Detecting Inconsistent Thrown Exceptions

Lin Xu
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
lin-xu@sjtu.edu.cn

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
zhonghao@sjtu.edu.cn

***Abstract*—Exception-handling is a critical mechanism in many programming languages. Although it is beneficial in many programming contexts, the bugs in this mechanism can lead to disastrous consequences. In the literature, researchers have proposed various approaches to detect the bugs in handling exceptions. Meanwhile, the thrown exceptions themselves can also be buggy, but to the best of our knowledge, no prior approach has ever been proposed to detect bugs in thrown exceptions, because it is even difficult for programmers to determine whether a thrown exception is buggy or not. In this paper, we propose the first automatic criterion (meta-oracle) to determine whether a thrown exception contains a bug. Our meta-oracle says that if the messages of two thrown exceptions are quite similar, the type of the two exceptions shall be the same. Based on this meta-oracle, we implement EXMINER that is able to detect bugs in thrown exceptions. It introduces sequence mining to identify similar exception messages, and groups thrown exceptions by the patterns that are mined from their messages. From each group of exception messages, it detects the ones whose types are different as bugs in thrown exceptions. For the first time, EXMINER detected three bugs from the latest versions of three Apache projects. We reported this new type of bugs to their developers, and all the bugs were confirmed and fixed.**

I. INTRODUCTION

Application Programming Interface (API) libraries have been an essential ingredient of software development [23]. When calling Application Programming Interface (API) libraries incorrectly, programmers can introduce API-related bugs [27], [29], and such bugs can cause exceptions. Exception handling is a critical mechanism in many languages [21]. With its support, if the inputs and the call sequences of APIs are incorrect, libraries can throw exceptions with *error messages* to actively warn programmers. Indeed, another recent empirical study [28] shows that about 80% of parameter rules are solely encoded in error messages of thrown exceptions. Although the exception-handling mechanism is critical, the mechanism can also introduce bugs. One type of such bugs occurs when thrown exceptions are incorrectly handled. For example, an exception can bypass the release method of a resource, and cause resource leaks. To detect such resource leaks, the prior approaches [11], [16] mine the acquisitions and releases of resources, and check whether acquired resources are released in `catch` clauses.

Besides the bugs in handling exceptions, a thrown exception itself can be also buggy. Although it is desirable to detect such bugs, to the best of our knowledge, no prior approach has ever been proposed to detect bugs in thrown exceptions. It is challenging to detect such bugs in thrown exceptions, because it is difficult to obtain the knowledge of throwing correct exceptions. The knowledge shall be correct, effective, and useful. It is difficult to obtain such knowledge for throwing exceptions, because exceptions are many and programmers can use them in different ways. For example, in Java, J2SE alone provides hundreds of exceptions, and it allows implementing more customized exceptions. Many exceptions do not have high-quality documents, and their definitions of exceptions can be overlapped. When programmers do not fully understand exceptions, they can throw wrong exceptions, and such exceptions lead to bugs in their written code. As shown Section II, such bugs can be critical.

To improve the state of the art, in this paper, we propose the first meta-oracle that is able to determine bugs in thrown exceptions. Our meta-oracle says that *if two thrown exceptions have similar error messages, they shall be of the same type*. Based on our meta-oracle, we propose the first approach called EXMINER that detects bugs in thrown exceptions. As error messages are written in natural languages, it is difficult to identify similar ones. To handle the problem, we introduce frequent item mining [24] to mine patterns from sentences, and invent two criteria to select useful ones. We used EXMINER to check the latest code of four popular projects. As shown in Section V, our tool found three bugs, and all the bugs were fixed by their developers.

II. MOTIVATING EXAMPLE

In this section, we use three real bugs to illustrate the importance of our wrong exceptions and the technical choice of EXMINER. TinkerPop [3] is a graph database. As a core functionality of storing graphs, its `serializeResponseAsString` method serialises objects to strings. A programmer called Jason Plurad reported a *critical bug* [2], and this bug report complains that this method throws a wrong exception. Figure 1 shows the patch of this bug report. The signature of this method declares that it can throw `SerializationException`. In the buggy code, it throws the `RuntimeException`, but in the fixed code, the thrown exception is modified. When a wrong exception is thrown, the problem of this exception may not be correctly handled. As a result, this problem can cause serious bugs. For example, in another database called Cassandra, its disk failure policy is not activated, because the buggy code throws a wrong exception when a failure occurs [7].

```
1 public String serializeRequestAsString(...)
      throws SerializationException { ...
2  } catch (Exception ex) {...
3 -  throw new RuntimeException("Error during
       serialization.", ex);
4 +  throw new SerializationException(ex);
5  }
6 }
```

Fig. 1: The patch of TINKERPOP-682

We notice that many exceptions are thrown with messages. For example, in Figure 1, the message in Line 4 indicates that the thrown exception is related to serialization. Furthermore, the messages of thrown exceptions shall be meaningful, because they are useful to debug bugs. We notice that programmers take effort to maintain such messages. For example, a bug report [1] complains that when an exception is thrown, its message asks users to set a wrong parameter. Based on the above two observations, we propose EXMINER that detect wrong thrown exceptions by comparing their messages. We next introduce our meta-oracle and technical details.

