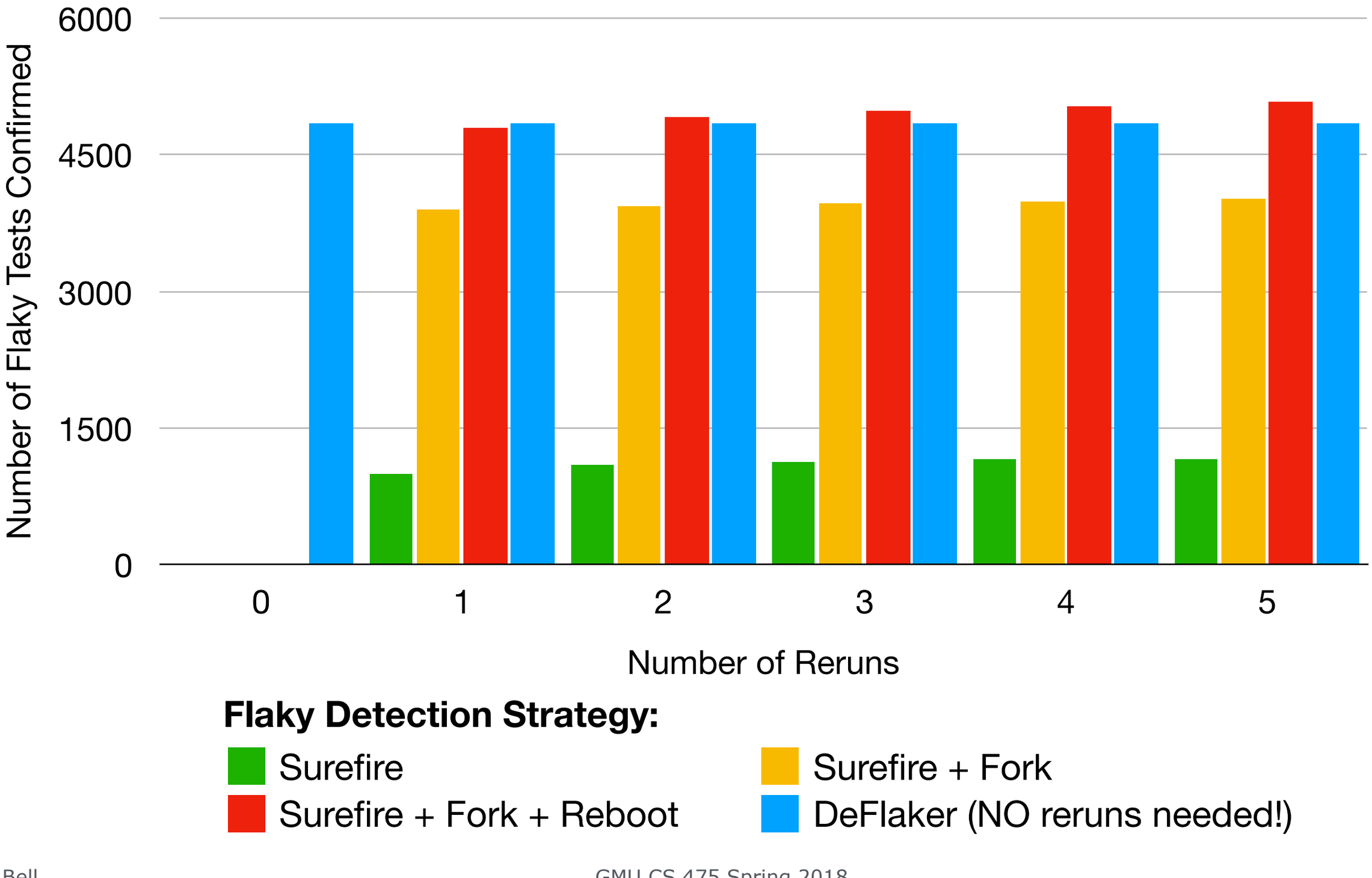
Evaluation

- What is the performance overhead of running DeFlaker?
- How many flaky tests does DeFlaker find in comparison to rerunning failed tests?

DeFlaker is Fast



DeFlaker Finds Flaky Tests



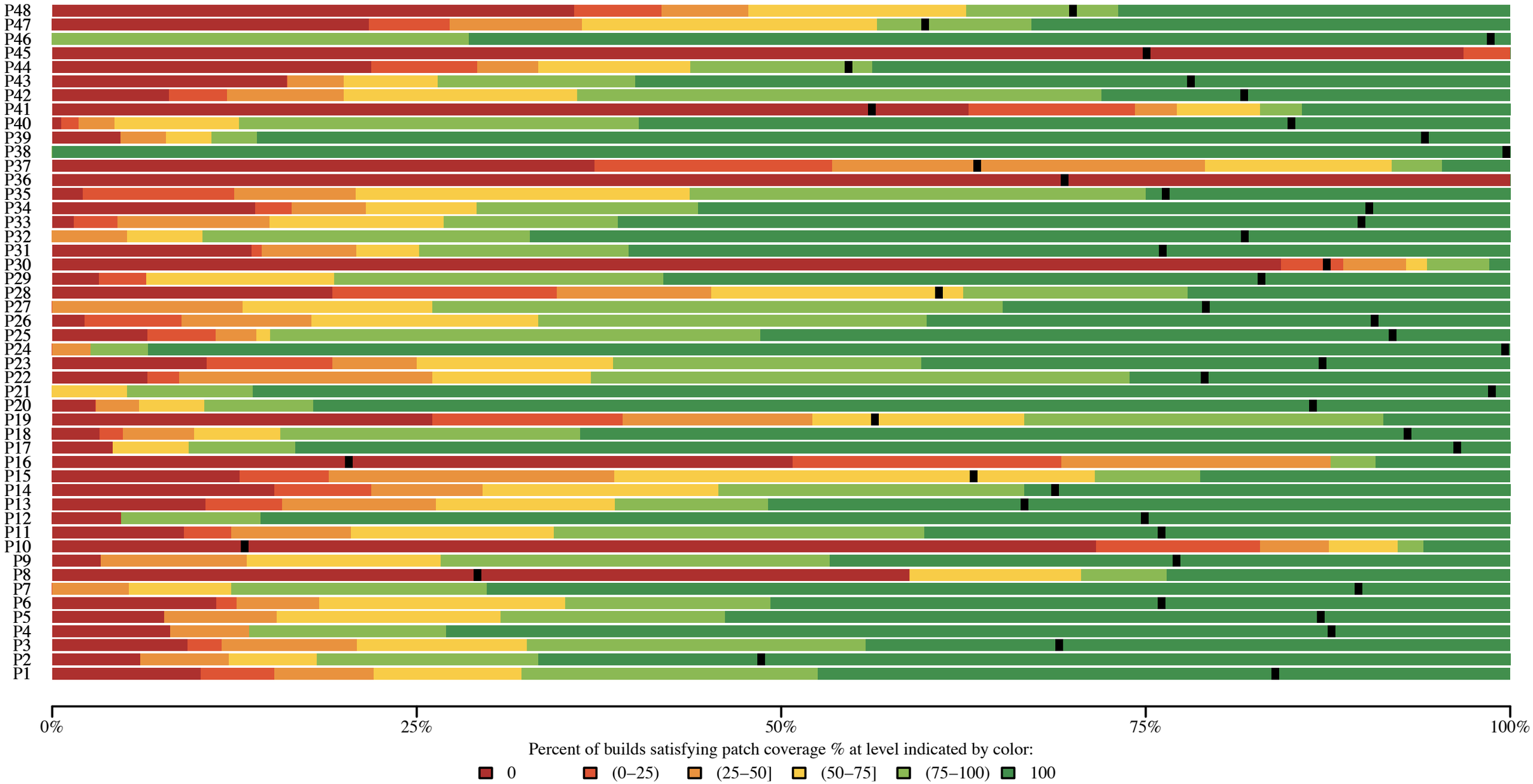
DeFlaker Findings

- HOW you re-run flaky tests matters much more than how many times you rerun them
- DeFlaker is extremely low overhead and can immediately identify flaky tests
- Also deployed shadowing live executions on TravisCI, found 87 new flaky tests and reported to developers, many now fixed
- Differential coverage may have many other useful applications as well
- Try it out! http://deflaker.org/

