Fully Automatic and Precise Detection of Thread Safety Violations

PLDI 2012

by

Michael Pradel and Thomas R. Gross

ETH Zurich

presented by

Martin Aigner

University of Salzburg

May 2, 2013

Overview

- The problem of testing multi-threaded code
- Proposed solution: automatically generate test cases
- Evaluation of the results

TESTING FOR THREAD-SAFETY

- Consider a (wanna-be thread-safe) class in an OO language...
- Try to argue about it’s thread-safety... How do you do that?
  - Verify correct concurrent behavior?
    - Too expensive in most cases
  - Write tests? How? Where to start? Where to end?
    - Also expensive

EXAMPLE

```
class StringBuffer implements CharSequence {

  /* init instance with given string */
  StringBuffer(String s) {...}

  /* modify this holding a lock */
  synchronized void deleteCharAt (int index) {...}

  /* modify this while holding a lock */
  synchronized void insert (int dstOffset, CharSequence s,
    int start, int end) {...}

  /* convenience overloading */
  void insert (int dstOffset, CharSequence s) {
    this.insert(dstOffset, s, 0, s.length());
  }
}
```

```
class StringBuffer implements CharSequence {

  /* init instance with given string */
  StringBuffer(String s) {...}

  /* modify this holding a lock */
  synchronized void deleteCharAt (int index) {...}

  /* modify this while holding a lock */
  synchronized void insert (int dstOffset, CharSequence s,
    int start, int end) {...}

  /* convenience overloading */
  void insert (int dstOffset, CharSequence s) {
    this.insert(dstOffset, s, 0, s.length());
  }
}

main: StringBuffer sb1 = new StringBuffer("thread-safe");
main: StringBuffer sb2 = new StringBuffer("not");
```

Thread 1	Thread 2
`sb1.insert(0, sb2);`	`sb1.deleteCharAt(0);`

Result?

```
class StringBuffer implements CharSequence {

  /* init instance with given string */
  StringBuffer(String s) {...}

  /* modify this holding a lock */
  synchronized void deleteCharAt (int index) {...}

  /* modify this while holding a lock */
  synchronized void insert (int dstOffset, CharSequence s,
    int start, int end) {...}

  /* convenience overloading */
  void insert (int dstOffset, CharSequence s) {
    this.insert(dstOffset, s, 0, s.length());
  }
}

main: StringBuffer sb1 = new StringBuffer("thread-safe");
```

Thread 1	Thread 2
`sb1.insert(0, sb1);`	`sb1.deleteCharAt(0);`

Result?

PROBLEMS WITH TESTING

- In general:
  - Cover the configuration space
  - Observe success of failure
- For concurrent tests:
  - Find out if a concurrent execution meets a sequential specification

Proposed Solution

Create, execute, and validate test cases automatically

REQUIREMENTS

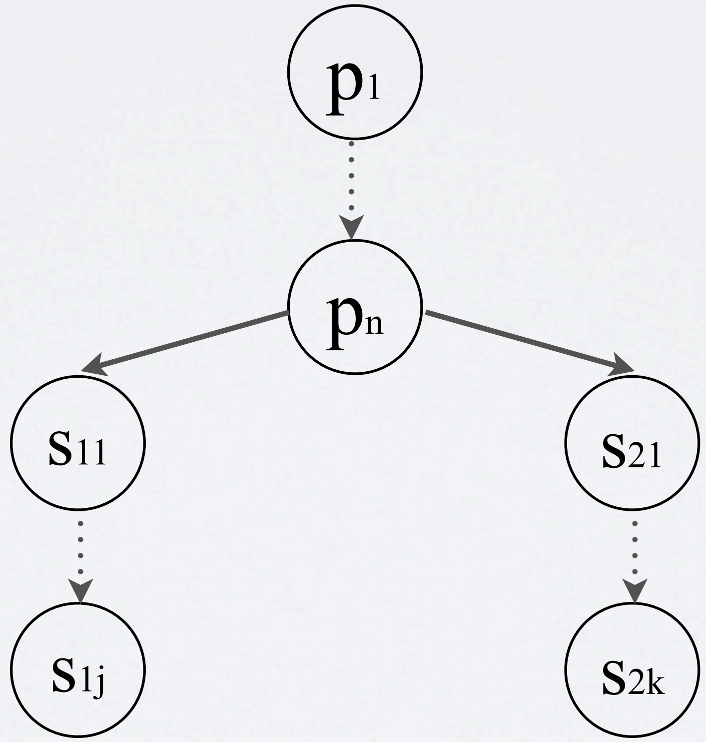
- We need:
  - The class under test (CUT)
  - A set of auxiliary classes the CUT depends on
  - A test that maybe triggers a bug (thread-safety violation)
  - An execution environment that exposes the bug
  - An oracle that recognizes an erroneous execution

