Amplifying Tests to Validate Exception Handling Code

Pingyu Zhang
University of Nebraska-Lincoln, pzhang@cse.unl.edu

Sebastian Elbaum
University of Nebraska-Lincoln, selbaum@virginia.edu

2012 34th International Conference on Software Engineering (ICSE)
Digital Object Identifier: 10.1109/ICSE.2012.6227157

Amplifying Tests to Validate Exception Handling Code

Pingyu Zhang and Sebastian Elbaum
Computer Science and Engineering Department
University of Nebraska - Lincoln
Lincoln, NE 68588, USA
{pzhang,elbaum}@cse.unl.edu

***Abstract*—Validating code handling exceptional behavior is difficult, particularly when dealing with external resources that may be noisy and unreliable, as it requires: 1) the systematic exploration of the space of exceptions that may be thrown by the external resources, and 2) the setup of the context to trigger specific patterns of exceptions. In this work we present an approach that addresses those difficulties by performing an exhaustive amplification of the space of exceptional behavior associated with an external resource that is exercised by a test suite. Each amplification attempts to expose a program exception handling construct to new behavior by mocking an external resource so that it returns normally or throws an exception following a predefined pattern. Our assessment of the approach indicates that it can be fully automated, is powerful enough to detect 65% of the faults reported in the bug reports of this kind, and is precise enough that 77% of the detected anomalies correspond to faults fixed by the developers.**

***Keywords*-Test transformation; exception handling**

I. Introduction

Exception handling constructs are meant to increase application robustness. In practice, however, the code handling exceptions is not only difficult to implement [15] but also challenging to validate (e.g., [11], [17], [22].)

We conjecture that the validation challenge is particularly relevant when dealing with systems that must interact with external resources that can be noisy and unreliable, and exhibit transient and unpredictable failures [16]. Consider, for example, an exception handling construct to check for the end of a sequential input stream. The reliability of the stream is rarely in doubt, the end of the file is a certainty, and standard testing frameworks provide mocking support for streams. In contrast, an exception handling construct interacting with noisy and often unreliable localization, communication, and sensor services, cannot make such simplifying assumptions, requiring more complex and hence harder to validate implementations.

In this work we set out to quantify and to mitigate such challenge. We start by studying bug reports of five popular ubiquitous open source applications for the Android phone platform (Section II). The study provides evidence of the fault proneness of and difficulties in validating exception handling constructs in this application domain. Almost a third of the bug reports that led to code fixes were caused by poorly implemented exception handling constructs that did not completely address the space of potential exceptional behavior. Of those bug reports, over half were caused by a small set of commonly noisy external resources (localization and connection). When we analyzed those reports in detail, we found that two thirds were caused by a series of exceptions following non-trivial patterns most of which could not be detected by existing validation approaches.

Based on those findings, we develop an automated approach to support the detection of faults in exception handling code that deals with external resources. Our approach is simple, scalable, and effective in practice when combined with a test suite that invokes the resources of interest. The approach first instruments the target program so that the results of calls to external resources of interest can be mocked at will to return exceptions. Then, existing test cases are systematically amplified by re-executing them on the instrumented program under various mocked patterns to explore the space of exceptional behavior. When an amplified test reveals a fault, the mocking pattern applied with the test serves as an explanation of the failure induced by the external resource. To control the number of amplified tests the approach prunes tests with duplicate calls and call-outcomes to the external resources, and bounds the number of calls that define the space of exceptional behavior explored.

The approach is built on two assumptions. First, it builds on the small scope hypothesis [13], often used by techniques that systematically explore a state space, which advocates for exhaustively exploring the program space up to a certain bound. The underlying premisse is that many faults can be exposed by performing a bounded number of program operations, and that by doing so exhaustively no corner cases are missed. Several studies and techniques have shown this approach to be effective (e.g., [2], [5], [6]) and we build on those in this work. In our approach the bound corresponds to the length of the mocked patterns. Second, we assume that the program under test has enough tests cases to provide coverage of the invocations to the resources of interest. The increasing number and maturity of automated test case generation techniques and tools support this assumption. If this assumption holds, then the approach can automatically and effectively amplify the exposure of code handling exceptional behavior.

Table I
SUMMARY OF ARTIFACTS

Application	Resource API	Version	LOC	Unit Test Suite	
				#	Exe Time (sec)
Barcode Scanner	android.net	r1220	18170	117	275
Keepassdroid	android.database / android.net	v0.9.3	7713	34	102
myTracks	android.database / android.location	r195	5918	55	367
SipDroidVoIP	android.net	r340	18150	39	135
XBMC remote	android.net / android.bluetooth	r317	10880	78	234

Our contributions are:

- A study quantifying the frequency and providing a characterization of the faults associated with exception handling code constructs and more specifically with those related to handling noisy resources that cannot be controlled with simple input manipulation.
- An approach for automatically amplifying the space of exceptional behavior associated with external resources covered by a test suite. We present a definition of the problem in terms of the space of potentially thrown exceptions by the external resources, the architecture of the approach, and our implementation.
- An assessment of the cost-effectiveness of the approach when applied to several real scenarios. With just a small set of external resources and the artifacts' original tests, the amplified suite reported anomalies that led to code fixes 77% of the time and that included 65% of the reported bugs.

II. MAGNITUDE OF THE PROBLEM

In this section we study the prevalence of faults associated with code that handles exceptions. The study focuses on free popular applications for the Android platform which often rely extensively on APIs that work with external resources like wireless connections, databases, GPS, or bluetooth.

The selection of artifacts for the study consisted of the following steps. First, we collected a pool of 210 candidate applications from the following sources: Wikipedia [14] (121), Le Wiki Koumbit [19] (40), Trac [20] (15), OpenStreetMapWiki [27] (8), and Android Open Source DB [7] (26). We then use statistics provided by Cyrket [21] to identify the applications with more than 50K downloads. This left us with 25 applications. Since we are interested in applications with certain level of development maturity, we refined our selection criteria by retaining applications that 1) had an active and public bug tracking repository (excluded 15 apps), 2) had multiple versions (excluded none), and 3) shipped with a unit test suite (excluded 5 more). These constraints left us with 5 applications: *myTracks*, a geotagging application; *XBMC remote*, a remote control for the XBMC media center; *Barcode Scanner*, a retriever of online information; *Keepassdroid*, a password keeper that syncs with cloud services; and *SipDroidVoIP*, a voice over IP client.

