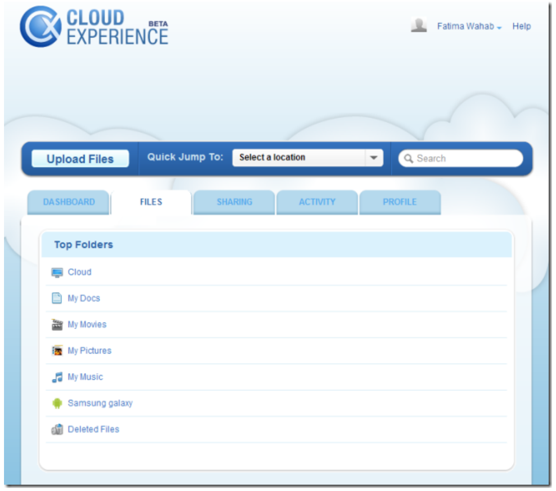
Finding Errors in Multithreaded GUI Applications

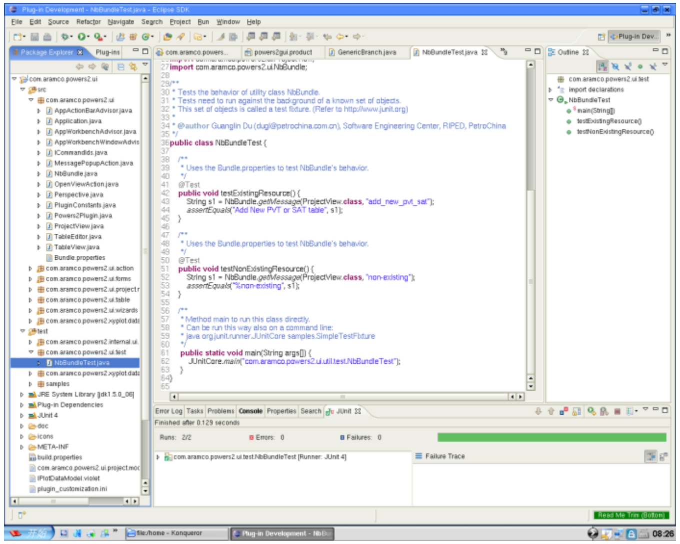
Sai Zhang

University of Washington

Joint work with: Hao Lu, Michael D. Ernst

GUIs are everywhere in modern software

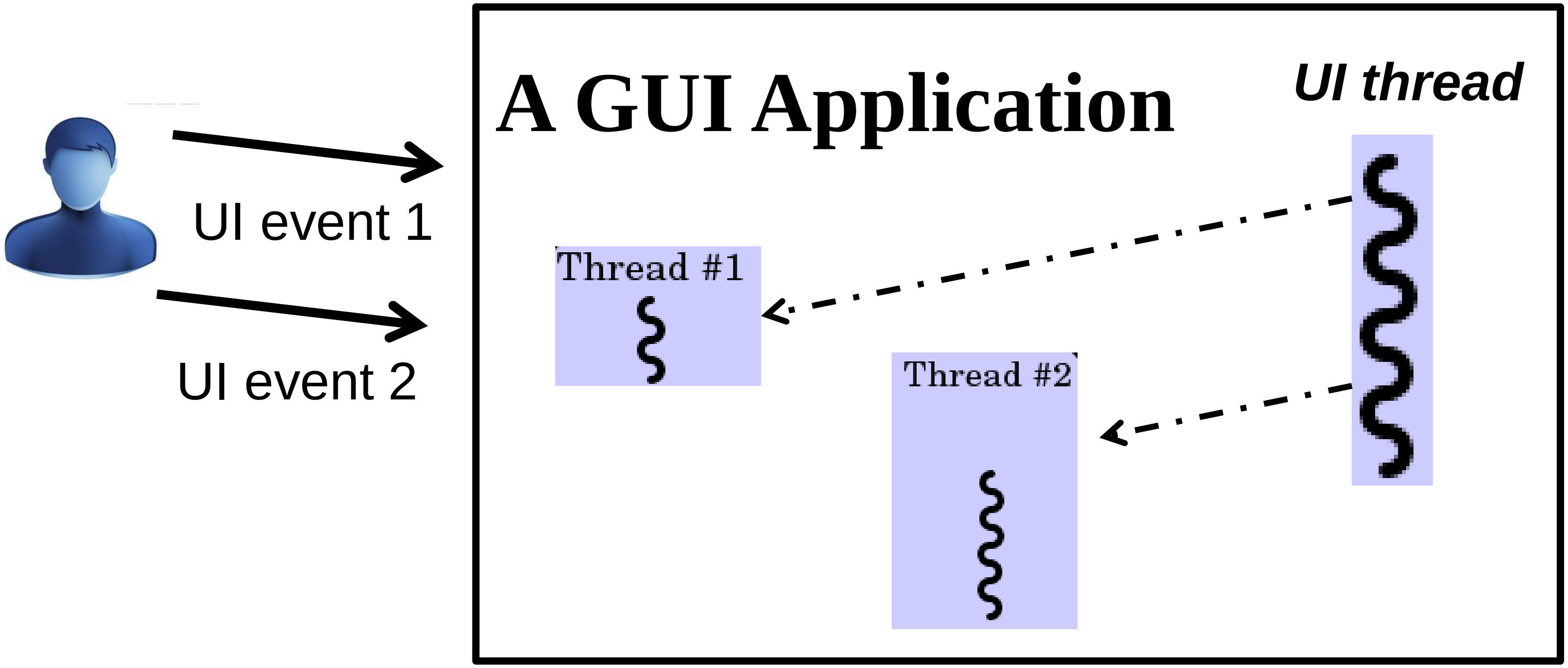




```
import com.aramco.powers2.ui.NbBundle;

/**
 * Tests the behavior of utility class NbBundle.
 * Tests need to run against the background of a known set of objects.
 * This set of objects is called a test fixture. (Refer to http://www.junit.org)
 *
 * @author Guanglin Du (dugl@petrochina.com.cn), Software Engineering Center, RIPED, PetroChina
 */
public class NbBundleTest {

   /**
    * Uses the Bundle.properties to test NbBundle's behavior.
    */
   @Test
   public void testExistingResource() {
      String s1 = NbBundle.getMessage(ProjectView.class, "add_new_pvt_sat");
      assertEquals("Add New PVT or SAT table", s1);
   }

   /**
    * Uses the Bundle.properties to test NbBundle's behavior.
    */
   @Test
   public void testNonExistingResource() {
      String s1 = NbBundle.getMessage(ProjectView.class, "non-existing");
      assertEquals("%non-existing", s1);
   }

   /**
    * Method main to run this class directly.
    * Can be run this way also on a command line:
    * java org.junit.runner.JUnitCore samples.SimpleTestFixture
    */
    public static void main(String args[]) {
      JUnitCore.main("com.aramco.powers2.ui.util.test.NbBundleTest");
   }
}
```

```
Finished after 0.129 seconds
Runs: 2/2    Errors: 0    Failures: 0
com.aramco.powers2.ui.test.NbBundleTest [Runner: JUnit 4]
```