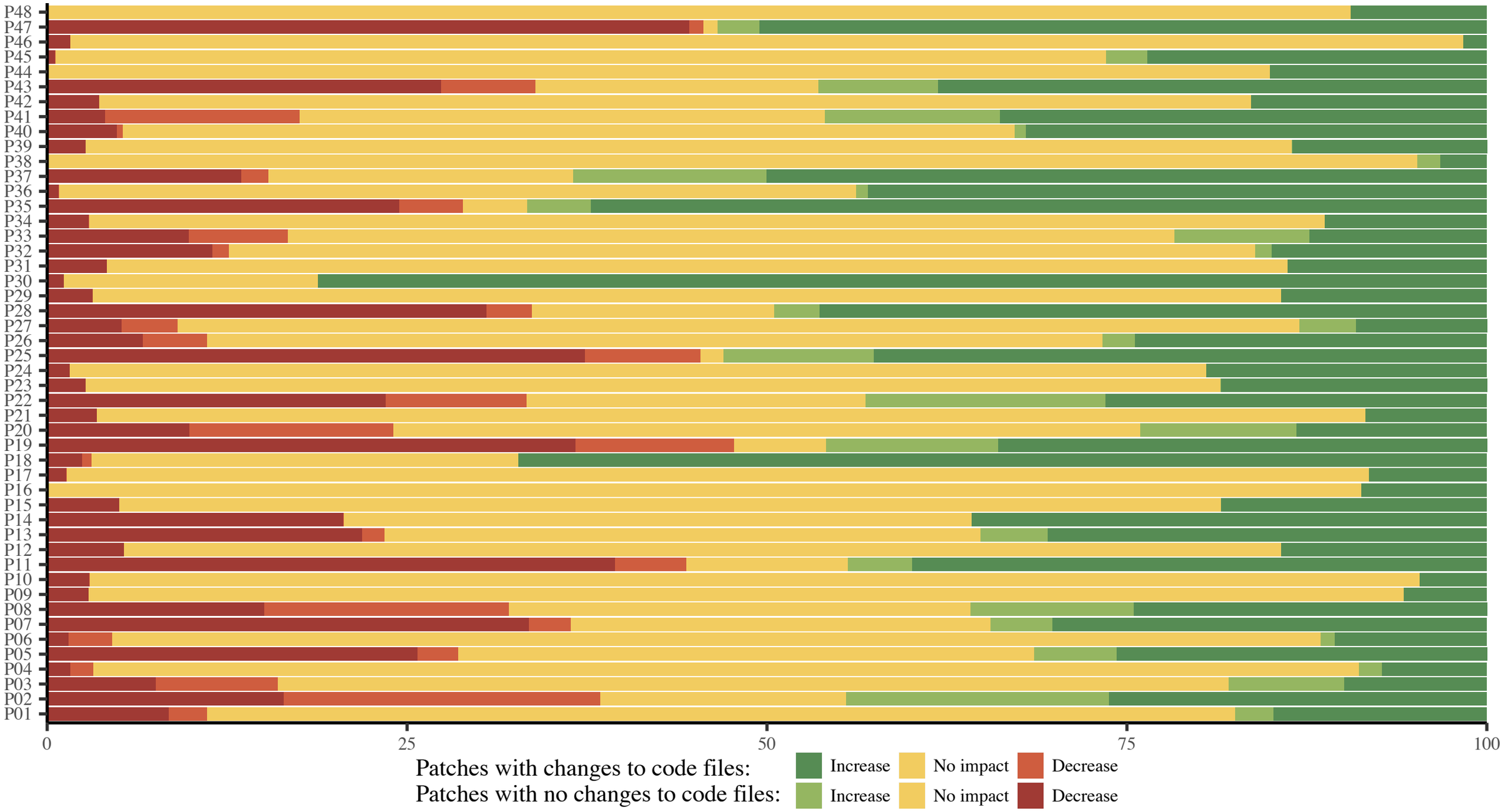
Measuring Code Coverage

	Project	Lang	Source	Builds	LoC	Time range (months)	Coverage		
							Start	Sparkline	End
P01	apache/commons-collections	java	[25]	189	12,765	40	84%		84%
P02	apache/commons-dbcp	java	[25]	164	5,662	19	48%		51%
P03	apache/commons-exec	java	[5]	212	971	65	67%		73%
P04	apache/commons-functor	java	[25]	248	2,693	69	83%		83%
P05	apache/commons-io	java	[25]	35	5,021	2	87%		88%
P06	apache/commons-jxpath	java	[25]	199	9,633	94	75%		77%
P07	apache/commons-math	java	[25]	217	45,034	16	90%		90%
P08	apache/commons-net	java	[25]	53	9,210	1	30%		29%
P09	apache/commons-validator	java	[25]	104	2,854	10	77%		77%
P10	apache/empire-db	java	[25]	241	21,258	60	14%		13%
P11	apache/httpcore	java	[5]	223	13,198	18	77%		77%
P12	ARMmbed/mbed-ls	python	C,IO	60	804	6	75%		75%
P13	bitwalker/timex	elixer	C,IO	128	2,615	16	69%		68%
P14	broadinstitute/firecloud-orchestration	Scala	C,IO	170	2,658	13	67%		65%
P15	containers/virtcontainers	go	C,IO	296	5,332	9	66%		63%
P16	coreos/alb-ingress-controller	go	C,IO	98	2,041	7	21%		22%
P17	damianszczepanik/cucumber-reporting	java	[17]	248	794	15	99%		99%
P18	dask/dask	python	C,IO	290	15,322	7	94%		94%
P19	doanduyhai/Achilles	java	[5]	111	11,008	9	38%		57%
P20	dropwizard/dropwizard	java	[5], [18]	246	7,700	9	87%		87%
P21	eBay/cors-filter	java	[17]	204	280	45	99%		100%
P22	F5Networks/k8s-bigip-ctlr	go	C,IO	103	5,621	5	76%		83%
P23	fasseg/exp4j	java	[17]	233	640	40	95%		95%
P24	Gillespie59/eslint-plugin-angular	node	C,IO	212	1,213	19	100%		100%
P25	goldmansachs/gs-collections	java	[25]	249	38,242	16	93%		93%
P26	google/jimfs	java	[5]	100	3,401	45	92%		88%
P27	HazyResearch/deepdive	Scala	C,IO	111	1,913	5	82%		82%
P28	hector-client/hector	java	[5]	137	8,583	25	64%		35%
P29	ikawaha/kagome	go	C,IO	91	1,099	27	87%		87%
P30	ilovepi/Compiler	dotNet	C,IO	96	1,874	1	86%		90%
P31	jhy/jsoup	java	[17]	246	6,615	28	76%		76%
P32	jknack/handlebars.java	java	[5]	100	3,935	9	84%		84%
P33	JodaOrg/joda-time	java	[18], [25]	248	14,789	35	90%		90%
P34	joel-costigliola/assertj-core	java	[5], [18]	241	11,104	9	91%		90%
P35	mailgun/kafka-pixy	go	C,IO	64	4,387	13	69%		80%
P36	ManageIQ/ui-components	node	C,IO	121	1,734	7	71%		70%
P37	MITLibraries/topichub	Scala	C,IO	102	5,644	11	65%		63%
P38	platinumazure/eslint-plugin-qunit	node	C,IO	62	628	22	100%		100%
P39	PragTob/benchee	elixer	C,IO	148	441	6	95%		94%
P40	raml-org/raml-java-parser	java	[17]	248	6,455	16	86%		86%
P41	ShiftForward/apso	Scala	C,IO	92	1,629	9	59%		56%
P42	spatialmodel/inmap	go	C,IO	55	5,983	9	83%		81%
P43	square/okhttp	java	[5]	247	11,854	10	78%		79%
P44	square/retrofit	java	[25]	181	2,479	17	54%		57%
P45	SteamDatabase/ValveResourceFormat	dotNet	C,IO	179	2,794	23	73%		80%
P46	terasolunaorg/terasoluna-gfw	java	C,IO	97	2,561	17	99%		99%
P47	undertow-io/undertow	java	[5]	238	51,388	9	60%		60%
P48	zxing/zxing	java	[5]	198	15,440	21	68%		74%
Total 48 projects, 389,325 LOC			Average	165	8,110	20	75%		76%

