Finding Errors in Multithreaded GUI Applications

Sai Zhang Hao Lü Michael D. Ernst
Department of Computer Science & Engineering
University of Washington, USA
{szhang, hlv, mernst}@cs.washington.edu

ABSTRACT

To keep a Graphical User Interface (GUI) responsive and active, a GUI application often has a main *UI thread* (or *event dispatching thread*) and spawns separate threads to handle lengthy operations in the background, such as expensive computation, I/O tasks, and network requests. Many GUI frameworks require all GUI objects to be accessed exclusively by the UI thread. If a GUI object is accessed from a non-UI thread, an *invalid thread access* error occurs and the whole application may abort.

This paper presents a general technique to find such *invalid thread access* errors in multithreaded GUI applications. We formulate finding invalid thread access errors as a call graph reachability problem with thread spawning as the sources and GUI object accessing as the sinks. Standard call graph construction algorithms fail to build a good call graph for some modern GUI applications, because of heavy use of reflection. Thus, our technique builds reflection-aware call graphs.

We implemented our technique and instantiated it for four popular Java GUI frameworks: SWT, the Eclipse plugin framework, Swing, and Android. In an evaluation on 9 programs comprising 89273 LOC, our technique found 5 previously-known errors and 5 new ones.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging.
General Terms: Reliability, Experimentation.
Keywords: Static analysis, invalid thread access error.

1. INTRODUCTION

End-user satisfaction depends in part on the responsiveness and robustness of a software application's GUI.

To make the GUI more responsive, GUI applications often spawn separate threads to handle time-consuming operations in the background, such as expensive computation, I/O tasks, and network requests. This permits the GUI to respond to new events even before the lengthy task completes. However, the use of multiple threads enables new types of errors that may compromise robustness. We now discuss a standard programming rule for multithreaded GUI applications, the consequences of violating it, and a technique for statically detecting such violations.

1.1 The Single-GUI-Thread Rule

Many popular GUI frameworks such as Swing [18], SWT [34], Eclipse plugin [9], Android [1], Qt [28], and MacOS Cocoa [22] adopt the *single-GUI-thread rule* for GUI object access:

> All GUI objects, including visual components and data models, must be accessed exclusively from the *event dispatching thread*.

The *event dispatching thread*, also called the *UI thread*, is a single special thread initialized by the GUI framework, where all event-handling code is executed. All code that interacts with GUI objects must also execute on that thread. There are several advantages to the single-GUI-thread rule:

- Concurrency errors, such as races and deadlocks, never occur on GUI objects. GUI developers need not become experts at concurrent programming. Programming the framework itself is also simpler and less error-prone.
- The single-GUI-thread rule incurs less overhead. Otherwise, whenever the framework calls a method that might be implemented in client code (e.g., any non-final public or protected method in a public class), the framework must save its state and release all locks so that the client code can grab locks if necessary. When GUI objects return from the method, the framework must re-grab their locks and restore states. Even applications that do not require concurrent access to the GUI must bear this cost.
- GUI events are dispatched in a predictable order from a single event queue. If the thread scheduler could arbitrarily interleave component changes, then event processing, program comprehension, and testing would be more difficult.

1.2 The Invalid Thread Access Error

The single-GUI-thread rule requires GUI application developers to ensure that all GUI objects are accessed only by the UI thread. If not, an *invalid thread access error* will occur. This may terminate the application — doing so is considered preferable to nondeterministic concurrency errors.

The single-GUI-thread rule can be easily violated, since a spawned non-UI thread often needs to update the GUI after its task is finished. In practice, invalid thread access errors are *frequent*, *severe*, and *hard to debug*.

Take the popular Standard Widget Toolkit (SWT) GUI framework as an example. Invalid thread access is one of the top 3 bugs in developing a SWT application, and is the source of many concurrency bugs [35]. A Google search for "SWTException:Invalid thread access" returns over 11,800 hits, consisting of numerous bug reports, forum posts, and mailing list threads on this problem. Eclipse, the IDE for Java development, is built on top of SWT. Searching for "SWTException:Invalid thread access" in Eclipse's bug repository and discussion forum returns over 2700 bug reports and 350 distinct discussion threads, respectively. We manually studied all 156 distinct *confirmed* bug reports, and found this error has been confirmed in at least 20 distinct Eclipse projects and 40 distinct Eclipse components. Even after over 10 years of active development, a recent release of Eclipse still contains this error (bug id: 333533, reported in January 2011). In addition, the invalid thread access error is severe. It is user-perceivable, it cannot be recovered by the program itself, and it often terminates the whole application. In many circumstances as described in the bug reports, users must

ISSTA '12, July 15-20, 2012, Minneapolis, MN, USA

In class: org.mozilla.gecko.gfx.LayerView

```
68.  public LayerView(Context context, LayerController controller) {
69.      super(context);
         ....
73.      mRenderer = new LayerRenderer(this);
74.      setRenderer(mRenderer);
         ...
     }
```

In Android library class: android.opengl.GLSurfaceView

```
272. public void setRenderer(Renderer renderer) {
         ...
282.     mGLThread = new GLThread(renderer);
283.     mGLThread.start();
     }
```

In class: org.mozilla.gecko.gfx.LayerRenderer

```
220. public void onSurfaceChanged(GL10 gl, int width, int height) {
221.     gl.glViewport(0, 0, width, height);
222.     mView.setViewportSize(new IntSize(width, height));
     }
```

Figure 1: Bug 703256 reported on 11/17/2011 in Fennec (Mozilla Firefox for Android) revision d7fa4814218d. On line 74, `LayerView`'s constructor calls method `setRenderer` which spawns a new thread on line 283. This newly created, non-UI thread calls back method `onSurfaceChanged` that accesses GUI objects on line 222 of `LayerRenderer`, causing an invalid thread access error. Our tool finds this error and generates a report as shown in Figure 2.

restart the application to recover from the error. Furthermore, many reported errors are non-trivial to diagnose and fix. Developers usually need non-local reasoning to find the specific UI interactions that can trigger the bug; it took developers 2 years to fix Eclipse bug 51757 and verify the patch.

The invalid thread access error is not unique to the SWT framework. Other recent GUI frameworks like Android suffer from similar problems. For example, Figure 1 shows a recently-reported bug in the Android version of Mozilla Firefox. This bug is particularly difficult to diagnose, since the code that spawns a new thread inside the `setRenderer` method is in an Android library.

1.3 Finding Invalid Thread Access Errors

To ensure that GUIs behave correctly, developers must prevent or detect invalid thread access errors. Current techniques are not effective.

It is infeasible for testing to cover the enormous space of possible interactions with a GUI. Each sequence of GUI events can result in a different state, and each GUI event may need to be evaluated in all of these states. A software system like Eclipse often has a test suite that achieves fairly high statement coverage, but many paths executed by bug-triggering UI sequences are still not covered.

Stylized coding patterns are also inadequate. One possible rule is to always access GUI objects via asynchronous message passing, to ensure a GUI object is accessed in the UI thread. For example, a developer could have prevented the bug in Figure 1 by wrapping line 222 inside a `post` message-passing method[1], as follows:

In class: org.mozilla.gecko.gfx.LayerRenderer

```
220. public void onSurfaceChanged(GL10 gl, int width, int height) {
221.     gl.glViewport(0, 0, width, height);
         mView.post(new Runnable() {
             public void run() {
222.             mView.setViewportSize(new IntSize(width, height));
             }
         });
     }
```

Such an approach is desirable for accesses from non-UI threads, but it is not necessary for all accesses. In our evaluation on real-world programs, we found that a simple analysis that requires GUI operations to be in a message issued an unacceptable number of warnings. Furthermore, this approach is dangerous for accesses from the UI thread. Asynchronous message passing offers no timing guarantee, so a GUI object may have already been disposed before the message sent to it arrives, causing other bugs.