Multithreading in GUI applications



UI event 1

UI event 2

A GUI Application

UI thread

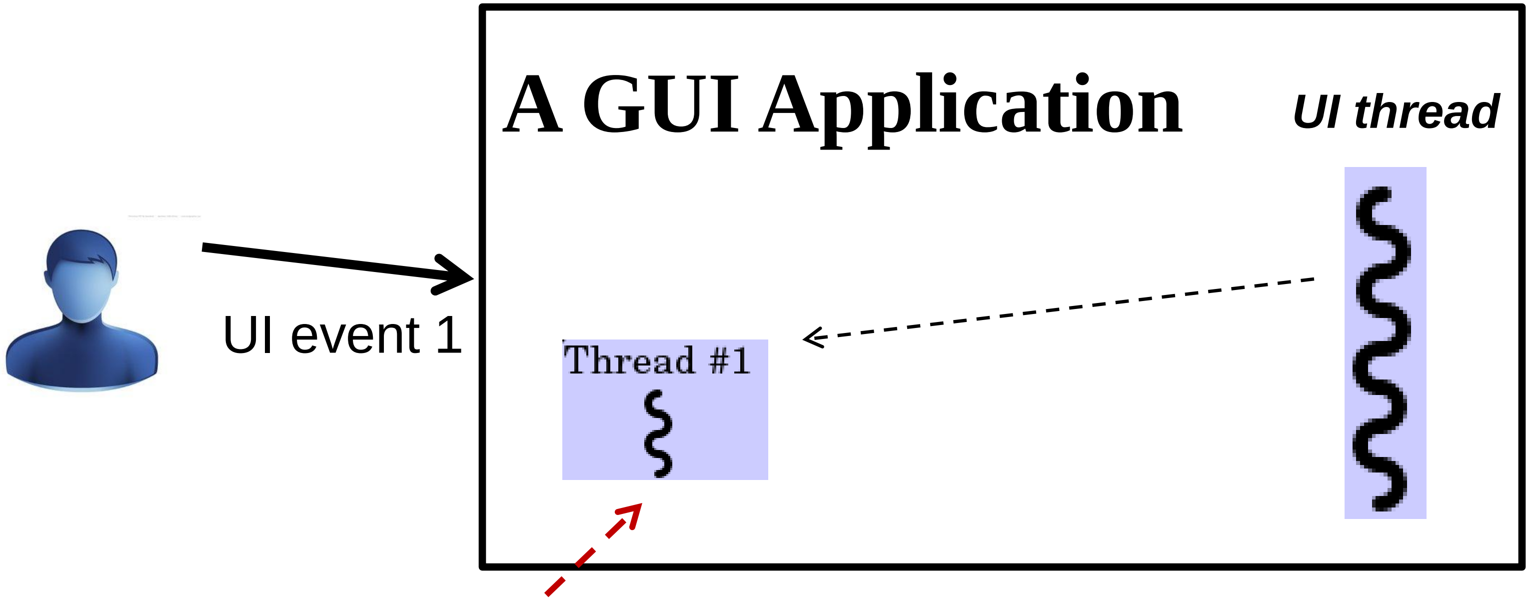
Thread #1

Thread #2

The Single-GUI-Thread Rule

*All GUI objects must be exclusively accessed by the **UI thread***

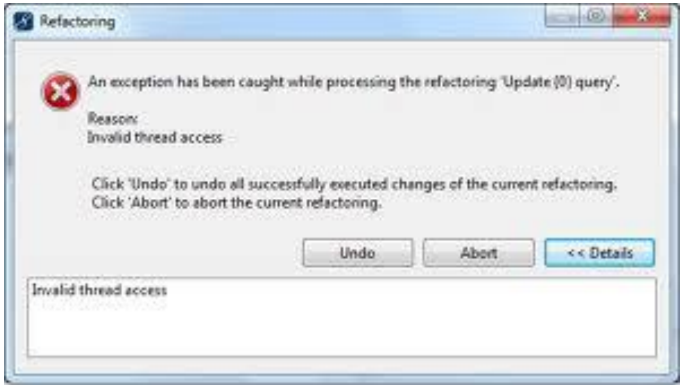
- **Required** by: SWT eclipse Java Swing ANDROID Mac OS X Cocoa Qt X Windows …



This ***non-UI thread*** must ***not*** access any GUI objects

Violation of the Single-GUI-Thread rule

- Triggers an “*Invalid Thread Access Error*”
- May abort the whole application



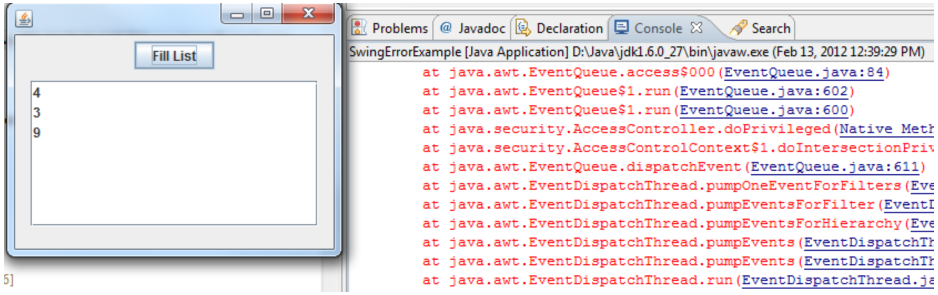
An exception has been caught while processing the refactoring 'Update (0) query'.