Generating concurrent tests

- Definitions: call sequence, prefix, suffixes
  - A call sequence is a n-tuple of calls $(c_1, \ldots, c_n)$ where a call is a method together with input and output parameters
  - Prefix: a call sequence that is executed sequentially to "grow" an object
  - Suffixes: a set of call sequences that are executed concurrently on an object to trigger a concurrency bug

Generating concurrent tests

- Definitions: test
  - A test is a triple $(\mathbf{p}, \mathbf{s}_1, \mathbf{s}_2)$ where $\mathbf{p}$ is a prefix and $\mathbf{s}_1$ and $\mathbf{s}_2$ are suffixes



EXAMPLE: CONCURRENT TEST

prefix **p**

```
main: StringBuffer sb1 = new StringBuffer("thread-safe");
```

Thread 1	Thread 2
`sb1.insert(0, sb1);`	`sb1.deleteCharAt(0);`

suffix s_1

suffix s_2

TASKS: CREATING CALL SEQUENCES

- A task takes a call sequence $s_{in} = (c_1, ..., c_i)$ and returns a call sequence $s_{out} = (c_1, ..., c_i, c_j, ..., c_n)$
- 3 types of tasks:
  - *instantiateCUTTask*: appends a call to instantiate the CUT
  - *callCUTTask*: appends a call to the CUT instance
  - *parameterTask*: provides a typed parameter by either:
    - selecting a previous output parameter
    - appending a call that returns the parameter

Test generation algorithm

- Three global variables:
  - P: a set of prefixes
  - M: a map assigning a prefix to its set of suffixes
  - T: a set of already generated tests

TEST GENERATION ALGORITHM

- Step 1: create a new prefix
  - invokes *instantiateCUTTask* to create an instance of CUT
  - repeatedly invokes *callCUTTask* to extend the prefix
  - the *parameterTask* is invoked whenever a call requires a parameter

Test generation algorithm

- Step 2: create a suffix for a prefix
  - repeatedly invokes ***callCUTTask*** and use the output parameters of the prefix as input the suffix
  - again, the ***parameterTask*** is invoked whenever a call requires a parameter
  - add the suffix to M for the given prefix

Test generation algorithm

- Step 3: test creation
- create tests based on the prefix, the suffix and all it's other suffixes (obtained from M) and store them in T
- randomly return a test from T

Algorithm 1 Returns a concurrent test (p, s_1, s_2)

```
P: set of prefixes                                      ▷ global variables
M: maps a prefix to suffixes
T: set of ready-to-use tests
if |T| > 0 then
    return randRemove(T)
if |P| < maxPrefixes then                               ▷ create a new prefix
    p ← instantiateCUTTask(empty call sequence)
    if p = failed then
        if P = ∅ then
            fail(”cannot instantiate CUT”)
        else
            p ← randTake(P)
    else
        for i ← 1, maxStateChangerTries do
            p_ext ← callCUTTask(p)
            if p_ext ≠ failed then
                p ← p_ext
    P ← P ∪ {p}
else
    p ← randTake(P)
s_1 ← empty call sequence                               ▷ create a new suffix
for i ← 1, maxCUTCallTries do
    s_1,ext ← callCUTTask(s_1, p)
    if s_1,ext ≠ failed then
        s_1 ← s_1,ext
M(p) ← M(p) ∪ {s_1}
for all s_2 ∈ M(p) do                                   ▷ one test for each pair of suffixes
    T ← T ∪ {(p, s_1, s_2)}
return randRemove(T)
```

Step 1

Step 2

Step 3

the parameters *maxPrefixes*, *maxStateChangerTries* and *maxCUTCallTries* are heuristically defined limits.

