A Lightweight Approach to Component-Level Exception Mechanism for Robust Android Apps

Kwanghoon Choi[1]

Computer & Telecommunication Engineering Division, Yonsei University, Wonju, Gangwon-Do, Korea, 26493

Byeong-Mo Chang[2]

Department of Computer Science, Sookmyung Women's University, Seoul, Korea, 04310

Abstract

Recent researches have reported that Android programs are vulnerable to unexpected exceptions. This is because the current design of Android platform solely depends on Java exception mechanism, which is unaware of the component-based structure of Android programs. This paper proposes a component-level exception mechanism for programmers to build robust Android programs with. With the mechanism, they can define an intra-component handler for each component to recover from exceptions, and they can propagate uncaught exceptions to caller component along the reverse of component activation flow. Theoretically, we have formalized an Android semantics with exceptions to prove the robustness property of the mechanism. In practice, we have implemented the mechanism with a domain-specific library that extends existing Android components. This lightweight approach does not demand the change of the Android platform. In our experiment with Android benchmark programs, the library is found to catch a number of runtime exceptions that would otherwise get the programs terminated abnormally. We also measure the overhead of using the library to show that it is very small. Our proposal is a new mechanism for defending Android programs from unexpected exceptions.

Keywords: Android, Java, Exception, Component, Semantics

1. Introduction

Exception handling in Java is an important feature to improve the robustness of Java programs. For example, Figure 1 shows a simplified Java program that may throw one of two exceptions, `NoSuchOperator` and `ArithmeticException` (*"divide by zero"*), when users try to do a calculation with an unsupported operator or with zero as a divisor. Once the *calc* method throws such an exception, it is propagated along the call stack to a caller, the *main* method, where it is handled by the catch block.

Email addresses: `kwanghoon.choi@yonsei.ac.kr` (Kwanghoon Choi), `chang@sookmyung.ac.kr` (Byeong-Mo Chang)

[1]This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2014R1A1A2053446).

[2]This research was supported by the Sookmyung Women's University (Research Grants 1-1403-0212).

```
class Calculator {
 void main() {
    int a, b, r;  char op;
    // read a, op, and b
    try {
       r = calc(a, op, b);
       // display the result
    }
    catch(NoSuchOperator e) {
       // Not support the operator
    }
 }
```

```
 int calc(int a,char op,int b) {
   if(op=='+') return a+b;
   else if(op=='-') return a-b;
   else if(op=='*') return a*b;
   else if(op=='/') return a/b;
   else
     throw new NoSuchOperator();
 }
}
```

Figure 1: A Java Program Using Exceptions

Many Android programs are written in Java with Android APIs, presumably using exception handling. Android is Google's open-source platform for mobile devices, and it provides the APIs (Application Programming Interfaces) necessary to develop applications for the platform in Java (http://developer.android.com). An Android program consists of *components* such as activities, services, broadcast receivers and content providers. Android components communicate with another by sending messages called *Intents*. For example, an activity can start other activities by sending Intents to Android platform, which invokes methods of callee activities.

How vulnerable Android components are due to Intents have been reported by experiments in [1, 2, 3, 4]. With Intent fuzzing, they generated random and semi-valid intents to test how components react to these exceptional conditions, focusing on uncaught exceptions that result in the crashes. One experiment by [2] measured the number of failed components for various types of components, reporting that 29(8.7%) out of total 332 Activities crash with generated semi-valid intents. The distribution of exception types are also measured to understand how components fail due to uncaught exceptions, showing that `NullPointerException` makes up the largest share of all the exceptions, and other exceptions like `ClassNotFoundException` and `IllegalArgumentException` are next significant ones.

We have also found by examining source code of programs that Android programs can be very vulnerable to exceptions. We examined 9 programs and found that 41 activities (51%) out of total 80 activities have no exception handlers like try-catch, as will be shown in our experiment later. Activities without exception handlers cannot handle any thrown exceptions, and so result in the crashes when any exceptions are thrown.

From these observations, we can be sure that it is necessary for developing more robust Android programs to handle uncaught exceptions from components. Currently, programmers only resort to the conventional Java exceptions: the try-catch construct to defend statements and the thread-level uncaught exception handler interface (`Thread.UncaughtExceptionHandler`) to catch exceptions escaping from a thread. They are still crucial for Android programs, but they only address too fine-grained level in statements or too coarse-grained level in a thread. They are not immediately useful for addressing defending Android components.

Our design of *component-level exception mechanism* naturally follows that of the conventional Java exceptions of separating error-handling code from "regular" code and of propagating errors up the call stack. First of all, our mechanism is designed for each Android component to have a designated "catch" facility to defend itself from any (unexpected) uncaught exceptions thrown by the "regular" code of the component. This feature will allow programmers to focus more on the main flow of components, never missing any exceptions attempting to escape the components. Second, our mechanism is designed to support the propagation of exceptions following up a component activation stack. This facility will make components more resilient even by catching exceptions propagated from other components. Particularly, many components in Android programs have a relationship on *"who activates whom"*, which is very similar to a caller-callee relationship in method invocation. Also, Android platform already has an activity stack internally, which is the same as a call stack of method invocation, to maintain the who-activates-whom relationship.

In this paper, we propose a mechanism for component-level exception handling and propagation in Android programs, which can be used to make them more robust by defending Android programs from unexpected events. We take a lightweight approach by providing new component APIs (e.g., ExceptionActivity class), which extends the existing Android components (e.g., Activity class) with the component-level exception mechanism. No Android platform needs to be modified to use our approach. Programmers can utilize component-level exception handling and propagation by writing components with the new extended APIs. This use of the exception mechanism preserves the structure of classes and methods in original programs. Our approach is also flexible in that programmers can take full control of deciding which components handle what exceptions and how they are recovered.

Following an overview of Android programs and our motivation in Section 2, we present our idea of the Android component-level exception mechanism in Section 3. We give a theoretical account on the mechanism by an Android semantics with exceptions to prove the robustness of the mechanism in Section 4. We also perform experiments in practice to show that Android programs can be more robust with the new API in Section 5. We count how many exceptions are caught with the new API. We also measure the marginal cost of the mechanism by changed lines of code, increased binary size, and startup time due to the adoption of the mechanism. Finally, after discussing related work in Section 6, we conclude in Section 7.

2. Motivation

An Android program is a Java program with APIs in Android platform. Using the APIs, one can build user interfaces to make a phone call, play a game, and so on. An Android program consists of components whose types are Activity, Service, Broadcast Receiver, or Content Provider. Activity is a foreground process equipped with windows such as buttons and text inputs. Service is responsible for background jobs, and so it has no user interface. Broadcast Receiver reacts to system-wide events such as notifying low power battery or SMS arrival. Content Provider supports various kinds of storage including database management systems.

Components in an Android program interact with each other by sending messages called *Intent* in Android platform. An Intent holds information about a target component to which it will be delivered, and it may hold data together. For example, a user interface screen provided by an activity changes to another by sending an Intent to Android platform, which will pause the UI screen and will launch a new screen displayed by a target activity specified in the Intent.

Currently, Android programs defend themselves against any exceptions only by the same ways as what plain Java programs do with. One is by a try-catch statement, and the other is by a thread-level uncaught exception handler (`Thread.UncaughtExceptionHandler`). The two conventional methods are still crucial for Android programs, but they are only useful for defending too fine-grained level in statements or too coarse-grained level in a thread. They do not immediately address defending components, which are an important aspect of Android

```
class Main extends Activity {
  void onCreate() {
    addTextInput(1); // 1st operand
    addTextInput(2); // operator
    addTextInput(3); // 2nd operand
    addButton(1); // (=) button
    /* initialize the main screen */
  }
  void onClick(int button) {
    int a, b;    char op;
    // read a,op,and b
    Intent i = new Intent();
    i.setTarget("Calc");
    // put a, op, and b into i
    i.setArg( ... a, op, b ... );
    try { startActivityForResult(i); }
    catch(NoSuchOperator exn)
       { /* handle the exception */ }
  }
  void onActivityResult(int resultCode,
                          Intent i) {
    if (resultCode == RESULT_OK) {
       // read r from i
       int r = ... i.getArg() ... ;
       // display the result
    }
  }
}
```

```
class Calc extends Activity {
  void onCreate() {
    addButton(1); // goback button
    // display the button
  }
  void onClick(int button) {
    int a, b, r;
    char op;
    Intent i = getIntent();
    // get a, op, b from i
    ... i.getArg() ... ;
    if (op=='+') r=a+b;
    else if (op=='-') r=a-b;
    else if (op=='*') r=a*b;
    else if (op=='/') r=a/b;
    else
      throw new NoSuchOperator();
    // put r into i
    i.setArg( ... r ... );
    // dismiss this activity
    // with i as a return value
    finish(RESULT_OK, i);
  }
}
```

Figure 2: An Android Calculator Program with Useless Exception Handling (in the onClick method of Main)

programs. Note that an Android program typically runs in a thread, making interleaved execution of several components.

What mechanism do we need to provide to defend Android components against any exceptions? The lessons from the conventional Java exception mechanism [5] can shed light on this question. A notable advantage is that Java exceptions allow to separate error-handling code from "regular" code cleanly. Similarly, every Android component may be required to have its own "catch" block that enables programmers to write the main flow of the component code and to deal with the exceptional cases elsewhere. For example, exceptions such as `NullPointerException` can be thrown by almost any pieces of codes in the component, but only one handler may be written in such a catch block of the component, rather than having a separate conditional statement or handler for each piece of the codes.