III. THE CRITERIA OF META-ORACLE

The oracle problem is well-known in software testing [12]. In this paper, we define a test oracle as follows:

Definition 1:

A test oracle is the knowledge to determine whether the execution result of a test case is correct or not.

Furthermore, we define a meta-test-oracle as follows:

Definition 2:

A meta-test-oracle is the knowledge to determine whether a type of test cases return correct results or not, and in static tools, a meta-oracle is the knowledge to determine whether a type of source files is correct or not.

A meta-oracle shall be correct, effective, and useful:

1. The correctness. This criterion requires that the knowledge itself shall be correct. If not, its downstream tools will produce many false alarms by their internal flaws.

2. The effectiveness This criterion requires that a meta-oracle shall be effective to detect bugs. A correct meta-oracle may not be an effective one, because it may be infeasible to find many violations of a correct meta-oracle. For example, source file shall have no compilation errors. Although it is a correct meta-oracle, it is ineffective to detect real bugs.

3. The usefulness This criterion requires that a meta-oracle is able to detect important bugs. A meta-oracle is considered to be less useful if its violations are not considered as bugs.

A correct, effective, and useful meta-oracle is difficult to be obtained, but it can motivate many research tools (*e.g.*, learning from repositories [20]). However, due to the complexity of the real world, it is infeasible to define a perfect meta-oracle. For example, although differential testing has been used to detect bugs [14], [15], [26], one of its limitations lies in undefined behaviors [17]. Even much simpler meta-oracles have limitations. For example, a meta-oracle says that a program shall not have infinite loops [13], but loops are designed to be infinite to host services. Our meta-oracle is that ***thrown exceptions shall be of the same type, if their messages are similar***.

IV. APPROACH

Figure 2 shows the overview of EXMINER. It takes the source files of a Java project as its input, and from the input, it extracts all the thrown exceptions. From the messages of thrown exceptions, it mines frequent word sequences. It then classifies thrown exceptions by their frequent word sequences, and detects abnormal exceptions.

A. Extracting Thrown Exceptions

From the source files of a library, EXMINER first extracts the types of the error messages of its thrown exceptions. We build EXMINER upon JDT [4], the Eclipse's built-in Java compiler. From each `throw` statement, EXMINER extracts a pair $\langle t, m\rangle$, where t denotes the type of a thrown exception and m denotes its message. JDT parses each source file into an abstract syntax tree (AST) and supports customized visitors. A customized visitor can capture different types of AST nodes. EXMINER implements a customized visitor to extract all the `throw` statements. To extract t and m from a `throw` statement, EXMINER analyzes three cases:

1. Instance creations. In most cases, `throw` keywords are followed by the creations of exceptions. Figure 3a shows such an example. For these statements, EXMINER extracts the type of the exception class as t. The creation of an exception is the method call of a constructor. EXMINER extracts m from all the arguments of the constructor. If an argument is a string constant, it adds the constant to m. For example, from Figure 3a, it extracts the pair, $\langle IllegalArgumentException$, $Property\ asked\ is\ not\ a\ Text\ Property\rangle$. The arguments can be the combination of texts and variable names. For example, a thrown exception is as follows:

```
throw new IllegalArgumentException("cannot change
      index sort from " + segmentIndexSort + " to
      " + indexSort);
```

The extracted message is as follows:

cannot change index sort from segmentIndexSort to indexSort

A constructor can have more than one argument, and an argument can be a variable. For example, a thrown exception is as follows:

```
} catch (MaxBytesLengthExceededException e) {
  String msg = "Document contains at least one
       ...";
  throw new IllegalArgumentException(msg, e);
}
```

If an argument is a variable, EXMINER checks the assignments to this variable. If a constant value is assigned to the variable, it adds the value to the message of the exception. If an exception has more than one argument, it adds their constant values one by one to the message of the exception. As EXMINER relies on static analysis to extract messages, it ignores variables whose values are determined at runtime. In

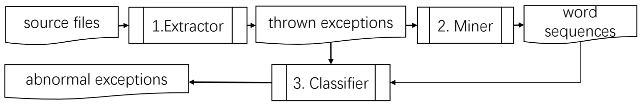

Fig. 2: The overview

the above example, the second argument is a thrown exception, and many methods can throw the exception with different messages. As its value is determined at runtime, EXMINER ignores this input, and the extracted message is as follows:

Document contains at least one ...