Reason:
Invalid thread access

Click 'Undo' to undo all successfully executed changes of the current refactoring.
Click 'Abort' to abort the current refactoring.

Invalid thread access

SWT / Eclipse plugin



```
SwingErrorExample [Java Application] D:\Java\jdk1.6.0_27\bin\javaw.exe (Feb 13, 2012 12:39:29 PM)
        at java.awt.EventQueue.access$000(EventQueue.java:84)
        at java.awt.EventQueue$1.run(EventQueue.java:602)
        at java.awt.EventQueue$1.run(EventQueue.java:600)
        at java.security.AccessController.doPrivileged(Native Meth
        at java.security.AccessControlContext$1.doIntersectionPriv
        at java.awt.EventQueue.dispatchEvent(EventQueue.java:611)
        at java.awt.EventDispatchThread.pumpOneEventForFilters(Eve
        at java.awt.EventDispatchThread.pumpEventsForFilter(EventI
        at java.awt.EventDispatchThread.pumpEventsForHierarchy(Eve
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchTh
        at java.awt.EventDispatchThread.pumpEvents(EventDispatchTh
        at java.awt.EventDispatchThread.run(EventDispatchThread.ja
```

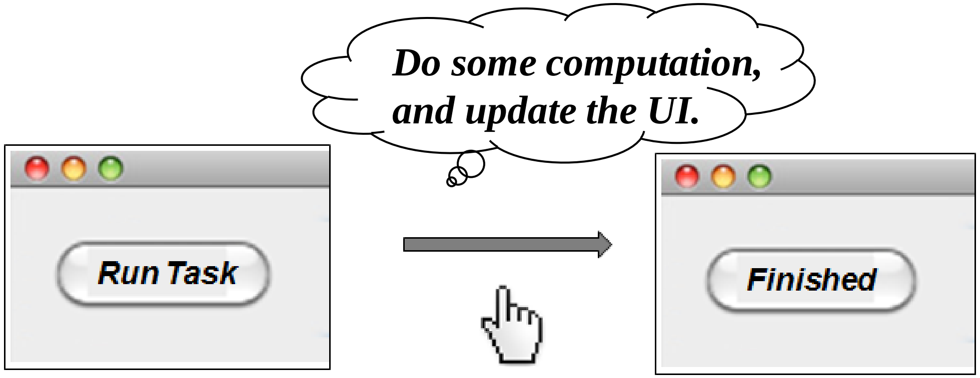
Swing

Sorry!

The application TestAndroid (process test.android) has stopped unexpectedly. Please try again.

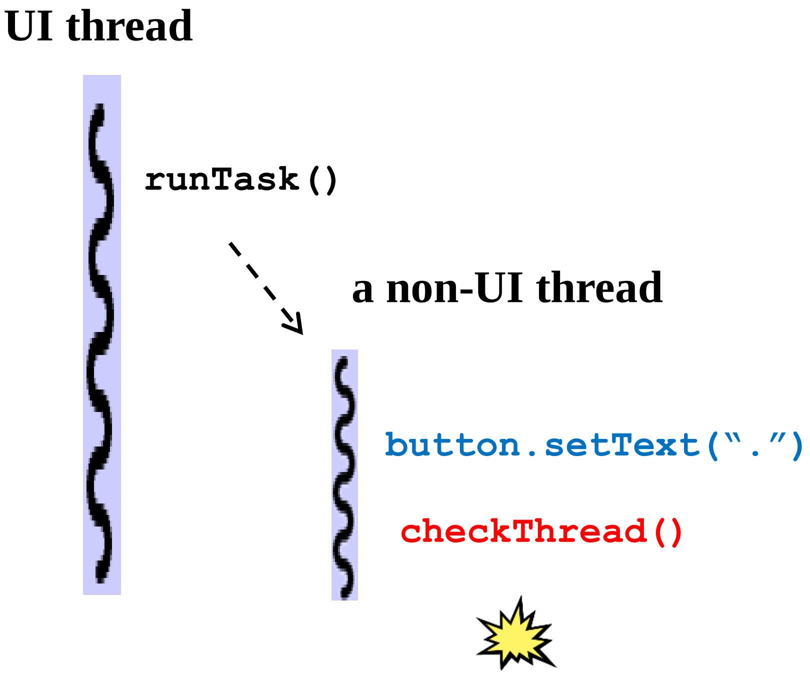
Android

An Example Violation



button's event handler:

```
public void runTask() {
  Runnable r = new Runnable() {
    public void run() {
      … //do some lengthy computation
      button.setText("Finished");
    }
  };
  new thread(r).start();
}
```



Create a new, non-UI thread

Access the `button` object to set text

```
//GUI framework code
public void setText(String) {
     checkThread();
     ...
}
```

Trigger an invalid-thread-access-error

***Invalid Thread Access Errors** in practice*

- ***Pervasive***
  - One of the *top 3* bug categories in SWT [Shanmugam 2010]
  - A Google search returns *11800+* entries (bug reports, FAQs, etc.)
  - In Eclipse
    - *2732* bug reports
    - *156 confirmed* bugs in 20+ projects, 40+ components
- ***Severe***
  - Often aborts the whole application
- ***Hard to debug***
  - Non-trivial effort to fix (e.g., *2 years* to resolve one bug in Eclipse)