Also similarly as using Java exceptions allows to propagate errors up the call stack automatically, Android components may be required to have an automatic mechanism on propagating exceptions up a component activation stack. For example, a callee Android component may propagate any exceptions to its caller component automatically when they are not handled by itself. Clearly, this will make more chances to defend Android components from any exceptions thrown by their callee components. Without such a propagation mechanism, programmers may have to write to return error codes between callee and caller components, which will be very tedious and likely to make new errors particularly when an activation chain is long.

Now we have identified two things. First, to make Android programs robust, there exists a new level of exception mechanism in terms of components in addition to the conventional Java exception mechanism. Second, to support a component-level exception mechanism, each component should provide a convenient way to catch any exceptions uncaught inside itself, which we call *intra-component exception handling*, and each caller component should also be able to handle any exceptions propagated from callee ones, which we call *inter-component exception handling*.

However, the current design of Android platform does not support such a component-level exception mechanism. To illustrate such a problem, Figure 2 presents an Android calculator program with two activities Main and Calc. In the example, there is no component-level "catch" dedicated to each activity, and there is no convenient construct enabling the caller activity Main to catch exceptions propagated from the callee activity Calc. Later, we will formalize this problem by a semantics for Android activities and exceptions in Section 4.

In Figure 2, Activity is a class that represents a screen in Android platform, and Main and Calc extending Activity are also classes representing screens. Initially, Android platform creates a Main object, it invokes the *onCreate* method to add three text input windows and one button with integer identifiers as arguments. After entering two integers and one operator, a user clicks the button. And then the *onClick* method is invoked to perform some action for the button. The new Intent specifies the name of an activity class that represents the new screen and some data passed to the callee. The *onClick* method sets *"Calc"* and values from the text input windows in the new Intent object, and then it requests launching by invoking *startActivityForResult*. Android accepts the request and changes the current UI screen by stacking Calc on Main, calling Calc's *onCreate* method, which setups a button to return the calculation result back to Main. On pressing the button, Android invokes Calc's *onClick* method, which gets the operator and operands from the Intent (obtained by *getIntent()*), calculates it to set the result to the Intent, and dismisses Calc. Then Main appears again, and Android invokes the *onActivityResult* method to pass `RESULT_OK` as *resultCode* and the intent with the result as *i*, and to display it.

As well as the normal execution flow as explained above, the exceptional execution flows might happen. When a user enters other than the four arithmetic operators, Calc's *onClick* method will throw an exception, `NoSuchOperator`. When a user enters, say, "1 / 0", the division in the method will throw `ArithmeticException` (*"divide-by-zero"*). In such exceptional cases, the exceptions thrown by the Calc's *onClick* method will be propagated to Android platform who invoked the method, not to (the *onClick* method of) Main who activated Calc. Android platform then catches the exceptions, but it has nothing to do sensibly but stops the execution abnormally. Hence, the exceptions will never reach the try-catch block surrounding the invocation *startActivityForResult(i)* in the Main's *onClick* method. This explains why the conventional exception handling is not effective for the Android component-based structure.

This is a limitation of the design of Android platform solely depending on the Java exception semantics. One might introduce a try-catch block surrounding the if statements in the Calc's *onClick* method to handle both of the exceptions. This can be a very tedious work because we have to search all potentially vulnerable codes. We might not be able to find all such codes in advance. Even if we did do so, the main flow of a component would be mixed with the exceptional flow. Also, due to the separate roles of activities, one could not proceed further, for example, for Calc to get alternative inputs without going back to Main. This limitation motivates us to propose a component-level exception mechanism.

3. A Component-Level Exception Mechanism

We propose a new mechanism to provide a method named *Catch* to handle intra-component exceptions, and to propagate uncaught exceptions to its caller along the activation stack, which we call inter-component exceptions. Figure 3 shows an example of the mechanism. Suppose Activity 1 starts Activity 2, which starts Activity 3. When an exception is thrown in Activity 3, it is passed to its *Catch* method. If it is not handled in the *Catch* method, then it is propagated to its caller Activity 2. If it is not handled in the *Catch* method of Activity 2, then it is propagated to Activity 1.

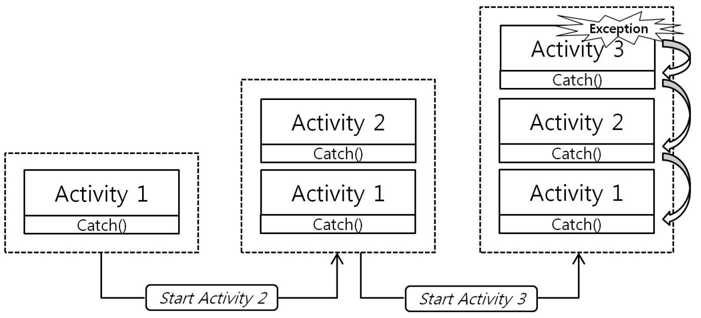

Figure 3: Component-Level Exception Handing and Propagation along the Activity Stack

We design a domain-specific library for the component-level exception mechanism by extending Activity APIs, as in Figure 4, which we call *CE* library. Basically, the library introduces new component classes (e.g., ExceptionActivity) by extending the existing ones (e.g., Activity). Users of this library must write one's own Activities by extending the new component class ExceptionActivity as in Figure 5. The new component classes have a method named *Catch* as a default intra-component exception handler, which Android programs can override to define their own handler. For example, Figure 4 shows ExceptionActivity's *Catch* method that takes an exception as an argument and returns `false`, which means the exception is not handled. In Figure 5, Calc class overrides the method to handle `ArithmeticException`.

• Inter-Component *Try-Catch* Construct: As well as having a single *Catch* method in each component to catch all intra-component exceptions, we can define an exception handler for each activation of components by *TryActivityForResult(intent, catcher)*, a substitute method of *startActivityForResult(intent)* in the CE library, as follows:

```
TryActivityForResult(intent, new Catch() {
  boolean handle(Throwable exn) {
    if (exn instanceof E) {
        // handles the exception E
        return true;
    }
    else return false;
  }
});
```

where Catch is a Java interface in Figure 4 for building a *catcher* object. The *handle* method returns `true` if the exception is processed. Otherwise it returns `false` to re-throw the exception.

```
class ExceptionActivity extends Activity {
  void onCreate() {
    try { OnCreate(); }
    catch(Throwable exn)
     { Throw(exn); }
  }
  void OnCreate() { }
  void onActivityResult
          (int resultCode ,Intent i) {
    if (resultCode == RESULT_OK) {
      try {
       OnActivityResult(resultCode , i);
      }
      catch(Throwable exn)
        { Throw(exn); }
    }
    else { // resultCode == RESULT_EXN
      Throwable exn = (Throwable)i.getData();
      try {
       if (catcher == null ||
            catcher.handle(exn)==false )
              throw exn;
      }
      catch(Throwable unhandled_exn)
        { Throw(unhandled_exn); }
    }
  }
  void OnActivityResult
    (int resultCode ,Intent i) { }
  void onClick(View v) {
    try { OnClick(v); }
    catch(Throwable exn)
     { Throw(exn); }
  }
  void OnClick(View v) { }
  void TryActivityForResult
          (Intent i, Catch c) {
    // set c as a catcher
    catcher = c;
    this.startActivityForResult(i);
  }
  void Throw(Throwable exn) {
    try {
      if (Catch(exn)==false)
        throw exn;
    }
    catch(Throwable exn) {
      // return to the caller
      Intent i = new Intent();
      // put exn into the intent i
      i.setData(exn);
      // dismiss this activity
      // with i as a return value
      finish(RESULT_EXN, i);
    }
  }
  boolean Catch(Throwable exn)
    { return false; }

  Catch catcher;
}

interface Catch {
  boolean handle(Throwable exn);
}
```

Figure 4: The Component-Level Exception Library for Activity

• Overriding Android Interface Methods: The CE library puts a fence at each border between each Android component and the platform to catch all exceptions from the component. For example, the *onCreate* method of ExceptionActivity in Figure 4 invokes its counterpart method *OnCreate*, which is surrounded by a try-catch block (i.e., a fence) to catch all uncaught exceptions from the method and to propagate them to the caller activity. Android programs are supposed to override the counterpart method (*OnCreate*) to be called by the interface method (*onCreate* of ExceptionActivity), which Android platform invokes. The other interface methods such as *onClick* and *onActivityResult* have the similar fence structure.