[1]The `post` method in class `android.widget.View` is a standard way to send asynchronous messages to the UI thread.

```
   org.mozilla.gecko.gfx.LayerView.<init>(Context;LayerController)
-> android.opengl.GLSurfaceView.setRenderer(GLSurfaceView$Renderer;)
-> java.lang.Thread.start()
-> android.opengl.GLSurfaceView$GLThread.run()
-> android.opengl.GLSurfaceView$GLThread.guardedRun()
-> org.mozilla.gecko.gfx.LayerRenderer.onSurfaceChanged(GL10;II)
-> org.mozilla.gecko.gfx.LayerView.setViewportSize(IntSize;)
   ...
-> android.view.ViewRoot.recomputeViewAttributes(View;)
-> android.view.ViewRoot.checkThread()
```

Figure 2: Our tool reports a method call chain that reveals the potential error in Figure 1. → represents the call relationship between methods, and `checkThread` is an Android library method that checks whether the current thread is the event dispatching thread before accessing a GUI object. 8 more methods in the call chain, shown as ". . ." above, are omitted for brevity.

Our approach: static analysis. This paper uses static analysis to find potential GUI errors. Static analysis has two advantages compared to dynamic approaches such as testing. First, a static analysis can explore paths of the program without executing the code, and without the need for a test suite. Second, a static analysis can verify the code: if a sound static analysis reports no warnings, the code is guaranteed to be bug-free.

Our static analysis formulates finding invalid thread access as a call graph reachability problem. Given a call graph, our technique traverses paths from its entry nodes, checking whether any path accesses a GUI object from a non-UI thread. If a suspicious path is found, the static analysis warns of a potential error. The warning is in the form of a method call chain from the starting point. As an example, Figure 2 shows a report produced by our static analysis for the buggy code in Figure 1. This report indicates how a new, non-UI thread is spawned and accesses GUI objects. The generated report allows developers to inspect the method call chain, understand how the error could be triggered, and fix it if it is a real bug.

Our static analysis is independent of the call graph construction algorithm. However, modern GUI applications tend to use reflection, and in the presence of reflection, existing call graph construction algorithms such as RTA [4] and k-CFA [13] fail to build a complete call graph. To alleviate this problem, we present an algorithm to build a reflection-aware call graph, and also compare its usefulness with existing call graph construction algorithms in our experiments.

Static analysis may report false positives due to its conservative nature, or may report multiple warnings that actually correspond to the same error. To address such limitations, we devised a set of error filters to remove likely false positives and redundant warnings. The filters introduce potential unsoundness to our algorithm, but in practice they work well and make our technique more usable.

1.4 Technique Instantiation and Evaluation

We implemented an invalid-thread-access-error detection tool that supports four popular GUI frameworks: SWT, Eclipse plugin framework, Swing, and Android. Swing and SWT are the two dominant GUI frameworks for desktop Java applications. Eclipse is the most widely-used IDE for Java. Android is the #1 platform for mobile applications with market share 56% as of September 2011. Although our technique is applicable to any GUI framework with the single-GUI-thread rule, each framework has its own definition of *UI thread* and *program entry points*. Thus, our implementation is parameterized with respect to those framework-specific parts (see Section 3.1).

Our tool works in an automatic manner and scales to realistic programs. We evaluated our implementation on 9 programs comprising 89273 LOC. The experimental results demonstrate that: our technique is effective (it found 10 real-world errors and produced only 10 false positive warnings); our reflection-aware call graph construction algorithm helps in finding errors in Android applications; and our proposed error filters significantly reduce the number of warnings and thus the programmer effort.

1.5 Contributions

This paper makes the following contributions:

- **Problem.** To the best of our knowledge, we are the first to address the invalid thread access error detection problem for multithreaded GUI applications.
- **Technique.** We formulate finding invalid thread access errors as an elegant call graph reachability problem, and present a general error detection technique. In addition, we use a reflection-aware call graph construction algorithm (Section 2).
- **Implementation.** We implemented our technique and instantiated it for four popular GUI frameworks: SWT, Eclipse plugin, Swing, and Android (Section 3). Our implementation is publicly available at: `http://guierrordetector.googlecode.com`.
- **Evaluation.** We applied our tool to 9 programs from 4 different frameworks, comprising 89273 LOC. The results show the usefulness of the proposed technique (Section 4).

2. TECHNIQUE

We first give a high-level formulation of the problem (Section 2.1), then present the error detection algorithm (Section 2.2). Finally, we show how to construct a reflection-aware call graph (Section 2.3) and filter the error reports (Section 2.4).

2.1 Problem Formulation

This section formulates the problem. We first define *UI thread*, *non-UI threads*, *UI-accessing methods*, *safe UI methods*, and *invalid thread access error*, and state two assumptions we make with regard to error detection.

DEFINITION 1 (UI THREAD). The UI thread is a special thread created by the GUI framework during GUI initialization. After the GUI becomes visible, the UI thread takes charge of the application to handle events from the GUI. It spawns new threads to process lengthy operations in the background.

ASSUMPTION 1. We assume that each multithreaded GUI application has a single UI thread. This is true for applications built on top of GUI frameworks adopting the single-GUI-thread rule. The only exception is that an application may fork a new process to launch another application with its own UI thread. In that case, we require the launched application to be analyzed separately.

DEFINITION 2 (NON-UI THREAD). Any other threads except for the UI thread in a multithreaded GUI application are called non-UI threads.

ASSUMPTION 2. We assume that each non-UI thread is (transitively) spawned by the UI thread. Under this assumption, we ignore all non-UI threads created by the GUI framework before the UI thread has been initialized. That is, we assume all post-initialization GUI work occurs in the UI thread. Once the GUI is visible, the application is driven by events, which are always handled in the UI thread. We believe this assumption is reasonable, since if a non-UI thread spawned during pre-initialization GUI work accesses a GUI object, an exception becomes immediately apparent and the whole application may abort even before the GUI is visible. This is highly unlikely for fielded GUI applications.

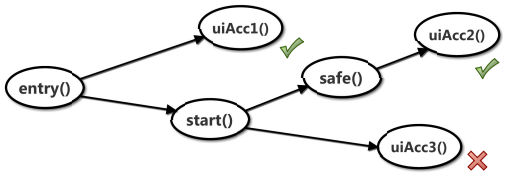

Figure 3: A simple call graph to illustrate the invalid thread access problem. The `entry` method is executed in the UI thread. Nodes `uiAcc1`, `uiAcc2`, and `uiAcc3` are UI-accessing methods. Node `start` is a (non-UI) thread-spawning method, and node `safe` is a safe UI Method. Methods `uiAcc1` and `uiAcc2` are executed in the UI thread, but method `uiAcc3` is executed in a non-UI thread, causing an invalid thread access error.

DEFINITION 3 (UI-ACCESSING METHOD). A method whose execution may read or write a UI object is called a UI-accessing method.

DEFINITION 4 (SAFE UI METHOD). GUI frameworks that adopt the single-GUI-thread rule must provide methods to permit non-UI threads to run code in the UI thread, typically by sending a message to the UI thread. We call such methods *safe UI methods*, since they can be invoked safely by any thread.

DEFINITION 5 (INVALID THREAD ACCESS). An invalid-thread-access error results when the UI thread may spawn a non-UI thread, and there exists a path from the non-UI thread's `start` method to any UI-accessing method without going through any safe UI method.

PROBLEM. At a high level, to detect an invalid thread access error, an analysis needs to track all non-UI threads spawned by the UI thread, and check whether those non-UI threads may invoke a UI-accessing method.

EXAMPLE. Figure 3 shows an example call graph to illustrate the invalid thread access problem.