Table II
CLASSIFICATION OF CONFIRMED AND FIXED BUG REPORTS

Application	Bug reports - Confirmed and Fixed		
	Total	Exceptions	Exceptions with External Resources
Barcode Scanner	136	47	17 (13%)
Keepassdroid	45	23	16 (36%)
myTracks	46	21	11 (24%)
SipDroidVoIP	252	39	27 (11%)
XBMC remote	105	39	27 (26%)

Table I provides a summary of the artifacts. Column "Resource API" shows the types of resource APIs the application calls. Since many external resources are managed through APIs, we decided to focus on the ones that use the Android API to manage communication and sensing. The version of each application that we use for the evaluation corresponds to the oldest version against which a bug report is available. Columns "Version" and "LOC" denote the chosen version and lines of code respectively. Column "Unit Test Suite" denotes the number of unit tests shipped with each application, and the time it takes to run those tests.

The repositories held thousands of bug reports. Among those, 584 were confirmed by the developers and addressed through a code revision. Utilizing the bug repositories' search facilities we then identified the bug reports that mentioned keywords such as *exception, throw, catch* which resulted in 282 reports. We then examined each of those bug reports to identify the ones that were caused by incomplete or erroneous coding of exception handlers. We found that 169 of the confirmed and fixed bug reports had to do with poor implementation of the exception handling constructs. Last, we analyzed each report in further detail to identify the ones that had to do with exceptions thrown when services associated with the localization, bluetooth, network resources were invoked.

Our detailed findings are reported in Table II. Column "Total" shows the total number of reports that were confirmed and linked to a fix. Columns "Exception" and "Exceptions with External Resources" report the counts of the more detailed analyses. Even for the small set of external resources we examined, the data suggests that 29% of the confirmed and fixed bugs have to do with poor exceptional handling code, and that 17% correspond to interactions with external resources.

These findings must be taken in the light of the scope of the study. We study five applications in a domain where we suspected troubles with exception handling because of the

```
 1 //M1: Sets the correct response format
 2 public boolean setResponseFormat(...) {
 3     while(!ret.contains('OK'))
 4         ret = query('setResponseFormat', ...);
 5 }
 6 //M2: Returns system info
 7 public String getSystemInfo(...) {
 8     return query('GetSystemInfo', ...);
 9 }
10 //M3: Returns list of media currently playing
11 public ICurrentlyPlaying getCurrentlyPlaying() {
12     list.add(query('GetCurrentPlaying','music'));
13     list.add(query('GetCurrentPlaying','video'));
14     mediaFiles = list.get('Filename');
15     ...
16 }
17 //Executes an HTTP API method
18 public String query(String method, String par) {
19     try {
20         URL url = formatQueryString(method, par);
21         URLConnection uc = url.openConnection();
22         uc.getInputStream();
23         ...
24     } catch (Exception e) {
25         mErrorHandler.handle(e);
26         return "";
27 }}
28 //Centralized exception handler
29 public void handle(Exception exception) {
30     try { throw exception; }
31     catch (NoSettingsException e) {...}
32     catch (NoNetworkException e) {...}
33     catch (WrongDataFormatException e) {...}
34     catch (HttpException e) {...}
35     catch (IOException e) {
36         startActivity(new Connection(...));
37 }}
```

Figure 1. Code excerpt (with comments added for readability) from XBMC Remote Revision 220 with a faulty exception handling mechanism.

types of resources being managed. Still, there are literally tens of thousands of applications like this deployed and we expect for these findings to generalize to them. We also echo the threat to validity often raised when dealing with such open source repositories in terms of their noise which is compounded by the fact that the analysis of the bug reports required some level of judgement by one of the authors and as such is subject to experimentation bias. The data is available to enable other researchers to control it [1].

Even in light of these limitations, these findings support what we had informally observed in terms of the magnitude of the problem associated with exceptional behavior caused by external resources. With these issues in mind, we proceed to illustrate what makes exposing these faults difficult and how the proposed approach addresses those challenges.

III. Difficulties in Exposing Faulty Exceptional Behavior

Figure 1 shows a code excerpt from the Android application XBMC Remote which implements a remote control for the XBMC media player and multimedia center application. Lines 1-16 of Figure 1 list three methods that are called sequentially during the initialization phase of the application. The method *setResponseFormat* attempts to format an http request for subsequent data fetching; method *getSystemInfo* fetches basic system information of the XBMC media center; and method *getCurrentlyPlaying* fetches a list of currently playing music and videos. All three methods use a subroutine *query* (Lines 17-27) to communicate with the server application. The *query* method attempts to establish an URL connection and fetch contents. The API call at line 21 can throw an *IOException* if the connection is down. This exception is caught by the **catch** clause (Line 24), which calls *mErrorHandler.handle*, a centralized error handler (Lines 28-37). When the **catch** clause for *IOException* (line 35) is executed, an attempt to renew the connection is performed.

A bug report issued against this code and deemed high priority by the developers, indicated that the application crashed at launch when the web connection became intermittent[2]. The report's trace log shows that the crash was caused by an elaborated series of successful and failed queries to the API managing the connection. For the failure to be exposed, queries at lines 4 and 8 needed to succeed, but queries at lines 12 and 13 had to throw an exception. These exceptions caused for line 14 to reference a null variable *mediaFiles*, and a subsequent dereference on that variable caused a *NullPointerException* and crashed the application.

Similar issues associated with the poor handling of exceptional events triggered by external resources represent 26% of the bugs reported in XBMC Remote.