For example, a calculator program in Figure 5 is obtained by rewriting with the library the previous Android example program in Figure 2. Both Main and Calc classes now have counterpart methods *OnCreate* and *OnClick* instead of the original interface methods *onCreate* and *onClick*. Whenever any of the counterpart methods in Android programs throws an exception, the surrounding try-catch construct of the corresponding interface method in ExceptionActivity will catch it and propagate it by invoking *"Throw(exn)"* in the catch block of the interface method in ExceptionActivity.

```
class Main extends ExceptionActivity {
   void OnCreate() { /* the same code as in Fig.2 */ }
   void OnClick(int button) {
      /* the same code as before the try-catch block in Fig.2 */
      TryActivityForResult(i, new Catch() {
         boolean handle(Throwable exn) {
            if (exn instanceof NoSuchOperator) {
               // handle the unsupported operator
               return true;
            }
            else return false;
         }});
   }
   void OnActivityResult(int resultCode, Intent i) {
      /* the same code as in Fig.2 */
   }
}
class Calc extends ExceptionActivity {
   void OnCreate() { /* the same code as in Fig.2 */ }
   void OnClick(int button) { /* the same code as in Fig.2 */ }
   boolean Catch(Throwable exn) {
      if (exn instanceof ArithmeticException) {
        // handle the divide-by-zero exception
        return true;
      }
      else return false;
   }
}
```

Figure 5: Rewriting the Calculator Program in Fig. 2 with the Component-Level Exception Library

• Inter-Component *Throw* Construct: The CE library defines a (private) method "*Throw(exn)*" as shown in Figure 4. It first passes an exception to the *Catch* method dedicated to intra-component exception handling. If the *Catch* method does not handle the exception, it then propagates the exception to a caller component via the standard Android return mechanism. For implementing this inter-component exception propagation, the library packages the exception into an Intent to set as a return value (*"Intent i = new Intent(); i.setData(exn); finish(RESULT_EXN,i);"*). Note that Intent class is assumed to have a field *data*, which holds an exception. In Appendix (Figure 8), a definition of Intent class is available. Also note that, to distinguish exceptional component results from normal ones, we tags the results with a result code, `RESULT_EXN`. Normal results are tagged with `RESULT_OK`. And then this component is dismissed (say, by invoking *finish* in ExceptionActivity). After the callee is finished with the intent holding the exception, Android platform will invoke the caller's *onActivityResult* method to receive the intent.

On a caller's receiving a result, the caller examines the result code in the *onActivityResult* method (of ExceptionActivity) to decide what to do next. If it is a normal result, an *OnActivityResult* method is invoked. If it is an exception, the catcher (exception handler) of this activation will try to handle the exception. When the catcher succeeds in the exception handling, we get back to the normal state. Otherwise, we re-throw the exception by *Throw* method.

Note that there may be no more (say, *ExceptionActivity* extended) caller. In such a situation, the CE library is designed to stop inter-component exception propagation by finishing the last activity. One may opt to display a user a dialog window to notice the end of the exception propagation and the termination of this Android program.

To get the benefit from *ExceptionActivity*, we need to rewrite an Android program into one using it. A basic transformation for a default recovery from any abnormal termination of an Android program is presented by Definition 1. For more fine-grained recovery, a programmer should extend the basic transformation by overriding a *Catch* method for intra-component exception handling of an activity of one's interest, or by installing an exception handler *catcher* for inter-component exception handling of an activity by $TryActivityForResult(intent, catcher)$. Otherwise, by default, *Catch* is defined to return $false$ and *catcher* is set to *null*, meaning not handling any exception but just propagating it.

Definition 1 (A Basic Android Exception Transformation)**.** *For an Android program, we build a new one by applying the following rules:*

- *Every occurrence of Activity is replaced with ExceptionActivity.*
- *For each class declaration* class *C* extends *D where D is Activity or is inherited from it, the declared methods of $onCreate$, $onClick$, and $onActivityResult$ are renamed as $OnCreate$, $OnClick$, and $OnActivityResult$, respectively.*
- *Every occurrence of an invocation in the form of $this.startActivityForResult(intent)$ is replaced with $this.TryActivityForResult(intent, null)$ when this points to an object of Activity class or its decendant.*
- *The others remain unchanged.*

Figure 5 shows an example of how our component-level exception library is used to improve the robustness of the Android example program in Figure 2. The library allows to handle `ArithmeticException` thrown by the *OnClick* method in Calc (when users enter, say, "1/0") by the intra-component exception handler (the *Catch* method in Calc). Also, the library helps to propagate `NoSuchOperator` exception (thrown by the same *OnClick* method when users enter other than the four arithmetic operators) to the inter-component exception handler (the *handle* method of a catcher created at the *TryActivityForResult* method invocation in Main). After the exceptions are handled, the program gets back to a normal state.

We have only described an exception mechanism extending Activity. The same idea can be applied to Fragment (which is a small detachable Activity). We also developed an exception mechanism for Service, Broadcast Receiver, and Content Provider as well. So, they are equipped with intra-component exception handling by the same idea of overriding interface methods of each component. These three types of components, however, have one's own life-cycle, not always following the who-activates-whom relationship, hence we will use the inter-component exception propagation for the components only when domain knowledge on activation flows is available.

Besides the improvement of robustness, our proposal has a few advantages. First, our approach is lightweight, demanding no change on Android platform, and so all commercial Android devices can get benefit from it immediately. It even allows the mixed uses of Activity and ExceptionActivity. Second, using the CE library does not change the structure of the classes and methods at all, as shown by comparing the two programs in Figure 2 and 5. Third, programmers can take full control of component-level exception handling and propagation by making use of *Catch* method of ExceptionActivity and Catch interface.

Using the CE library incurs only a little overhead as will be shown by our experiments later. The costs are threefold; efforts to rewrite programs with the library, increased binary sizes linked with it, and increased startup time due to the extra size and the exception handling layer. The experiment will show our approach is very effective enough to catch a number of runtime exceptions but only with a little costs.

Our CE library only concerns Java-based Android programs now. One can mix Java with other programming languages such as JavaScript or C/C++ to build Android programs. how to propagate exceptions across codes written in different languages is beyond our topic, but it will be interesting to research on exceptional interaction of Android-Java with, for example, JavaScript, extensively used for cross-platform development.

The proposed implementation works reasonably well with the component-based structure in Android programs. In the structure, for example, activities in a single program are loosely coupled by intents, and activities in two different programs can be seamlessly activated by one another as they are in the same program. Android components may be more tightly coupled with each other when any caller component has to know the kinds of exceptions thrown by its callee components. Programmers may need to have a trade-off in this respect. Nonetheless, such a tight coupling between components may be mitigated by catching an exception based on its group or general type rather than a very specific exception, according to the conventional grouping of Java exceptions by class hierarchy. For example, one can specify `IOException` to represent any type of error that can occur when performing I/O. Even when no knowledge on potential exceptions is available, one can specify *Throwable*, which is the root class of all Java exceptions. In addition, the usefulness of exception handling among components has been reported similarly by other researches [6, 7, 8, 9], though they have dealt with server systems, which will be discussed later.

4. A Formal Semantics for Android-Java with Exceptions and Its Theoretical Properties

In this section, we give a theoretical account for the robustness property of Android programs using the component-level exception mechanism, which we explained by example in the previous section. We first present an Android semantics, which is based on an imperative version of the featherweight Java [10, 11] extended with Java exceptions. The semantics supports the behavior of Activity in Android platform, such as the life-cycle of Activity and user interaction. This is expressive enough to run two Android examples in Figure 2 and 5. After presenting the semantics, we prove the robustness property in this formal setting.

4.1. An Android Semantics with Exceptions

The purpose of our semantics is to describe the execution of an Android program by a sequence of state transitions as $state_1 \Longrightarrow state_2$ where $state_i$s are Android program states. For a simple modeling, we identify four actions on Android platform to make a state transition as this: starting an Android program, activating an activity, user's pressing some button, and finishing an activity to come back. Each state transition in the semantics exactly coincides to performing one of these actions. Each action q on Android platform is defined as,

$$q \ ::= \ Run\ C \mid Activate(l) \mid Press\ btn \mid Return(c, l)$$

where C is an activity class, l is an (Intent) reference, btn is an integer identifier for a button, and (c, l) is an integer result code and a reference for a return value.

To describe an activity screen in the Android program, Android program states have the form of a triple (t, q, h) where t is an activity stack, q is an action on Android platform, and h is an object heap. An activity stack t is $(l_1, w_1) \cdots (l_n, w_n)$ where l_i is an activity reference and w_i is a set of button windows in the activity. Only the top activity (l_1, w_1) is visible to a user and the next top activity (l_2, w_2) will be visible when we finish the top activity to remove from the stack. An action q is one of what is described above or it can be empty as $\emptyset$. An object heap h is a mapping of references onto objects as $\{l_1 \mapsto obj_1, \cdots, l_n \mapsto obj_n\}$ where $obj = C\{f_1 = l_1, \cdots, f_m = l_m\}$ with the fields f_is and their values l_is. An object may be written as $C\{\bar{f} = \bar{l}\}$ in shorthand.