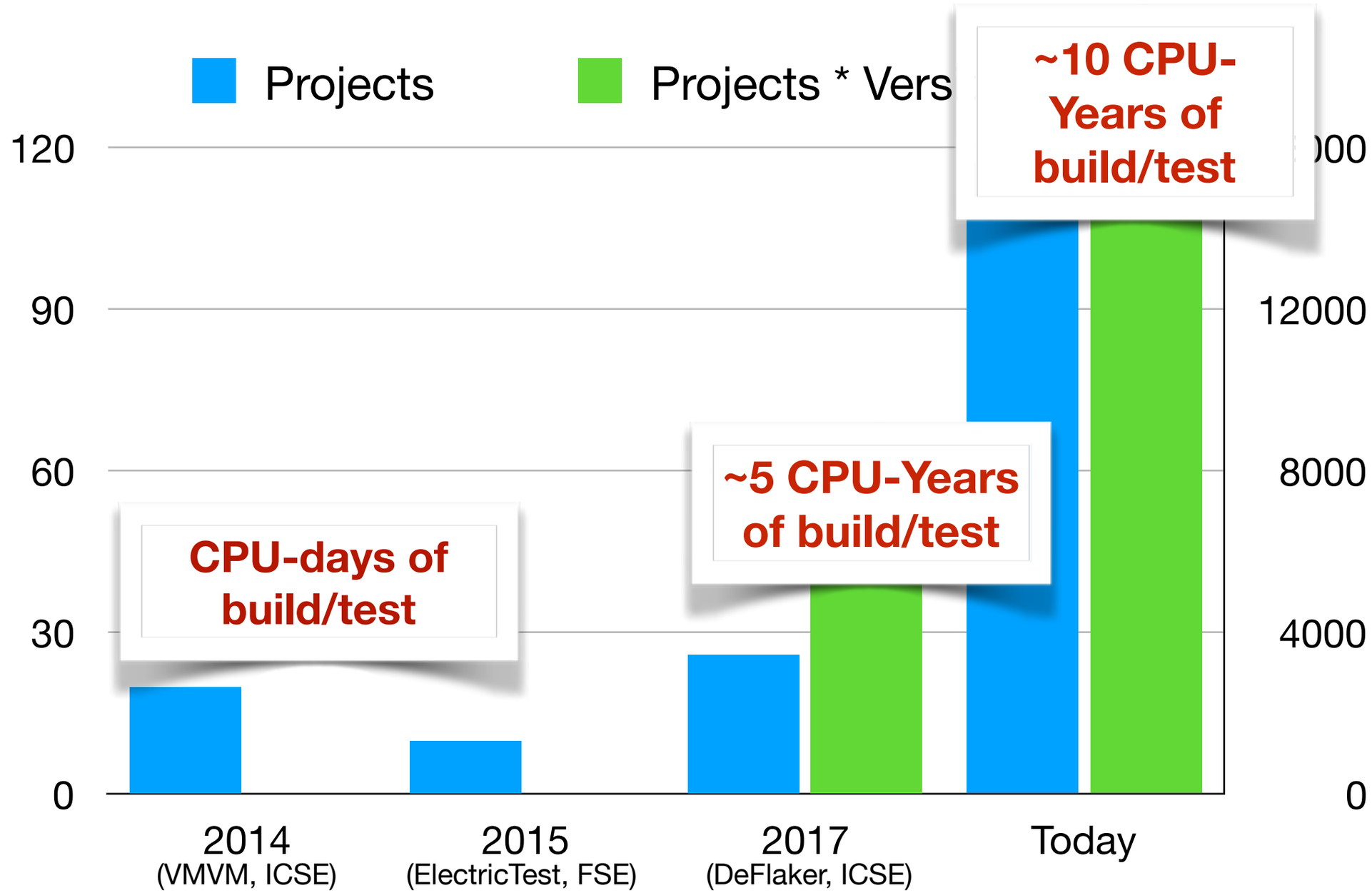
Coverage of Changes



Changes of Coverage



Testing Infrastructure



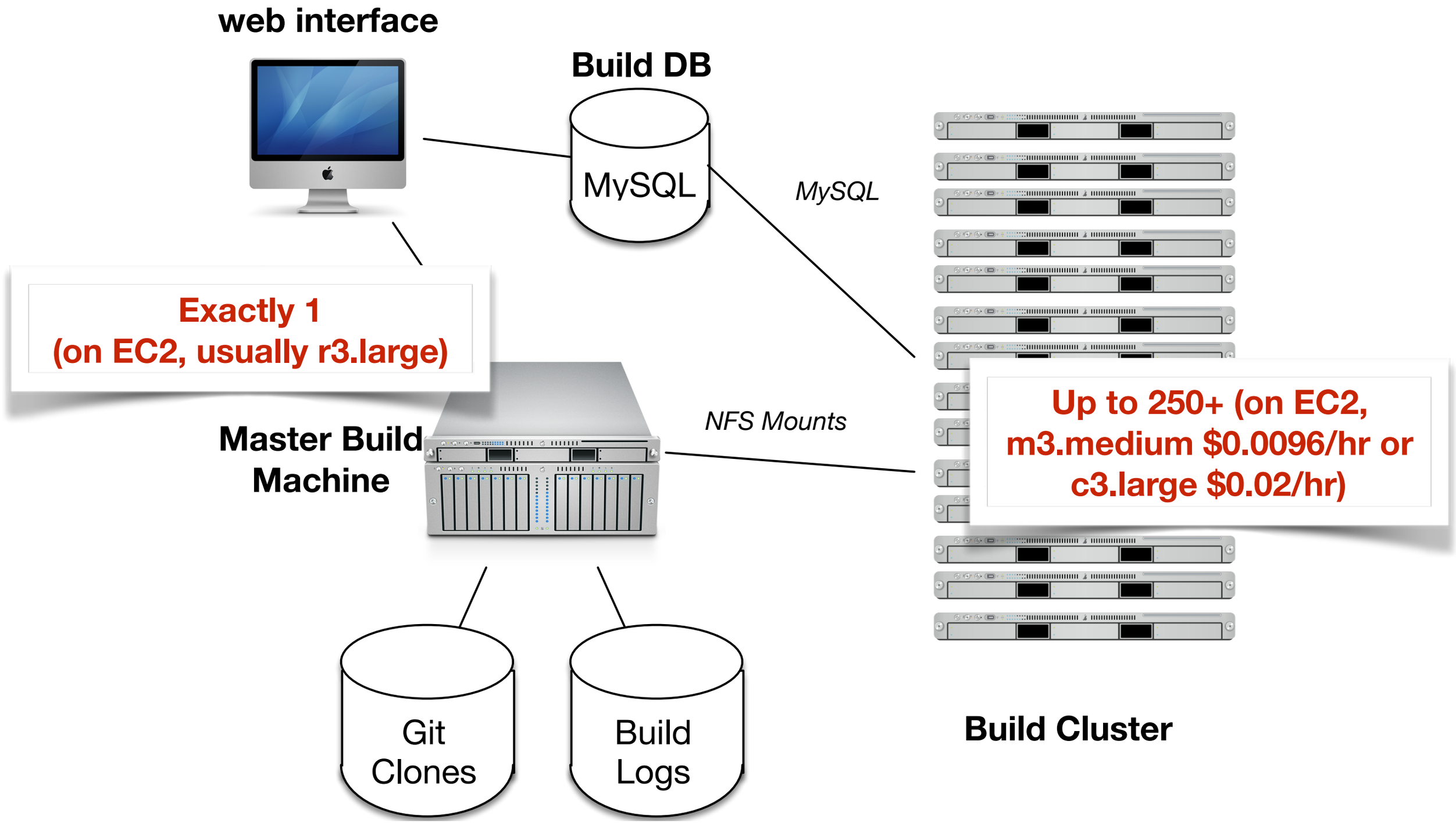
Testing Infrastructure

- How do we do 15,000 build+test cycles quickly and cheaply?
- An embarrassingly parallel problem
- Each commit of each project can be built with no dependency on any other build
- Hence, can trivially scale to do as many at once as makes sense…

Backend Infrastructure

- Master/slave architecture
- Failure assumptions: master can not fail, slaves may fail by crashing
- In practice, we've seen worker nodes fail by crashing from underlying hardware problems
- Slave nodes boot up, bring up a shared NFS mount, run an "on boot" script that is provided by the master to determine how to run builds…

Testing Infrastructure



Example Interface: Flaky Tests

Build History

View grouped by builds | Export to CSV

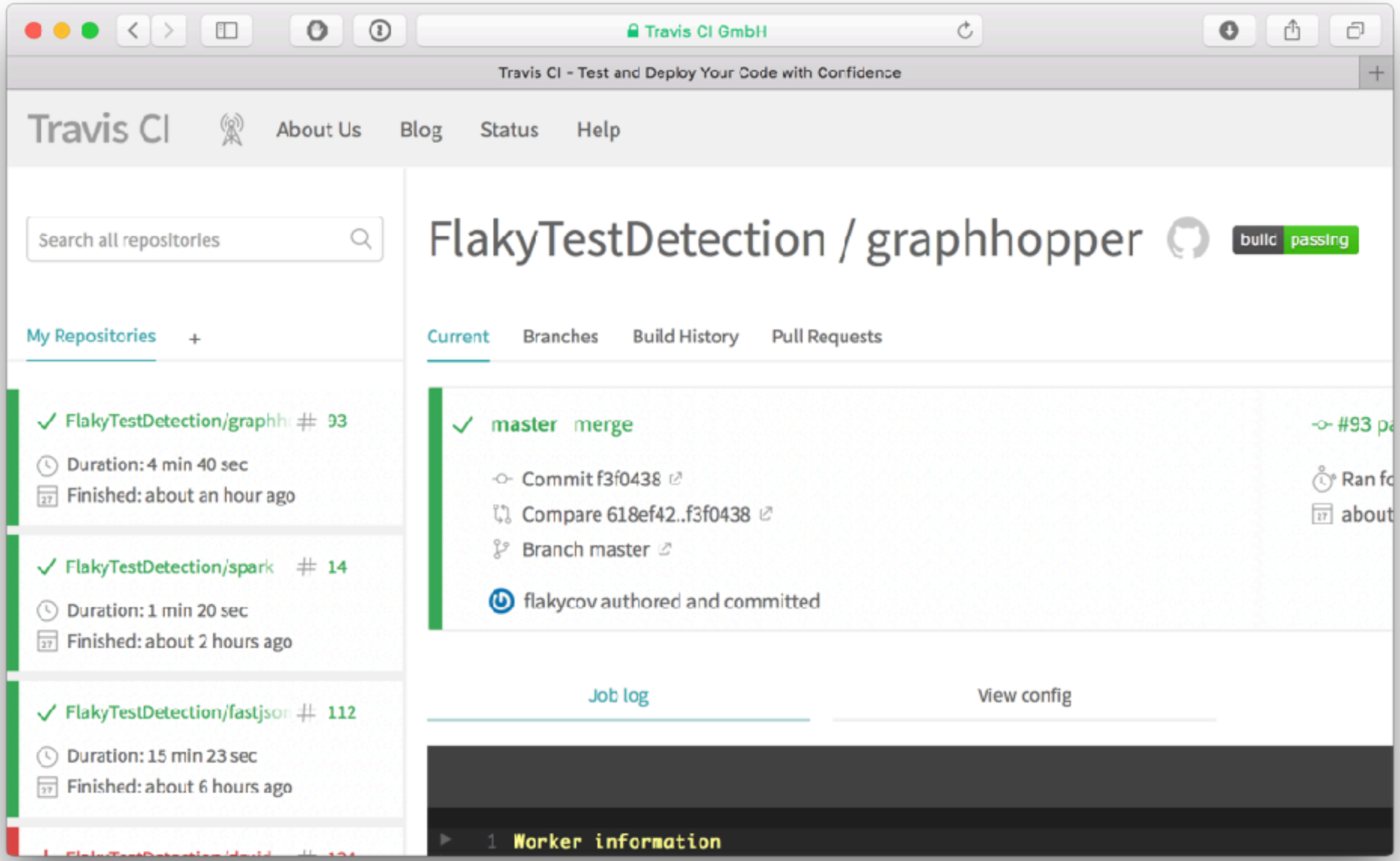
Selected Run: 88 - First pass, all new projects

Can select an experiment to visualize/export

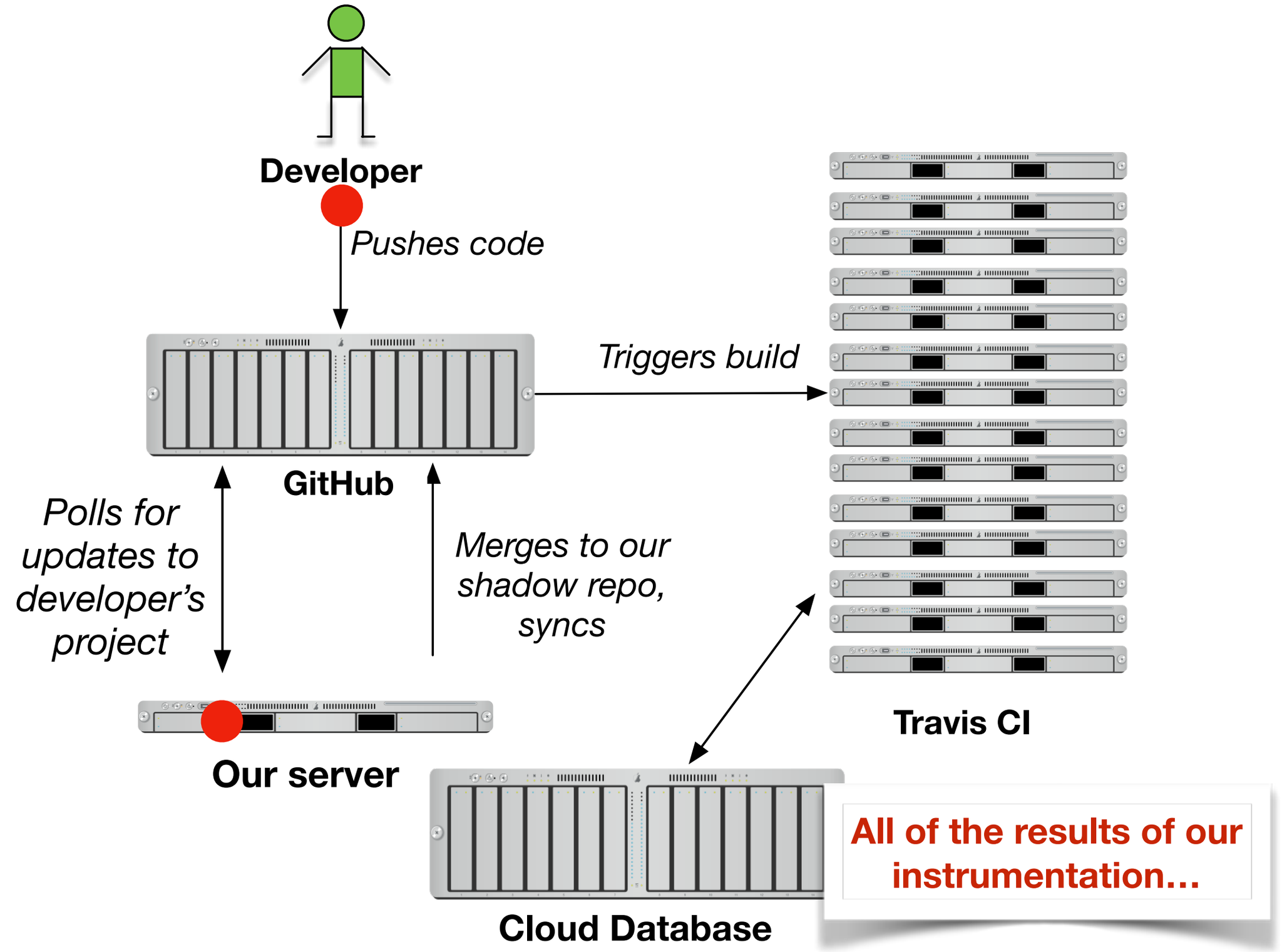
View results of re-running same build several times

Project	SHA	Log File	Test Classes	Test Methods	Test Method Failures
− Activiti	1 total	10 total	298	1828	0.1
	− 8e124654ba44c4a4cc1ee53fcb2b82cfa1e9fec6	10 total	298	1828	0.1
		#182379	298	1828	0
		#182483	298	1828	1
		#182491	298	1828	0
		#182703	298	1828	0
		#182794	298	1828	0
		#182828	298	1828	0
		#183124	298	1828	0
		#183169	298	1828	0
		#183236	298	1828	0
		#183261	298	1828	0
+ stream-lib	1 total	10 total	23	139	0
+ jitwatch	1 total	10 total	29	236	0
+ jstorm	1 total	10 total	0	0	0
+ tachyon	1 total	10 total	264.5	1537.5	1
+ ant	1 total	10 total	0	0	0
+ cassandra	1 total	10 total	0	0	0
+ derby	1 total	10 total	0	0	0
+ hbase	1 total	10 total	0	0	0
+ hive	1 total	10 total	0	0	0