This example conveys two interesting points. First, regarding the difficulties of developing tests for such exceptional scenarios, detecting such faults would require: 1) the control of an external resource (connection) to turn it on and off in a prescribed manner, and 2) the systematic exploration of the space of exceptional program behavior that can be triggered through the invocation of an external resource.

Second, regarding the capabilities of existing validation techniques, we note that more precise program representations that include exceptional edges may help to detect components that require additional tests to cover exceptional behavior but assistance to develop such tests is lacking. We also observe that simply covering exceptional edges may not be enough as some of the sequences of throws resulting in failures are quite elaborate. Alternative approaches that mine common patterns of exception handling and use those to detect potential anomalies present different tradeoffs as they may be effective for simpler patterns, but struggle as the space of exceptional constructs becomes richer. We discuss and compare some of these approaches later in the paper.

IV. Test Amplification

We propose an approach for detecting faulty implementations of exception handling constructs through the exhaustive

[1]http://cse.unl.edu/~pzhang/data/icse12.zip

[2]http://code.google.com/p/android-xbmcremote/issues/detail?id=84

amplification of the space of exceptional program behavior explored by each test and associated with an external resource up to a user-defined number of invocations. Conceptually, the approach first instruments the target program by adding a mocking device to take the place of the external resources of interest, it then amplifies each original test by exposing it to all possible resource thrown exceptions while monitoring for program failures.

A. Overview with Example

Following with the example of the previous section, the external resource of interest is *connection*, which corresponds to API calls on *URLConnection* in the *query* method (line 21). The approach instruments the program to enable the mocking of the invoked API methods so they can throw an exception. The approach is exhaustive in that it explores all the possible mocking patterns bounded by a specified number of invocations to the *connection* resource API, which we call the *mocking length*.

The exploration process with a mocking length of five is illustrated in Figure 2. The nodes correspond to calls to the API containing the resource of interest, the edges represent whether an exception is thrown (1) or not (0), and the tree height corresponds to the mocking length. To simplify the explanation, we label the nodes with the line number of the *query* method call sites. A path from the root to a leaf node represents a specific mocking pattern explored by an amplified test. So, for example, the most-left path corresponds to a normal execution of an amplified test where no exceptions are thrown. The right-most path corresponds to a pattern where all calls to the target API throw an exception.

Out of the 2^5 patterns explored, four patterns (labeled FP1 - FP4) revealed terminating program failures (marked with the bolder edges and ending in a star). FP1, for example, corresponds to the mocking pattern that operates normally for the calls to the API launched in lines 4 and 8, but throws an exception at lines 12 and 13 that lead to a crash. We note that this pattern, which shows queries that work normally when invoked at lines 4 and 8, and fail at lines 12 and 13, matches the situation described in the bug report.

For each failure, the approach generates a report that records 1) type of resource being mocked, 2) mocking pattern, 3) type of exception being thrown, and 4) call trace after the exception is thrown. Figure 3 contains the failure report corresponding to FP 1 in Figure 2. Such failure reports are used to communicate with the user and also as a basis for various types of filters to control the number of tests kept or shown to the user. For example, a simple failure-filter prunes all reports that did not lead to an exceptional termination caused by the mocked resource. A distinct-failure filter prunes reports with the same type of exception thrown and the same call trace prefix, only reporting the tests with the shorter mocking pattern (the intuition behind this decision is that a shorter pattern is easier to understand and helps debug the failure.) According to this latest filter, in our example, all failure reports had the same type of exception and trace so we just keep FP1.

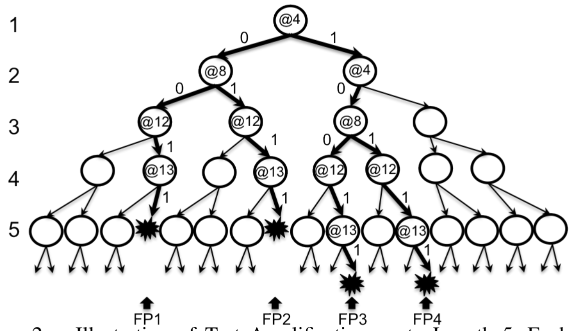

Figure 2. Illustration of Test Amplification up to Length 5. Each path from the root to a leaf corresponds to a mocking pattern. 32 patterns are explored and 4 failing patterns (marked as FP1-FP4) are found.

```
Mocked Resource API: java.net.URLConnection
Mocking Pattern: Normal(4), Normal(8), Throw(12),
  Throw(13)
Exception Type: IOException
Trace:
java.io.IOException thrown
xbmc.android.util.ErrorHandler.handle
xbmc.android.util.Connection.query
xbmc.httpapi.client.ControlClient.getCurrentPlaying
xbmc.httpapi.type.ICurrentPlaying.add
xbmc.httpapi.type.ICurrentPlaying.get
java.lang.NullPointerException thrown
...
Thread main exiting due to uncaught exception
```

Figure 3. Failure Report corresponding to FP1

Equipped with an intuition of the challenges and the approach, we now proceed to define them more formally.

B. Problem Definition

Given program P, test suite T, and API R managing the resources of interest to the tester, we formally define the problem domain as follows.

Definition 4.1: Resource-sensitive function calls: set of calls F in P to functions in R. Each call has an associated target function name and location.

In the previous example, F would contain calls such as $openConnection_{21}$.

Definition 4.2: Resource-sensitive function call sequence: $\overline{seq_i} = [f_1, f_2, \ldots, f_n]$, where $f_j \in F$, generated by the execution of $t_i \in T$ on P. More generally, $SEQ_T = \{\overline{seq} | \forall t \in T, seq = exec(P_F, t)\}$.

For example, assuming that the bug report had an associated test t_{84} then $\overline{seq_{84}} = \{oC_{21}, oC_{21}, oC_{21}, oC_{21}\}$ (oC is the abbreviation of $openConnection$.)

Definition 4.3: Space of Exceptional Behavior: each call in $\overline{seq_i}$ either returns normally or raises an exception defining a space of exceptional behaviors $S_{t_i} = \overline{seq_i} \times (normal, exception)$. We call each one of the products an exception mocking pattern.