Thread safety oracle

- A class is said to be thread-safe (in Java methodology) if:
  - multiple threads can use it without synchronization
  - <u>and</u> the observed behavior of a concurrent execution is equivalent to and sequential execution of a linearization of the calls preserving per-thread ordering

EXAMPLE

```
main: CopyOnWriteArrayList l = new CopyOnWriteArrayList();
```

Thread 1	Thread 2
`l.add(“a”);`	`l.add(“b”);`
`println(l.toString());`	

Results: `[a]`, `[a,b]`, or `[b,a]`

Linearizations:

```
add(“a”); → println(); → add(“b”); gives [a]
add(“a”); → add(“b”); → println(); gives [a, b]
add(“b”); → add(“a”); → println(); gives [b, a]
```

Definitions

- operator $\oplus$ concatenates call sequences
- For calls $\mathbf{c}$ and $\mathbf{c'}$, the notion $\mathbf{c} \rightarrow \mathbf{c'}$ indicates that call $\mathbf{c}$ precedes $\mathbf{c'}$

LINEARIZATION

Definition 1 (Linearization)**.** *For a test* (p, s_1, s_2)*, let* $\mathcal{P}_{12}$ *be the set of all permutations of the call sequence* $s_1 \oplus s_2$*. The set of linearizations of the test is:*

$$\begin{aligned}\mathcal{L}_{(p,s_1,s_2)} = \{p \oplus s_{12} \mid s_{12} \in \mathcal{P}_{12} \wedge \\ (\forall c, c' \;\; (c \rightarrow_{s_1} c' \Rightarrow c \rightarrow_{s_{12}} c') \wedge \\ (c \rightarrow_{s_2} c' \Rightarrow c \rightarrow_{s_{12}} c'))\}\end{aligned}$$

Intuitively: a linearization of a test (p, s_1, s_2) appends to p all calls from s_1 and from s_2 in a way that preserves the order of calls in s_1 and s_2

EXECUTION

Definition 2 (Execution). *For a test (p, s_1, s_2), we denote the set of all distinguishable executions of this test as $\mathcal{E}_{(p,s_1,s_2)}$. Each $e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}$ represents the sequential execution of p followed by a concurrent execution of s_1 and s_2. Likewise, we denote the sequential execution of a call sequence s as e_s.*

Note: due to the non-determinism of concurrent executions a single test can have multiple distinguishable executions

THREAD SAFETY

Notation: for executions $\mathbf{e}_1$ and $\mathbf{e}_2$, the notion $\mathbf{e}_1 \cong \mathbf{e}_2$ indicates that $\mathbf{e}_1$ and $\mathbf{e}_2$ are equivalent (discussed later)

Definition 3 (Thread safety). *Let $\mathcal{T}_C$ be the set of all possible tests for a class C. C is thread-safe if and only if:*

$$\forall (p, s_1, s_2) \in \mathcal{T}_C \;\; \forall e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}$$
$$\exists l \in \mathcal{L}_{(p,s_1,s_2)} \text{ so that } e_{(p,s_1,s_2)} \cong e_l$$

Thread safety

Notation: for executions e_1 and e_2, the notion $e_1 \cong e_2$ indicates that e_1 and e_2 are equivalent (discussed later)

Definition 3 (Thread safety). *Let $\mathcal{T}_C$ be the set of all possible tests for a class C. C is thread-safe if and only if:*

$$\forall (p, s_1, s_2) \in \mathcal{T}_C \;\; \forall e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}$$
$$\exists l \in \mathcal{L}_{(p,s_1,s_2)} \text{ so that } e_{(p,s_1,s_2)} \cong e_l$$

Problem: this is expensive! All executions times all tests times all linearizations... Let's try the opposite

THREAD-UNSAFETY

We call a class thread-unsafe if:

$$\exists (p, s_1, s_2) \in \mathcal{T}_C \;\; \exists e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}$$
$$\text{so that } \forall l \in \mathcal{L}_{(p,s_1,s_2)} \;\; e_{(p,s_1,s_2)} \not\cong e_l$$

Intuitively: the oracle tries to find a test that exposes behavior not possible with any linearization of the test

Still expensive, but we are lucky this time. We only need to check buggy concurrent executions for equivalence to their linearizations