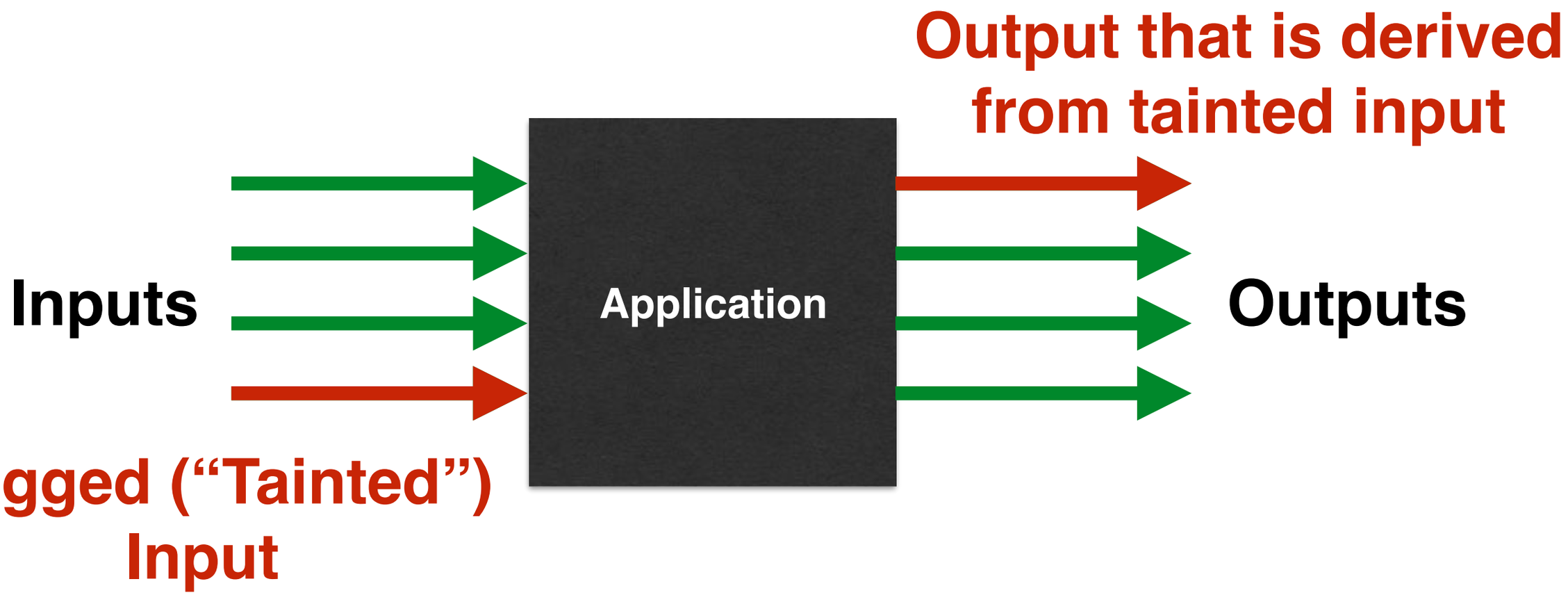
“Live” Infrastructure



"Live" Infrastructure

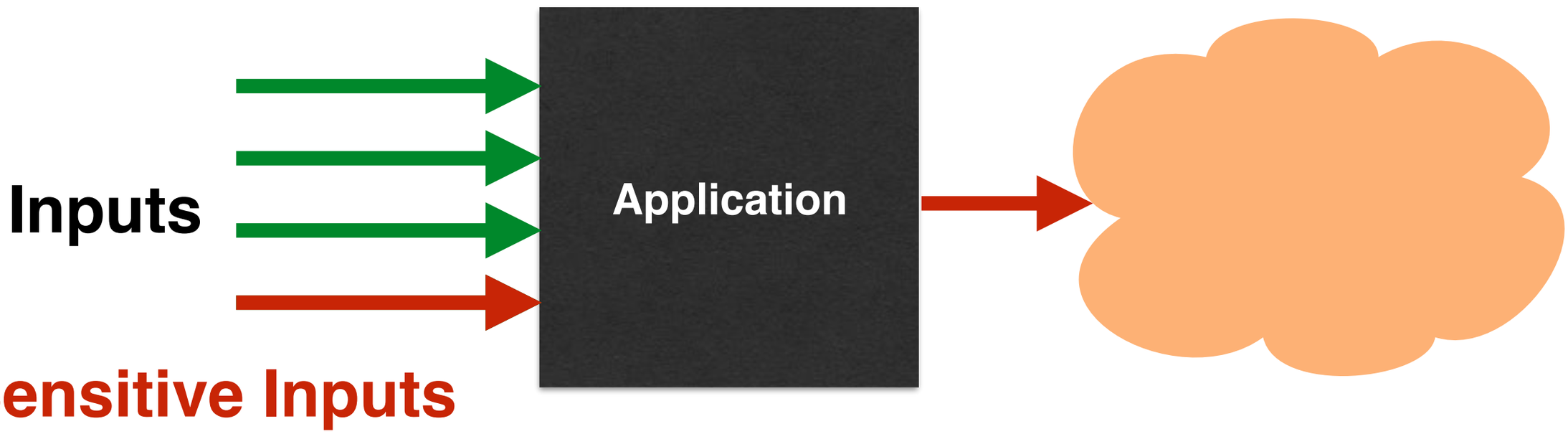


Dynamic Data Flow Analysis: Taint Tracking



Taint Tracking: Applications

End-user privacy testing: Does this application send my personal data to remote servers?



Taint Tracking: Applications

Testing: Are my test cases overly specified?

```
@Test
public void testEnrolled() throws Exception {
   Student s = new Student();
   s.id = 5;
   s.name = "Bob";
   s.setEnrolled();
   assertTrue(s.isEnrolled);
}
```

No assertion depends on these values

Problem: No Tool Support

“Normal” Taint Tracking

- Associate tags with data, then propagate the tags
- Approaches:
  - Operating System modifications [Vandebogart ’07], [Zeldovich ’06]
  - Language interpreter modifications [Chandra ’07], [Enck ’10], [Nair ’07], [Son ’13]
  - Source code modifications [Lam ‘06], [Xu ’06]
  - Binary instrumentation of applications [Clause ’07], [Cheng ’06], [Kemerlis ’12]

Not portable

Hard to be sound, precise, and performant

Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs

Jonathan Bell and Gail Kaiser
Columbia University

[OOPSLA 2014; Artifact Evaluated & Approved]

Phosphor

- Leverages benefits of interpreter-based approaches (information about variables) but *fully portably*
- Instruments *all* byte code that runs in the JVM (including the JRE API) to track taint tags
  - Add a variable for each variable
  - Adds propagation logic

Key contribution:

How do we efficiently store meta-data for **every** variable **without** modifying the JVM itself?

Phosphor: Instrumentation Strategy

```
double pie = 3.14;
double more = 1;
double more_pie = pie + more;
int ret = callSomeMethod(pie);
```

```
double pie = 3.14;
int pie_tag = 0;
double more = 1;
int more_tag = 0;
double more_pie = pie + more;
int more_pie_tag = pie_tag | more_tag;
TaintedInt tmp = callSomeMethod(pie_tag, pie);
int ret = tmp.val;
int ret_tag = tmp.tag;
```

(Of course, we do this all at byte code, not source code)

```
jon@lrrr:~$ instrumented-jre/bin/java test.Test
Segmentation Fault
```

It’s not so easy.

(Yes: You can seg-fault the JVM trivially)

JVM Type Organization

- Primitive Types
  - int, long, char, byte, etc.
- Reference Types
  - Arrays, instances of classes
  - All reference types are assignable to java.lang.Object

Phosphor's Taint Tag Storage

	Local variable	Method argument	Return value	Operand stack	Field
Object	Stored as a field of the object				
Object array	Stored as a field of each object				
Primitive	Shadow variable	Shadow argument	"Boxed"	Below the value on stack	Shadow field
Primitive array	Shadow array variable	Shadow array argument	"Boxed"	Array below value on stack	Shadow array field

Taint Propagation

- Modify all byte code instructions to be taint-aware by adding extra instructions
- Examples:
  - Arithmetic -> combine tags of inputs
  - Load variable to stack -> Also load taint tag to stack
  - Modify method calls to pass taint tags

Complications

- Primitives on the stack
- Non-modifiable classes (e.g. Object, StackTraceElement, Byte, some others)
- Arrays of primitive values
  - And multi-dimensional arrays, and upcasting arrays
- Native methods

Phosphor Doubles Stack Size

Code Snippet

```
void foo(int i, int j)
{
 int k = i + j;
}
```

Bytecode

```
ILOAD 1
ILOAD 2
IADD
ISTORE 3
```

Code Snippet (instrumented)

```
void foo(int i_tag, int i,
   int j_tag, int j)
{
 int k = i + j;
 int k_tag = i_tag | j_tag;
}
```

Instrumented Bytecode

```
ILOAD 1
ILOAD 2
ILOAD 3
ILOAD 4
DUP2_X1
POP2
IADD
SWAP
IOR
ISTORE 6
ISTORE 5
```

Methods can get long, but avoids need for expensive shadow stack

Non-Modifiable Classes (and Arrays)

- Candidate approach:
  - Use a HashMap, with each untrackable object as key
  - Very, very slow (need to access a globally-locked HashMap for EVERY operation you do involving an object)
- Phosphor:
  - Special case everything
  - Primitive arrays: get their own shadow array, tracked through upcasting
  - Non-modifiable classes: HashMap

Challenge 1: Upcasting

Solution 1: Box taint tag with array when upcasting

```
byte[] array = new byte[5];
Object ret = array;
```

```
int[] array_tag = new int[5];
byte[] array = new byte[5];

Object ret = new TaintedByteArray(array_tag,array);
```

```
byte[] foo = (byte[]) ret;
```

```
int[] foo_tag = ((TaintedByteArray) ret).tag;
byte[] foo_tag = ((TaintedByteArray) ret).val;
```

Challenge 1: Upcasting

Also needed for multi-dimension primitive arrays

```
byte[] array = new byte[5];
byte[][] ret = new byte[][]{array};
```

```
int[] array_tag = new int[5];
byte[] array = new byte[5];

TaintedByteArray[] ret = new TaintedByteArray[1];
ret[0] = new TaintedByteArray(array_tag,array);
```

```
byte[] foo = ret[0];
```

```
int[] foo_tag = ret[0].tag;
byte[] foo_tag =ret[0].val;
```

Challenge: Native Code

We can’t instrument everything!