An argument can be string constants that are defined in other classes, and it is difficult to determine their values. To extract more meaningful error messages, EXMINER builds a table for all defined string constants, and replaces them if they appear in error messages.

2. Variables. If a `throw` keyword is followed by a variable, EXMINER resolves the type of the variable as t, and checks the initializer of the variable. If a constant value is assigned to the variable, it extracts the value as m.

3. Method invocations. If a `throw` key word is followed by a method invocation, EXMINER extracts the return type of the method as t. The extraction of error messages is identical to the process of instance creations.

B. Mining Frequent Word Sequences

The second step is to mine the commonality among thrown exceptions. EXMINER achieves this research goal by mining frequent word sequences from exception messages. A sentence (S) can be considered as a sequence of words ($w_1 \dots w_n$). In natural language processing, a word is a token, and the task of identifying the words from a sentence is known as tokenization [25]. We use the tokenizer of lingpipe [5], and it is a rule-based tokenizer. EXMINER considers an exception message as a sequence, and extracts its words as items.

After a sentence is split into words, we use frequent item mining [24] to mine frequent words. In frequent item mining, a subsequence is defined as follows:

Definition 3: Given two sequences $S_a = a_1 \dots a_n$ and $S_b = b_1 \dots b_m$, if there exists integers $1 \leq i_1 < i_2 \dots < i_n \leq m$ such that $a_1 = b_{i1}, \dots, a_n = b_{in}$, S_a is a subsequence of S_b, denoted as $S_a \subseteq S_b$. If $S_a \subseteq S_b$ and S_a is shorter than S_b, S_a is a proper subsequence of S_b, denoted as $S_a \subset S_b$.

The above definition needs to determine whether two items are equivalent. The first word of a sentence is capitalized. As sentences in error messages can be incomplete, after their words are extracted, their capitalized letters can be different. When mentioning code names, some sentences use upper cases and some use lower cases. To handle the problem, when EXMINER compares two words, it ignores their cases. The plural forms of most nouns are different from their single forms, and verbs have different tenses. To handle the problem, EXMINER uses a stemmer [22] to transfer all words to their roots. Given a set of sequences, the frequency of a sequence is calculated as follows:

Definition 4: Given a set of sequences $SD = S_1 \dots S_n$, the frequency of S_a is the number of $S_i \in SD$ and $S_a \subseteq S_i$.

The frequency of a sequence is also known as its support, denoted as $sup^{SD}(S_a)$. Given a support threshold min_sup, the problem of mining frequent sequences from SD is to locate all sequences Si such that $sup^{SD}(S_i) \geq min\_sup$.

Definition 5: If S_a is frequent and $\nexists S_b$ such that $S_a \subset S_b$, S_a is a frequent closed sequence.

As defined in Definition 3, a subsequence of a sentence can be discontinuous. Although the definition is proper in many applications, a discontinuous subsequence of a sentence can be difficult to understand and sometimes even misleading. A subsequence can consist of an adjective from the subject and a noun from the object. Such subsequences are less informative. To handle the problem, we revise sparesort to mine true subsequences:

Definition 6: Given two sequences $S_a = a_1 \dots a_n$ and $S_b = b_1 \dots b_m$, if there exists integers $1 \leq i, i+1 \dots, i+n \leq m$ such that $a_1 = b_i, \dots, a_n = b_{i+n}$, S_a is a true subsequence of S_b, denoted as $S_a \preceq S_b$. If $S_a \preceq S_b$ and S_a is shorter than S_b, S_a is a proper true subsequence of S_b, denoted as $S_a \prec S_b$.

For example, "property not" is a proper subsequence, but is not a proper true subsequence of the exception message in Figure 3. For simplicity, we call a mined frequent proper true sequence as a pattern. EXMINER extends sparesort [18], and it mines patterns from exception messages. We select sparesort, because this library is written in Java and open source.

C. Identifying Abnormal Exceptions

In natural languages, the meaning of a word is limited. To express richer meanings, word phases and sentences are constructed. As frequent item mining does not consider the meanings of its mined patterns, its mined patterns can be too short to convey meanings, because shorter patterns typically have higher frequencies than longer ones. The problem cannot be resolved by simply requiring longer patterns. Even if a pattern is long, it can contain no informative words. For example, we find that a pattern is "is the combination of a". Although this pattern has five words, it is not quite meaningful. Indeed, if different subjects and objects are added to this pattern, it can convey quite different messages.