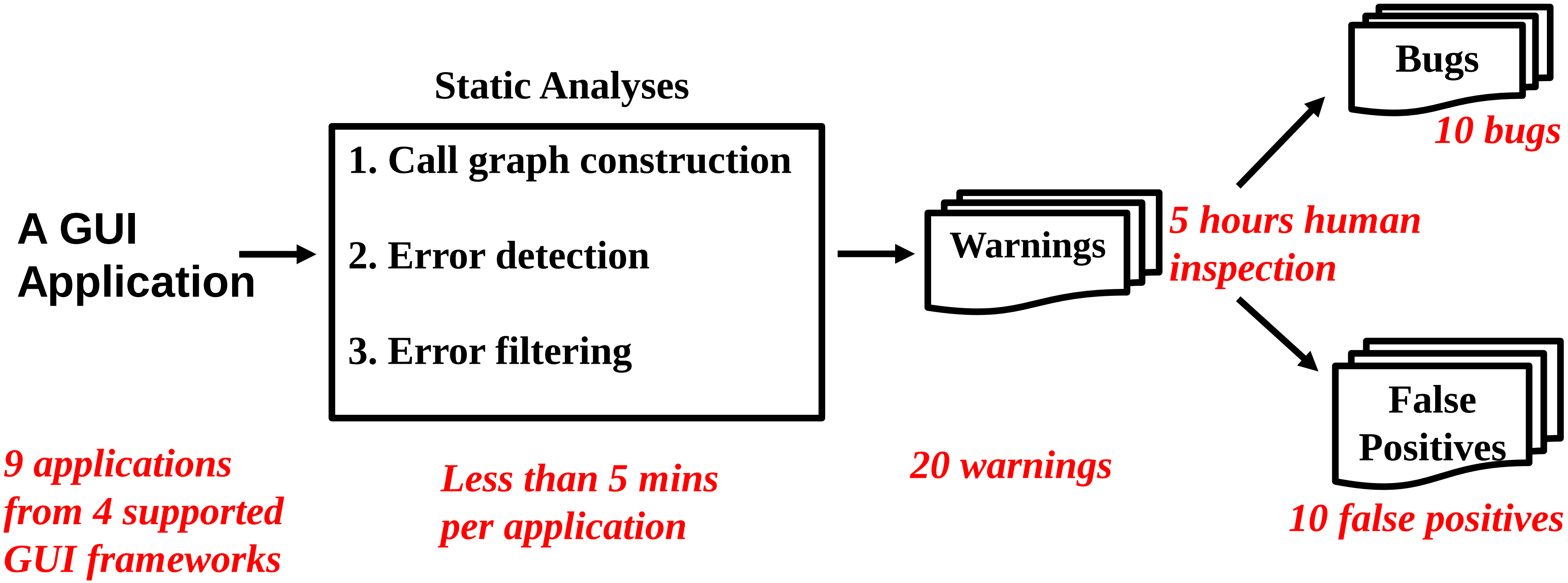
Why the Single-GUI-Thread Rule?

- Simpler programming
  - No datarace nor deadlock on GUI objects

- Less overhead
  - No locking when accessing GUI objects

- A single event queue can dispatch UI events
  - Easy event processing, program comprehension, and testing

Our Error Detection Technique



A GUI Application

Static Analyses

1. Call graph construction

2. Error detection

3. Error filtering

Warnings

Bugs

False Positives

9 applications from 4 supported GUI frameworks

Less than 5 mins per application

20 warnings

5 hours human inspection

10 bugs

10 false positives

-**Automated**: ***no need*** for a test harness

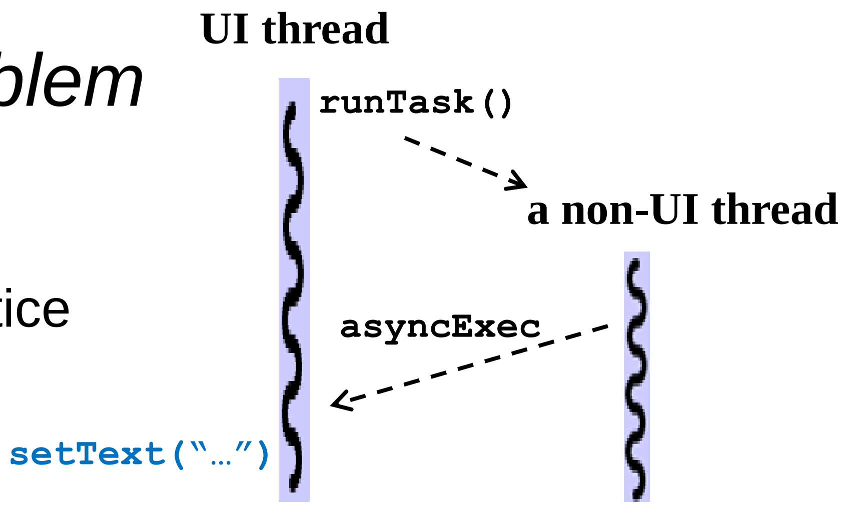
-**General**: instantiated it for ***4 GUI frameworks:*** **SWT** eclipse Java Swing ANDROID

-**Scalable**: evaluated on 9 applications, over ***1.4 M LOC*** with lib code

-**Practical**: found ***10 bugs*** with ***10 false positives***

Existing Solutions for this Problem

- Testing
  - Misses many corner cases in practice
- Stylized programming rules



```
public void runTask() {
  Runnable r = new Runnable() {
    public void run() {
      … //do some lengthy computation
      button.setText(“Finished”);
    }
  };
  new thread(r).start();
}
```