Native Code

```
public int hashCode() {
  return super.hashCode() * field.hashCode();
}
```

```
public native int hashCode();
```

What caller expects → `public TaintedInt hashCode();`

Solution: Wrappers. Rename *every* method, and leave a wrapper behind. Always call wrapper version.

```
public TaintedInt hashCode$$wrapper() {
    return new TaintedInt(0, hashCode());
  }
```

Native Code

Wrappers work both ways: native code can still call a method with the old signature

```
public int[] someMethod(byte in)
        {
          return someMethod$$wrapper(0, in).val;
        }


public TaintedIntArray someMethod$$wrapper(int in_tag, byte in)
{
  //The original method "someMethod", but with taint tracking
}
```

Native Code

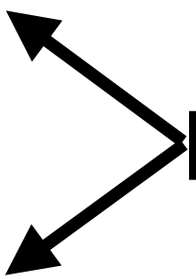
- The main design limitation
- Return value's tag becomes combination of all parameters (heuristic); not found to be a problem in our evaluation
- Note: reflection is NOT a limitation, is easiest of all challenges to work around (runtime wrappers)

Configuration Options

- Tag propagation modes:
  - Data flow `int c = a + b;`
  - Control flow `if (a == 0) c = 0;`
- Tag format:
  - Integer (bit vectors)
  - Object (maintain relationships sets)
- Automatic Tagging and Checking

Implicit vs. Explicit Data Flow

```
String secretStr ="secret";
String leakedStr = "";
switch(secretStr.charAt(i))
{
case 'a':
   leakedStr += 'a';
   break;
case 'b':
   leakedStr += 'b';
   ...
}
```



No “explicit” data flow!

Phosphor: API

Getting and setting tags on objects

```
Interface TaintedWithObjTag.class
public Taint getPHOSPHOR_TAG();
public void setPHOSPHOR_TAG(Object o);
```

Getting and setting tags on primitives

```
MultiTainter.class
public static Taint getTaint(<Primitive Type> c);
public static float tainted<Primitive Type>(<Primitive Type> f, Object tag);
```

Getting relationships between tags

```
Class Taint.class
public LinkedList<Taint> getDependencies();
public Object getLabel();
```

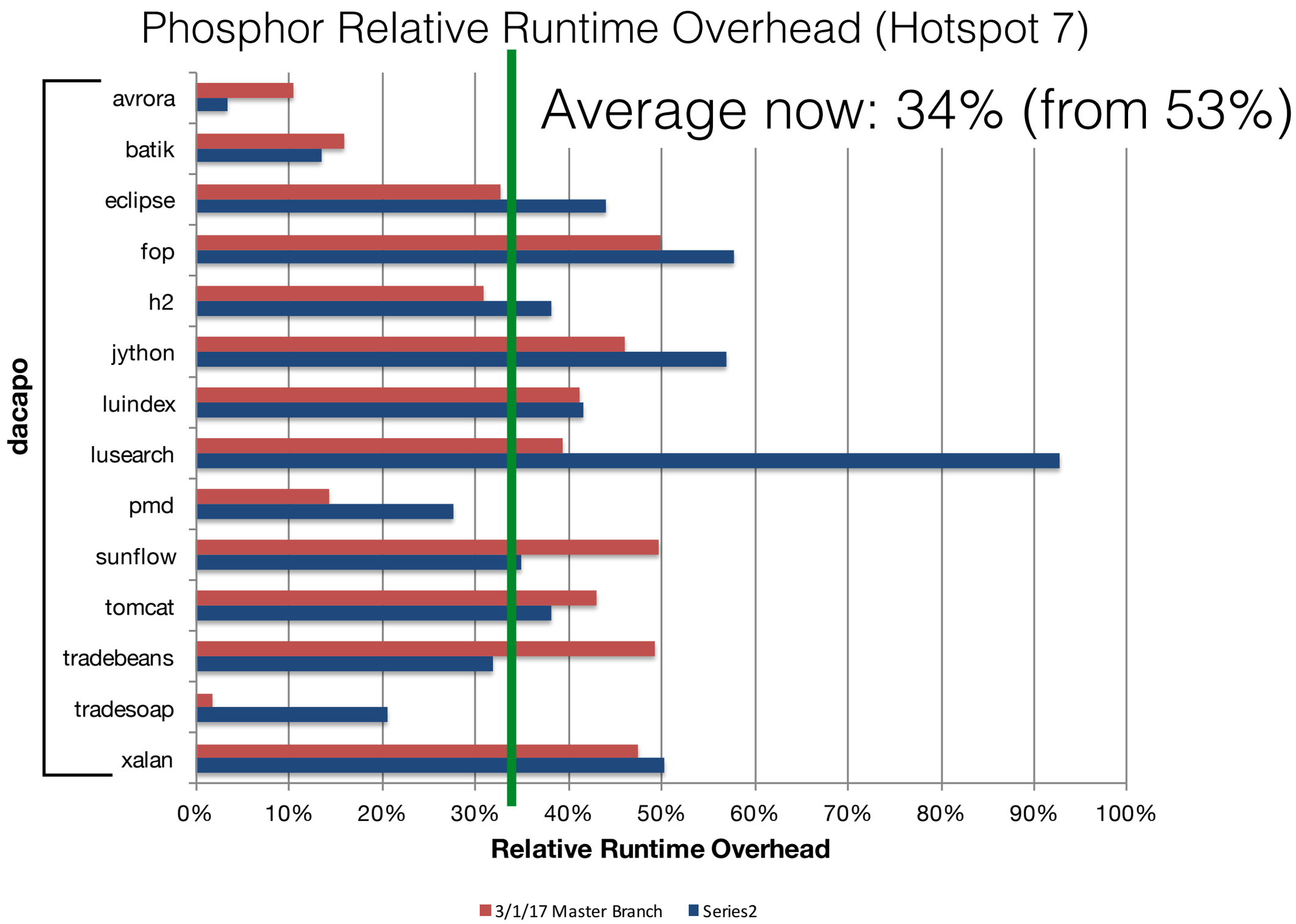
Evaluation

- Soundness & Precision
- Performance
- Portability

Soundness & Precision

- DroidBench - series of unit tests for Java taint tracking
  - Passed all except for implicit flows (intended behavior)

Macrobenchmarks



Portability

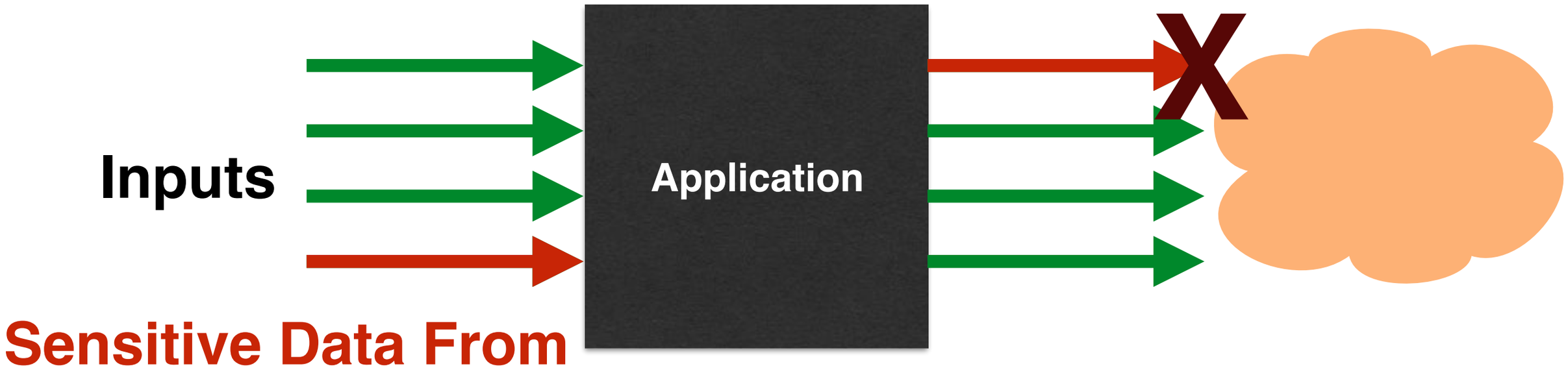
JVM	Version(s)	Success?
Oracle (Hotspot)	1.7.0_45, 1.8.0_0	**Yes**
OpenJDK	1.7.0_45, 1.8.0_0	**Yes**
Android Dalvik	4.3.1	**Yes**
Apache Harmony	6.0M3	**Yes**
Kaffe VM	1.1.9	**Yes**
Jikes RVM	3.1.3	No, but may be possible with more work

Phosphor

- MIT License on GitHub
- Already adopted as a tool by 3+ independent research groups across the world (UW, UCLA, U Lisbon)
- Actively maintained beyond OOPSLA publication
- Array handling as presented now is much simpler than in 2014
- Tags can be anything (not just ints)
- Preliminary support for implicit flows

Data Flow Analysis for Data Security Testing

Data Flow Analysis for Data Security



Sensitive Data From Device

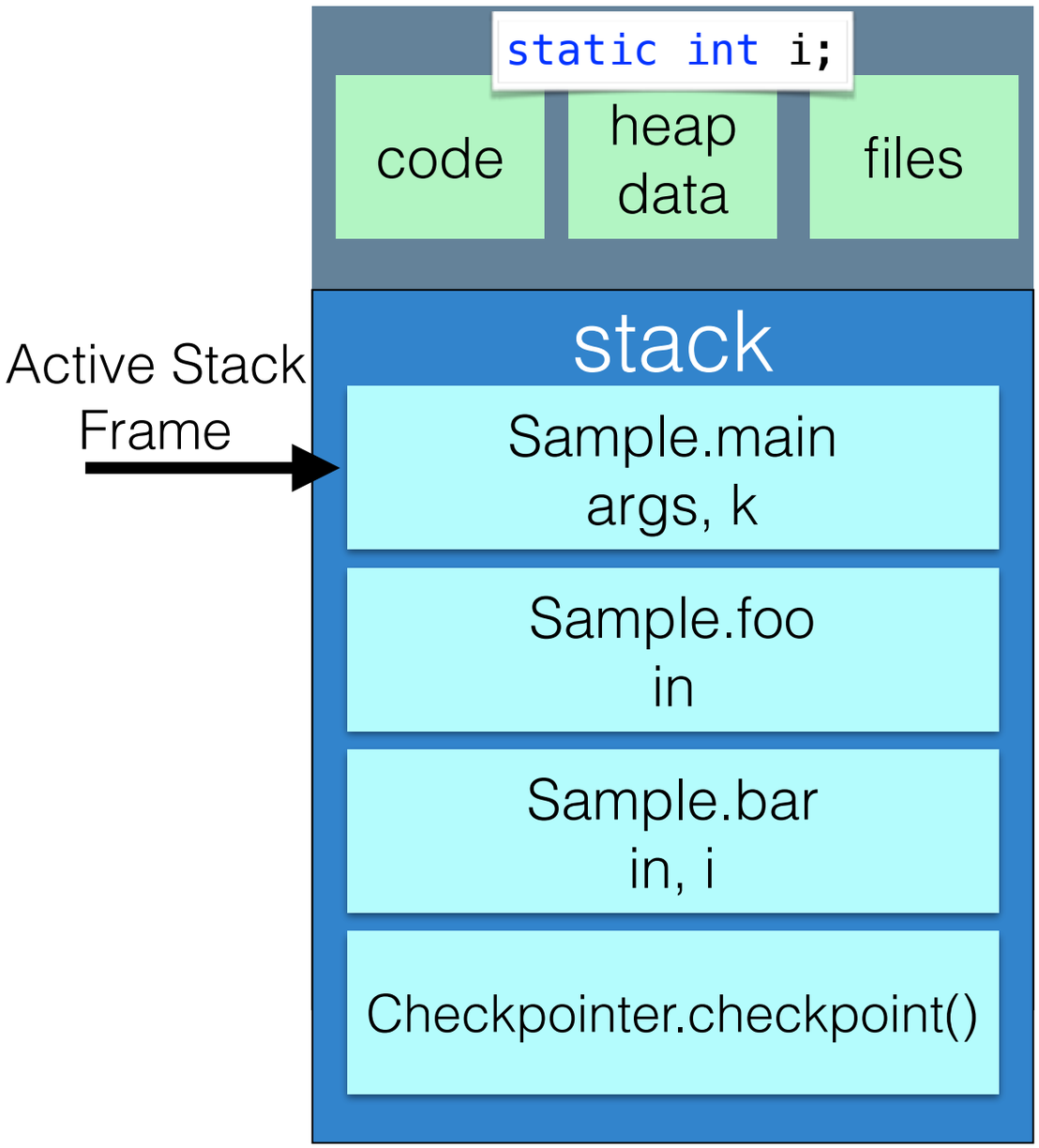
Ex. GPS, IMEI

Building a Dynamic Analysis Toolkit

Checkpoint/Rollback

- What is it?
  - Checkpoint an app captures its state
  - Rollback an app returns it to that same state
- Why?
  - Fault Tolerance
    - Input generation/state exploration
    - Speculative Execution