Equivalence of execution

Definition 4 (Equivalence of executions). *Two executions e_1 and e_2 are equivalent if*

- *neither e_1 nor e_2 results in an exception or a deadlock, or*
- *both e_1 and e_2 fail for the same reason (that is, the same type of exception is thrown or both executions end with a deadlock).*

Algorithm 2 Checks whether a test (p, s_1, s_2) exposes a thread safety bug

```
repeat
    e_(p,s1,s2) ← execute(p, s1, s2)
    if failed(e_(p,s1,s2)) then
        seqFailed ← false
        for all l ∈ L(p, s1, s2) do
            if seqFailed = false then
                e_l ← execute(l)
                if failed(e_l) ∧ sameFailure(e_(p,s1,s2), e_l) then
                    seqFailed ← true
        if seqFailed = false then
            report bug e_(p,s1,s2) and exit
until maxConcExecs reached
```

consider only failed concurrent executions

no concurrency bug if linearization fails for the same reason

concurrency bug only if the linearizations did not fail

Limitations

- Only exceptions and deadlocks are considered
- Testing is still incomplete
- No semantical equivalence of concurrent and linearized executions
- The proposed approach is based on two assumptions:
  - Uncaught exceptions and deadlocks that occur in concurrent executions but not in sequential ones are a problem
  - Sequential execution is deterministic

Contributions

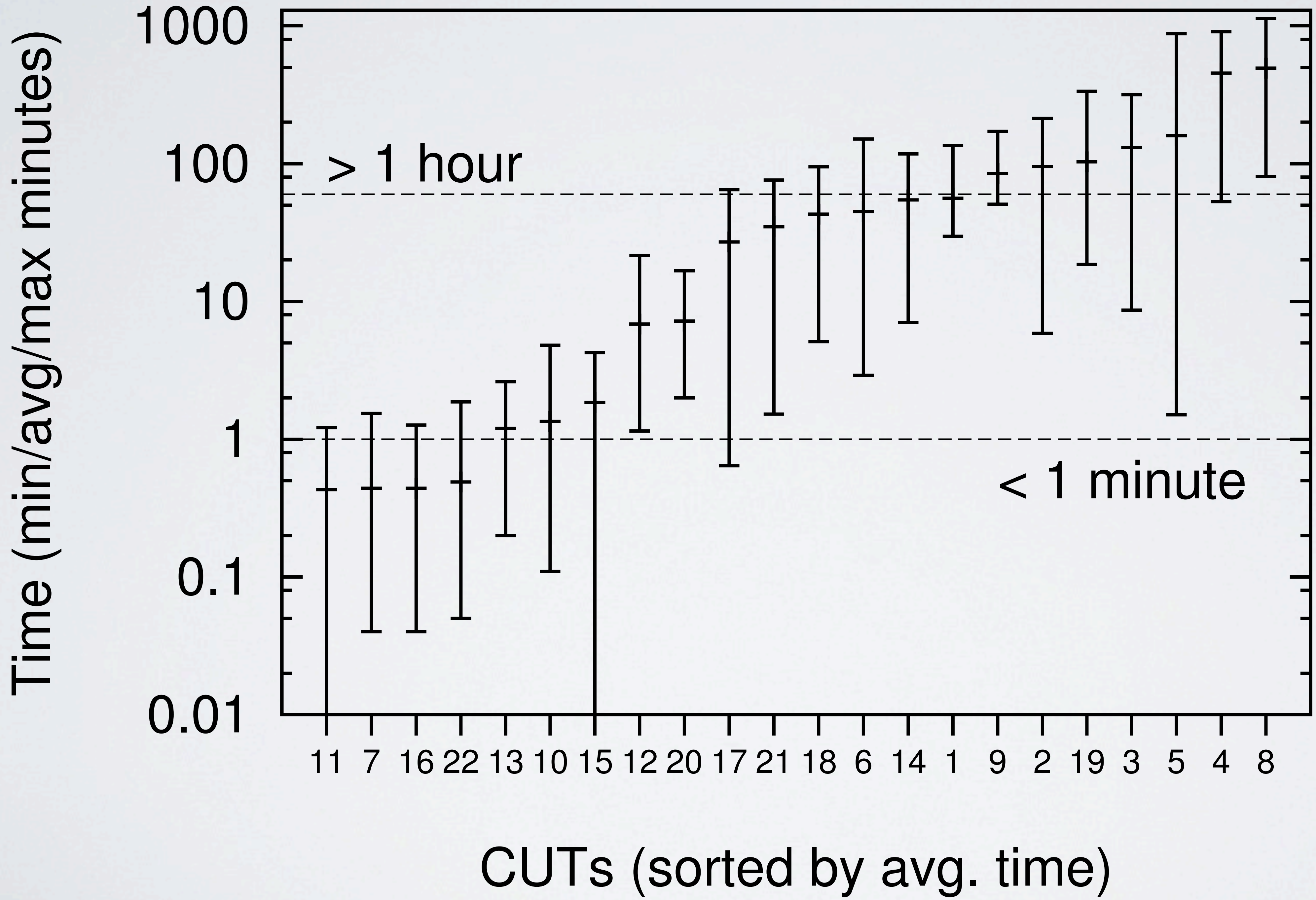
- Fully automatic
- Only true positives are reported
- No (or very little) human effort for testing and evaluation
  - This makes it cheap!
- It does indeed find bugs