2.2 Error Detection Algorithm

Figure 4 shows the algorithm for detecting potential invalid thread access errors. Our algorithm uses a static call graph as the program representation for a multithreaded GUI application. A Java call graph represents calling relationships between methods. Specifically, each node represents a method and each edge (f, g) indicates that method f may call method g. A theoretically ideal call graph is the union of the dynamic call graphs over all possible executions of the program. A conservative, or sound, static call graph is a superset of the ideal call graph; it over-approximates the dynamic call graph of every possible execution. Because it is based on the call graph, our algorithm's reports are in terms of methods, as shown in Figure 2.

Our algorithm in Figure 4 first constructs a static call graph for the tested program (line 2), then specifies entry nodes, UI-accessing nodes, and safe UI nodes for it on lines 3, 4, and 5, respectively. Lines 3–5 are GUI-framework-specific. As an example, for a SWT desktop application, the entry nodes include the single main method, UI-accessing nodes include methods `Widget.checkWidget` and `Display.checkDevice`, and safe UI methods include two SWT helper methods `Display.asyncExec` and `Display.syncExec` for message passing. Section 3.1 explains the detailed instantiation for each supported framework.

The algorithm first performs graph traversal to find all reachable `Thread.start()` nodes from each entry node (line 7). Each `Thread.start()` node in a call graph indicates that a new, non-UI thread is spawned. The `Thread.start()` nodes act as the sources in the main graph traversal. The algorithm uses Breadth-First Search (BFS) to search for reachable UI-accessing methods (lines 9–23). The algorithm stops the traversal if it reaches a UI-accessing method (lines 15–17) or a safe UI method (lines 18, 19).

```
Input: a Java program P
Output: a set of potential invalid thread access errors
 1: errors ← ∅
 2: cg ← constructCallGraph(P)
 3: entryNodes ← getEntryNodes(cg)
 4: uiAccessingNodes ← getUIAccessingNodes(cg)
 5: safeUINodes ← getSafeUINodes(cg)
 6: for each entryNode in entryNodes do
 7:    worklist ← getReachableStarts(entryNode)
 8:    visited ← ∅
 9:    while worklist ≠ ∅ do
10:       node ← worklist.dequeue()
11:       if node ∈ visited then
12:          continue
13:       end if
14:       visited ← visited ∪ node
15:       if node ∈ uiAccessingNodes then
16:          newError ← createErrorReport(node)
17:          errors ← errors ∪ newError
18:       else if node ∈ safeUINodes then
19:          continue
20:       else
21:          worklist.enqueueAll(getSuccNodes(cg, node))
22:       end if
23:    end while
24: end for
25: return errors
```

Figure 4: Algorithm for detecting invalid thread access errors in multithreaded GUI programs. Any call graph construction algorithm can be used (line 2). The algorithm is parameterized by the three methods in lines 3–5 which are specific to each GUI framework as described in Section 3.1.

The error message created by the method `createErrorReport` on line 16 includes the method call chain from the entry node to the UI-accessing node as the error report. For brevity, the algorithm does not show the data structures that store the current call chain. The error report in Figure 2 displays a method call chain from the entry node `LayerView.<init>` to the UI-accessing node `ViewRoot.checkThread`.

The algorithm in Figure 4 uses BFS to search for potential error-revealing paths, since BFS always returns the shortest path to the UI-accessing node, permitting smaller error reports. However, other graph search strategies such as Depth-First Search (DFS) or exhaustive path search can also be employed. In our experiment (Section 4.3.4), we empirically compared three different graph search strategies, and demonstrated that using BFS, the algorithm found more errors than DFS and was more practical than exhaustive path search.

2.3 Call Graph Construction

For some modern GUI applications, computing a good call graph in the presence of reflection and native methods is non-trivial. To alleviate this problem, we next present a reflection-aware call graph construction algorithm in Section 2.3.1 and annotation support for native methods in Section 2.3.2.

2.3.1 Reflection

GUI applications built on top of the Android framework use configuration files and reflection to specify GUI layout. As a result, call graph construction algorithms such as RTA [4] and k-CFA [13] fail to build a sufficiently complete call graph. The example code in Figure 5 from an Android application illustrates the limitations. In Figure 5, line 6 uses reflection to create a `Button` object by looking

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

```
1. public class MyActivity extends Activity {
2.    @Override
3.    public void onCreate(Bundle savedInstanceState) {
4.        super.onCreate(savedInstanceState);
5.        setContentView(R.layout.main);
6.        Button button = (Button) findViewById(R.id.button_id);
7.        button.setOnClickListener(new Button.OnClickListener() {
8.            @Override
9.            public void onClick(View v) {
10.               button.setText("Button Clicked.");
11.           }
12.       });
13.   }
14. }
```

Figure 5: Sample GUI application code on the Android platform. The layout XML file (the top) specifies a Button object declaratively, and the Java code (the bottom) first loads the XML file (line 5) and then uses reflection to create a Button object by its ID (line 6, the `findViewById` method).

```
Input: a Java program P
Output: a call graph cg
1: for each expression in P do
2:    if isReflectionCall(expression) then
3:       objectSet ← getAllObjectsThatMayBeCreated(expression)
4:       newExpr ← createObjectCreationExpression(objectSet)
5:       replace expression with newExpr
6:    end if
7: end for
8: cg ← constructCallGraph(P)
9: return cg
```

Figure 6: A reflection-aware call graph construction algorithm. Lines 1–7 are a simple program transformation to replace reflection calls with object creations, and line 8 builds the call graph using an existing call graph construction algorithm. This algorithm is parameterized by two methods on lines 2 and 3. How to instantiate it for the Android framework is presented in Section 2.3.1.

up its id declared in the associated XML file. When this button is clicked, its event handling code (lines 9–11) updates the text.

When analyzing the code in Figure 5, existing call graph algorithms fail to conclude that the variable `button` declared on line 6 points to a *non-null* `Button` object due to their limitations in handling the reflection call `findViewById`. The resulting call graph omits the edge corresponding to the `setText` method call on line 10.

To address this limitation, we present an algorithm to construct a reflection-aware call graph based on a simple program transformation. The basic idea is to replace reflection calls with explicit object creation expressions, pretending that the corresponding concrete object has been created. The algorithm is shown in Figure 6. It consists of two steps. The first step (lines 1–7) is a simple program transformation to replace reflection calls with object creation expressions, and the second step (line 8) uses an existing call graph construction algorithm to build the graph on the transformed program. In our context, a reflection call represents a framework-specific helper method invocation that uses Java reflection to create desirable objects, such as the `findViewById` method in Android applications, instead of the methods in the `java.lang.reflection` package. When it sees a reflection call, the algorithm determines a set of possible objects that might be created. After that, the algorithm creates an expression that non-deterministically returns a new object from the object set.

This algorithm is parameterized by two methods on lines 2 and 3. When instantiated for the Android framework, the predicate isReflectionCall on line 2 returns true if the *expression* is a `findViewById(id)`

method call, and the method getAllObjectsThatMayBeCreated on line 3 parses the associated XML configuration file to extract the class declaration corresponding to the given `id` value. If the `id` value is dynamically generated, this method will conservatively return instances of all subclasses of the declared type.

Take the code in Figure 5 as an example. When the algorithm sees the reflection call `findViewById(R.id.button_id)`, it parses the XML configuration file to determine that the `button_id` value is mapped to a `Button` instance. Then, it replaces the reflection call with an explicit object creation expression: `new Button(null)`. After that, the algorithm employs existing call graph construction algorithms to analyze the transformed program, and permits them to include the call edge `setText` in the resulting graph.

As demonstrated in our experiments, this reflection-aware call graph construction algorithm helps in detecting errors in Android applications (Section 4.3.3).