Following with the previous example, assuming the resource can raise just one type of potential exception, $\overline{seq_{84}}$ may reveal a space of up to 16 (2^4) behaviors depending on whether each invocation returns normally or not. Consequently, $S_T = \bigcup_{\forall t_i \in T} S_{t_i}$, the union of all exceptional behavior spaces bounded by each t_i, forms the space of exceptional behaviors due to the execution of F under T that P must be able to handle.

Definition 4.4: Test Amplification for Exceptional Behavior: transformation $T \rightarrow Amplified_T$ where $\forall S_{t_i} \in S_T$ $\exists amp_{t_i} \in Amplified_T$ designed to cover S_{t_i}.

There are three aspects of $Amplified_T$ worth noticing. First, program dependencies among the invocations may limit the reachability of the S_T nodes. There may also be some patterns that are explored through mocking, but could not be experienced in practice given the design of the external resource. Second, $Amplified_T$ may reveal invocation sequences that were not in SEQ_T either because of their order or because they include new calls. Such sequences can be translated into new tests that enrich T and consequently S_T. Third, as defined, the number of paths in S_T can grow exponentially. To control this growth, we bound the size of each $\overline{seq_i}$.

Definition 4.5: Bounded Test Amplification for Exceptional Behavior: given a bound k, we define bounded test amplification for exceptional behaviors as the test transformation, as per Definition 4.4, but on the following exceptional space:

$$S_T^k = \bigcup_{\forall \overline{seq_i} \in SEQ_T} head(\overline{seq_i}, k) \times (normal, exceptional)$$

where $head(\overline{seq_i}, k)$ returns the first k elements of each $\overline{seq_i}$.

C. Approach Architecture

Figure 4 illustrates the architecture for the systematic amplification of tests. There are five core components.

The sequence *collector*, takes as input P, R, and T. It instruments P to capture all calls to R, and it then runs all tests in T to produce SEQ_T. Tests that do not contribute a sequence are dropped so that only $T' \subset T$ are further considered. The exceptional space *builder* takes as input SEQ_T, P, R, and bound k. The builder analyzes R to derive the types of exceptions that the resource can generate. Given those exception types and SEQ_T, the builder generates S_T^k, a space of the exceptions that may be raised. k is used to bound the space depth. The *mocking* component takes P and R and it generates P' so that all invocations to R can be forced to return an exception of the allowed types. This component facilitates the exploration of a mocking pattern consisting of a sequence of invocations to R that may return normally or raise an exception.

The *explorer* component systematically attempts to amplify the tests in T' to cover S_T^k by mocking the behavior

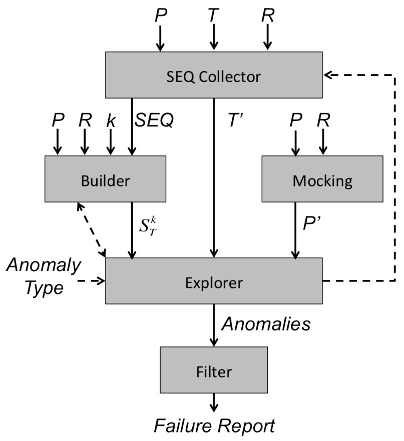

Figure 4. Amplification Architecture

of R as perceived by P' while re-executing the tests. As an amplified test executes, the explorer will check for two types of anomalies: abnormal termination (program terminates due to uncaught exceptions) or abnormal execution time (duration of amplified test is greater than the non-amplified by a certain threshold). If either anomaly is detected, the explorer will generate a trace of invocations to R together with their outcome. The *filter* component will then take those anomalies and report the ones meeting a predefined criteria such as whether to include amplified tests whose mocking patterns and outcome were already revealed by other amplified tests.

The description of the approach architecture overlooks some interesting aspects that we have considered but not fully developed. First, in this work we pursue exhaustive exploration of the space in S_T^k. However, the architecture also allows more selective exploration of the space to accommodate cost-effectiveness tradeoffs. For example, regular expressions could be provided to the *builder* to constrain the space S_T^k to specific exceptional patterns that are known to be problematic for a particular system or resource. Second, the dotted line from the explorer to the collector in Figure 4 alludes to the potential for establishing a feedback loop where the new invocations of the resources or the new outcomes of existing invocations revealed by the explorer are used to enrich the S_T^k. Third, the dotted line between builder and explorer indicates that these processes may be coupled so that the space is defined incrementally as it is being explored. So the *builder*, for example, could define a space for one pattern and pass it to the *explorer*, which in turn will influence the *builder* in the formation of the rest of S_T^k. This type of lazy space definition may be particularly effective at early stages of the amplification where the size of the space is unknown. Fourth, the types of anomalies considered could be extended by monitoring for invocation of unexpected handlers due to exception inheritance, or exceptions that are subsumed without proper handling. Last, our architecture does not prescribe how the instrumentation and exploration should occur. As we shall see, We use AspectJ to provide mocking capability, but other established frameworks, such as JMock[9] or EasyMock[24], could also be used.

D. Implementation

In this section we briefly describe the most interesting aspects of an implementation of our approach in the context of Java 1.6 programs and JUnit test suites.

Collection and Mocking. We use AspectJ [25] 1.6.10 to instrument the artifacts to collect SEQ_T, mock the API calls and inject exceptions, and to detect anomalies at run-time. We define point cuts for the sites of the calls to the target API and an advice that consists of a *throw* statement for each type of exception that the invocation can throw. The exception types are obtained by scanning the API methods signatures.

Building and Exploration. Our builder and explorer work independently. First, the builder analyzes the signatures of the methods in R invoked by P as indicated by SEQ to determine the type of exceptions they can throw. It then derives the space of exceptions. The explorer then attempts to perform a depth-first space search, amplifying each test with each of its mocking patterns, and running them one at a time. For each $t_i \in T'$, the explorer will clone each test up to 2^{m^k} times, where m is the number of types of exceptions the invocations can throw, and k is the bound set by the user. Each cloned test is then coupled with a unique mocking pattern to amplify its behavior.

