1 Abstract

Large-scale distributed systems are designed to be highly available, but like all human-coded systems, inevitably fall prey to bugs and experience outages. Outages in such large-scale systems however, can be very impactful, as seen for example in the 2011 Amazon Web Services outage that brought down major sites like The New York Times, Reddit, and Quora. The paper “Simple Testing Can Prevent Most Critical Failures” (2014) from the University of Toronto presents this problem in great detail, outlining that in the majority of cases, the root cause of most critical failures lies directly in developer bugs in error handling code. To address the issue, they developed Aspirator, a proof-of-concept bytecode analysis tool that targets Java try-catch blocks in order to warn developers about potential sources for bugs in distributed systems. Aspirator primarily targets cases of empty error handlers, handlers containing "TODO" or "FIXME" statements, and cases where the exception is too general, causing the system to abort in more cases than necessary. This project implements ExceptCheck, a user-friendly Java Exception handler checker in a command-line interface tool in Python. ExceptCheck, unlike Aspirator, is an exploratory source code analysis tool, built primarily off of an open-source Java lexer/parser for Python. ExceptCheck mirrors most of the functionality of Aspirator, to the degree that source code analysis allows, and introduces extra features to ease debugging and encourage further source code analysis in this area. ExceptCheck finds similar coding patterns and bad practices to that of the original paper, but also exposes key differences between bytecode and source code analysis.
2 Background

Researchers at the University of Toronto have found that over 90% of critical failures in distributed data-intensive systems result from incorrect failure handling code. Paper link: https://zoo.cs.yale.edu/classes/cs426/2017/bib/yuan14simple.pdf. In Java code, this typically corresponds to incorrect or bad practice implementation of the `catch` block in `try-catch` blocks. As a result, they developed their own simple error reporting tool called Aspirator, built on top of Chord, an open-source bytecode analysis framework, and presented their findings from running their tool on several open-source distributed systems.

2.1 Catastrophic Errors in Java Exception Handling

The end-to-end sequence involving Java Exceptions is that a fault happens due to a root cause, which then leads to an error, triggering an exception. Finally, this leads to a failure, which is visible to the user or administrator. According to the paper, of the 72 exceptions found during testing, 48 were catastrophic, almost all of which were the result of incorrect error handling. About 35% of those failures were trivial mistakes. 23% were system specific but easily detectable, and 34% were complex bugs. Both Aspirator and ExceptCheck target the trivial mistakes.

Part of the initial stages of this project involved looking through the details and fixes of all 72 of the bug reports in the dataset in order to determine patterns in these bugs in hopes of coming up with ways to potentially generate code to fix some of these bugs. Unfortunately, since the dataset was small, this meant that only around 15 of the reports actually visibly showed trivial mistakes. Upon reconsideration, and since the reports were not very detailed, the code-generation project direction was abandoned in favor of an implementation-based approach.

2.2 Aspirator

Aspirator is a static checker for Java bytecode. This means that it runs on compiled code and uses information from generated .class and .jar files. It is built on top of Chord, a framework for conducting bytecode analysis, and can be found at: https://github.com/diy1/aspirator. It primarily targets and reports the following 3 cases for exception handling:

- An empty error handler (around 25% of catastrophic failures)
- The exception is too general, causing the system to abort in more cases than necessary (around 8% of catastrophic failures)
• The error handler contains "TODO" or "FIXME" statements (around 2% of catastrophic failures)

• Obviously incorrect logic in the error handling that would have been caught by simple testing

Unfortunately, since much of Aspirator’s dependency code is now deprecated and unmaintained, it cannot be run or easily built without major code changes. As a result, the source code could not be extended as originally planned. However, most of the major features were well documented in the source code after a quick inspection of the relevant files: CheckEmptyHandler.java, CheckTerminatingHandler.java, and CheckUnhandledExceptions.java. The main features include:

• If a handler only prints an error message, it will be treated as an empty one

• Tracking position and count information for try-catch blocks

• Tracking the registers pertaining to where return values are stored

• Handling edge cases for when Chord splits up basic blocks incorrectly

• Printing log information for when a potential error is detected out to a .txt file

• Reducing false positives by tracking whether variables assigned in a try block are checked after the catch blocks in an if statement; determining whether the try contains a return, break, or continue statement in order to bypass the catch block

• Identifying whether thrown exceptions are subclasses of handled exceptions in cases of abort or exit overcatch

• Miscellaneous other tasks not related to handled exceptions

This tool was applied to 9 systems (Cassandra, HBase, HDFS, Hadoop MapRed2, Cloudstack, Hive, Tomcat, Spark, Zookeeper) and detected 171 new bugs.