We notice that meaningful patterns contain more nouns, verbs, adjectives, and adverbs. Based on this observation, we use the number of the above words as a criterion to remove less meaningful patterns. In natural language processing, the task of identifying the part of speech tags is known as POS tagging, and the state-of-the-art taggers can be roughly divided into supervised taggers and unsupervised taggers [19]. Supervised taggers learn statistical models from corpora with labels, and unsupervised taggers are often based on predefined rules. EXMINER uses lingpipe [5] to identify the part of speech tags. The POS tagger of lingpipe is an unsupervised tagger, and its statistical model is Hidden Markov Model. We select lingpipe as our underlying POS tagger, because it is trained on a large corpus, *i.e.*, the Brown Corpus [8]. This corpus contains a million words.

As our model is Hidden Markov Model and it is trained on complete sentences, lingpipe is more effective to tag complete

```
public getUnqualifiedTextProperty(String name){
 AbstractField prop = getAbstractProperty(name)
     ;...
 if (prop instanceof TextType){
   return (TextType) prop;
 }else{
   throw new IllegalArgumentException("Property
       asked is not a Text Property");
 }...}
```

(a) A method throws `IllegalArgumentException`

```
public getUnqualifiedArrayList(String name){...
  for (AbstractField child : getAllProperties()){
    if (child.getPropertyName().equals(name)){
      if (child instanceof ArrayProperty){
        array = (ArrayProperty) child; break;
      }
      throw new BadFieldValueException("Property
          asked is not an array");
    }...
}}
```

(b) The other method throws `BadFieldValueException`

Fig. 3: Our found bug

sentences than patterns. To extract better tagging results, instead of patterns, ExMiner extracts the tagging results from complete error messages, and maps them to mined patterns.

For each frequent word sequence S_a, ExMiner searches for a subset of thrown exceptions $\langle e_1, m_1\rangle, \dots, \langle e_n, m_n\rangle$ such that $S_a \subseteq m_i$. As the sentences of exception messages are already tokenized, it is straightforward to compare them to determine whether S_a is a subsequence of m_i. After that, ExMiner compares all the e_i. A mined frequent word sequence indicates the same type of problems, and its located exceptions are thrown because of the same problem. In a project, programmers shall throw the same exception for the problem. If they do not, it is confusing, and it can lose the same type of problems when programmers try to catch this exception. Due to the above considerations, if ExMiner finds more than one thrown exception in the located subset, it determines that an abnormal exception is detected. After that, programmers can inspect such abnormal exceptions for bugs.

V. Early Results

We used ExMiner to check the three Apache projects such as `common_io`, `pdfbox`, and `shiro`. ExMiner detected three bugs. After we reported them, they are all confirmed and fixed. Figure 3 shows an example of our found bugs. We find this bug in the latest version of `pdfbox` [6]. Figure 3 shows two methods of `pdfbox`, and each method has a parameter. Given an illegal input, the method in Figure 3a throws `IllegalArgumentException`, but the method in Figure 3b throws `BadFieldValueException`. We determine that this shall be a bug, because it is strange that the same error triggers two different exceptions. After we reported the problem, it was fixed by the developers of `pdfbox` [10].

It is challenging to detect this type of bugs, because the definitions of exceptions are vague and can have overlaps. In this example, according to their documents [9], `IllegalArgumentException` indicates that an illegal or inappropriate argument is passed to a method. As the exceptions in Figure 3 are triggered by illegal inputs, it does not violate its document to throw `IllegalArgumentException`. However, from the perspective of client programmers, if the same type of errors throw different exceptions, their catch statement can fail to catch exceptions. The developers of `pdfbox` also agree that it is important to throw the identical exception, if the error is identical. We summarize this criterion as our meta-oracle to detect bugs in thrown exceptions.

However, it is difficult to determine whether two exceptions are thrown for an identical problem. In this example, the two checked variables have different names (`prop` vs `child`), and their values are checked against different types (`TextType` vs `ArrayProperty`). One exception is thrown from an `else` statement, but the other exception is thrown inside an `if` statement. Even their thrown messages are different. ExMiner resolves the problem by mining the thrown messages. In this example, ExMiner mines that the phrase, *property asked is not*, often appear in thrown messages, but their thrown exceptions are different. In this way, it detects the abnormal thrown exception, and we determine that it indicates a bug. This bug is already fixed, after we report it.