The Android Environment. Android applications compile into dex format (Android bytecode) and execute on the Dalvik VM and the Android Virtual Device (AVD) [8]. This required for us to change their building process so that AspectJ could apply the advices before the platform-specific file formats were generated.

V. Evaluation

In this section we address the following research questions

- RQ1: How cost-effective are the amplified tests in detecting anomalies in exceptional handling code?
- RQ2: To what extent do the detected anomalies represent real faults?

A. Study Design and Implementation

We studied the Android applications and the resources listed in Table I. For each of those resources, we collected the checked exceptions their public methods could throw (i.e., for the *android.bluetooth* methods that would include *ConnectionTimeoutException*, *SocketTimeoutException*, *UnknownHostException*).

We set the approach parameters as follow. To set the approach mocking length we mimic the process a tester would follow. Ideally a tester would select the smallest mocking pattern length that still detects all the faults. This value, however, is *not* known in advance, and it is different across programs, tests, and resources. So the tester must pick a reasonable starting value that may be refined over time. Similarly, we select a length of 10 which seems reasonable considering the time it takes to explore the exceptional space of these applications. This decision also echoes with Jackson's small scope hypothesis [13] which conjectures that exhaustive testing up to a small bound will detect most faults. As described next, our findings confirm this hypothesis, as the majority of the faults can be detected with a length of 5. We set the filtering component to remove duplicate reports (those that include the same failing trace of invocations to R, thrown exceptions, and test outcome), whether produced by a single test or across multiple tests.

We assess the effectiveness of the approach by generating $Amplified_T$ from the unit test suites that came with the artifacts, and then running the amplified tests on the respective artifact. We analyze the anomalies revealed by the amplified tests from two perspectives. First we compare them against the bug reports in terms of precision (the degree to which a detected anomaly maps to a bug report in the repository) and recall (the degree to which bug reports in the repository are included in the set of detected anomalies). Second, we check whether anomalies detected on earlier versions disappear in later versions as the code may have been fixed but such fix may not have been reported. We measure costs in terms of the *size* of the amplified test suite and the *time* required to generate and execute it. We discuss other costs in Section V-D.

The Android applications required different Android API versions ranging from 1.6 to 2.2. The study was conducted using a 2.4 GHz Intel Core 2 Duo machine with 4 GB memory, running Mac OS X 10.6.6.

B. RQ1: Cost-Effectiveness in Detecting Anomalies

We study this research question by amplifying the unit tests that came with the artifacts, executing them and analyzing the behaviors they expose.

We start by providing a characterization of the amplified test suite, $Amplified_T$, in Table III. For each artifact-resource combination, we report the number of original tests exercising each one of the resources, the number of transformed tests, and the time required to execute them. We omitted the test amplification time for $Amplified_T$ because it is trivial (e.g., for the biggest application with the most tests, Barcode Scanner, the generation time was 24 seconds).

First, we note that the initial test screening process helped to eliminate many original tests that do not reach the target exceptional constructs. For example, for Barcode Scanner, just 18% of the original tests were able to exercise the external resource *android.net*. Second, as expected for an exhaustive testing exploration approach, the number of amplified tests and time required to execute them are generally large, with the most prominent case being the *myTracks* and *android.location* combination which took more than 27 hours to finish. As we later show, our bound choice may have been too conservative and hence unnecessarily costly.

The cost is compensated, however, with noticeable coverage gains and anomalies detected. The $Amplified_T$ suite provides on average a gain of 62% of coverage on the catch-blocks, hinting at the potential of the automated amplification to expose new exceptional behavior.

Next, we analyze the impact of these newly explored behaviors. Table IV accounts for the anomalies the amplified test suite found, and provides a characterization of the mocking patterns that revealed those anomalies. The amplified test suite identified 115 unique anomalies, an average of 23 per application. Of those anomalies, 65% corresponded to mocking patterns that consisted of more than just a single throw in response to a call to the API. Still, we found that the average mocking pattern length is just over 3, indicating that the cost of the approach could be drastically reduced by selecting much tighter exploration bound.

To better convey the effects of the mocking pattern length on testing costs and fault detection effectiveness, we reported the study with lengths 1 and 5. The results were consistent throughout. The test suite generated with a length of 1 failed to detect any anomalies as it did not explore a large part of the interesting exceptional behavior. With a length of 5, 94% of the anomalies detected with a length of 10 were found with less than 5% of the cost. However, a bound of 5 would have missed 7 anomalies that corresponded to faults in *Keepassdroid* and *XBMC remote*, so the gains may come at a significant cost for some artifacts.

These data suggest that an amplified test suite can provide significant coverage gains of exception handling code, detect many anomalies in existing popular applications, and do so through the exploration of patterns that are non-trivial yet concise. We now proceed to investigate these anomalies.

C. RQ2: Anomalies & Failures

First, we assess the anomalies by executing the amplified tests revealing an anomaly on the newest version of the applications, evaluating whether the anomalies are still present.

Table V summarizes our findings. The new versions of the applications corresponds to the latest version available on June 2011. Column "Found" was copied from column "# Anomalies" in Table IV for comparison. Column "Remain" reports the number of anomalies that still exist in the newest versions, and columns "# More than one throw" and "Length" show the characteristics of the mocking patterns that are used to detect these anomalies. *Table V suggests that the majority of the anomalies (77%) that appear in older versions are not present in the latest versions, which adds credibility to the value of the anomalies detected by the approach.* For example, the amplified test that exposed the fault in Figure 1 did not fail on a newest version. We also note that the "remaining" anomalies are all induced by more than one throw in response to the API call. This result may indicate that the "remaining" anomalies represent a type of faults that is harder to find. For example, in *Keepassdroid*, one "remaining" anomaly requires a mocking pattern that throws an exception on the second of three consecutive *execSQL* calls, but succeeds on the first and third calls. This failing scenario, although possible, is very rare among this type of resource APIs.