2.3.2 Native Methods

A GUI application may use native methods to interact with the underlying operating system or platform. Native methods are often beyond the ability of a static analysis but should be considered to make the call graph more complete. To do so, we provide an annotation `@CalledByNativeMethods` for users to specify which native methods may call the current method. For example, the following code snippet indicates that native methods `native1()` and `native2()` may call method `javaMethod()`.

```
@CalledByNativeMethods(callers={"native1", "native2"})
public void javaMethod() { ... }
```

Our static analysis takes the call relationship specified by this annotation into consideration when traversing the call graph.

Adding annotations for native methods is optional and requires manual effort. In our experiments, 1 out of 9 programs (SGTPuzzler) uses native methods. We manually searched the Java source files to find all native methods, inspected the C code to determine possible target methods that may be called by a native method, and added 7 annotations for this program. We found such annotations were useful: one error is only reported when using the user-provided annotations.

2.4 Filtering the Error Reports

A static analysis can check possible error paths without executing the code, but it may report paths that do not actually exist (false positives) or multiple paths that have the same error cause (redundant warnings). We devised 5 error filters to remove likely false positives and redundant warnings. The first 2 filters are sound in that they will not filter real bugs, and the other 3 filters are based on heuristics. Orthogonally, filters 2 and 3 are for reducing false positives, and filters 1, 4, and 5 are for reducing redundant warnings.

1. Filter Lexically Redundant Reports. One reported method call chain can lexically subsume another one. For example, suppose that two reported method call chains, `a()` → `b()` → `c()` and `d()` → `a()` → `b()` → `c()`, both lead to a potential error, since `d()` and `a()` are two distinct entry methods. The second call chain can be removed without missing any errors, since the first chain reveals the same error and is shorter for programmers to interpret.

2. Filter Reports with User-Annotated Methods. Users are permitted to explicitly annotate specific methods that will never trigger an error. Assuming the user annotations are accurate, a reported method call chain containing an annotated method can be safely removed.

In our experiments, we produced a list of 19 annotated methods for two subjects (MyTracks and Fennec), since they employ a customized pattern to interact with the GUI framework. For example, in MyTracks, all non-UI threads are initialized via using the library method `android.os.handler.handleCallback`. This method can be invoked from any thread and will never cause an invalid thread access error, because it checks whether the current thread is the UI thread or not before execution. If not, this method will use safe UI methods to run the code in the UI thread.

3. Filter Reports Containing Library Calls. A method call chain containing certain library calls like `Runtime.shutDown` are unlikely to be buggy. For example, the `Runtime.shutDown` method is called when the JVM terminates, and uses multithreading to dispose all GUI objects when the program exits. Our experiments use a list of 42 such library calls.

4. Filter Reports with the Same Head Methods from the Entry Node to `Thread.start()`. A method can call multiple methods that access GUI objects, such as:

```
public void m() {
    accessUIObject1();
    accessUIObject2();
}
```

If method `m()` is invoked by a non-UI thread, our algorithm will report method call chains that only differ in the last few method nodes, such as:

`a()` → ...`Thread.start()` ... → `m()` → `accessUIObject1()` ...

`a()` → ...`Thread.start()` ... → `m()` → `accessUIObject2()` ...

These two chains may have the same error root cause: knowing that method `m()` is called by a non-UI thread is sufficient to understand the error. This filter compares two reported chains that have the same head methods from the entry node to `Thread.start()`, and removes the longer one.

5. Filter Reports with the Same Tail Methods from `Thread.start()` to the UI-accessing Method. A method can have multiple callers, so method call chains with the same tail are likely to just represent different ways to trigger the same error. This filter compares two reported method call chains that share the same tail from `Thread.start()` to the UI-accessing methods, and removes the longer one. Using filters 4 and 5 together results in only one error per `Thread.start()`.

In our experiments (Section 4.3.5), these filters remove 99.96% of the reported warnings. We also found that using sound filters alone was insufficient, because an overwhelming number of warnings remained. This motivated our use of heuristic filters.

3. IMPLEMENTATION

We implemented an error detection tool on top of the WALA framework [36]. We instantiated the proposed technique for four widely-used GUI frameworks, namely SWT, Eclipse plugin framework, Swing, and Android.

3.1 Instantiation for Different Frameworks

When instantiating our error detection technique for different frameworks, the major framework-specific parts, corresponding to lines 3–5 in Figure 4, are identifying **call graph entry nodes**, **UI-accessing nodes**, and **safe UI methods** for each framework.

3.1.1 SWT

We instantiated our technique for SWT applications as follows:

- **Call graph entry nodes:** the main method. It is executed in the UI thread after the GUI is initialized.
- **UI-accessing nodes:** the `Widget.checkWidget` and `Display.checkDevice` methods. If the current thread is a non-UI thread, these methods throw a `RuntimeException`.
- **Safe UI methods:** `Display.asyncExec` and `Display.syncExec`. These methods execute code (a)synchronously in the UI thread.

3.1.2 Eclipse Plugin

We instantiated our technique for the Eclipse plugin framework as follows:

- **Call graph entry nodes:** all user code methods that override SWT GUI event handling methods. Eclipse calls back the overridden methods to handle the events. All SWT GUI event handling methods (i.e., the overridden methods in a class that implements `org.eclipse.swt.internal.SWTEventListener`) are always called back from the UI thread.
- **UI-accessing nodes** and **safe UI methods** are the same as SWT.

3.1.3 Swing

A Swing application has a single main method, but contains three kinds of threads: the *initial thread* that executes initial application code from the main method, the *UI thread*, where all GUI manipulation code is executed, and the *worker thread* where time-consuming background tasks are executed. After a Swing program starts, its initial thread exits and the UI thread takes charge of the application and starts to execute event-handling code or spawn new worker threads. We instantiated our technique for Swing as follows:

- **Call graph entry nodes:** all user code methods that override Swing GUI event handling methods. Those event handling methods are always called back from the UI thread.
- **UI-accessing nodes:** all methods defined in each Swing GUI class except for three thread-safe methods: `repaint()`, `revalidate()`, and `invalidate()`.
- **Safe UI methods:** `SwingUtilities.invokeLater` and `SwingUtilities.invokeAndWait`, which execute code in the UI thread.

3.1.4 Android

Android is a Java-based platform for embedded or mobile devices. An Android program does not have a single entry point. It uses *activities* (i.e., instances of the `Activity` class) to interact with users through a visual interface and handle GUI events. We instantiated our technique for Android programs as follows:

- **Call graph entry nodes:** in an Android application, an `Activity` object is created and manipulated by the UI thread. Thus, we treat all public methods defined in the `Activity` class and any overriding definitions in its subclasses as call graph entry nodes. We also add all user code methods that override Android GUI event handling methods as entry nodes, since they are called back from the UI thread.
- **UI-accessing nodes:** the `ViewRoot.checkThread` method. If the current thread is a non-UI thread, the `ViewRoot.checkThread` method throws a `RuntimeException`.
- **Safe UI methods:** `View.post` and `View.postDelay`, which execute code in the UI thread.

Given a GUI application using a supported framework, our tool takes its Java bytecode as input, plugs in the corresponding instantiation parameters at runtime, and automatically detects potential invalid thread access errors.

3.2 Android-Specific Implementation Details

We implemented the reflection-aware call graph construction algorithm (Section 2.3.1) using WALA's *bypass logic*. Unlike other tools [26], our tool does not require a separate pass for program instrumentation; instead, it parses the configuration file in an Android application, and then intercepts the call graph construction process on-the-fly to replace all reflection calls with object creation expressions. Since Android applications are often fully encrypted and shipped in Dalvik bytecode as a single apk file, our tool first uses android-apktool [2] to decrypt the apk file, and then uses the ded translator [7] to convert Dalvik bytecode to Java bytecode before feeding to WALA. The Android system library (i.e., `android.jar`) uses many "stub" classes as placeholders for the sake of efficiency. We manually re-compiled `android.jar` from its source code, so it contains real class files rather than stubs.