VI. Work Plan

As our effectiveness and usefulness have not been analyzed in a quantitative manner yet. In future work, we plan to extend this work from three perspectives:

1. Building a benchmark of our target bugs. We plan to build a benchmark where true labels of bugs are provided. It can be feasible to identify our target bugs from past bug fixes and bug reports. With such a benchmark, we can calculate the precision and recall of ExMiner.

2. Detecting bugs of the latest versions. Another way is to detect bugs in the latest versions, but we cannot calculate recalls in this way, because it is infeasible to obtain all the bugs of a real project. In addition, from a decent project, it is unlikely for a tool to detect many bugs and some programmers are reluctant to check too many reported bugs. We are still working towards a better way to solve this problem.

3. Improving our approach. In some projects, exception messages are built at runtime, and cannot be extracted throw static analysis. For example, a project can store all exception messages in a property file, and load them at runtime. We plan to introduce dynamic analysis to collect such messages. In addition, we notice that if a message is associate with more than one exception, its minority group can be correct, instead of the majority group. We plan to explore more informative oracles to determine which one shall be correct.

Acknowledgement

We appreciate the anonymous reviewers for their insightful comments. Hao Zhong is the corresponding author. This work is sponsored by the National Key R&D Program of China No. 2018YFC083050.

References

[1] FSNamesystem.setTimes throws exception with wrong configuration name in the message. https://issues.apache.org/jira/browse/HDFS-2790, 2012.

[2] JsonMessageSerializerV1d0 throws RuntimeException instead of SerializationException. https://issues.apache.org/jira/browse/TINKERPOP-682, 2015.

[3] APACHE TinkerPop. https://tinkerpop.apache.org/, 2020.

[4] Eclipse Java development tools. http://www.eclipse.org/jdt/, 2020.

[5] LingPipe: a tool kit for processing text. http://alias-i.com/lingpipe, 2020.

[6] Pdfbox: A java pdf library. https://pdfbox.apache.org, 2020.

[7] Running oos should trigger the disk failure policy. https://issues.apache.org/jira/browse/CASSANDRA-11448, 2020.

[8] The Brown Corpus. http://www.helsinki.fi/varieng/CoRD/corpora/BROWN, 2020.

[9] The J2SE API documents. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/package-summary.html, 2020.

[10] XMPSchema#getUnqualifiedArrayList throws a different exception. https://issues.apache.org/jira/browse/PDFBOX-4803, 2020.

[11] M. Acharya and T. Xie. Mining API error-handling specifications from source code. In *Proc. FASE*, pages 370–384, 2009.

[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering*, 41(5):507–525, 2014.

[13] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proc. ASE*, pages 161–169, 2009.

[14] S. R. Choudhary, H. Versee, and A. Orso. WEBDIFF: Automated identification of cross-browser issues in web applications. In *Proc. ICSM*, pages 1–10, 2010.

[15] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. ESEC/FSE*, pages 185–194, 2007.

[16] S. Jana, Y. J. Kang, S. Roth, and B. Ray. Automatically detecting error handling bugs using error specifications. In *Proc. USENIX*, pages 345–362, 2016.

[17] T. Kapus and C. Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proc. ASE*, pages 590–600, 2017.

[18] H. Kazato, S. Hayashi, T. Oshima, S. Miyata, T. Hoshino, and M. Saeki. Extracting and visualizing implementation structure of features. In *Proc. APSEC*, pages 476–484, 2013.

[19] D. Kumawat and V. Jain. Pos tagging approaches: A comparison. *International Journal of Computer Applications*, 118(6), 2015.

[20] H. Mei and L. Zhang. Can big data bring a breakthrough for software automation? *Science China Information Sciences*, 61(5):1–3, 2018.

[21] T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in java. In *Proc. OOPSLA*, pages 83–95, 2001.

[22] M. F. Porter et al. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[23] S. Raemaekers, A. Van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Proc. ICSM*, pages 378–387, 2012.

[24] J. Wang and J. Han. BIDE: efficient mining of frequent closed sequences. *ICDE*, pages 79–90.

[25] J. J. Webster and C. Kit. Tokenization as the initial phase in nlp. In *Proc. COLING*, pages 1106–1110, 1992.

[26] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. PLDI*, pages 283–294, 2011.

[27] H. Zhong and H. Mei. Learning a graph-based classifier for fault localization. *Science China Information Sciences*, 63:1–22, 2020.

[28] H. Zhong, N. Meng, Z. Li, and L. Jia. An empirical study on api parameter rules. In *Proc. ICSE*, page to appear, 2020.

[29] H. Zhong, X. Wang, and H. Mei. Inferring bug signatures to detect real bugs. *IEEE Transactions on Software Engineering*, 2020.