Now we shift to the second step of the assessment on the detected anomalies, mapping the anomalies to the bug reports. We compute the percentage of bug reports associated with the anomalies found and the percentage of anomalies that are included in the bug reports. The first metric gives us a notion of the approach completeness (also refer to as recall) while the second provides a lower bound on the approach preciseness. We note that both metrics are inherently limited (e.g., they assume that all faults have been found and that all found faults have a bug report) but they are useful to pinpoint strengths and weaknesses of the approach.

The process to map anomalies to bug reports is as follows. First, we search among the bug repository for instances of the location (call to target API or exception) where an anomaly was detected. If the resulting bug report includes a stack trace, which is the majority of cases, we will match it with the stack trace associated with the anomaly. Otherwise, we retrieve the submitted fix to the bug, and inspect the code to determine if the fix was applied to the ill coded exception handling module that was captured by the anomaly reported by the amplified test.

Figure 5 shows the mapping of anomalies to bug reports. Each bar represents the total number of anomalies detected by the amplified suite for one artifact, and the three levels of shade indicate the number of anomalies that were 1) detected by our approach, fixed by the developers, and matched to bug reports; 2) detected and fixed in a later version (as per the previous assessment), but could not be matched; 3) just detected. Note that an anomaly that was detected, fixed but not matched could be the result of either: 1) the issue was fixed as a side effect of other submitted code changes; 2) the problematic code module was refactored during program evolution; 3) it was not reported.

Figure 5 suggests that, on average, 66% of the anomalies can be traced to faults reported in the bug repository. Furthermore, 11% of the anomalies, although not matched, are fixed in later versions, indicating that these detected anomalies do expose exceptional behaviors in practice. On average, only 23% of the detected anomalies have unconfirmed status, which may constitute faults that are yet to be found or false positives. For example, myTracks has a simulation mode through which locations are defined via a KML file. If such mode is activated, location API calls cannot fail because they do not interact with real location providers. Our tool overlooks this possibility and mocks such API calls, which may lead to false positives.

Table III
COST AND COVERAGE CHARACTERIZATION

Application	Resource	# Tests		$Amplified_T$	Coverage	
		T'	$Amplified_T$	Exec. Time (hrs)	Original	$Amplified_T$
Barcode Scanner	java.net	21	21,504	13.1	21%	81%
Keepassdroid	android.database	12	12,288	9.5	12%	75%
	java.net	27	27,648	20.6		
myTracks	android.database	21	21,504	16.1	7%	81%
	android.location	39	39,936	27.6		
SipDroidVoIP	java.net	32	32,768	21.9	10%	74%
XBMC remote	java.net	41	41,984	21.7	23%	76%
	android.bluetooth	19	19,456	11.4		

Table IV
ANOMALIES DETECTED

Application	# Anomalies Detected	Mocking Pattern	
		# More than one throw	Length (avg/max)
Barcode Scanner	19	9 (47.3%)	2.6 / 5
Keepassdroid	11	9 (81.8%)	5.1 / 10
	18	10 (55.6%)	2.1 / 5
myTracks	19	15 (78.9%)	3.4 / 5
	9	5 (55.6%)	3.2 / 5
SipDroidVoIP	16	9 (56.2%)	1.9 / 4
XBMC remote	14	11 (78.6%)	4.8 / 10
	9	6 (66.7%)	3.1 / 4

Table V
STATUS OF DETECTED ANOMALIES IN NEW VERSIONS OF THE APPS

Application	Newest Version	# Anomalies		Mocking Pattern Remain	
		Found	Remain	# More than one throw	Length (avg/max)
Barcode Scanner	r1692	19	3	3 (100%)	3.3 / 5
Keepassdroid	v1.8	11	2	2 (100%)	4 / 5
		18	3	3 (100%)	3.7 / 5
myTracks	r314	19	3	3 (100%)	3.7 / 5
		9	4	4 (100%)	3 / 5
SipDroidVoIP	r613	16	4	4 (100%)	3.8 / 4
XBMC remote	r713	14	4	4 (100%)	3.3 / 5
		9	3	3 (100%)	3.7 / 4

Figure 6 shows the mapping from bug reports associated with external resources to the anomalies. The shaded portions of the bars represent bug reports that were detected by $Amplified_T$, and the white parts represent bugs that were reported as anomalies. *On average, 65% of the reported bugs are matched to the anomalies detected by $Amplified_T$.* We then proceeded to analyze those numbers in more detail to determine under which circumstances our approach failed to detect a reported bug. The most common reason was the limited coverage of the available unit test suite. For example, myTracks Issue #172 describes a crash when saving a new marker to a track. The triggering condition for this bug requires pausing and resuming tracking before inserting a new marker. This work flow, however, was not covered by any of the original tests. A second reason was the lack of control on some of the external factors other than the invocation of resource APIs. For example, myTracks issue #137 describes a bug where the user gets many error messages when trying to upload tracks to the Google Maps service. Reproducing the bug requires controlling two factors: a Google authentication API that fails all the time, and a specific scheduling order for two threads. Our approach controls the first factor, but does not have control over the second.

While the first shortcoming can be addressed by devoting more testing efforts, the second issue requires extending our approach to include a more sophisticated instrumentation mechanism to capture and replay the threads schedule.

D. Threats to Validity

In addition to the limitations we mentioned in Section II regarding the scope of the programs we studied and the potentially noisy nature of analyzing bug reports, we introduced some other threats in this section. More specifically, our choice of versions was deliberate to maximize the number of faults that could be detected. In practice, the deltas will be smaller and is not certain how the collected metrics will be affected. Second, the metrics we utilized are just partial proxies for the cost-effectiveness of the approach and are highly context dependent. In a more realistic setting, the cost of the approach would also include the time required by developers to interpret the tool's outputs and exclude the false positives. Third, our focus was on particular types of exceptions that we deemed interesting based on our experience. Although the approach is applicable to other exception handling constructs and resources, its cost-effectiveness may vary according to the difficulties associated with particular the resources.