4. EMPIRICAL EVALUATION

Our experimental objective is three-fold: to demonstrate the effectiveness of our approach in detecting real errors in multithreaded GUI applications, to compare our call graph construction algorithm with existing ones, and to evaluate the usefulness of the proposed error filters.

First, we describe our subject programs (Section 4.1) and the experimental procedure (Section 4.2). We then show that our technique detects bugs in real-world GUI applications (Section 4.3.1). We also compare our technique with a straightforward approach (Section 4.3.2), compare different call graph construction algorithms (Section 4.3.3), and evaluate various graph search strategies in error detection (Section 4.3.4). Finally, we show that the proposed filters are effective in removing warnings (Section 4.3.5).

4.1 Subject Programs

We used 9 open-source projects from SourceForge, Google Code, and the Eclipse plugin marketplace as evaluation subjects.

Five subjects (EclipseRunner, HudsonEclipse, SGTPuzzler, Fennec, and MyTracks) are selected because they have known invalid thread access errors (1 error per subject) and all errors have been fixed in later revisions. We used the buggy versions to check whether our tool can correctly identify those known errors. For the other four subjects, we selected them by first searching for the framework keywords (e.g., "Java Swing" or "Java SWT") in the above source repositories, and then choosing subjects based on the following criteria. First, the subject must be a Java application, not an open library. Second, the subject must use multithreading in its implementation. Third, the subject is listed in the first 5 result pages. This permits us to exclude immature subjects that may contain obvious errors. For each selected subject, we ran our tool on the latest stable release to find new errors.

The subjects used in our experiment (Figure 7) include end-user applications, programming tools, and games.

- **SWT desktop applications.** FileBunker [12] is a file backup application that uses one or more GMail accounts as its backup repository. ArecaBackup [3] offers a local file backup solution for Linux and Windows.
- **Eclipse plugins.** HudsonEclipse [16] monitors Hudson build status from Eclipse. EclipseRunner [10] extends Eclipse's capability of running launch configurations.
- **Swing applications.** S3dropbox [30] allows users to drag and drop files to their Amazon S3 accounts. SudokuSolver [33] computes Sudoku solutions using multithreaded execution.
- **Android applications.** SGTPuzzler [32] is a single-player logic game. Fennec [11], developed by Mozilla, is the Mozilla Firefox web browser for mobile devices. MyTracks [24], developed by Google, records users' GPS tracks, and provides interfaces to visualize them on Google Maps.

4.2 Experimental Procedure

We ran our tool on each subject program with three call graph construction algorithms: RTA [4], 0-CFA, and 1-CFA [13]. When running each call graph construction algorithm on three Android

Program (version)	LOC	Classes	Methods
SWT desktop applications			
FileBunker (1.1.2)	14237	150	1106
ArecaBackup (7.2)	23226	444	4729
Eclipse plugins			
EclipseRunner (1.0.0)	3101	48	354
HudsonEclipse(1.0.9)	11077	74	649
Swing desktop applications			
S3dropbox (1.7)	2353	42	224
SudokuSolver (1.06)	3555	10	62
Android mobile applications			
SGTPuzzler (v9306.11)	2220	16	148
Fennec (d7fa4814218d)	8577	51	620
MyTracks (01d5c1e1cd47)	20297	143	1374
Total	89273	978	9266

GUI framework (version)	LOC	Classes	Methods
SWT (3.6)	129942	999	9643
Eclipse plugin development (3.6.2)	460830	6630	37183
Swing (1.6)	167961	878	13159
Android (3.2)	683289	5085	10584
Total	1442022	13592	70569

Figure 7: Open-source programs used in our evaluation. Column "LOC" is the number of lines of code, as counted by LOCC [21]. Each program is analyzed together with its GUI framework, as listed in the bottom table.

applications, we used two configurations: with and without our enhancements (Section 2.3). We did not use more expensive algorithms like k-CFA ($k > 1$), because they do not scale to our subject programs.

Subject SGTPuzzler uses native methods to interact with the underlying operating system. For it, we manually checked the possible Java methods that a native method may call, and then added 7 `@CalledByNativeMethods` annotations for it. Two subjects (MyTracks and Fennec) use a customized pattern to interact with the GUI framework. For them, we added 19 user-defined filters, described in Section 2.4. In this experiment, none of the paper authors was familiar with the subject programs, but we found it was quite easy to add extra annotations and user-defined filters. All these manual parts took less than a total of 60 minutes.

We manually determined the validity of each warning. For a known error, we compared the generated report (i.e., an error-revealing method call chain as shown in Figure 2) against the actual bug fix to check whether the tool identified the buggy method. For a previously-unknown error, we submitted a new bug report to its developers, and wrote a test driver to reproduce it.

4.3 Results

4.3.1 Errors in Multithreaded GUI Applications

As shown in Figure 8, our tool found errors in each subject program. Using the 1-CFA call graph algorithm, our tool issued 20 warnings, among which 10 warnings reveal 10 distinct errors (5 were previously unknown, and the 5 known errors were correctly identified), 2 warnings are false positives, and the remaining 8 warnings are redundant. Section 4.3.3 compares with other call graph construction algorithms.

We submitted all 5 new errors to the respective developers. As of May 2012, 1 error in S3dropbox has been confirmed, and we have reproduced all the other errors. All found bugs and our experimental results are publicly available at `http://www.cs.washington.edu/homes/szhang/guierror/`.

In class: com.tomczarniecki.s3.gui.DeleteBucketAction

```
59.private void deleteBucket() {
60.    executor.execute(new Runnable() {
61.        public void run() {
62.            try {
63.                controller.deleteCurrentBucket();
64.            } catch (Exception e) {
65.                logger.info("Delete failed", e);
66.                deleteError();
67.            }
68.        }
69.    });
70.}

77.private void deleteError() {
78.    String text = "Cannot delete folder .....";
79.    display.showErrorMessage("Delete failed",
        String.format(text, controller.getSelectedBucketName()));
80.}
```

Figure 9: An invalid thread access error reported by our tool in the S3dropbox Swing application. The error occurs when the method `deleteCurrentBucket` invoked on line 63 throws an exception, which causes method `deleteError` to access a Swing GUI object `display` on line 79 from a non-UI thread. This error was previously unknown, and has been confirmed by the S3dropbox developers.

In class: com.eclipserunner.views.impl.RunnerView

```
179.private void initializeResourceChangeListener() {
180.  ResourcesPlugin.getWorkspace().addResourceChangeListener(
        new IResourceChangeListener() {
181.      public void resourceChanged(IResourceChangeEvent event) {
182.        refresh();
183.      }
184.  }, IResourceChangeEvent.POST_CHANGE);
185.}

414.public void refresh() {
415.  getViewer().refresh();
416.}
```

Figure 10: An invalid thread access error reported by our tool for the EclipseRunner plugin. In Eclipse, the callback method `resourceChanged` on line 181 is invoked by non-UI threads when a `ResourceChangeEvent` happens. However, the `refresh` method directly accesses GUI objects (to refresh the view on line 415) without any protection and thus triggers the error. This error was reported by other users 13 months after the buggy code was checked in, and fixed by developers.