Evaluation

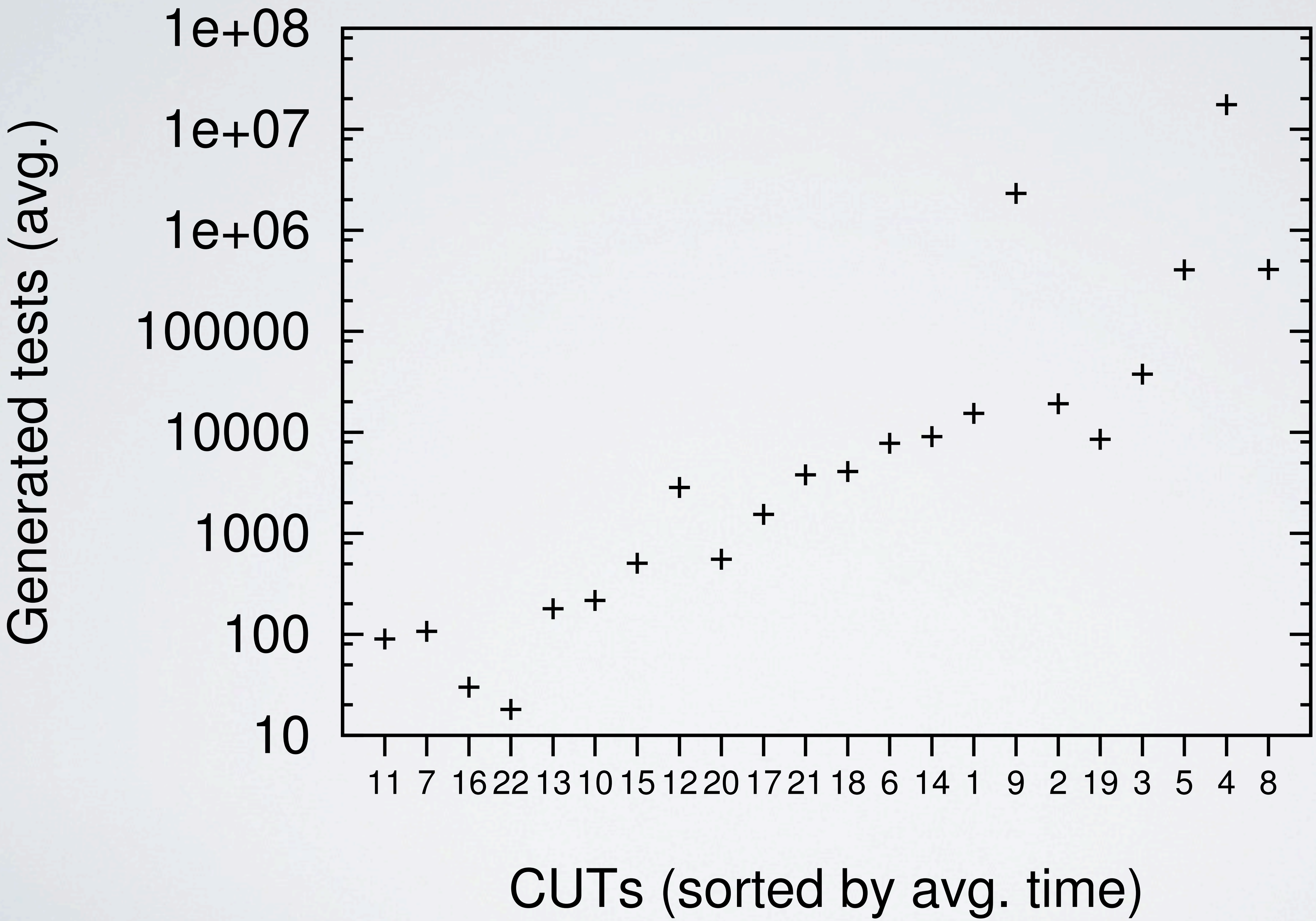
- Tests were performed on six popular code bases
  - Java standard library
  - Apache Commons DBCP
  - XStream (serialization library)
  - LingPipe (text processing toolkit)
  - JFreeChart (chart library)
  - Joda-Time (library for handling data and time)