We allow two different forms of states as: *run C* for an initial state to start with an activity class C, and $\perp$ for the abnormal termination state due to some uncaught exception.

$$\text{(run)}\ \frac{\mathcal{A}[Run\ C]\ (\emptyset, \emptyset, \emptyset) = (Success\ r,\ (t, q, h))}{Run\ C \Longrightarrow t, q, h}$$

$$\text{(launch)}\ \frac{\mathcal{A}[Activate\ l]\ (t, \emptyset, h) = (Success\ r,\ (t', q', h'))}{t, Activate\ l, h \Longrightarrow t', q', h'}$$

$$\text{(button)}\ \frac{\begin{array}{l}\text{Button } btn \text{ is pressed}\\ \mathcal{A}[Press\ btn]\ (t, \emptyset, h) = (Success\ r,\ (t', q', h'))\end{array}}{t, \emptyset, h \Longrightarrow t', q', h'}$$

$$\text{(back)}\ \frac{\mathcal{A}[Return(c, l)]\ (t, \emptyset, h) = (Success\ r,\ (t', q', h'))}{t, Return(c, l), h \Longrightarrow t', q', h'}$$

$$\text{(exception)}\ \frac{\begin{array}{l} q \in \{Run\ C, Activate\ l, Return(c, l)\} \wedge q_0 = q \quad \text{or} \quad q \in \{Press\ btn\} \wedge q_0 = \emptyset \\ \mathcal{A}[q]\ (t, \emptyset, h) = (Exception\ exn,\ (t', q', h'))\end{array}}{t, q_0, h \Longrightarrow \perp}$$

Figure 6: State Transition Relations for Actions on Android Platform

Figure 6 defines a state transition relation for the four actions and for uncaught exceptions. An Android program runs as $run\ C \Longrightarrow t_1, q_1, h_1 \Longrightarrow \cdots \Longrightarrow state_{final}$ where either the program terminates normally with the empty activity stack ($state_{final}$ is $(\emptyset, \emptyset, h_{final})$) or it stops abnormally with an uncaught exception ($state_{final}$ is $\perp$).

Each state transition rule in Figure 6 defines the execution of the corresponding action q by the semantic function $\mathcal{A}[q]$, which will be explained soon. This semantic function is a state transformer of the form

$$\lambda state.\ (SuccOrExn,\ state')$$

where $SuccOrExn\ ::=\ Success\ r\ \mid\ Exception\ exn$ to distinguish normal return values r from exceptions exn. In (run), (launch), (button), and (back), the intended execution of each action is performed successfully while, in (exception), the execution of the action causes some uncaught exception.

The semantic function $\mathcal{A}[-]$ for actions, which is assumed to be written in a call-by-value functional language like ML, is shown in Figure 7. The semantic function takes an action q, and it changes states as the intended behavior of the action, resulting a normal result or an exception. $\mathcal{A}[Run\ C]$ starts an Android program by creating an activity C, pushing it on the activity

$$
\begin{array}{lcl}
\mathcal{A}[\mathit{Run}\ C] & = & \mathit{do}\ \ l_{new} \leftarrow \mathcal{E}[C\ x = \mathsf{new}\ C();x]\ \emptyset \\
 & & \quad \mathit{pushOntoActivityStack}\ (l_{new}, \emptyset) \\
 & & \quad \mathcal{E}[x.\mathit{onCreate}()]\ \{x \mapsto l_{new}\} \\
\mathcal{A}[\mathit{Activate}\ l] & = & \mathit{do}\ \ C \leftarrow \mathit{targetActivityClassFromIntent}(l) \\
 & & \quad l_{new} \leftarrow \mathcal{E}[C\ x = \mathsf{new}\ C();\ x.\mathit{intent} = \mathit{intent};\ x]\ \{\mathit{intent} \mapsto l\} \\
 & & \quad \mathit{pushOntoActivityStack}\ (l_{new}, \emptyset) \\
 & & \quad \mathcal{E}[x.\mathit{onCreate}()]\ \{x \mapsto l_{new}\} \\
\mathcal{A}[\mathit{Press}\ \mathit{btn}] & = & \mathit{do}\ \ l \leftarrow \mathit{getTopActivityRef} \\
 & & \quad \mathcal{E}[x.\mathit{onClick}(b)]\ \{x \mapsto l, b \mapsto \mathit{btn}\} \\
\mathcal{A}[\mathit{Return}(c,l)] & = & \mathit{do}\ \ t \leftarrow \mathit{getActivityStack} \\
 & & \quad \mathit{if}\ \mathit{length}(t) \geq 2\ \mathit{then}\ \mathit{do} \\
 & & \quad\quad \mathit{let}\ (l_1, w_1)\cdot(l_2,w_2)\cdot t_0 = t \\
 & & \quad\quad \mathit{popFromActivityStack} \\
 & & \quad\quad \mathcal{E}[y.\mathit{onActivityResult}(rc,rv)]\ \{y \mapsto l_2, rc \mapsto c, rv \mapsto l\} \\
 & & \quad \mathit{else}\ \mathit{if}\ \mathit{length}(t) = 1\ \mathit{then}\ \mathit{do} \\
 & & \quad\quad \mathit{popFromActivityStack} \\
 & & \quad \mathit{else} \\
 & & \quad\quad \mathcal{E}[\mathit{throw}\ \mathit{new}\ \mathit{RuntimeException}(\text{“}\mathit{EmptyActivityStack}\text{”})]\ \emptyset
\end{array}
$$

Figure 7: Semantic Functions for Actions on Android Platform

stack, and initializing it by invoking its interface method *onCreate*(). For an Intent reference l, $\mathcal{A}[\mathit{Activate}\ l]$ retrieves a target activity class C from the intent reference and it launches the activity similarly as the steps for $\mathcal{A}[\mathit{Run}\ C]$ except assigning the intent reference l to the intent field of the new activity. $\mathcal{A}[\mathit{Press}\ \mathit{btn}]$ invokes the interface method *onClick*() of the top activity. $\mathcal{A}[\mathit{Return}\ (c,l)]$ finishes the top activity and moves back to the second top activity if there is any one. On moving back, it invokes the interface method $\mathit{onActivityResult}(rc, rv)$ of the second top activity where rc is an integer result code and rv is a return value from the top activity. When there is no more activity on the stack except the top one, it stops the execution of an Android program with the activity stack empty.

It is easy to extend this semantic function further to support the full life-cycle of Activity [12] by including the rest of the Activity interface methods such as *onDestroy*, *onResume*, and *onPause*, but we omit the extension due to the unnecessary complexity in our presentation.

Note two things to finish explaining the semantic function for actions on Android platform. First, the semantic function for Java expressions such as $\mathcal{E}[x.\mathit{onCreate}()]\ \{x \mapsto l_{new}\}$ is extensively used in the semantic function for actions. We define $\mathcal{E}[e]\ \mathit{env}$ as the Java semantics including exception constructs (*throw* and $\mathit{try} - \mathit{catch}$), where e is a Java expression and *env* is an environment mapping identifiers to references. The details are available in Appendix. Second, both of the semantic functions $\mathcal{A}[q]$ and $\mathcal{E}[e]\ \mathit{env}$ are written in monadic do notation to simplify passing states and checking whether a computation is performed successfully or not. For example, the definition of $\mathcal{A}[\mathit{Press}\ \mathit{btn}]$ is equivalent to one without do notation as:

$$
\begin{array}{lcl}
\mathit{do}\ l \leftarrow \mathit{getTopActivityRef} & & \lambda \mathit{state}_0.\ \mathit{bind}\ \mathit{getTopActivityRef} \\
\quad \mathcal{E}[x.\mathit{onClick}(b)][x \mapsto l, b \mapsto \mathit{btn}] & \equiv & \quad (\lambda l.\lambda \mathit{state}.\ \mathit{bind}\ \mathcal{E}[x.\mathit{onClick}(b)][x \mapsto l, b \mapsto \mathit{btn}] \\
 & & \quad\quad (\lambda l'.\lambda \mathit{state}'.\ \ \mathit{return}\ l'\ \mathit{state}')\ \mathit{state})\ \mathit{state}_0
\end{array}
$$

where $\mathit{bind}\ m_1(\lambda l.m_2)$ does case analysis on whether or not a computation m_1 is successful to decide to perform the next m_2. The full details on do notation and some auxiliary monadic functions such as *pushOntoActivityStack* are also available in Appendix.

Now we are ready to run, for example, two Android programs in Figure 2 and 5. Let us first consider the Android program using Activity in Figure 2. Assume a state that *Calc* activity is visible where the activity stack is the form of $(l_{Calc}, w_{Calc}) \cdot (l_{Main}, w_{Main})$. When a user presses a button *btn* of an activity referenced by l_{Calc}, we invoke *onClick* method of *Calc* by evaluating $\mathcal{E}[x.onClick(b)]\ \{x \mapsto l_{Calc}, b \mapsto btn\}$, according to $\mathcal{A}[Press\ btn]$. As explained in Section 2, this invocation may throw *NoSuchOperator* exception when an illegal arithmetic operator is given. Throwing an exception is interpreted by the semantic function $throw\ exn = \lambda state.\ (Exception\ exn, state)$. In the case, this invocation evaluates to $(Exception\ l_{exn}, state)$, ignoring the execution of whatever follows the invocation, where l_{exn} is an exception reference to $NoSuchOperator\{\cdots\}$ object. Consequently, we are forced to choose (exception) to make a state transition to $\perp$. In the alternative Android program using ExceptionActivity in Figure 5, however, such a situation will never happen by propagating the exception (l_{exn}) to the second top activity *Main*, which will be shown in the following.

4.2. Robustness Property

Every Android program using *ExceptionActivity* is more robust than the original program using *Activity*. We show this robustness property by proving that every Android program extending *ExceptionActivity* stops normally even when the original program extending *Activity* is terminated abnormally due to some uncaught exception thrown by itself.

Let us denote an Android program as $\bar{N}$, a set of class declarations where N is a class declaration in the form of $class\ C\ extends\ D\ \{\ \cdots\ \}$. $\bar{N}^*$ is an Android program rewritten using *ExceptionActivity* by the basic Android exception transformation in Definition 1. Then the robustness property is formulated as this theorem.