Figure 9 shows an invalid thread access error our tool found in S3dropbox. This error happens when the `deleteCurrentBucket` call on line 63 throws an exception, making it hard to detect by testing. We reported this error to the S3dropbox developers. Tom Czarniecki, a key developer of S3dropbox, confirmed this single-GUI-thread violation. He mentioned that the S3dropbox project uses certain design patterns to avoid such violations (e.g., actions for UI interaction are encapsulated into a `Worker` interface), but the developers still overlooked the error our tool found. Another reason they overlooked this violation is because some GUI frameworks like Swing do not provide any runtime checks for invalid thread accesses. The Swing GUI does not exhibit user-visible faults on some erroneous executions. However, as clearly stated in the official documentation [18], accessing Swing GUI objects from non-UI threads risks thread interference or memory-consistency errors.

Figure 10 shows an error found in the EclipseRunner plugin. This error is event-related. It happens when a `ResourceChangeEvent` happens, which then invokes the `refresh` method on line 415 to update the user interface. In EclipseRunner, the `refresh` method is called by 6 different methods from the same non-UI thread. Thus, our tool issues 6 separate warnings to indicate 6 different ways to trigger this error. 5 of the warnings are redundant.

Subject Program	Our Technique									Requiring Wrappers (Section 4.3.2)
	RTA			0-CFA			1-CFA			
	CG Size	#Warning	#Bug	CG Size	#Warning	#Bug	CG Size	#Warning	#Bug	#Warning
SWT desktop applications										
FileBunker	18951	1	0	15743	0	0	76088	2	1	693
ArecaBackup	20882	1	0	19697	1	1	116398	1	1	3021
Eclipse plugins										
EclipseRunner	11248	6	1	7201	6	1	26911	6	1	202
HudsonEclipse	18473	2	1	15814	2	1	56645	3	1	182
Swing desktop applications										
S3dropbox	37751	0	0	30609	0	0	115324	1	1	210
SudokuSolver	27730	3	2	20907	3	2	39299	2	2	356
Android mobile applications										
SGTPuzzler	13631 / 13865	1 / 16	0 / 0	9546 / 9682	0 / 4	0 / 1	35198 / 35756	0 / 1	0 / 1	104
Fennec	14058 / 14387	1 / 1	0 / 0	8263 / 8898	1 / 1	0 / 0	29125/ 31759	3 / 3	1 / 1	433
MyTracks	24036 / 24036	161 / 220	0 / 0	10803 / 13645	119 / 119	0 / 0	39235 / 110977	1 / 1	0 / 1	1192
Total	186760 / 187323	176 / 250	4 / 4	138583 / 142196	132 / 136	5 / 6	534223 / 609158	19 / 20	8 / 10	6393

Figure 8: Experimental results in finding invalid-thread-access errors in multithreaded GUI programs. Column "CG Size" is the number of nodes in the call graph. Column "#Warning" is the number of warnings issued by our tool. Column "#Bug" shows the actual bugs found. Column groups "RTA", "0-CFA", and "1-CFA" show the results of using different call graph construction algorithms. For the Android applications, a slash "/" separates the result of using standard call graph construction algorithm and our call graph construction algorithm in Section 2.3 (dealing with reflection calls and adding 7 native method annotations for SGTPuzzler). The 5 errors found in FileBunker, ArecaBackup, S3dropbox, and SudokuSolver are previously unknown. As a comparison, the results of the "Requiring Wrappers" approach (Section 4.3.2), are shown at the far right.

Besides the above two examples, other errors our tool reported are also subtle to find. For example, our tool found two new errors in SudokuSolver. One error only happens when the given Sudoku is unsolvable. The other one happens when the program fails to launch the mail-composing window of the user's default mail client (i.e., the `java.awt.Desktop.mail()` method throws an exception after a user clicks the "eMail Me" button).

We found that GUI developers have already used design patterns, runtime checks, and testing to avoid violating the single-GUI-thread rule. However, due to the huge space of possible UI interactions, hard-to-find invalid thread access errors still exist.

Summary. Our technique can find real errors in multithreaded GUI applications with acceptable accuracy.

4.3.2 Comparison to Requiring Wrappers

As mentioned in Section 1.3, one way to prevent invalid thread access errors is to wrap every GUI-accessing operation with message passing (i.e., via the safe UI methods). A wrapper is not always necessary, and indiscriminate wrapping can give rise to other types of errors. Nonetheless, a straightforward and sound way to detect potential invalid thread access errors is to issue a warning whenever a GUI-accessing operation is not wrapped.

This approach identifies every error that our technique found. However, requiring wrappers issues a huge number of warnings, most of which are probably false positives (see the far right column of Figure 8). The primary reason is that this simple approach does not globally reason about the calling relationship between threads, UI-accessing methods, and safe UI methods, and thus it often incorrectly classifies GUI accessing operations which will never be executed in a non-UI thread as erroneous. Furthermore, our technique outputs a method call chain with each reported error, which can help developers understand how an invalid thread access error is triggered.

Summary. Our technique provides richer contextual information for the reported error, and is significantly more precise than requiring each GUI access to be wrapped.

```
void entry() {
  f();
  g1();
}
void f() {
  new Thread(new Runnable() {
    public void run() {
      safe();
    }
  }).start();
}
void safe() {
  uiAcc();
}
void g1() { g2(); }
void g2() {
  new Thread(new Runnable() {
    public void run() {
      uiAcc();
    }
  }).start();
}
```

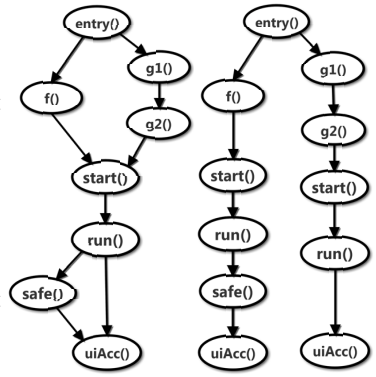

Figure 11: (a) shows example code, in which `safe()` is a Safe UI method and `uiAcc()` is a UI-accessing method. (b) shows a less precise call graph built by RTA or 0-CFA, and (c) shows a more precise call graph built by 1-CFA. Nodes for constructors are omitted for brevity. Using the less precise call graph, our error detection algorithm (Figure 4) reports an invalid method call chain: `entry()` → `f()` → `start()` → `run()` → `uiAcc()`. It does not report the actual error path because it is longer: `entry()` → `g1()` → `g2()` → `start()` → `run()` → `uiAcc()`. The algorithm outputs the actual error path when using the more precise call graph.

4.3.3 Comparing Call Graph Construction Algorithms

We next compare the 3 call graph construction algorithms (RTA, 0-CFA, and 1-CFA) used in our experiments. As shown in Figure 8, 1-CFA found more errors than the other two algorithms. This is because RTA and 0-CFA do not consider the calling context when constructing a call graph: they mix calls to the same method from different callers into a single node, thus introducing imprecision. Although the error paths exist in the less precise graphs, our algorithm does not report them because of its heuristics for outputting the shortest possible call chain. Figure 11 illustrates this point.

The results in Figure 8 also show that using the call graph construction algorithm in Section 2.3 helps in finding errors in Android

applications. One error from the MyTracks Android application can only be found by using the reflection-aware call graph construction algorithm (Section 2.3.1). This is because the error-related GUI object (`msgTextView`) in MyTracks is created reflectively as follows.

In class: com.google.android.apps.mytracks.StatsUtilities

```
97.  public void setLatLong(int id, double d) {
98.    TextView msgTextView = (TextView) activity.findViewById(id);
99.    msgTextView.setText(LAT_LONG_FORMAT.format(d));
100. }
```

Another error in SGTPuzzler is only reported when adding native method annotations (Section 2.3.2).

Summary. Using the 1-CFA algorithm finds more errors than 0-CFA and RTA, and our reflection-aware call graph construction algorithm helps to find errors in some Android applications.

4.3.4 Comparing Graph Search Strategies

The error detection algorithm in Figure 4 uses a separate BFS for each entry node. This section evaluates three variants: a multi-source BFS; a separate DFS for each entry node; and exhaustive search.