ID	Code base	Class	Declared thread-safe	Found unsafe	Reason for failing	Implicit
Previously unknown bugs:						
(1)	JDK 1.6.0_27 and 1.7.0	StringBuffer	yes	yes	IndexOutOfBoundsException	yes
(2)	JDK 1.6.0_27 and 1.7.0	ConcurrentHashMap	yes	yes	StackOverflowError	yes
(3)	Commons DBCP 1.4	SharedPoolDataSource	yes	yes	ConcurrentModificationException	yes
(4)	Commons DBCP 1.4	PerUserPoolDataSource	yes	yes	ConcurrentModificationException	yes
(5)	XStream 1.4.1	XStream	yes	yes	NullPointerException	yes
(6)	LingPipe 4.1.0	MedlineSentenceModel	yes	yes	IllegalStateException	no
Known bugs:						
(7)	JDK 1.1	BufferedInputStream	yes	yes	NullPointerException	yes
(8)	JDK 1.4.1	Logger	yes	yes	NullPointerException	yes
(9)	JDK 1.4.2	SynchronizedMap	yes	yes	Deadlock	yes
(10)	JFreeChart 0.9.8	TimeSeries	yes	yes	NullPointerException	yes
(11)	JFreeChart 0.9.8	XYSeries	yes	yes	ConcurrentModificationException	yes
(12)	JFreeChart 0.9.12	NumberAxis	yes	yes	IllegalArgumentException	no
(13)	JFreeChart 1.0.1	PeriodAxis	yes	yes	IllegalArgumentException	no
(14)	JFreeChart 1.0.9	XYPlot	yes	yes	ConcurrentModificationException	yes
(15)	JFreeChart 1.0.13	Day	yes	yes	NumberFormatException	yes
Automatic classification of classes as thread-unsafe:						
(16)	Joda-Time 2.0	DateTimeFormatterBuilder	no	yes	IndexOutOfBoundsException (10x)	yes
(17)	Joda-Time 2.0	DateTimeParserBucket	no	yes	IllegalArgumentException (9x)	no
					NullPointerException (1x)	yes
(18)	Joda-Time 2.0	DateTimeZoneBuilder	no	yes	NullPointerException (6x)	yes
					ArrayIndexOutOfBoundsException (2x)	yes
					IllegalFieldValueException (2x)	yes
(19)	Joda-Time 2.0	MutableDateTime	no	yes	IllegalFieldValueException (9x)	no
					ArithmeticException (1x)	yes
(20)	Joda-Time 2.0	MutableInterval	no	yes	IllegalArgumentException (10x)	no
(21)	Joda-Time 2.0	MutablePeriod	no	yes	ArithmeticException (10x)	yes
(22)	Joda-Time 2.0	PeriodFormatterBuilder	no	yes	ConcurrentModificationException (5x)	yes
					IndexOutOfBoundsException (4x)	yes
					IllegalStateException (1x)	no
	Joda-Time 2.0	ZoneInfoCompiler	no	no	(stopped after 24h)	–

COSTS on 8-core 3GHz Xeons



REQUIRED TEST CASES



REFERENCE

Michael Pradel and Thomas R. Gross. 2012. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation* (PLDI '12). ACM, New York, NY, USA, 521-530.

THANKS A LOT