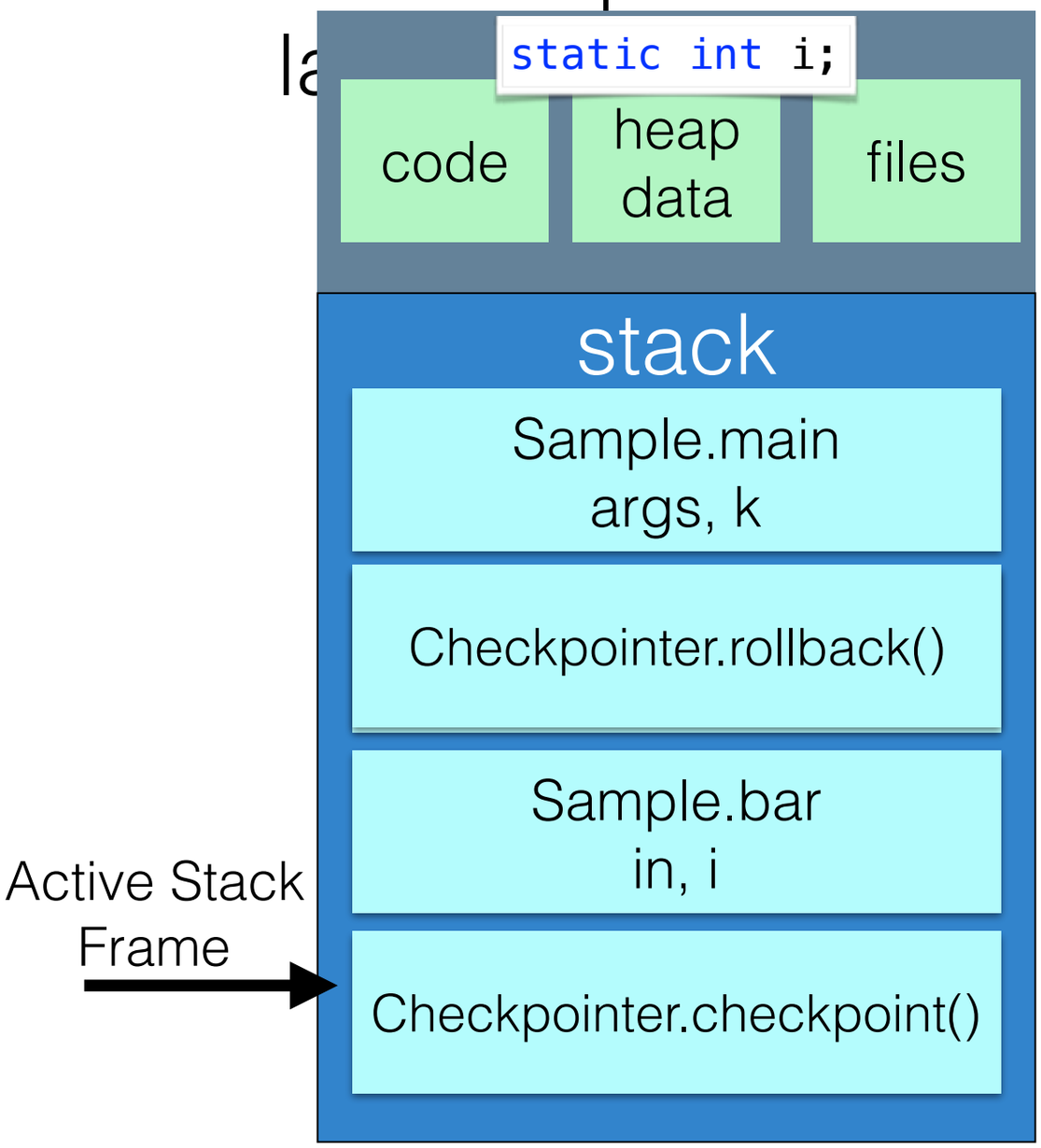
Checkpoint/Rollback



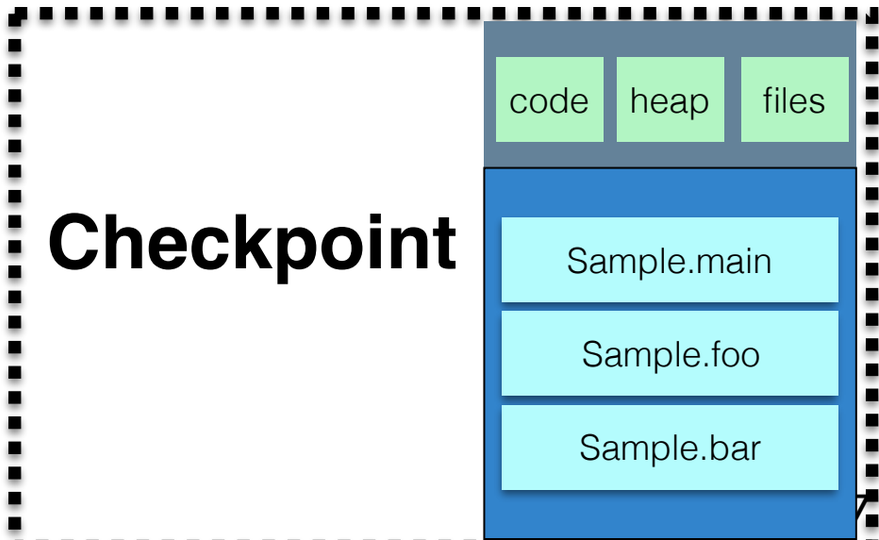

```
public class Sample
{
    static int i;
    public static void main(String[] args)
    {
        int k = 10;
        foo(k);
        Checkpointer.rollback();
    }
    public static void foo(int in)
    {
        bar(in);
    }
    public static void bar(int in)
    {
        i = in;
        Checkpointer.checkpoint();
    }
}
```

Checkpoint/Rollback

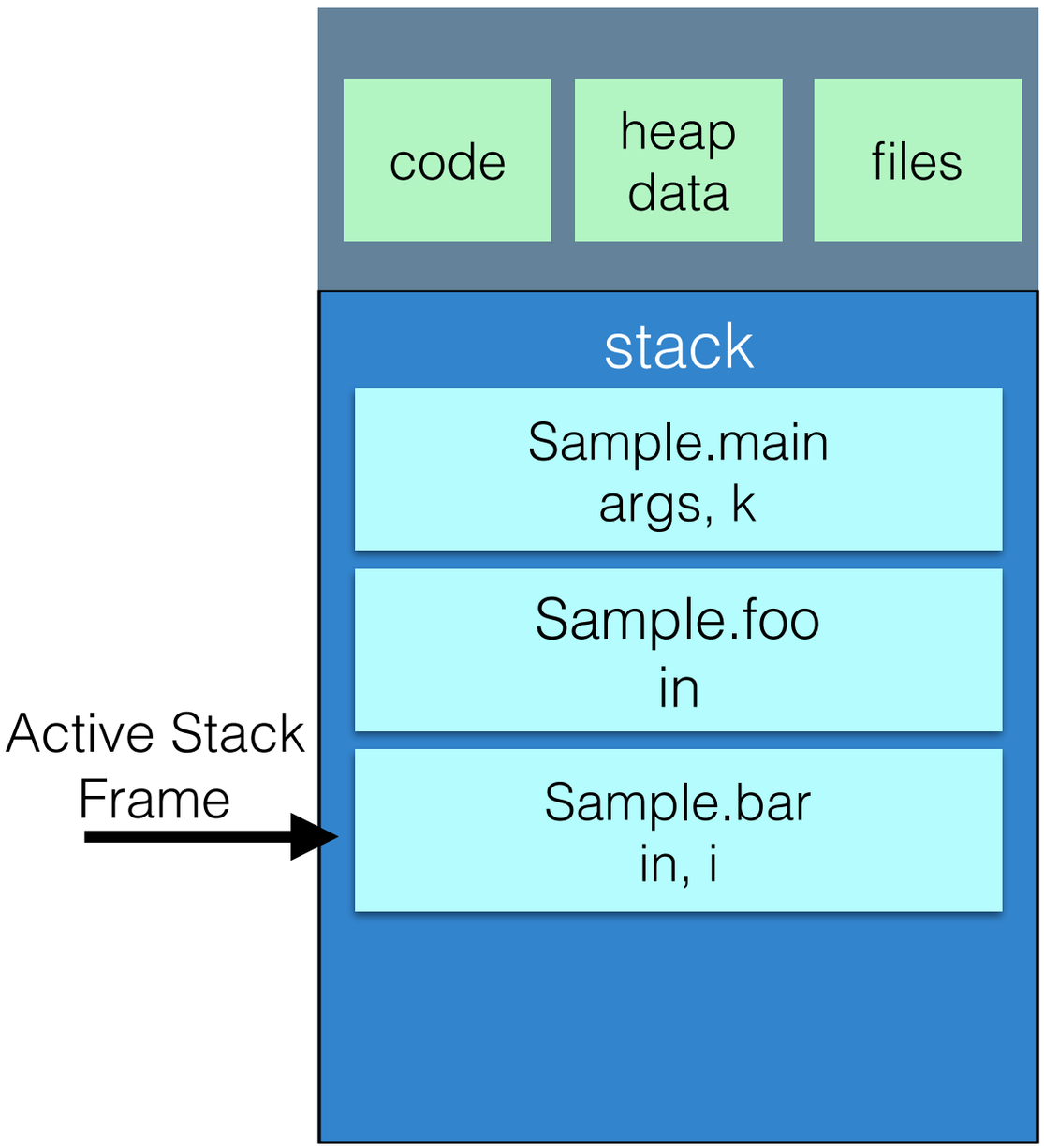
- Often implemented with page fault handlers to la




```
public class Sample
{
    static int i;
    public static void main(String[] args)
    {
        int k = 10;
        foo(k);
        Checkpointer.rollback();
    }
    public static void foo(int in)
    {
        bar(in);
    }
    public static void bar(int in)
    {
        i = in;
        Checkpointer.checkpoint();
    }
}
```



Checkpoint/Rollback




```
public class Sample
{
    static int i;
    public static void main(String[] args)
    {
        int k = 10;
        foo(k);
        Checkpointer.rollback();
    }
    public static void foo(int in)
    {
        bar(in);
    }
    public static void bar(int in)
    {
        i = in;
        Checkpointer.checkpoint();
    }
}
```

How to implement Checkpoint/Rollback?