Requiring Wrappers

```
Display.asyncExec(new Runnable(){
  public void run() {
    button.setText(“Finished”);
  }
};
```

- **Unnecessary**: ***if already on the UI thread***
- **Dangerous**: ***may introduce new concurrency errors***

Results on 9 evaluation programs

	#Warnings	#Bugs
Requiring Wrappers	6393	?
Our technique	20	10

Outline

- Problem
- → Error detection technique
- Implementation
- Experiments
- Related work
- Conclusion and future work

Terminology

- **UI thread**:

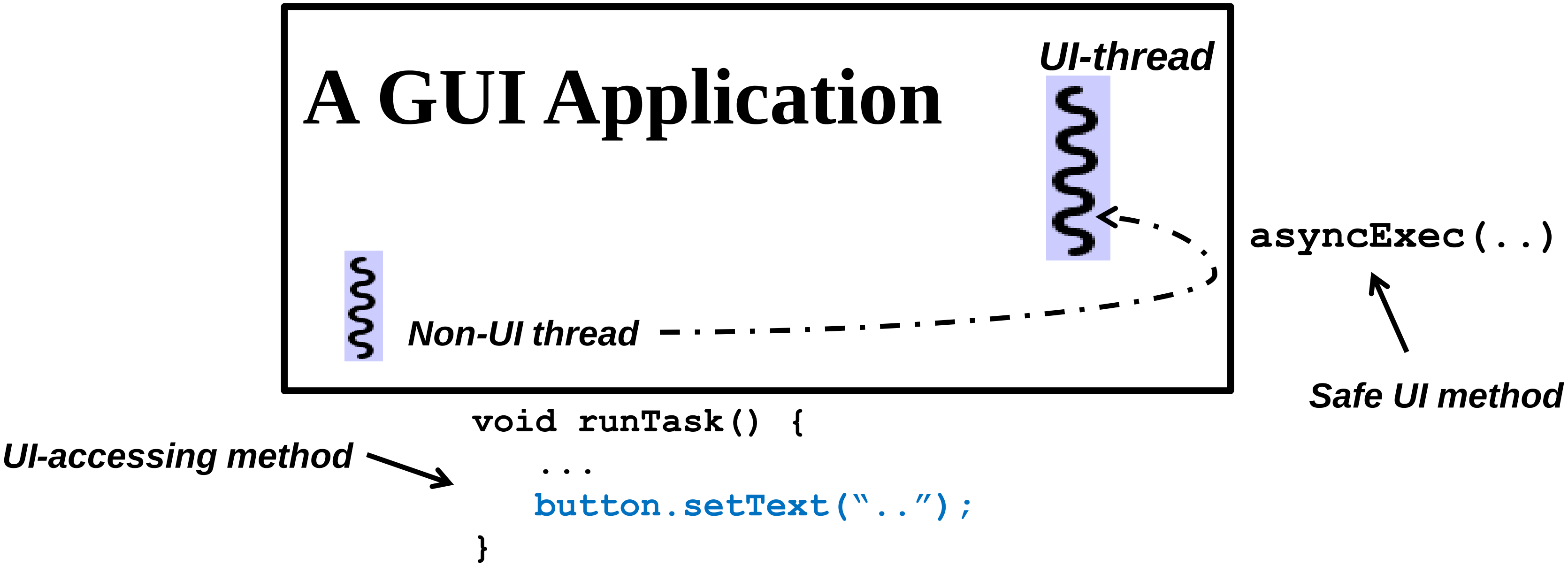
  a single special thread to handle UI events

- **Non-UI thread**: *other threads*
- **UI-accessing method:**

  a method whose execution may read or write a GUI object

- **Safe UI method:**

  message-passing methods to execute code on the UI thread



A GUI Application

UI-thread

Non-UI thread

`asyncExec(..)`

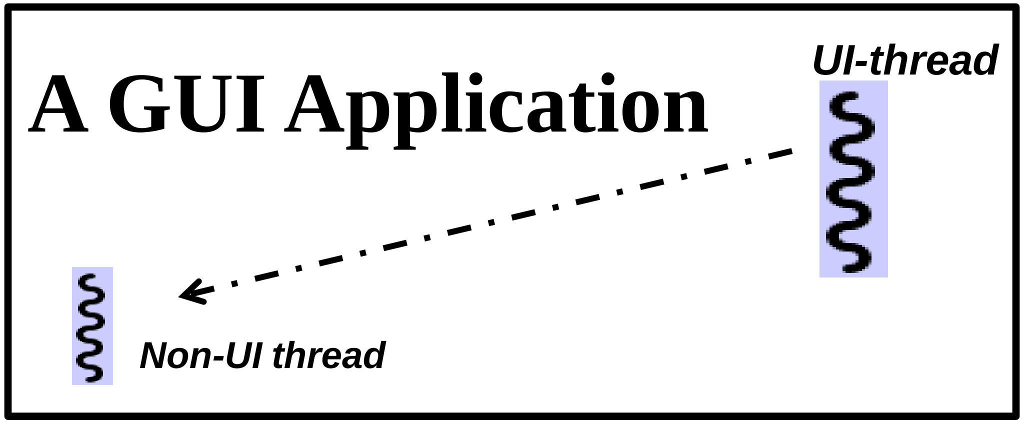
Safe UI method

UI-accessing method

```
void runTask() {
   ...
   button.setText(“..”);
}
```

Assumptions

- Single UI thread
- Thread spawning:
  - Every non-UI thread is (transitively) spawned by the UI thread



Problem formulation: call graph reachability

- An invalid thread access error occurs when:

  a non-UI thread invokes a UI-accessing method ***without*** going through a Safe UI method