E. Extended Domain and Alternative Approach

In this section we compare the proposed approach against the CAR-Miner tool developed by Thummalapenta et al. [22]. This tool represents one of the latest attempts targeting the detection of errors in exception handling code. Instead of amplifying or generating a test suite, CAR-Miner mines exception handling rules from the source code of a pool of applications and then checks whether a target program violates those rules.

Our comparison with CAR-Miner is focused on HsqlDB, the artifact on which CAR-Miner detected the most faults [22]. HsqlDB is a database application with almost 30KLoc in version 1.7.1 (the one used in the original study) and 551 unit tests. We take advantage of the public availability of this application to examine its bug reports as we did for the

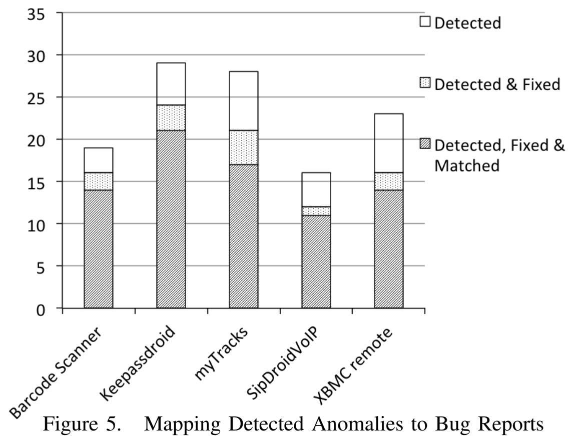

Figure 5. Mapping Detected Anomalies to Bug Reports

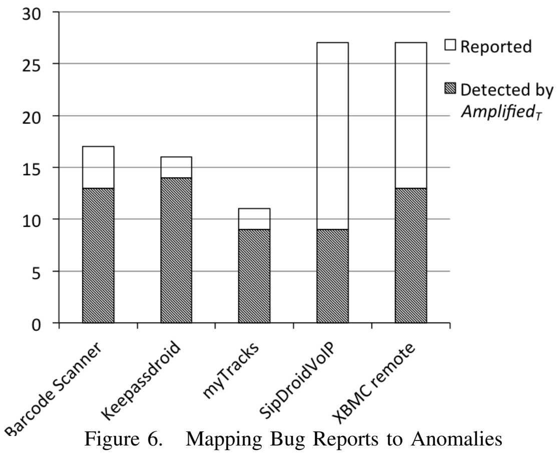

Figure 6. Mapping Bug Reports to Anomalies

Android applications in Section II. The examination of 178 confirmed bug reports that led to code revisions revealed that 58 of them (32%) were caused by poorly handled exceptions and 14 of them were caused by the external resource *java.db* (the core external resource used by this application). This seems to indicate that the proper handling of exceptions in HsqlDB is as challenging as for the Android applications, but the effect of external resources is smaller as the Android applications seem to rely more heavily on external sensors and communication services.

CAR-Miner detected 51 instances of broken rules in HsqlDB and the authors were able to map 10 of those to bug reports. Upon closer examination we noticed that three of those bug reports were later rejected by the developers, which leaves CAR-Miner with seven broken rules that map to reported bugs. One of these three instances, #1896443 is particularly interesting because it points to one of the limitations of this type of approaches caused by the use of intraprocedural analysis which means that complicated exception handling patterns are often missed.

Amplifying the HsqlDB test suite with our approach resulted in 97,280 amplified tests that take 23.7 hours to execute and find 22 anomalies. Among the anomalies found are the 7 confirmed faults found by CAR-Miner, and two other faults from the repository. One such instance, bug report #1800705, shows a case where a raised exception caused a DB connection not to close properly. Again, since the exception is not thrown by an API call in the method but rather by a a custom function that re-throws exceptions for the API nested inside, CAR-Miner is not able to detect it. In terms of false positives, as expected, a mining approach is less beneficial with over 85% of false positives (51 anomalies reported from which 7 were confirmed faults). For our approach, the same criteria gives a false positive rate of 59% (9 of 22 were confirmed bugs).

F. Preliminary Case Study

To start addressing some of the limitations we identified in terms of the scope of the work and its lack of development context, we performed a case study of the approach assisting Android applications developers. Our case study occurs in the convenient context of BusLinc [12], an application for the Android platform being developed by a team of senior Computer Science students at University of Nebraska-Lincoln, two professional Android developers, and the IT division of the Lincoln StarTran transportation service. The application communicates with StarTran's location server and, combined with a smartphone's current location, can provide users detailed bus route, nearest bus stop, and real-time bus schedule information.

The application primarily uses two types of resource APIs, a network API that communicates with a server, and a location API that is used to obtain the device physical location. We use our approach to assist with the testing of exception handling behaviors of the network API, which was used more extensively than the location API. We generated 2048 amplified tests based on just two automated system tests provided by the developers. In 24 minutes the approach reported 4 distinct anomalies, all of which were terminations caused by poorly handled exceptions, and involved complex mocking patterns.

One such anomaly reflected a situation where the network communication succeeded in checking server availability and route updates, but failed at retrieving the actual bus routes. Consequently, the route object was stored as a null pointer and a subsequent reference to the object crashed the application. Two other anomalies of similar type were detected, one for checking out the bus stops, the other for vehicles. The last anomaly was associated with the logic for displaying a route on the Google Maps overlay, where the waypoints on the route were null objects due to a network failure in updating them.

We met with two of the developers to gain further insights on these anomalies. During the meeting, the developers were directed towards the code locations with the poorly implemented exception handlers that cause the crashes, and were asked to construct failing scenarios for the network API usage that could lead to these crashes. After 15 minutes, the developers failed to identify scenarios for any of the four failures. We then provided and explained the failure reports. With those at hand, the developers recognized and confirmed the problems. Based on these preliminary findings, it seems that the approach was useful in revealing non-obvious problems with their exception handling constructs.

VI. Related Work

Our work relates to techniques aimed at detecting faults in exception-handling constructs. This includes efforts in two areas: mining specifications and coverage representations.