Theorem 1 (Robustness). *Suppose Android programs never try to start any activity that is absent. Whenever run* $C \Longrightarrow^n t, q, h$ *or run* $C \Longrightarrow^n \perp$ *in an Android program* $\bar{N}$, *run* $C^* \Longrightarrow^{n+m} (t_{final}, q_{final}, h_{final})$ *in the basic Android exception transformed program* $\bar{N}^*$ *for some* $n, m \geq 1$.

We prove the theorem by two propositions below. For the proof, we need to extend $(-)^*$ to states (t, q, h) as (t^*, q^*, h^*). t^* and q^* are simply t and q. h^* is the same as h but every activity object is extended with an extra field of $catcher = null$ as: when C is *Activity* or its descendant, $C\{\bar{f} = \bar{l}\}$ is replaced by $D\{\bar{f} = \bar{l}, catcher = null\}$ such that D is *ExceptionActivity* when C is *Activity* or D is the same as C when C is its descendant.

For the better recovery of exceptions, one could extend the basic transformation of Definition 1 by overriding a *Catch* method in an activity or by installing an exception handler *catcher* in an activation of an activity. This will give rise to the better robustness than what one gets by the basic transformation.

First, the sound simulation proposition says that, whenever an Android program $\bar{N}$ arrives at a normal state, the transformed Android program $\bar{N}^*$ also arrives at another state equivalent under $(-)^*$.

Proposition 1 (Sound Simulation of Normal Execution). *If run* $C \Longrightarrow^n t, q, h$ *then run* $C^* \Longrightarrow^n t^*, q^*, h^*$ *for* $n \geq 1$.

Second, whenever an Android program $\bar{N}$ gets stuck throwing an exception, the transformed Android program $\bar{N}^*$ will stop normally, propagating the exception to the last activity.

Proposition 2 (Complete Handling of Exceptional Execution)**.** *Suppose Android programs never try to start any activity that is absent. If run* $C \Longrightarrow^{n} \perp$ *then run* $C^{*} \Longrightarrow^{n+m} \emptyset, \emptyset, h$ *for some heap* h *and* $n, m \geq 1$.

Note that, when an exception is thrown outside of the fence of the interface methods, our CE library (*ExceptionActivity*) will never be able to catch it. `ActivityNotFoundException` is one of such exceptions. This is thrown when one cannot find a target activity class to launch by *targetActivityClassFromIntent*(l) from an intent reference l in Figure 7, due to the absence of the class.

The detailed proofs for the two propositions are available in Appendix.

5. Experiments

Our benchmarks consist of nine Android programs [13]: one commercial program (Bitcoin-Wallet), two sample programs (BluetoothChat and NotesList) developed by Google, one student project program (Cafe), and the rest five programs excerpted from advanced Android books [12, 14]. Table 1 shows a catalog of these programs. It also shows the numbers of Android components in each benchmark and the numbers (in the parentheses) of those using no try-catch blocks at all.

Android Apps	Activity	Service	Broadcast	Provider
AndroidSecurity	11(6)	0	0	0
Bitcoin-Wallet	33(10)	2	5(2)	2
BluetoothChat	2(2)	0	1	0
Cafe	5(4)	0	0	2(1)
Contacts	5(4)	1(1)	0	1
MigrateClinic	6(4)	0	0	0
NotesList	4(3)	0	0	1
MediaPlayer	4(3)	0	3(1)	0
Earthquake	10(5)	2	3(2)	1(1)

Table 1: Android Benchmark Programs: # of Components (without Try-Catch)

Our library is found to be very effective in catching a number of otherwise uncaught exceptions in runtime, as in Table 2. With our exception library, we have rewritten all benchmark programs, which are also available in [13]. We have found that the library catches 30 uncaught exceptions in the benchmarks to prevent them from terminating abnormally. Most of the exceptions caught by the library are by referencing `NULL`, using indices out of bounds, and passing illegal arguments.

Table 2 also classifies the causes of the exceptions as malformed Intents, data/query format errors, mis-configurations, and so on. A malformed Intent may miss filling some field of the intent that the receiving component expects to have. Some benchmark improperly handles the format of XML data sent from a remote server due to the change of its XML schema. Also, some user-entered text may cause syntactic errors in query statements to build. Android platform versions of mobile devices may be different from what programs were developed with, which can

Android Apps	Exns	C1	C2	C3	C4
AndroidSecurity	0	0	0	0	0
Bitcoin-Wallet	7	7	0	0	0
BluetoothChat	3	2	0	1	0
Cafe	1	0	0	0	1
Contacts	0	0	0	0	0
MigrateClinic	1	0	0	1	0
NotesList	11	10	0	0	1
Earthquake	7	0	5	1	1

Types of Exceptions	Exns
NullPointer	16
IndexOutOfBounds	3
ArrayIndexOutOfBounds	1
CursorIndexOutOfBounds	1
IllegalArgument	5
OutOfMemory	2
NumberFormat	1
IllegalState	1

Table 2: Exceptions Caught by Our Library and the Causes (C1:Intents, C2:Format, C3:Config, C4:Etc.)

also cause exceptions. The ratio of exceptions due to the malformed intents is higher than any others, which is consistent with the observation by [2].

We discuss how exceptions are thrown and how our library handles them in some of the benchmarks, BluetoothChat and NotesList. BluetoothChat is a text-based communication program using bluetooth. It consists of two activities and one broadcast receiver where the main activity launches the other discovery activity to find out any near bluetooth equipped devices to chat with. Once such a device is found, the broadcast receiver will receive an intent holding information about the device. NotesList is a memo program to create, to edit, and to delete memos in mobile database. It has three activities, one for listing memos, another for editing a memo, and the third for editing a memo title.

Both of them have been developed very robust by Google since the initial release of Android platform (2009), but it was not difficult to find out some vulnerability to cause exceptions. For example, we can construct a malformed intent for discovery of near-by bluetooth devices to be sent to the broadcast receiver of BluetoothChat to raise `NullPointerException`. Also, we can make NotesList raise the out-of-memory exception by copying a memo into itself repeatedly, that makes it large exponentially, terminating the memo editor activity with `OutOfMemoryError`. In both of the exceptional situations, the original Android programs terminate abnormally, but the rewritten ones catch them and they are able to return to the normal program state. We also found that these rewritten programs benefit from inter-component exception handling. This confirms our claim that the proposed component-level exception mechanism makes Android programs more robust.

The application of our Android exception library to our benchmarks involves rewriting them, which can be a criterion on how burdensome programmers are to use the library. Table 3 summarizes the changes of lines of code in the benchmarks. On average, we have changed 132 lines of code, which amounts to about 4.5% of the LOC of the original benchmarks. Compared with the LOC of each benchmark, the number is relatively small. It is straightforward to rewrite Android programs with the library under the basic Android exception transformation of Definition 1. It requires little domain knowledge on the Android programs.

The application of the exception library to our benchmarks also incurs some overhead. Table 3 shows increased binary sizes due to static linking with the library and increased startup time due to the extra size and the exception handling layer. In most of the benchmarks, the increased binary sizes range from 4K to 5K bytes, which do not take much storage at all even in mobile devices. Moreover, these increased binary sizes could be diminishing if Android platform supported a component-level exception mechanism as ours in a form of shared library.

Android Apps	LOC	%	Binary size	%	Startup	%
AndroidSecurity	1297 (078)	6.01	323070(+4001)	1.24	140(+08)	5.71
Bitcoin-Wallet	57721 (592)	1.03	3799290(+13693)	0.36	1211(+86)	7.10
BluetoothChat	1124 (048)	4.27	27087(+4837)	17.86	172(+15)	8.72
Cafe	2356 (110)	4.67	1472783(+5892)	0.40	321(+47)	14.64
Contacts	1781 (051)	2.86	35663(+5801)	16.27	156(+08)	5.13
MigrateClinic	1111 (058)	5.22	53545(+4916)	9.18	n/a	n/a
NotesList	3221 (088)	2.73	59983(+5174)	8.63	312(+16)	5.13
MediaPlayer	664 (055)	8.28	153833(+4586)	2.98	172(+16)	9.30
Earthquake	1934 (111)	5.74	50676(+8736)	17.24	257(+16)	6.23

Table 3: Lines of Code, Binary Size in bytes, and Startup Time in millisecond of the Benchmarks (With Changes and Increments When Using the Component-Level Exception Library)

Another overhead is the startup time of an Android program, defined as an interval between the time when user presses the launching icon and the time when the execution of the *onCreate* method of the main Activity finishes. The average startup time increases less than 0.03 seconds on Samsung Galaxy Nexus and Android Ver. 4.1.1. Even the longest difference (0.086 sec. in Bitcoin-Wallet) is never noticeable.

Finally note that it will be more effective when one starts developing one's own Android programs with the CE library. During the development, our library can help to catch exceptions and to fix some code for the cause of the exceptions earlier. Also, the development of Android programs using the component-level exception mechanism might lead to somewhat more beneficial Android program architecture, which will be an interesting future work.

6. Related Work

Android applications can be vulnerable due to Intents, if input from Intents are not validated sufficiently. A malformed Intent delivered to a receiver exposes attack surfaces as pointed out by [4]. For example, unauthorized receipt of an implicit Intent can be made by malicious component, and Intent spoofing can be made, by which a malicious application sends an Intent to an exported component that is not expecting Intents from that application. A static analysis tool like ComDroid detects statically these potential vulnerabilities in Android applications [4].