2.3 SpotBugs (Formerly FindBugs)

SpotBugs/FindBugs is also a static bytecode checker for Java. In the original University of Toronto paper, the researchers suggested extending FindBugs to include the features implemented in Aspirator, which also constituted an early attempt in this project. It has now been moved to Spotbugs (https://github.com/spotbugs/spotbugs)
and is an ongoing project. This open source code also ended up being difficult to work with, given that there are so many dependencies and irrelevant features to catching exception handling bugs.

After encountering difficulties in both Aspirator and SpotBugs, the project direction shifted to making a more standalone extension in ExceptCheck, which is hopefully a more accessible and up-to-date resource/tool for detecting Java try-catch related errors.

3 ExceptCheck

ExceptCheck is a Python command-line interface tool built on a Java parsing library that can be run on large files and directories. It supports user-friendly reporting features and requires minimal setup aside from installing the python package six. Since it focuses on catching bugs related to exception handling specifically, it is easily extendable to include even more considerations of common coding practices. Unlike Aspirator, it performs its analysis based on the source code, rather than the bytecode. This poses limitations in some directions, while introducing more possibilities in others.

3.1 Implementation Details

ExceptCheck performs source code analysis, and is built using and extending an open-source library called javalang (https://github.com/c2nes/javalang). Javalang provides a lexer and parser for Java 8 and is based on the Java language specification found at: http://docs.oracle.com/javase/specs/jls/se8/html/. This project involved deeply understanding the inner workings of javalang as well as Java syntax in order to modify the exceptCheck/javalang/parse.py and exceptCheck/javalang/tree.py files to track position information among other variables to better suit ExceptCheck’s specific purpose. The bulk of the exception handling-specific work has been done in the Python files under the top-level exceptCheck/ directory.

In order to robustly analyze source code, ExceptCheck must first build an abstract syntax tree (AST) of the source code in order to represent the major components of a syntactically correct Java file. As a simple example, a piece of code that looks like this:
3.1 Implementation Details

```java
public class Division {
    public void callDivide() {
        try {
            int result = divide(2, 1);
            System.out.println(result);
        } catch (BadNumberException e) {
            // do something clever with the exception
            System.out.println(e.getMessage()).log();
        }
        System.out.println("Division attempt done");
    }
}
```

would be represented as a tree like this:
3.2 Features

ExceptCheck can be run directly from the command line and takes the following options:

- `-f, --filename` followed by a list of Java files to check
- `-d, --dirname` followed by a list of Java directories to check
- `--sourceCheck` to check the source code of the specified files and directories
- `--summary` to print a summary of ExceptCheck findings, which includes a list of files where certain warnings were reported, as well as summary counts for those same warning types, for data aggregation
- `--skip` to specify a list of files to skip checking. This is especially useful for omitting a handful of files in large directories that halt the program due to Java syntax errors.
- `-v, --verbose` for printing all occurrences of `try-catch` blocks as well as warning occurrences as files are checked. This is very useful when inspecting a small set of files to find patterns in the blocks.
- `-dbg, --debug` for printing out in-line with verbose output. This is very useful when inspecting a small set of files to find patterns in the code.

In terms of functionality, ExceptCheck implements all of the characteristics described of Aspirator above, except for tracking registers for return values, and identifying whether thrown exceptions are subclasses of handled exceptions. This is due to the nature of the difference between bytecode and source code analysis, in that only bytecode analysis allows for accessing register information and determining which exceptions can be thrown by a specific method. Additionally, ExceptCheck also explicitly identifies lines where "TODO" or "FIXME" comments exist, and points out cases of Throwable and abort over-catch. The tool also makes sure that the most
specific exceptions are caught first if there are multiple catch blocks for a single try statement.