The first variant uses a single BFS, but starting at multiple sources. It deletes lines 6 and 24 in Figure 4 and changes line 7 to:

worklist $\leftarrow \bigcup_{entry \in entryNodes}$ getReachableStarts(*entry*)

This variant returns the shortest path from any `Thread.start()` node to any UI-accessing node, rather than the shortest path from each `Thread.start()` node to any UI-accessing node as the algorithm in Figure 4 does. This variant reported 8 errors and 12 false positives. It missed 1 error in S3dropbox and 1 error in SudokuSolver. It visits every node in the call graph only once, rather than potentially once per entry node, but doing so prunes out real error paths.

The second variant uses one DFS per entry node, by changing the *worklist* on line 7 of Figure 4 to a stack, and changing queue operations dequeue and enqueueAll on lines 10 and 21 to pop and pushAll. This variant reported 9 errors and 10 false positives. It missed 1 error in FileBunker. DFS tends to search deeper into the graph and return longer paths that are more likely to be infeasible or to be removed by the filters.

The third variant uses exhaustive search to find potential errors. This variant enumerates all non-cyclic paths from all reachable `Thread.start()` nodes to each UI-accessing node, and then checks whether each path spawns a new thread and accesses GUI objects without using safe UI methods. We ran this variant on each subject program for 1 hour. It explored 5.1×10^9 paths on average, but did not output any potential errors before it terminated. The number of non-cyclic paths in a graph is exponential in the graph size, and it is infeasible to enumerate all paths for a realistic call graph. In our experiments, the smallest call graph contains 7201 nodes and the average out-degree of each node is 2.15. The number of distinct non-cyclic paths is astronomically large; a rough estimate is $2.15^{7201} \approx 1.07 \times 10^{2107}$.

Given a sound call graph, suppose that there exists some error-revealing path between an entry node E and a UI-accessing node U. A sound search strategy is one that reports some error-revealing path between E and U, while an unsound strategy might report a non-error-revealing path between the nodes (Figure 11 shows an example). Exhaustive search is sound because it does not miss any possible (non-cyclic) paths. However, it is impractical. BFS and DFS are unsound because they visit every node, but do not traverse every path to visit that node. Using BFS or DFS can be viewed as a heuristic filtering step, akin to the filters of Section 2.4.

Summary. Our algorithm finds more errors than using multi-source BFS, DFS, or exhaustive search.

Subject Program	Number of Warnings					
	Before Filtering	Sound Filters		Heuristic Filters		
		F_1	$F_{1,2}$	$F_{1,2,3}$	$F_{1,2,3,4}$	$F_{1,2,3,4,5}$
SWT desktop applications						
FileBunker	4494	4494	4494	3210	10	2
ArecaBackup	6219	438	438	438	1	1
Eclipse plugins						
EclipseRunner	1644	1644	1644	1644	6	6
HundsonEclipse	1367	567	567	567	3	3
Swing desktop applications						
S3dropbox	45528	31978	31978	30975	9	1
SudokuSolver	58	58	58	58	2	2
Android mobile applications						
SGTPuzzler	2	1	1	1	1	1
Fennec	122	84	80	80	9	3
MyTracks	1176	1176	483	441	69	1
Total	60610	40440	39753	37414	110	20

Figure 12: Number of warnings after applying a set of sound and heuristic error filters. Column "Before Filtering" shows the number of warnings by the reflection-aware 1-CFA algorithm. Other algorithms show similar patterns, which are omitted for brevity. Column "$F_{i,...,j}$" represents the number of remaining warnings after applying the ith to jth filters as defined in Section 2.4. The numbers in the last column are the same as the subcolumn "#Warning" under column "1-CFA" in Figure 8.

4.3.5 Evaluating Error Filters

Figure 12 measures the effectiveness of the error filters of Section 2.4. The five error filters removed 99.96% of the reported warnings as likely false positives or redundant warnings. Specifically, the two sound filters 1 and 2 removed 34.44% of the warnings, and the three heuristic filters 3, 4, and 5 removed a further 65.52% of the warnings. The most effective filter is #4, for removing reports with the same head methods from the entry node to `Thread.start()`.

Summary. Our proposed error filters are effective in reducing the number of warnings.

4.4 Discussion

Performance and scalability. Our tool has been evaluated on 9 subject programs with 89273 LOC and frameworks with 1.4 MLOC, showing good scalability. Our evaluations were conducted on a 2.67GHz Intel Core PC with 4GB physical memory (1GB is allocated for the JVM), running Windows 7. For the most time-consuming subject, S3dropbox, our tool finished the whole analysis within 252 seconds using the most expensive 1-CFA call graph construction algorithm. Analyzing other subjects or using different algorithms took less time. Call graph construction took 33–91% of the total time.

Threats to validity. There are two major threats to validity in our evaluation. One threat is the degree to which the subject programs used in our experiment are representative of true practice. In our evaluation, we only selected subjects from open-source repositories. Another threat is that we only employed three widely-used call graph construction algorithms (i.e., RTA, 0-CFA, and 1-CFA) in our evaluation. Using other call graph construction algorithms might achieve different results.

Limitations. Our technique is limited in three aspects. First, it only considers non-UI threads that are spawned by the UI-thread after the GUI is initialized, and ignores other possible non-UI threads (quite unusual) that are created during the pre-initialization GUI work. One way to remedy this limitation is to design an analysis to identify those non-UI threads created before a GUI is launched. Second, like

many bug-finding techniques, our technique is neither sound nor complete. It may issue false positives due to the conservative nature of a static analysis. It may miss true positives due to the graph search strategy, and it never reports cyclic paths. Furthermore, for the sake of scalability, our tool implementation uses the default configuration of WALA and ignores part of the AWT library. Thus, it missed one error in S3dropbox which we later found by using pluggable type-checking [8]. Designing better call graph construction algorithms, graph search strategies, and filtering heuristics may alleviate this limitation. Third, our tool cannot compute call relationships for inter-process communication between components. It also requires users to manually add annotations to characterize call relationships that involve native methods. This limitation may lead to false negatives. Investigating the false negative rate is ongoing work.

Experimental Conclusions. Invalid thread access errors can be subtle to detect in many cases. The technique presented in this paper offers a promising solution. Our technique finds real-world errors and issues few false positive warnings. Our proposed filters are useful in reducing the number of warnings.

5. RELATED WORK

Work related to this paper falls into three main categories; (1) analyzing and testing GUI applications; (2) bug-finding techniques for multithreaded programs; and (3) call graph construction algorithms.

5.1 Analyzing and Testing GUI Applications

Automated GUI testing is a challenging task. Various techniques automate GUI testing including test generation [39], test execution [38], and test script repairing [6, 15]. For example, Guitar [38, 39] is a GUI testing framework for Java and Microsoft Windows applications. Yuan and Memon [39] generated event-sequence-based test cases for GUI programs using a structural event generation graph. However, testing is often insufficient to detect many potential errors in a GUI application due to the huge space of possible UI interactions. In contrast, a sound static analysis can explore all paths to find potential errors missed by testing. Compared to software testing, a static analysis tool such as ours may report false positives and redundant warnings due to its conservative nature. In our experiments, simple error filters reduced the number of warnings to an acceptable level.

Michail and Xie [23] proposed a tool-based approach to help users avoid bugs in GUI applications. Their approach monitors a user's actions in the background, and gives a warning as well as the opportunity to abort the action, when a user attempts an action that has led to problems in the past. Their work aims to prevent an existing bug from happening again. By contrast, our work aims to find unknown errors.