It is also shown by experiments in [2] how vulnerable components are due to Intents. By extending the basic Intent fuzzer [1], they generated random and semi-valid intents and tested how components handle these exceptional conditions. In particular, they focused on uncaught exceptions, because they result in the crashes. In the experiment, they measured the number of failed components for various types of components. For instance, 29(8.7%) out of total 332 Activities crash with generated semi-valid intents on Android 4.0 emulator. The distribution of exception types are also measured to understand how components fail due to uncaught exceptions. It is shown that `NullPointerException` makes up the largest share of all the exceptions. In case of implicit Intents, the number of crashes due to `NullPointerException` is 32(38.5%) out of the total 83 crashes. Other exceptions like `ClassNotFoundException` and `IllegalArgumentException` are next significant ones. This experiment justifies a component-level exception mechanism to reduce abnormally failed components.

A combination of static analysis and random fuzzing was also proposed to dynamically test Android applications in [3]. A path-insensitive, inter-procedural CFG analysis is employed to automatically extract the expected intent structure that a component is expecting to receive. A set of intents are generated with the static intent structure information to explore more execution paths. Target components are executed with these fuzzed intents, and both code coverage and crashes due to exceptions are monitored.

An intent specification language was proposed by [15] for a common Intent fuzzing based Android testing framework, providing a flexible way to express the shape information of intents and to generate test artifacts in various testing contexts. In a preliminary result on an application of the testing framework, it reported in 10 real-world applications the intent vulnerability caused by `NullPointerException`, `RuntimeException`, and `IllegalArgumentException`,

The traditional exception propagation in Java is applied only to the inside of a thread [16], while the propagation of uncaught exceptions in Android-Java is confined to the inside of components. In [17, 18], an inter-procedural static analysis of Java programs was proposed to estimates their exception flows along the method call stack independently of the programmer's specifications. By extending the work [17], the exception propagation analysis was implemented along with its visualization tool in [19], which visualizes possible propagation paths of exceptions using the static analysis information. Other works on Java exceptions like [20] studied how to improve resilience against unanticipated exceptions by program transformation.

There have been several research works [6, 7, 8, 9] including one by Huang and Wu [6], which recognizes the similar problem as ours, to design a middle-ware approach atop EJB and/or CORBA container for (implementation language-neutral) exception handling at architecture level. Contrary to this, our component-level exception mechanism is for the mobile platform, Android, in the form of an easy-to-use and user-extensible Java library using class inheritance to intercept exceptions systematically.

Another contribution of this paper is the semantic definition for combining Android, Java, and exceptions together. There have been researches on the semantics for each or two of the three features, but not for considering all of them. Also, it is noticeable that the notion of Monad is used for modeling exceptions and states in our Android-Java semantics, not in functional language semantics [21, 22] where the technique of Monad is well known for structuring.

Starting with researches for Android semantics, Chaudhuri, Fuchs, and Foster [23, 24] had presented an operational semantics for very abstract form of Android applications for the first time. They had used the semantics to prove the soundness of a type system for a permission-based security model and to formalize an information-flow analysis used in ScanDroid. Palamidessi and Ryan also defined another operational semantics for abstract Android security framework [25]. Payet and Spoto defined a non-standard operational semantics for a subset of Dalvik byte code instructions, particularly emphasizing the detailed life-cycle of Activity [26]. Jeon, Micinski, and Foster took a step forward to define a precise operational semantics of Dalvik byte code instructions for dynamic analysis through symbolic execution in Symdroid [27]. Another operational sematnics for Dalvik byte code instructions by Wognsen, Karlsen, and Olesen [28, 29] had took into account the feature of Java reflection and the WebView JavaScript interface, both of which are very important in defining the behavior of Android applications in practice. Choi and Chang [11] defined a featherweight Android semantics by extending a featherweight Java semantics [10], and they proved a type soundness of a type and effect analysis for activation flow in Android programs.

As to researches for Java and exception semantics, Drossopoulou and Eisenbach firstly defined a formal semantics for Java [30]. Nipkow and Oheimb proposed a semantics named Java-light for a large subset of Java including exceptions to prove the Java type soundness [31]. Igarashi, Pierce, and Wadler proposed an operational semantics for core calculus of Java and Generic Java (GJ) [10]. Bierman, Parkinson, and Pitts defined a semantics for an imperative core calculus of Java, employing an effect system to deal with the imperative features properly [32], which is similar to the use of Monad to model exceptions in our semantics. Jones [33] defined monadic functions for Java byte code instructions only over states, not exceptions. Stärk, Börger, and Schmid defined the most complete semantics for Java 1.0 using Abstract State Machines (ASMs), they also defined the compiler and the byte code format to prove the compiler correctness [34]. Another large-scale semantics for Java was defined by Farzan, Chen, Meseguer, and Roşu using term rewriting in Maude [35]. This semantics is executable, and it was applied to model-checking of Java programs. Recently, a complete semantics of Java 1.4, called K-Java, has been presented by Denis Bogdănaş and Grigore Roşu [36], and this work has been applied to model-checking multi-threaded programs.

7. Conclusion

We proposed a new idea of component-level exception mechanism for robust Android programs, and realized our idea as the flexible form of a domain-specific library. Theoretically, we have shown that the mechanism improves the robustness of Android programs by designing an Android semantics with exceptions. Also, in practice, six out of nine Android benchmark programs become more robust by using the library. In addition, the overhead of using the library in Android programs turns out to be small in terms of lines of code, binary sizes, and startup time, according to our experimental assessment.

Our proposal is a new Android program development methodology for recovering unexpected exceptions which is different from testing or static/dynamic/hybrid analyses [1, 2, 3, 4, 37]. As well as the prevention of abnormal termination, the proposal can also help to prevent information exposure through any exception messages, as suggested by one of a set of software weaknesses(CWE-209, http://cwe.mitre.org).

In future, first of all, the component-level exception mechanism can guide a new design of Android platform architecture to enhance the current limitation of Android platform. Second, the proposed library can be used for more robustness of applications in the course of application development, even though our experiments show its effectiveness just by transforming already developed applications with the exception library. Third, the design of exception propagation among codes written in different programming languages, particularly in JavaScript for cross-platform development, will be quite interesting.

References

[1] Intent Fuzzer (2009).
URL `https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx`

[2] A. K. Maji, F. A. Arshad, S. Bagchi, J. S. Rellermeyer, An Empirical Study of the Robustness of Inter-component Communication in Android, in: Proceedings of the International Conference on Dependable Systems and Networks, 2012, pp. 1–12. doi:10.1109/DSN.2012.6263963.

[3] R. Sasnauskas, J. Regehr, Intent Fuzzer: Crafting Intents of Death, in: Proceedings of the Joint International Workshop on Dynamic Analysis(WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), ACM, San Jose, CA, 2014, pp. 1–5.

[4] E. Chin, A. Felt, K. Greenwood, D. Wagner, Analyzing Inter-application Communication in Android, in: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, ACM, New York, 2011, pp. 239–252. doi:10.1145/1999995.2000018.

[5] S. Microsystems, The Java Tutorial: a Practical Guide for Programmers (1999).

[6] G. Huang, Y. Wu, Towards Architecture-Level Middleware-Enabled Exception Handling of Component-based Systems, in: Proceedings of the 14th International ACM SIGSOFT Symposium on Component-Based Software Engineering, ACM, New York, NY, 2011, pp. 159–168. doi:10.1145/2000229.2000252.

[7] A. Romanovsky, Exception Handling in Component-based System Development, in: 25th Annual International Computer Software and Applications Conference. (COMPSAC), Chicago, Illinois, 2001, pp. 580–586. doi:10.1109/CMPSAC.2001.960671.

[8] F. C. Filho, P. A. d. C. Guerra, V. A. Pagano, C. M. F. Rubira, A Systematic Approach for Structuring Exception Handling in Robust Component-based Software, Journal of the Brazilian Computer Society 10 (3) (2005) 5–19.

[9] C. Dellarocas, Toward Exception Handling Infrastructures in Component-based Software., in: International Workshop on Component-based Software Engineering, ACM/IEEE, Kyoto, Japan, 1998, pp. 25–26.

[10] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a Minimal Core Calculus for Java and GJ (2001). doi:10.1145/503502.503505.

[11] K. Choi, B.-M. Chang, A Type and Effect System for Activation Flow of Components in Android Programs, Information Processing Letters 114 (11) (2014) 620–627.

[12] R. Meier, Professional Android 4 Application Development, 3rd Edition, Wrox, Hoboken, NJ, USA, 2012.

[13] Android Benchmarks and their Rewritten Versions with Android Exception Library.
URL `http://mobilesw.yonsei.ac.kr/paper/android_exception.html`

[14] Z. Mednieks, G. Blake Meike, L. Dornin, Enterprise Android : Programming Android Database Applications for the Enterprise, John Wiley & Sons, Somerset, NJ, USA, 2013.

[15] M.-P. Ko, K. Choi, B.-M. Chang, Intent Specification Langauge and Its Tools for Testing the Vulnerability of Android Components, Tech. Rep. TR-JUL-2015-3, Yonsei University, Wonju, Wonju, Korea (2015).

[16] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Addison-Wesley, 2005.

[17] B.-M. Chang, J.-W. Jo, K. Yi, K.-M. Choe, Inter-procedural Exception Analysis for Java, in: Proceedings of ACM Symposium on Applied Computing, ACM, Las Vegas, USA, 2001, pp. 620–625.