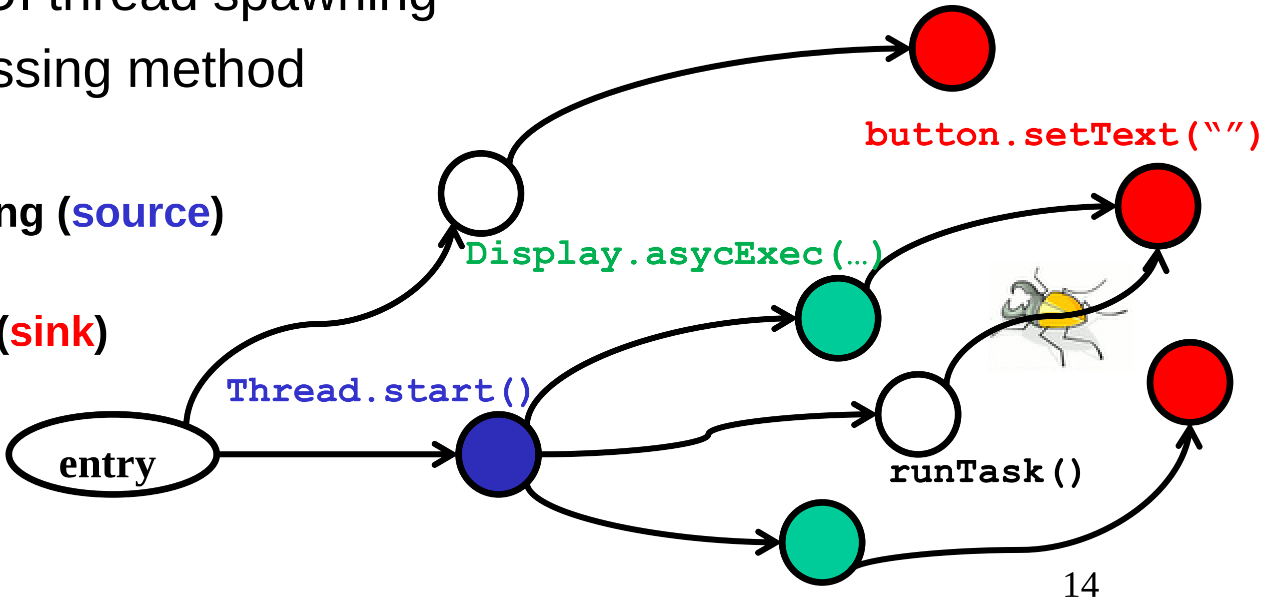
- A reachability problem
  - **Source**: non-UI thread spawning
  - **Sink**: UI-accessing method

🔵 **Non-UI thread spawning (source)**

🔴 **UI-accessing method (sink)**

🟢 **Safe UI method**

⚪ **Other method**



Error detection algorithm

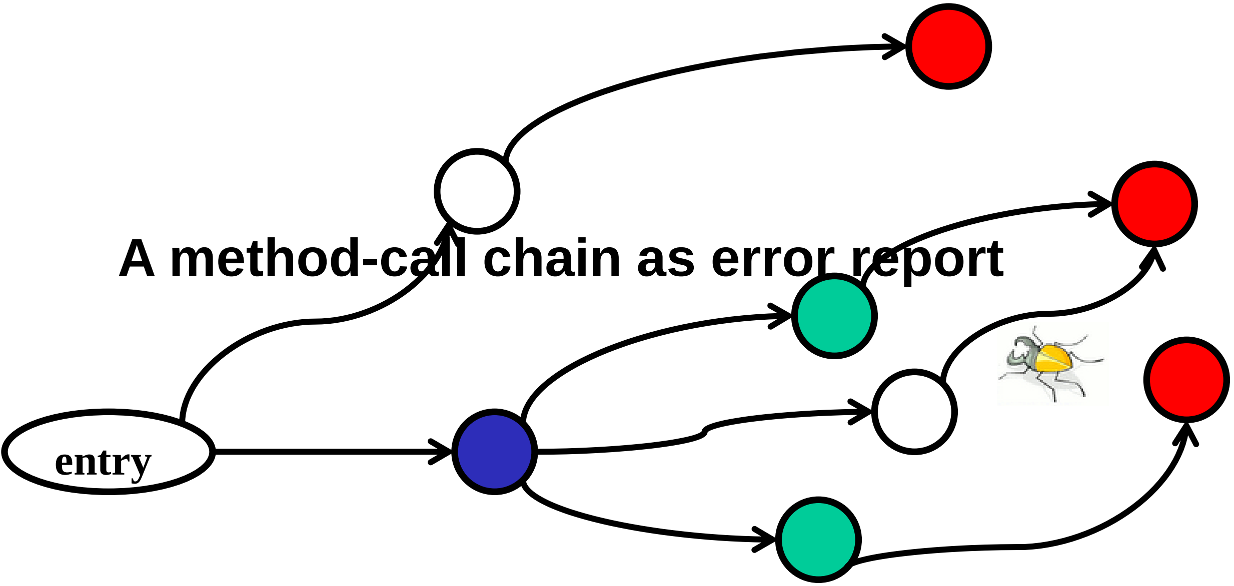
1. Construct ***a call graph*** for the tested program
2. Find paths from **`Thread.start()`** to UI-accessing methods <u>*without*</u> going through a safe UI method

🔵 **Non-UI thread spawning (i.e., `Thread.start()`)**

🔴 **UI-accessing method**

🟢 **Safe UI method**

⚪ **Other method**



A method-call chain as error report

Reflection in Constructing Call Graphs

Android Application:

```
<LinearLayout>
    <Button android:id="@+id/button_id" android:text="A Button" />
</LinearLayout>


...
Button button = (Button) findViewById(“button_id”);
button.setText(“This is a button”);
...
```

- `findViewById` does not **explicitly** construct a button object
- A call graph construction algorithm may:
  - fail to conclude the variable `button` points to a **concrete** object
  - exclude a `setText` edge to the call graph (that should exist)
  - miss 1 bug in our experiments

Reflection-aware call graph construction

Android Application:

```
<LinearLayout>
    <Button android:id="@+id/button_id" android:text="A Button" />
</LinearLayout>
```