- Expensive to capture all of this data!
- Lazy approach: use page fault handlers - fork the process, trap whenever process tries to write a page, make a copy of the page so you can restore later
  - Still expensive!
  - Annoying to do in complex runtimes (e.g. JVM)…

Checkpoint/Rollback + JVM

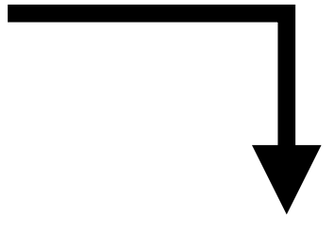
- Prior work hooked in between a modified JVM's garbage collector and some system functionality for page faults
- Not portable, still slow (what if objects are sparsely populated amongst pages)

Checkpoint/Rollback + JVM

- Alternative idea: why not copy **individual variables**?
- Avoids copying JVM state (which we can assume we can reproduce other ways)
- How do you decide to copy an individual variable though?

Checkpoint/Rollback + JVM

```
int field;
public void foo(){
    int x = field;
}
```



```
                int field;
boolean shouldCopy;
boolean shouldRestore;
public void foo(){
    if(this.shouldCopy)
        makeCopyOfThis();
    else if(this.shouldRestore)
        restoreThisFromCopy();
    int x = field;
}
```

Add this before every field access?

Slow!

Traverse the whole heap to set shouldCopy?

Slow!

CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs

Jonathan Bell and Luís Pina
George Mason University

ECOOP 2018

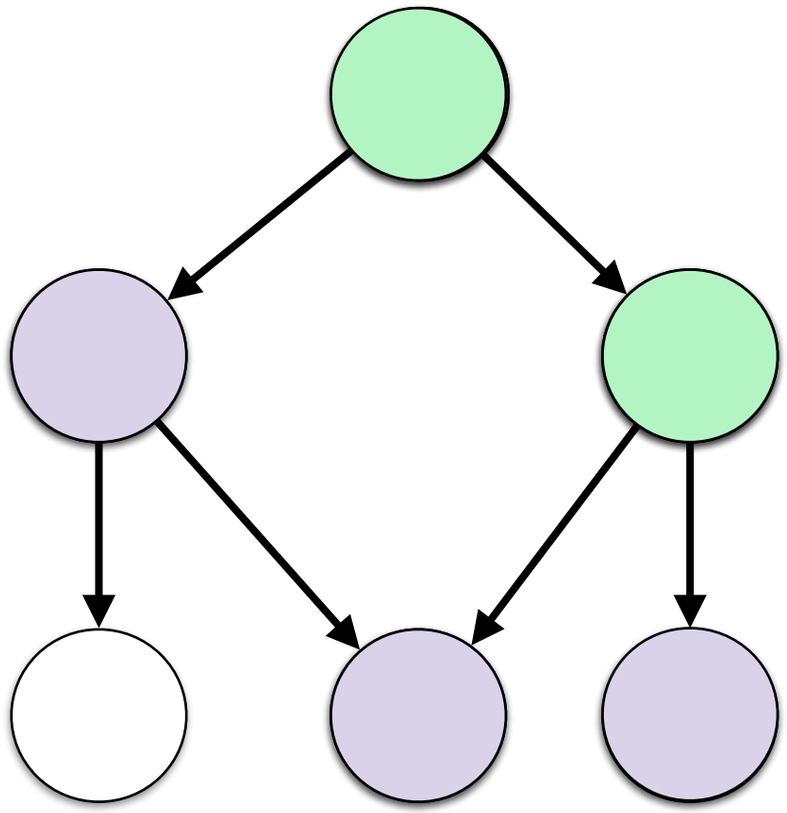
Key Insight

Enabling inisght:

CROCHET

- Uses this key insight (that we can change the type of objects in the JVM on-the cheap) to perform a **lazy heap traversal** to visit all objects, only as they are accessed by the app
- Much like page faults
- Adds a method, **$$$access()** to all objects, calls it before any read or write to the object
- `$$$access()` is an empty method, and gets optimized away in common case
- As it reaches new objects, changes type of the frontier of objects to define `$$$access()` to self-propagate

Lazy Heap Traversal



Starting state: no objects traversed

Step 1: Find all roots, change them into a proxy

Step 2: Once you dereference a proxy, it turns everything it can directly reference into a proxy and disables itself

Lazy Heap Traversal

- Guarantees propagation just before any edge of the graph is traversed
- As we transition each object, we copy its fields too (checkpoint) or restore them (rollback)
- Patch around reflection, Unsafe, static fields
- And it works for stack frames too!

Java Stack Smashing

- Enter: JVMTI - the JVM Tooling Interface
- Used to implement debuggers
- Can use to fetch stack trace, pop stack frames, get/set local variables

Java Stack Smashing

Checkpoint code:

```
//Original Code
void someFunc(int i, int[] ar)
{
    int j = i + 1;
    ar[i] = ar[i] - j;
    j--;
    otherFunc(j, ar); //Checkpoint is called by otherFunc
    ar[i] = 10;
}
//Checkpoint code
void someFunc(int i, int[] ar)
{
    boolean captureStack = false;
    int j = i + 1;
    ar[i] = ar[i] - j;
    j--;
    otherFunc(j, ar);
    if(captureStack)
       Checkpointer.captureStack();
    ar[i] = 10;
}
```

Java Stack Smashing

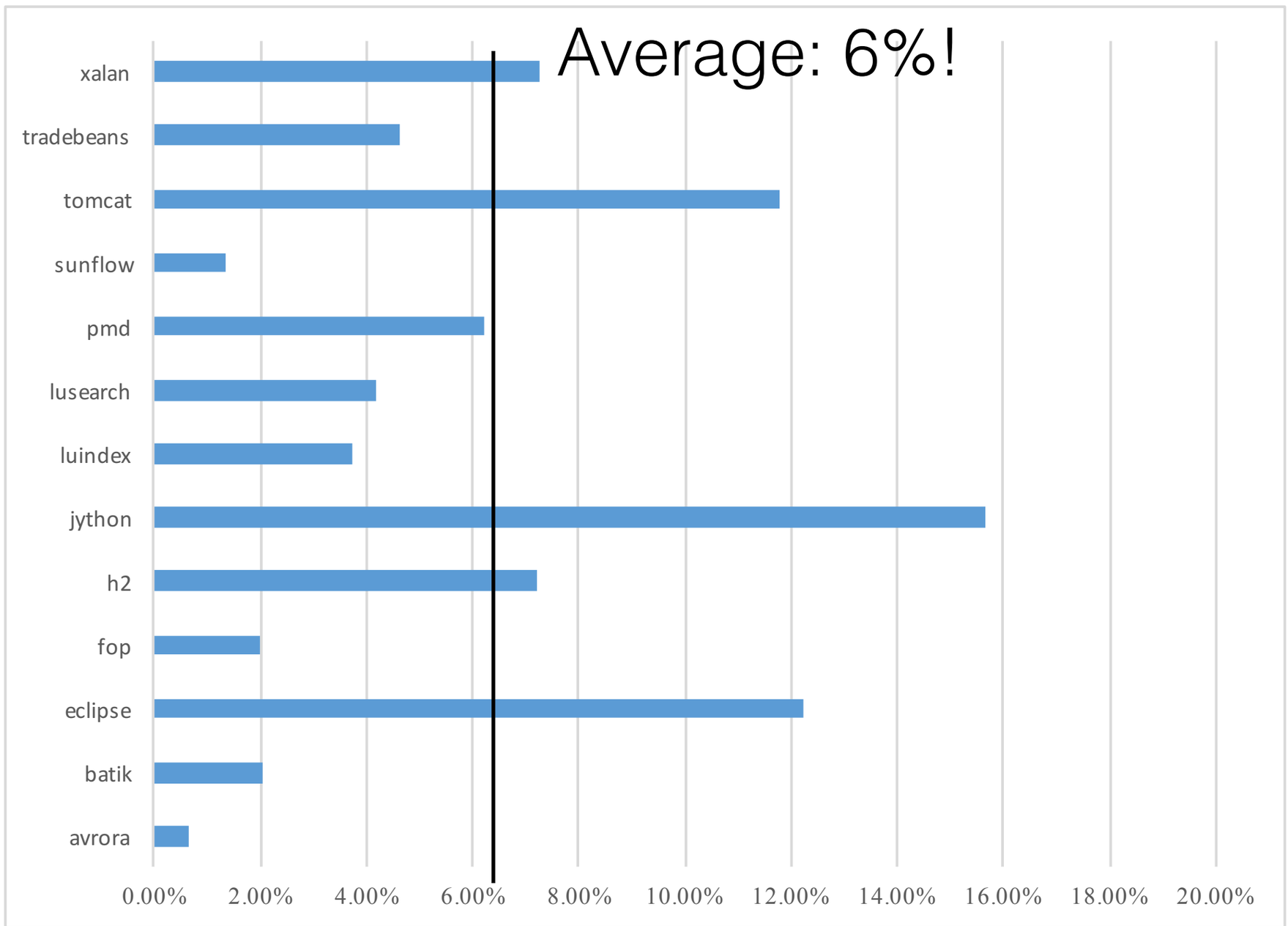
```
//Original Code
void someFunc(int i, int[] ar)
{
     int j = i + 1;
     ar[i] = ar[i] - j;
     j--;
     otherFunc(j, ar); //Checkpoint is called by otherFunc
     ar[i] = 10;
}
```

```
//Rollback code
void someFunc(int i, int[] ar)
{
    int j;
    boolean captureStack = false;
    if(Rollbacker.doRollback())
    {
       i = Rollbacker.localInt();
       ar = Rollbacker.localIntArray();
       j = Rollbacker.localInt();
       Rollbacker.removeRollbackCode();
    }
    else
    {
       //Original Code up-to otherFunc()
    }
    ar[i] = 10;
}
```