Techniques to mine specifications of exceptional behavior (e.g., [1], [22], [26]) operate by mining rules from a pool of source code and then checking a target program for violations of the mined rules. The existing techniques vary in the type of rule structure they target, the scope of the analysis, how the pool is built, and the challenges introduced by the target programming language. So, for example, while Thummalapenta et al. [22] use conditional rules, intraprocedural analysis, a pool of code enriched with code from a public repository, and Java code, Acharya et al. [1] use association rules in C code which does not support explicit try-catch structures. All these approaches' performance depends on the quality of the pool of source code, the precision and completeness of the analysis, and the training parameters that define what constitutes an anomaly. It is not evident from the reported studies whether exception handling constructs that can take so many different forms and occur in such large scopes can be effectively mined as rules. Our approach is different from these techniques in that it transforms the problem into a space exploration problem and systemically traverses the space to find faults.

The second thread of related work focuses on the development of more precise flow representations and analyses that include control flow edges to and from exception structures [4], [10], [11], [15], [17]. Sinha et. al [17] were among the first to build a program representation with explicit exception constructs, i.e. throw statements and try-catch-finally structures, and propose the use of this representation to calculate links between exceptions and their corresponding handlers. Later, they propose to use this information to build a toolset that helps with test case selection and maintenance [18]. Choi et.al [4] proposed one of the many refinements that followed, either to improve efficiency (e.g., by grouping edges by types) or preciseness (e.g., by combining the static analysis with some form of dynamic analysis for refinement [3]). Robillard et al. introduced a model and a static analysis tool, Jex, that adopted a similar approach but oriented towards providing development support [15]. Fu et. al [10], [11] extended the control-flow analysis by considering re-throwing exceptions which they argue is common among layered software, and by using the results of exception-flow analysis to improve the coverage of exception handling code through dynamic fault injection. This last piece of work is similar to ours in that we both adopt mechanisms for mocking the APIs. The difference is that this approach is guided to cover the exceptional edges while ours attempts an exhaustive coverage of mocking patterns.

Our work was inspired by Jackson et al. small scope hypothesis [13] and by its application to testing by Coppit et al. [5]. In addition, although not targeting exceptions, we are aware of complementary approaches that target unreliable and noisy resources used by ubiquitous context aware applications [16], [23], [28].

VII. Conclusion

We have introduced a simple yet cost-effective approach aimed at amplifying existing tests to validate exception handling code associated with external resources. The technical merit of the approach resides in defining the challenge as a coverage problem over the space of potential exceptional behavior, and the systematic manipulation of the environment to cover that space. Although our focus was motivated by faults triggered by noisy and unreliable external resources, the approach could be beneficial in other scenarios where there is limited understanding or confidence on an API.

The findings of our studies indicate that amplified suites are powerful enough to report anomalies for 65% of the reported faults of this kind, and precise enough that 77% of the detected anomalies led to code fixes. Our approach outperforms a state of the art approach in precision and recall. In addition, the preliminary case study illustrates the approach potential to assist developers.

There are several directions to build on this work. We want to enable more selective exploration of the exceptional space by introducing a language to specify the mocking patterns. Going a bit further, we are interested in studying the interplay between the proposed approach and more precise control flow representations and analysis that may help inform what mocking patterns are worth exploring, and what tests may not be worth transforming given the paths they traverse. In the longer term, we are interested in investigating other test transformations that exploit existing test suites to cost-effectively improve the validation process.

Acknowledgments

This material is based in part upon work supported by NSF Award CCF-0915526, and by AFOSR Award #9550-09-1-0687. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFOSR, or NSF. We thank D. Rosenblum for his feedback.

References

[1] Mithun Acharya and Tao Xie. Mining api error-handling specifications from source code. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2009.

[2] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[3] Raymond P.L. Buse and Westley R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 273–282, 2008.

[4] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. *SIGSOFT Software Engineering Notes*, 24:21–31, 1999.

[5] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, and Kevin J. Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31(4):328–339, 2005.

[6] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the Joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 185–194, 2007.

[7] Android Open Source DB. http://www.aopensource.com/.

[8] Android Developers Doc. http://developer.android.com/index.html.

[9] JMock Framework. http://www.jmock.org/index.html.

[10] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *Proceedings of the International Conference on Software Engineering*, pages 230–239, 2007.

[11] Chen Fu, Barbara G. Ryder, Ana Milanova, and David Wonnacott. Testing of java web services for robustness. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 23–34, 2004.

[12] BusLinc Code Host. http://code.google.com/p/android-unl2011/.

[13] Daniel Jackson and Craig Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996.

[14] Wikipedia: List of open source Android applications. http://en.wikipedia.org/wiki/List_of_open_source_Android_applications.

[15] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12:191–221, 2003.

[16] Michele Sama, Sebastian G. Elbaum, Franco Raimondi, David S. Rosenblum, and Zhimin Wang. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, 36(5):644–661, 2010.

[17] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, 26:849–871, 1999.

[18] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proceedings of the International Conference on Software Engineering*, pages 336–345, 2004.

[19] Le Wiki Koumbit: Open source Android applications. https://wiki.koumbit.net/AndroidFreeSoftware.

[20] Trac: Open source Android applications. http://trac.osuosl.org/trac/replicant/wiki/ListOfKnownFreeApps.

[21] Cyrket Mobile App Statistics. http://www.cyrket.com/m/android/.

[22] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the International Conference on Software Engineering*, pages 496–506, 2009.

[23] Zhimin Wang, Sebastian G. Elbaum, and David S. Rosenblum. Automated generation of context-aware tests. In *Proceedings of the International Conference on Software Engineering*, pages 406–415, 2007.

[24] EasyMock Website. http://easymock.org/.

[25] The AspectJ Project Website. http://www.eclipse.org/aspectj/.

[26] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, 2005.

[27] Open Street Map Wiki. http://wiki.openstreetmap.org/wiki/Android#OpenSource.

[28] Chang Xu and Shing-Chi Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proceedings of the the Joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 336–345, 2005.