Before transformation:

```
Button button = (Button) findViewById(“button_id”);
button.setText(“This is a button”);
```

After transformation:

```
Button button = new Button(null);
button.setText(“This is a button”);
```

- ***Program transformation***: replace reflection calls with explicit object creation expressions
- Use an *off-the-shelf call graph construction algorithm* on the ***transformed*** program

Annotation support for native methods

- Relationships between native methods and Java methods

```
@CalledByNativeMethods(callers = {“init”})
 public void addTypeItem(int id, String label) { …}
```

  - Manually specified
  - **`addTypeItem(int, String)`** may be called by native method “**`init`**”
  - Our technique will miss 1 bug without such annotations

Filtering the error reports

- A static analysis may report:
  - false positives
  - redundant warnings with the same error root cause

- A set of error filters to remove likely invalid reports
  - 2 sound filters
  - 3 heuristic filters

*2 **sound** error report filters*

- **Filter 1**: remove syntactically subsumed reports

  ~~`a() → b() → thread.start() → d()`~~

  `b() → thread.start() → d()`

- **Filter 2**: remove reports containing user-annotated, project-specific methods

  `util.runOnUIMethod(Runnable r)`

  - Checks whether the current thread is UI thread or not

*3 **heuristic** error report filters*

- **Filter 3**: remove reports containing specific library calls

  e.g., `Runtime.shutDown`

- **Filter 4**: remove longer reports with the same “entry node → `Thread.start()`” head nodes

  ~~`a() → b() → Thread.start() → c() → d() → e()`~~

  `a() → b() → Thread.start() → c() → f()`

- **Filter 5**: remove longer reports with the same “`thread.start()` → ui-accessing node” tail nodes

  ~~`a() → b() → Thread.start() → c() → d() → e()`~~

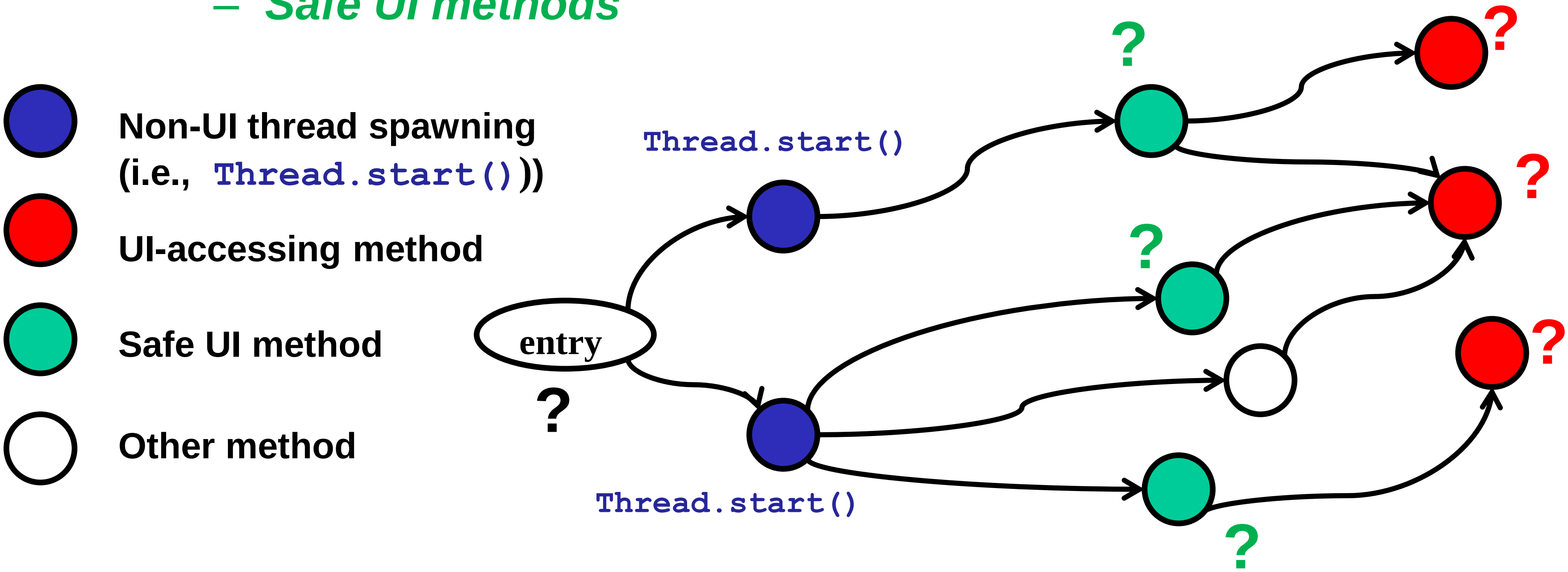
  `f() → Thread.start() → c() → d() → e()`

Outline

- Problem
- Error detection technique
- Implementation
- Experiments
- Related work
- Conclusion and future work

Instantiation for different frameworks

- Need to customize
  - ***program entry points***
  - ***UI-accessing methods***
  - ***Safe UI methods***



🔵 **Non-UI thread spawning (i.e., `Thread.start()`))**

🔴 **UI-accessing method**

🟢 **Safe UI method**

○ **Other method**

Instantiation details for 4 GUI frameworks

Frameworks	Entry node	UI-accessing methods	Safe UI method
SWT	`main()`	`checkWidget /` `checkDevice`	`asyncExec /` `syncExec`

Outline

- Problem
- Error detection technique
- Implementation
- Experiments
- Related work
- Conclusion and future work

Subject programs

Programs	Line of Code
SWT desktop applications	
FileBunker	14,237
ArecaBackup	23,226
Eclipse plugins	
EclipseRunner	3,101
HundsonEclipse	11,077
Swing applications	
S3dropbox	2,353
Sudoku Solver	3,555
Android applications	
SGTPuzzler	2,220
Mozilla Firefox	8,577
MyTracks	20,297
Total:	**89, 273**