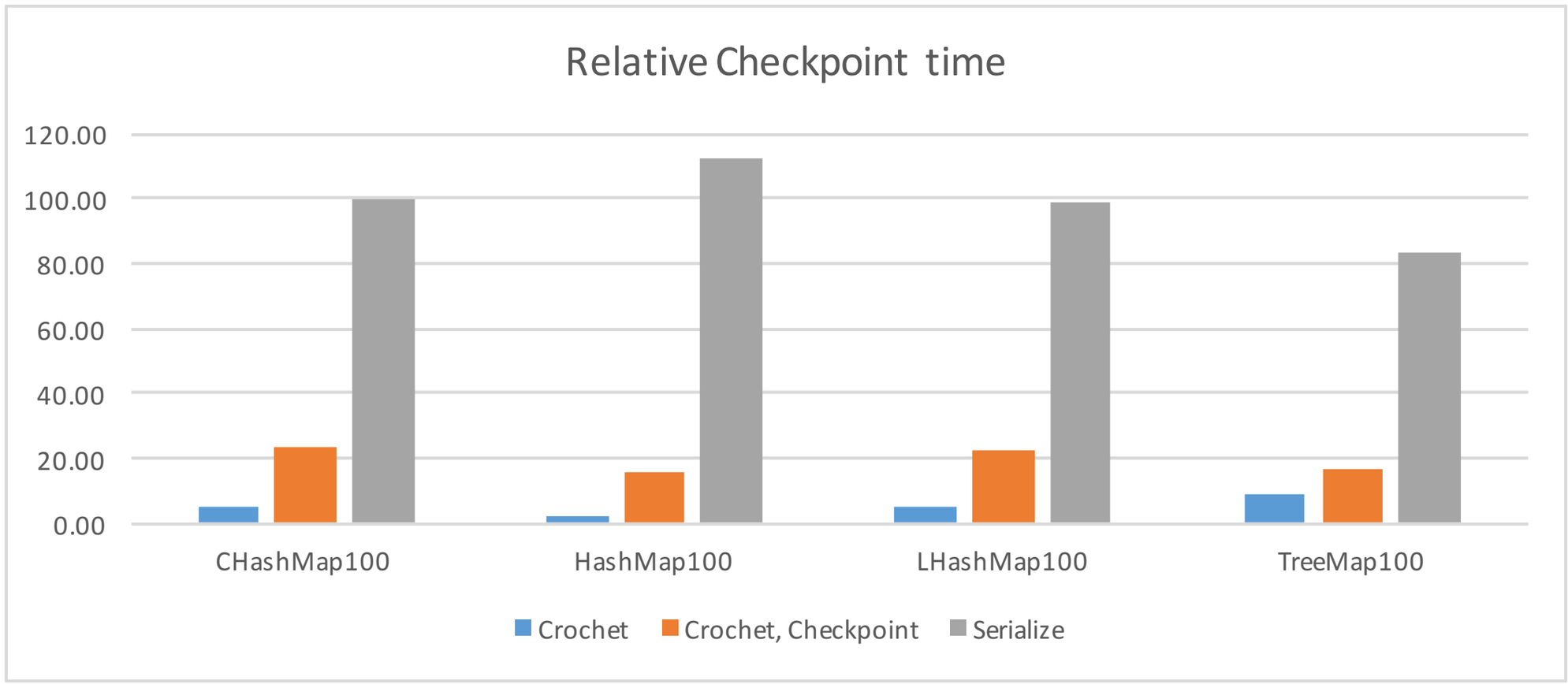
Java Stack Smashing

- We can use this to **completely** reproduce the stack trace and state of a thread
- Not shown: we grab everything sitting on the operand stack too!
- Not shown: we grab all of the monitors that each thread holds too!

Overhead on DaCapo, no checkpoint



Time to checkpoint



4 Microbenchmarks, each traversing a map with 100,000 entries

Future Work

- Short term:
  - Hybrid Static-Dynamic Dataflow Analysis
  - Using Phosphor for other analyses (e.g. tracking Path Constraints)
  - Abusing JVM for checkpoint-rollback WITH Phosphor

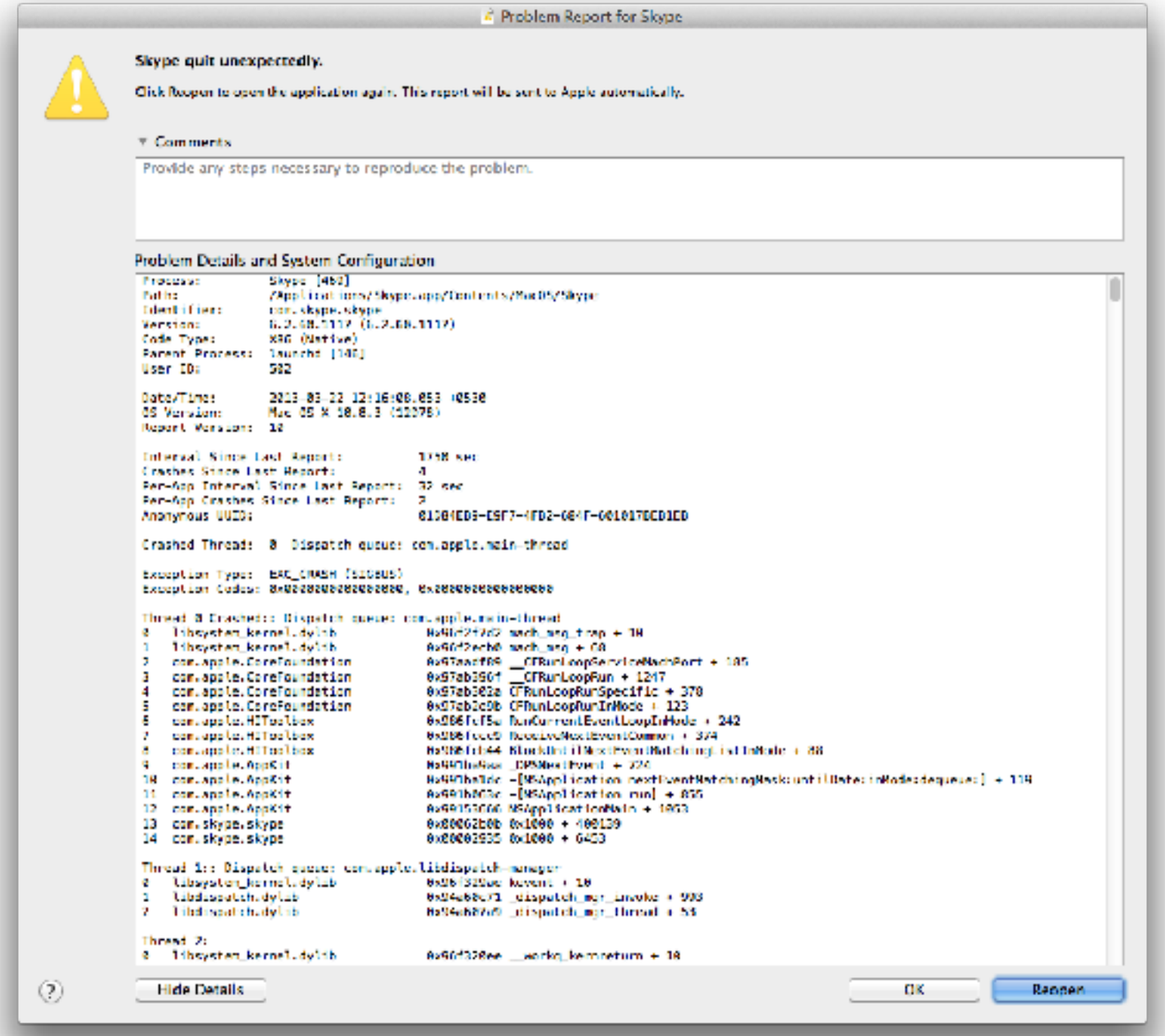
Tools and Approaches to Make Reliable Software Easier

Dependency Analysis for Debugging and Behavioral Clones

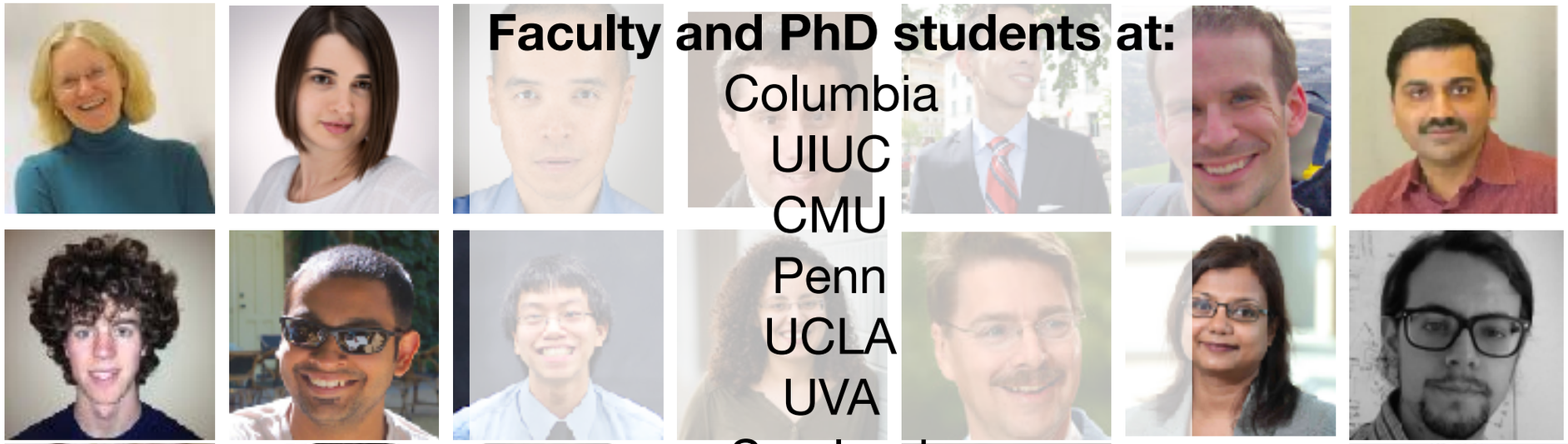
```
302  static void setUpFullSourceWorkspace(boolean large) throws Exception {
303    // Get wksp info
304    IWorkspace workspace = ResourcesPlugin.getWorkspace();
305    final IWorkspaceRoot workspaceRoot = workspace.getRoot();
306    String targetWorkspac
307
308    // Modify resources w
309    // running them
310    IEclipsePreferences r
311    resourcesPreferences.
312    workspace.getDescript
313    workspace.getDescript
314
315    // Get projects direc
316    File wkspDir = new Fi
317    FullSourceProjectsFilter filter = new FullSourceProjectsFilter();
318    File[] directories = wkspDir.listFiles(filter);
319    int dirLength = directories == null ? 0 : directories.length;
```

```
workspaceRoot= WorkspaceRoot  (id=409)
    path= Path  (id=456)
    projectTable= Collections$SynchronizedMap<K,V>  (id=730)
    workspace= Workspace  (id=731)
R/
```

Runtime Monitoring of Deployed Software



My Collaborators



Faculty and PhD students at:

Columbia
UIUC
CMU
Penn
UCLA
UVA
Saarland
Padderbor
Potsdam
ElectricCloud, Inc

Plus: Masters Students

Zhimao Liu
Namo Lu
Mandy Wang
Winnie Narang
Nikhil Sarda
Ethan Hann
Jason Halpern
Evgeny Fedetov
John Murphy

Plus: Undergraduate Students

Sravya Kalva
Pakeezha Arfany
Monica Jeyasan
Jonathan Barrios
David Rincon-Cruz
Emilia Pakulski
Alana Ramjit
Jennifer Lam
Xingzhou Derek He
Sidharth Shanker
Miriam Melnick
Alison Yang