A typical workflow might start by running a summary check on a large directory:

Which outputs a summary listing filenames organized by the type of warning detected in them:

As well as a summary of the statistics at the bottom of the output:
3.2 Features

Then it might make sense to pick a few files to look at more closely:

And view the verbose (in blue and yellow) and debug (in white) output for those files and try-catch blocks:

```
>>> try statement found at line 186 column 9
  no catch blocks found for try statement at line 186
>>> try statement found at line 211 column 9
  >>> catch block for Throwable
    block begins at line 215 column 9
    potential logging only in catch at line 215
    catch (Throwable e)
    {
      logger.error(String.format("Failed to release index %s", index.getName()), e);
    }
    Throwable overcatch at line 215
    block ends at line 218 column 9
Checking source code for .../cassandra/src/java/org/apache/cassandra/index/sasl/conf/IndexMode.java
>>> try statement found at line 70 column 9
  >>> catch block for InstantiationException
  >>> catch block for IllegalAccessException
    block begins at line 88 column 9
    potential logging only in catch at line 88
    catch (InstantiationException | IllegalAccessException e)
    {
      logger.error("Failed to create new instance of analyzer with class [{}", analyzerClass.getName(), e);
    }
    block ends at line 91 column 9
>>> try statement found at line 101 column 13
  >>> catch block for ClassNotFoundException
  >>> catch block for ClassCastException
    block begins at line 105 column 13
    catch (ClassNotFoundException e)
    {
      throw newConfigurationException(String.format("Invalid analyzer class option specified [%s]", indexOptions.getIndexANTALYZER_CLASS_OPTION));
    }
    block ends at line 109 column 13
>>> try statement found at line 125 column 9
```
One of the goals for this project was to make the tool relatively user friendly by displaying organized output. Unlike Aspirator, which outputs results to .txt files, using a Python tool like this allows for very quick alterations of and access to the output.

4 Discussion

The completed ExceptCheck tool was tested on a handful of simple testcases in `exceptCheck/textCases/`, as well as the open source versions of Cassandra, HBase, Cloudstack, and Hive. As shown above, the tool found 2 TODO warnings, 29 empty catch warnings, 12 logging only warnings, and 24 abort warnings for Cassandra. The rest of the numbers were (23, 454, 36, 104) for HBase, (106, 91, 18, 50) for Cloudstack, and (52, 521, 25, 34) for Hive. Cloudstack appears to have an abnormally high rate of TODO warnings, but on the whole, the proportion of trivial (potential) mistakes seems to reflect the percentages reported in the paper, which is a good sign since ExceptCheck’s features are based largely off of Aspirator’s. Additionally, reporting cases of caught Throwable seems to lead to too many false positives for the feature to be useful.

4.1 Comparisons

As expected, there appear to be a lot of false positives for abort warnings in ExceptCheck, due to the difference between how the warnings are reported between bytecode and source code analysis. In Aspirator, the paper authors were able to identify abort overcatch based on whether the handler aborted on a superclass of the thrown Exception, whereas ExceptCheck simply identifies all cases of exiting. Similarly, the number of empty catch warnings here may be an overestimate as well, since with bytecode analysis, a tool like Aspirator can trace actual registers and look at which exact Exceptions can be thrown from certain messages. Though some measures are taken in ExceptCheck to reduce false positives, some are simply not feasible without knowledge gained from bytecode analysis.

On the other hand however, source code analysis gives more intuitive line-by-line reporting and debugging. The reporting of TODO and FIXME statements are also extremely accurate, and tend to report very likely sources of bugs that bytecode analysis may miss, where the comments are not directly taken into account. Source code analysis also creates more opportunity to focus on common code patterns.

For ExceptCheck in particular, implementing the check for whether an assigned variable in the try block was checked later in an immediately following if statement
was very interesting. If found, an empty catch block in between would be ignored. This required looking through the AST to at sibling nodes to the `TryStatement` in order to find an adjacent `IfStatement`, and then recursing through expressions to determine whether the variables in question were checked.

### 4.2 Extensions

It would be ideal to combine the pros of both source code analysis and bytecode analysis into a single tool. With more time, it would be interesting to see how using something like the open-source library python-javatools (https://github.com/obriencj/python-javatools) library in ExceptCheck on .class or .jar files could improve the tool as it stands now.

Additionally, ExceptCheck works well as a basic source code checker for exception handling errors, as well as a meta-tool for further finding common bug patterns in exception handling code. With more statistics, it may be possible to identify even more patterns to then tackle code-generation.