Framework size:
1.4 MLOC

Experimental Procedural

- Run the error detection algorithm on each application
  - 3 call graph construction algorithms

RTA **0-CFA** **1-CFA**

precision

  - 2 configurations for Android applications
    - **with** / **without** call graph enhancement
      (handle reflection + annotations for native methods)

- Tool performance
  - *Less than 5 minutes* per application
- Manual result inspection
  - Spent *5 hours* in total to check the output validity

Experimental Results

- Output **20 warnings**, in which **10** are bugs (**5** are new)

Call graph algorithm	# Warnings	#Bugs
RTA with enhancement	250	4
0-CFA with enhancement	136	6
1-CFA with enhancement	**20**	**10**

- More precise call graph ⇒ more bugs found
  - 1-CFA found the most

Call graph algorithm	# Warnings	#Bugs
1-CFA	19	8
1-CFA with enhancement	**20**	**10**

- Call graph enhancement are useful (2 more bugs)

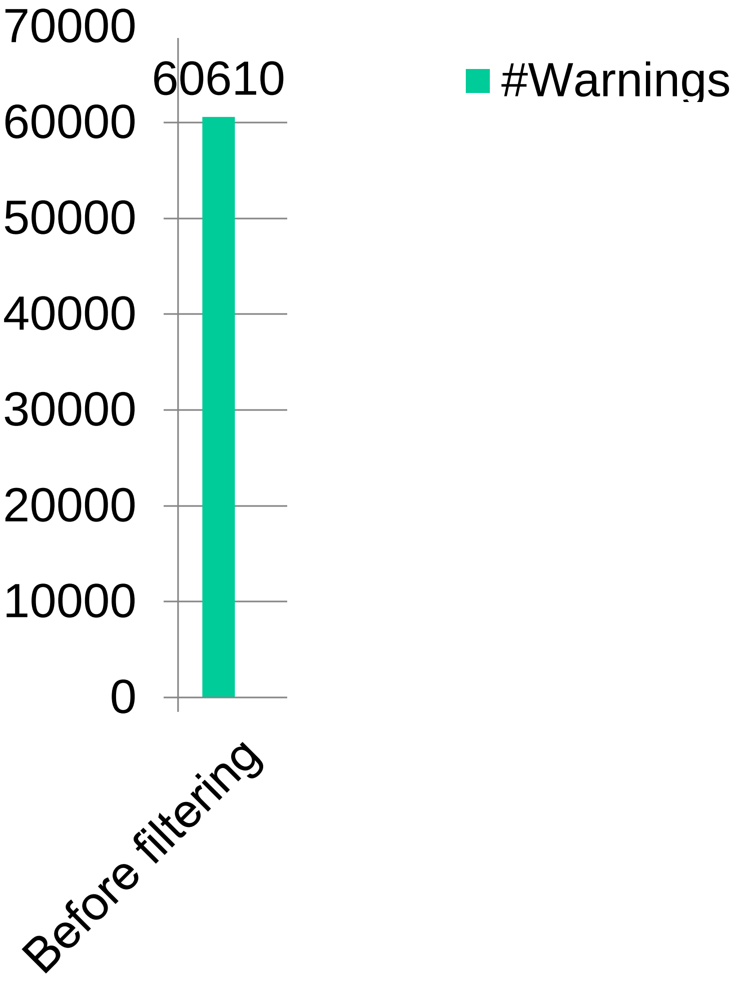
Comparing graph search strategies

- Our technique uses BFS, compare it with alternatives

Strategies	#Warnings	#Bugs
BFS	**20**	**10**

- Observations from our subject programs
  - Multi-source BFS **omits** bugs
  - DFS searches deeper, and returns **longer** paths (more likely to be invalid, due to the conservative call graph)
  - Exhaustive search is **sound** but **infeasible** in practice

Evaluating error filters



Sound Filters:

F1: remove lexically subsumed reports

F2: remove annotated reports

Heuristic Filters:

F3: remove specific library calls

F4: merge common heads

F5: merge common tails

Experimental conclusion

- Our technique:
  - Finds real invalid-thread-access errors
  - Detects more errors as the call graph precision increases
  - Uses BFS to find more errors than other search strategies
  - Reduces likely invalid reports via 5 error filters

Outline

- Problem
- Error detection technique
- Implementation
- Experiments
- ➡ Related work
- Conclusion and future work

Related Work

- **Analyzing and testing GUI applications**

  Guitar [Memon ‘00], Stabilizer [Michail ‘05], Julia [Payet ‘11] …

  Focus on test generation, error predication, and code verification; but does not support finding invalid thread access errors

- **Finding bugs in multithreaded programs**

  Eraser [Savage ‘97], Chord [Naik ‘05], Goldilocks [Elmas ‘07], FastTrack [Flanagan ‘09] …

  Different goals (dataraces + deadlocks), algorithms, and, abstractions

- **Call graph construction algorithms**

  RTA [Bacon ‘97], k-CFA [Might ‘10], TamiFlex [Bodden ‘11] …

  Does not support reflection, or need dynamic information

Outline

- Problem
- Error detection technique
- Implementation
- Experiments
- Related work
- Conclusion and future work

Future Work

- Incorporate dynamic and symbolic analysis
  - Filter out infeasible paths
  - Identify more entry points

- Automatically fix the invalid-thread-access errors
  - Counterexample-guided reasoning
  - Heuristic reasoning

- Unit testing of multithreaded GUI applications
  - Test abstraction
  - Event simulation

Contributions

- A general *technique* to find invalid thread access errors
  - Formulate error detection as a call graph reachability problem
- A tool *implementation* supporting 4 GUI frameworks
  - Available at:

GUI Error Detector

https://guierrordetector.googlecode.com/

- An *evaluation* on 9 subjects shows its usefulness