Recently, Payet and Spoto [26] presented a static analysis framework for Android programs based on abstract interpretation. Their framework focuses on the Android platform, and consists of 7 existing static analyses such as nullness analysis, class analysis, and termination analysis. However, their framework does not support detecting invalid thread access errors, and uses a quite different abstraction than ours. To the best of our knowledge, we are the first to address the invalid thread access error detection problem for multithreaded GUI applications, and our core technique has been tailored for four GUI frameworks.

5.2 Finding Bugs in Multithreaded Programs

A rich body of techniques have been developed to detect bugs in multithreaded programs [14, 19, 25, 31]. Runtime analysis tools such as Eraser [31] dynamically detect concurrency bugs using lockset algorithms. Static analysis tools such as Chord [25] exploit a key property of Java - namely the scoped use of locks, to further improve the precision of lockset computations. However, finding invalid thread access errors is quite different than detecting data races. A data race occurs when two concurrent threads access a shared variable and when at least one access is a write and the threads use no explicit mechanism to prevent the accesses from being simultaneous. In contrast, an invalid thread access error occurs when a non-UI thread accesses (reads or writes) a GUI object. Unlike detecting data races, finding an invalid thread access error does not require monitoring every shared-memory reference to verify that consistent locking behavior is observed among different threads. A technique only needs to track whether a non-UI thread can accesses a GUI object or not, and is much cheaper. Leveraging data race detection to improve our technique is future work.

An alternative way to find bugs in multithreaded programs is using model checking [19]. By exhaustively exploring the thread scheduling space, a model checker can report counterexamples as bug reports. Unfortunately, due to the exponential size of the search space, it is hard for model checking approaches to scale to a realistic multithreaded GUI application without compromising the error detection capability. We are not aware of any software model checking approach that scales to programs as large as those used in our experiments (including the library code). The technique presented in this paper is specifically designed to find invalid thread errors instead of being a general property checking tool. It chooses the call graph as a coarse-grained program representation with a set of error filters, to achieve good scalability with reasonable accuracy.

5.3 Call Graph Construction Algorithms

Call graph construction algorithms have been well studied in the literature. However, standard algorithms such as RTA [4] and k-CFA [13] do not build a sound call graph in the presence of reflection. As reflected in our experiments, using standard call graph algorithms misses errors in some Android applications. Livshits et al. [20] presented a static analysis to reason about reflective calls. The analysis attempts to infer additional information stored in string constants to resolve reflective calls statically. Their approach focuses on standard Java reflection calls (e.g., `Class.forName`) instead of framework-specific ones (e.g., `View.findViewById`). TamiFlex [5], a pure dynamic approach, records all reflectively-created class instances by intercepting JVM system calls, and re-inserts those recorded class into a program. However, TamiFlex requires a set of representative program executions and is only sound with respect to the given executions. Perhaps the closest work to our call graph construction algorithm is Payet and Spoto's Julia system [26]. The Julia system needs to first instrument Android's library code that performs the XML inflation, and then replaces the `findViewById` call with the corresponding object creation expressions. In addition, the Julia system does not handle native methods when building call graphs. In contrast, our technique provides annotation support for native methods, and our tool does not need a separate pass of off-line instrumentation. The reflection-aware call graph is created online by intercepting the standard call graph construction process.

6. CONCLUSION AND FUTURE WORK

This paper presented a simple, general, effective technique to find invalid thread access errors in multithreaded GUI applications. Our technique statically explores paths in a call graph to check whether a non-UI thread can access a GUI object. It uses a reflection-aware call graph construction algorithm to build a good call graph, and employs a set of error filters to filter likely false positives and redundant warnings. We demonstrated the usefulness of our technique by evaluating it on 9 programs built on 4 popular GUI frameworks.

The source code of our tool implementation is available at: http://guierrordetector.googlecode.com

Besides general issues such as performance and ease of use, our future work will concentrate on the following topics:

Integration with dynamic and symbolic analyses. The technique presented in this paper is a call-graph-based, pure static analysis. It suffers from false positives for many GUI applications, due to the conservative nature of static analysis. A possible way to reduce the number of false positives is to integrate with dynamic analyses [40] or symbolic analyses [27, 37] by using more accurate information to guide call graph exploration.

Unit testing multithreaded GUI programs. Besides a static analysis, software testing is another way to improve software quality. Although many testing techniques have been developed recently, few of them can be applied to unit test multithreaded GUI programs to find potential errors *earlier*. We plan to investigate how to apply recent advance in automated testing [17, 40] to the context of multithreaded GUI applications.

Fixing potential GUI errors. After an error is revealed, fixing it and verifying the patch is often time-consuming. Fixing concurrency bugs has become especially critical in the multicore era. Recently, work has been done on automatically repairing test scripts for GUI applications [6, 15]. However, none of them focuses on repairing the GUI program to patch a revealed error. Thus, we are interested in developing automated error fixing techniques for multithreaded GUI applications.

Generalization. We have applied our graph reachability analysis to one specific, but important problem. There is a rich history of testing and analysis problems being cast as reachability (e.g., [29]). In future work, we plan to formulate other problems, such as detecting security vulnerabilities, as path reachability, and to apply our scalable analysis to them.

7. ACKNOWLEDGEMENTS

We thank Stephen Fink and Manu Sridharan for answering our questions about WALA. This work was supported in part by ABB Corporation and NSF grant CCF-1016701.

8. REFERENCES

[1] Android website. http://www.android.com/.

[2] Android-Apktool. http://code.google.com/p/android-apktool/.

[3] ArecaBackup. http://www.areca-backup.org/.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA*, 1996.

[5] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE*, 2011.

[6] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè. Automated GUI refactoring and test script repair. In *ETSE*, 2011.

[7] Ded Decompiler. http://siis.cse.psu.edu/ded/.

[8] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE*, 2011.

[9] The Eclipse IDE for Java. http://eclipse.org/.

[10] EclipseRunner. http://andrei.gmxhome.de/filesync/.

[11] Fennec. http://www.mozilla.org/en-US/mobile/.

[12] FileBunker. http://filebunker.sourceforge.net/.

[13] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *OOPSLA*, 1997.

[14] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *Proc. ISSTA '11*, 2011.

[15] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proc. ICST*, 2010.

[16] HundsonEclipse. http://code.google.com/p/hudson-eclipse/.

[17] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *FSE*, 2011.

[18] JDK Swing Framework. http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[19] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009.

[20] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proc. APLAS '05*, 2005.

[21] LOCC. http://csdl.ics.hawaii.edu/Plone/research/locc/.

[22] Macos Cocoa. http://developer.apple.com/technologies/mac/cocoa.html.

[23] A. Michail and T. Xie. Helping users avoid bugs in GUI applications. In *Proc. ICSE*, May 2005.

[24] MyTracks. http://code.google.com/p/mytracks.

[25] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. PLDI*, 2006.

[26] E. Payet and F. Spoto. Static analysis of Android programs. In *Proc. CADE' 11*, 2011.

[27] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA*, 2011.

[28] The Qt Framework. http://qt.nokia.com/products/.

[29] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[30] S3dropbox. http://s3dropbox.googlecode.com.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *Proc. SOSP*, 1997.

[32] SGTPuzzles. http://chris.boyle.name/projects/android-puzzles.

[33] SudokuSolver. http://sudokupuzzlesol.sourceforge.net/.

[34] The SWT toolkit. http://eclipse.org/swt/.

[35] Top 3 SWT exceptions. Presentation by Lakshmi P Shanmugam, Eclipse SWT team at Eclipse Day India. http://www.slideshare.net/lakshmip/top-3-swt-exceptions-3951224.

[36] WALA. http://wala.sourceforge.net.

[37] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS*, 2005.

[38] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *IEEE TSE*, 37(4), 2011.

[39] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proc. ICSE*, 2007.

[40] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proc. ISSTA*, 2011.