[18] M. P. Robillard, G. C. Murphy, Static Analysis to Support the Evolution of Exception Structure in Object-oriented Systems, ACM Transactions on Software Engineering and Methodology 12 (2) (2003) 191–221.

[19] B.-M. Chang, J.-W. Jo, S. H. Her, Visualization of Exception Propagation for Java Using Static Analysis, Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation-doi:10.1109/SCAM.2002.1134117.

[20] B. Cornu, L. Seinturier, M. Monperrus, Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience against UnAnticipated Exceptions, Information and Software Technology 57 (1) (2015) 66–76.

[21] E. Moggi, Notions of Computation and Monads (1991). doi:10.1016/0890-5401(91)90052-4.

[22] P. Wadler, Monads for Functional Programming, in: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1992, pp. 1–14. doi:10.1145/143165.143169.

[23] A. Chaudhuri, Language-based Security on Android (2009). doi:10.1145/1667209.1667211.

[24] A. P. Fuchs, A. Chaudhuri, J. Foster, SCANDROID: Automated Security Certification of Android Applications (2009).

[25] C. Palamidessi, M. D. Ryan, Formal Modeling and Reasoning about the Android Security Framework, Lectures Notes in Computer Science 8191 (2013) 64–81.

[26] E. Payet, F. Spoto, An Operational Semantics for Android Activities, in: Proceedings of ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, ACM, San Diego, California, 2014, pp. 121–132.

[27] J. Jeon, K. K. Micinski, J. S. Foster, SymDroid: Symbolic Execution for Dalvik Bytecode (2012).

[28] E. R. Wognsen, H. S. n. Karlsen, Static Analysis of Dalvik Bytecode and Reflection in Android, Master thesis, Aalborg University (2012).

[29] E. R. Wognsen, H. S. n. Karlsen, M. C. Olesen, R. R. Hansen, Formalisation and Analysis of Dalvik Bytecode, Science of Computer Programming 92 (PART A) (2014) 25–55. doi:10.1016/j.scico.2013.11.037.

[30] S. Drossopoulou, S. Eisenbach, Java is Type Safe - Probably, in: 11th European Conference on Object-Oriented Programming (ECOOP), Vol. 1241, Springer, Jyväskylä, Finland, 1997, pp. 389–418.

[31] T. Nipkow, D. von Oheimb, Java-light is Type-Safe - Definitely, in: Proceedings of 25th ACM Symposium on Principles of Programming Languages, ACM, 1998, pp. 161–170.

[32] G. M. Bierman, M. J. Parkinson, A. M. Pitts, MJ: an Imperative Core Calculus for Java with Effects, Tech. Rep. 563, University of Cambridge (2003).

[33] M. P. Jones, The Functions of Java Bytecode, in: Proceedings of the OOPSLA '98 Workshop on Formal Underpinnings of Java, Vancouver, BC, Canada, 1998, pp. 1–21.

[34] R. F. Stärk, J. Schmidt, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer, 2001.

[35] A. Farzan, F. Chen, J. Meseguer, G. Roşu, Formal Analysis of Java Programs in JavaFAN, in: Proceedings of Computer-aided Verification (CAV'04), Vol. 3114 of LNCS, 2004, pp. 501–505.

[36] D. Bogdănaş, G. Roşu, K-Java: a Complete Semantics of Java, in: Proceedings of 4th ACM Symposium on Principles of Programming Languages, ACM, 2015, pp. 445–456.
[37] D. Octeau, P. McDaniel, J. Somesh, A. Bartel, E. Bodden, J. Klein, Y. L. Traon, Effective Inter-component Communication Mapping in Android with EPICC: an Essential Step Towards Holistic Security Analysis, in: 22nd USENIX Security Symposium, USENIX, Washington, D.C., USA, 2013, pp. 543–558.

Appendix

We present the syntax and semantics of a featherweight Android-Java, making complete the presentation of the semantics in Section 4. Our syntax and semantics start with those of the featherweight Java [10], but in an imperative version that makes such as heaps and reference updates explicit [11]. We extend it with the conventional Java exceptions, and support the behavior of Activity in Android platform, such as the life-cycle of Activity and user interaction.

A.1 A Syntax of Android-Java with Exceptions

We define a minimal syntax of a featherweight Android-Java, which allows to write all examples in this paper, by extending the featherweight Java [11, 10].

$$\begin{array}{lll}
N & ::= & \mathsf{class}\ C\ \mathsf{extends}\ C\ \{\bar{C}\ \bar{f};\ \bar{M}\} \\
M & ::= & C\ m(\bar{C}\ \bar{x})\ \{\ e\ \} \\
e & ::= & x \mid x.f \mid \mathsf{new}\ C() \mid x.f = x \mid (C)x \mid x.m(\bar{x}) \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid Cx = e;\ e \mid prim(\bar{x}) \\
 & \mid & \mathsf{try}\ e\ \mathsf{catch}(Cx)\ e \mid \mathsf{throw}\ x
\end{array}$$

An Android program is a set of class declarations $\bar{N}$. A block expression $C\ x = e; e'$ declares a local binding of a variable x to the value of e for later uses in e'. It is also used for sequencing $e;\ e'$ by assuming omission of a dummy variable $C\ x$. The conditional expression may be written as $\mathsf{ite}\ e\ e\ e$ for brevity. We write a string object as a "string literal." Also, x.m("...") means $String\ s =$ "..."; $x.m(s)$ in shorthand. A recursive method offers a form of loops. The primitive functions $prim(\bar{x})$ are interfaces between an Android program and the Android platform, which will be explained later.

For example, Figure 8 shows our definition of *Activity* class and *Intent* class in the explained syntax. In the definition, the primitive functions *primFinish*, *primAddButton*, and *primStartActivity* are introduced in order to model the interaction between Android programs and platform. Their semantics will be defined in the next section.

A.2 A Semantics for Android-Java with Exceptions and Proofs of Its Properties

Figure 9 shows basic semantic functions. Monadic functions (*return* and *bind*) make several semantics functions into a sequential one. Exception relevant functions (*throw*, *trycatch*) models how to throw exceptions and how to catch them in the semantics. State relevant functions (*get* and *put*) allow to read and write the current state.

- Using the monadic functions *bind* and *return*, the do notation can be defined where *Stmts* is a sequence of $x \leftarrow exp$, exp, and $let\ x = exp$ separated by a semicolon, as follows.

```
class Activity {
    Intent intent;

    void onCreate() { }
    void onPause() { }
    void onResume() { }
    void onDestroy() { }
    void onClick(int button) {
       if (button==BACK) primFinish(RESULT_CANCEL, null);
       else {}
    }
    void onActivityResult(int resultCode, Intent intent) { }
    void addButton(int button) { primAddButton(button); }
    void finish(int resultCode, Intent i) { primFinish(resultCode,i); }
    void tryActivityForResult(Intent i) { primStartActivity(i); }
    Intent getIntent() { this.intent; }
}

class Intent {
    String target;
    Object data;
    // The setter and getter methods
    // for the above fields
}
```

Figure 8: Android Classes: Activity and Intent

$$\begin{array}{lcl}
do\ \{\ x \leftarrow exp;\ Stmts\ \} & = & bind\ exp\ (\lambda x.\ do\ \{\ Stmts\ \}) \\
do\ \{\ exp;\ Stmts\ \} & = & bind\ exp\ (\lambda\_.\ do\ \{\ Stmts\ \}) \\
do\ \{\ exp\ \} & = & bind\ exp\ (\lambda x.\ return\ x) \\
do\ \{\ let\ x = exp;\ Stmts\ \} & = & let\ x = exp\ in\ do\ \{\ Stmts\ \}
\end{array}$$

- As variants of *get* and *put* functions, the four semantic functions (*pushOntoActivityStack*, *popFromActivityStack*, *getActivityStack*, and *getTopActivityRef*) used in Figure 7 can be easily defined.
- Two semantic functions *throw* and *trycatch* are introduced to support Java exception constructs later. *throw exn state*$_0$ always returns (*Exception exn*, *state*$_0$). *trycatch m h state*$_0$ performs a computation by *m state*$_0$. After that, we do case analysis on whether the computation is successful or not. When it succeeds, *m state* becomes the result of evaluation of *trycatch m h*, ignoring an exception handler *h*. When it throws an exception *exn* with *state*$_1$, we give it to the handler *h* as *h exn state*$_1$.
- A function *targetActivityClassFromIntent*(*l*) is used in Figure 7 to pick a target activity class from an Intent reference. Suppose $h(l) = Intent\{target = l_t, ...\}$ for the current heap *h*. The function returns $Class(h(l_t))$ if $l_t \neq null$ where $Class(\text{“C”}) = C$ such that *C* is an activity class. When the function fails to find any activity class due to the null Intent reference or the absence of any designated activity class, the function is defined to throw an exception by $\mathcal{E}[throw\ new\ ActivityNotFoundException]\ env_{empty}$.

```
bind m k        = λstate. let (x, state′) = m state in
                      case x of
                          Exception e → (Exception e, state′)
                          Success r → k r state′
return r        = λstate. (Success r, state)
get             = λstate. (Success state, state)
put state_0     = λstate. (Success (), state_0)
throw e         = λstate. (Exception e, state)
trycatch m h    = λstate. let (x, state′) = m state in
                      case x of
                          Exception exn → h exn state′
                          Success r → (x, state′)
```

Figure 9: Basic Semantic Functions

The semantic function $\mathcal{E}[e]$ *env* for Java expressions *e* and environment *env* is defined in Figure 10. The semantic function is a state transformer of the form

$$\lambda state.(SuccOrExn, state')$$

which is mostly standard [10]. The notable difference is to add an activity stack and an action to states for modeling Android platform. The standard Java constructs such as variable, field, and method invocation do not access nor change them while primitives change them as:

- *primStartActivity(x)* replaces the current intent reference *q* with a new intent reference bound to *x*.
- *primAddButton(x)* adds a new button whose identifier is bound to *x*.
- *primFinish(x,y)* dismisses the current activity to get back to its caller with a result code *x* and and an intent *y*.

The semantic function use some auxiliary functions defined in [10]. $\mathsf{mbody}(m, C)$ returns the body expression of the method of the class, and $\mathsf{fields}(C)$ gathers all fields belonging to the class, if necessary, following up the inheritance tree. $D <: C$ tests if D is any descendant class of C. The five semantic functions (*getFromHeap*, *updateHeapWith*, *addToHeap*, *putAction*, and *addToWindows*) used are also variants of the *get* and *put* functions.

Lemma 1. *If* $\mathcal{E}[e]$ *env* $(t_1, q_1, h_1) = (se, (t_2, q_2, h_2))$*, then* $\mathcal{E}[e^*]$ *env* $(t_1^*, q_1^*, h_1^*) = (se, (t_2^*, q_2^*, h_2^*))$ *where se is either a normal result or an exception.*

Proof. We prove this by induction on the depth of method invocation. For base cases, we can verify this proposition over the semantic functions for all kinds of expressions except $x.m(\bar{y})$ in Figure 10. For inductive case, the proposition holds for method invocation expressions by induction. Note that, every occurrence of $z.startActivityForResult(intent)$ in e is replaced by $z.TryActivityForResult(intent, null)$ in e^*. By the definition of the method $TryActivityForResult$ (in Figure 4), both of the method invocations do the same except that the latter one sets the catcher field of the activity z to null. The difference is identified by the definition of $(-)^*$ over heaps. □

$$
\begin{array}{lcl}
\mathcal{E}[x]\ env & = & do\ \ return\ env(x) \\
\mathcal{E}[x.f_i]\ env & = & do\ \ let\ l_x = env(x) \\
 & & \qquad C\{\bar{f} = \bar{l}\} \leftarrow getFromHeap(l_x) \\
 & & \qquad return\ l_i \\
\mathcal{E}[x.f_i = y]\ env & = & do\ \ let\ l_x, l_y = env(x), env(y) \\
 & & \qquad C\{\bar{f} = \bar{l}\} \leftarrow getFromHeap(l_x) \\
 & & \qquad updateHeapWith\ \{l_x \mapsto C\{\bar{f} = \bar{l}_{1,i-1} l_y \bar{l}_{i+1,n}\} \\
\mathcal{E}[new\ C()]\ env & = & do\ \ let\ \bar{D}\ \bar{f} = \mathsf{fields}(C) \\
 & & \qquad let\ l = \mathsf{fresh} \\
 & & \qquad addToHeap\ \{l \mapsto C\{\bar{f} = \overline{null}\}\} \\
\mathcal{E}[(C)x]\ env & = & do\ \ let\ l = env(x) \\
 & & \qquad D\{\bar{f} = \bar{l}\} \leftarrow getFromHeap(l) \\
 & & \qquad if\ D <: C \\
 & & \qquad\quad then\ return\ l \\
 & & \qquad\quad else\ \mathcal{E}[throw\ new\ ClassCastException()]\ env \\
\mathcal{E}[ite\ e_0\ e_1\ e_2]\ env & = & do\ \ l_0 \leftarrow \mathcal{E}[e_0]\ env \\
 & & \qquad if\ l_0 == True \\
 & & \qquad\quad then\ \mathcal{E}[e_1]\ env \\
 & & \qquad\quad else\ \mathcal{E}[e_2]\ env \\
\mathcal{E}[C\ x = e_0;\ e]\ env & = & do\ \ l_0 \leftarrow \mathcal{E}[e_0]\ env \\
 & & \qquad \mathcal{E}[e]\ env\{x \mapsto l_0\} \\
\mathcal{E}[x.m(\bar{y})]\ env & = & do\ \ let\ l = env(x) \\
 & & \qquad C\{\bar{f} = \bar{l}'\} \leftarrow getFromHeap(l) \\
 & & \qquad let\ \bar{l}_i = \overline{env}(y_i) \\
 & & \qquad let\ \bar{B}\ \bar{z}.e = \mathsf{mbody}(m, C) \\
 & & \qquad \mathcal{E}[e]\ \{this \mapsto l, \bar{z} \mapsto \bar{l}_i\} \\
\mathcal{E}[throw\ x]\ env & = & do\ \ throw\ env(x) \\
\mathcal{E}[try\ e_1\ catch(C\ x)\ e_2]\ env & = & do\ \ trycatch\ (\mathcal{E}[e_1]\ env)\ (\lambda l.\ if\ l\ instanceof\ C \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad then\ \mathcal{E}[e_2]\ env\{x \mapsto l\} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad else\ throw\ l) \\
 & & \\
\mathcal{E}[primStartActivity(x)]\ env & = & do\ \ putAction\ (Activate\ env(x)) \\
\mathcal{E}[primFinish(x, y)]\ env & = & do\ \ putAction\ (Return\ (env(x), env(y))) \\
\mathcal{E}[primAddButton(x)]\ env & = & do\ \ addToWindows\ \{env(x)\}
\end{array}
$$

Figure 10: Semantic Functions for Java Expressions Including Primitives

Now it is time to show the robustness theorem by proving the sound simulation of normal execution and the complete handling of exceptions as follows.

Proposition 1 (Sound Simulation of Normal Execution)**.** *If run $C \Longrightarrow^n t, q, h$ then run $C^* \Longrightarrow^n t^*, q^*, h^*$ for $n \geq 1$.*

Proof. By the condition, every transition from *run C* is made by one of (run), (launch), (button), and (back), meaning that the corresponding application of the semantic functions $\mathcal{A}[Run]$, $\mathcal{A}[Activate]$, $\mathcal{A}[Press]$, or $\mathcal{A}[Return]$ leads to $(Success\ result, state)$. This implies that all the sub-semantic functions such as $\mathcal{E}[x.onCreate()]$ in $\mathcal{A}[Run]$ and $\mathcal{A}[Activate]$, $\mathcal{E}[x.onClick(b)]$ in $\mathcal{A}[Press]$, and $\mathcal{E}[x.onActivityForResult(rc, rv)]$ in $\mathcal{A}[Return]$ must evaluate to a successful result, too. By Lemma 1, the corresponding sub-semantic functions in the $(-)^*$-transformed Android program evaluate to the same successful result. By applying (run), (launch), (button), and (back), we can get a successful transition in the trasformed program, too.

□

Proposition 2 (Complete Handling of Exceptional Execution)**.** *Suppose Android programs never try to start any activity that is absent. If run $C \Longrightarrow^n \perp$ then run $C^* \Longrightarrow^{n+m} \emptyset, \emptyset, h$ for some heap h and $n, m \geq 1$.*

Proof. By the condition of the proposition, there exists a transition with (exception) during the n transitions from *run C*. This means that some of the semantic functions $\mathcal{A}[Run]$, $\mathcal{A}[Activate]$, $\mathcal{A}[Press]$, or $\mathcal{A}[Return]$ leads to $(Exception\ exn, state)$. To have this kind of an exception that is uncaught by Android programs, there must exist some sub-semantic functions such as $\mathcal{E}[x.onCreate()]$ in $\mathcal{A}[Activate]$ that evaluates to the same exception and state. By Lemma 1 and by the definition of *onCreate* in *ExceptionActivity*, in the $(-)^*$-transformed Android program, $x.OnCreate()$ will be invoked, and $\mathcal{E}[x.OnCreate()]$ will evaluate to the same exception and state. In this situation, the try-catch block surrounding the invocation of *OnCrate* in *onCreate* of *ExceptionActivity* will catch the exception and will pass it to the next top activity by $x.Throw(exn)$.

Note that every $(-)^*$-transformed program is defined to have only default component-level exception handlers, not explicitly made by programmers, as follows:

- The default *Catch* method of all (exception) activities is defined to return false.
- The default *catcher* field of all (exception) activities is set to null.

by the definition of *ExceptionActivity* class in Figure 4.

Due to the definition of the default component-level exception handlers, any uncaught exception will be propagated across activities on the stack by repeating invoking *Throw* and then *onActivityResult* of *ExceptionActivity* until the activity stack becomes empty. During the propagation, the transformed Android program will create extra intent objects used for passing an exception to the previous activities by *Throw* of *ExceptionActivity*. Therefore, h is a union of h_0^* and $\{\overline{l \mapsto Intent\{...\}}\}$ where h_0 is a heap in *state*.

In this case, it is easy to prove that the $(-)^*$ transformed Android program will terminate normally in a finite time after the original Android program abnormally terminates. This is because the length of the activity stack t in *state* is finite. Therefore, there exists extra $m \geq 1$ transitions for the transformed Android program to take for the normal termination after n transitions to keep pace with the original program. □