Understanding the Non-Repairability Factors of Automated Program Repair Techniques

Bo Lin†, Shangwen Wang†*, Ming Wen⋆, Zhang Zhang†, Hongjun Wu†, Yihao Qin†, Xiaoguang Mao†
†*National University of Defense Technology, Changsha, China*
⋆*Huazhong University of Science and Technology, Wuhan, China*
{linbo19, wangshangwen13, zhangzhang14, wuhongjun15, qinyihao15, xgmao}@nudt.edu.cn, mwenaa@hust.edu.cn

***Abstract*—Automated Program Repair (APR) is becoming a hot topic in Software Engineering community with many approaches being proposed and experiments being performed over the years. The results obtained from different experiments can be used as practical guidance to advance APR techniques. However, researchers have generally ignored the biases with respect to the unexpected results generated by various APR techniques, in which case the repair process cannot be finished normally and is terminated with unexpected exceptions (referred to as the *non-repairability factors*). In this paper, we aim to thoroughly understand the reasons for such non-repairability factors of various APR techniques, thus to provide practical insights for diverse stakeholders to establish an unbiased evaluation of APR techniques. To achieve so, we performed a systematic study on the existing execution logs that are ended with unexpected exceptions collected from different APR studies. Specifically, we investigated different types of exceptions with their frequencies of occurrence, the behind reasons of such occurrences, as well as the impact of such exceptions on the repairability of APR techniques. Our experimental results reveal that: 1) non-repairability factors happen in 25.7% of our studied logs and are widespread among diverse combinations of APR tools with FL strategies; 2) *Inherent defect of APR tools* is the most common reason for the occurrence of the non-repairability factors; 3) the impact of the non-repairability factors on the performance of APR tools can be rather significant. Our empirical study indicates that it is of great importance to eliminate the biases from the non-repairability factors. We also highlight several implications for actions that we can take to eliminate such biases.**

***Index Terms*—Program Repair, Logs, Exceptions**

I. Introduction

Automated Program Repair (APR) [1], which aims to generate patches for software defects automatically, has attracted widespread attentions in recent years due to its huge potential in reducing debugging costs. The technique mainly contains three steps, which include Fault Localization (i.e., to generate a ranked list of suspicious elements in the program), Patch Generation (i.e., to drive the generation of a patched version of the buggy program which meets the desired behavior specified by the test suite through various kinds of methods such as heuristics [2] and constraints solving [3]), and Patch Validation (i.e., to assess whether the generated patches are correct). Over the years, the state-of-the-art in this field has been advanced dramatically with numerous test-suite-based techniques being proposed and evaluated in well-established benchmarks [4].

Each time a new technique is proposed, its authors made a comprehensive comparison against the studies in the literature to demonstrate its advances [5], [6]. Besides, there are also plenty of large-scale empirical studies concerning certain under-valued aspects such as patch correctness [7], benchmark overfitting [8], and efficiency [9]. The results of these comparisons play a significant role since it can guide the future direction in this field, e.g., if a study reveals certain factors that contribute most to the effectiveness of existing APR techniques, future studies may focus on advancing the parts with respect to such factors.

Generally, researchers only concern on how many bugs a tool can repair, while often ignore those failures of the fixing attempts of other bugs. In other words, few attentions has been paid to the factors that cause the repair process to end with unexpected results, in which case the fixing attempt is terminated abnormally and no results are generated (referred to as the *non-repairability factors* in this study). Recently, Liu et al. revealed that two tools (i.e., Nopol and DynaMoth) are unable to fix some bugs due to the configuration problems (cf. Table 4 in [9]), which also indicates that APR techniques may generate no patch for a bug for other reasons rather than the capability of the tool itself. Such issues indeed bring great threats to the fair comparisons among various APR techniques. For example, if a tool fails to fix a bug in an evaluation due to configuration problems while it can actually fix it when all the environments have been appropriately handled, it may achieve a low performance in the comparison and thus to be underestimated. Therefore, it is of great emergency to investigate what factors may cause the non-repairability factors and furthermore, how to avoid such factors to eliminate potential evaluation bias in the future studies.

To fill this gap, we performed a large-scale study to comprehensively understand the factors contributed to the non-repairability factors. Particularly, we investigated the exceptions in the execution logs generated by APR tools targeting the Java programs. We focus on Java repair systems since they are dominant in the APR field in recent years [10], [11]. We choose to investigate the execution logs of APR tools since the execution results, including those ended with unexpected results, can be directly reflected by the generated logs. Specifically, we analyzed all the execution logs generated

* Corresponding author.

This work is supported by the National Natural Science Foundation of China No.61672529, No.62002125, as well as the Fundamental Research Funds for the Central Universities (HUST) No.2020kfyXJJS076.

by existing studies that are publicly available, which include in total **11,818** logs. We summarized different types of exceptions occurred and deeply analyzed the causes behind them. Specifically, we classified them into 5 categories. Eventually, we revisited the performance of existing APR techniques and compared them without bias after removing all the observed non-repairability factors. Overall, our primary findings are:

F1: So far, execution logs generated by APR techniques have not been widely studied. Among numerous studies on APR, only five have released all their logs.

F2: Exceptions happen in a considerable number of fixing attempts (25.7% of our studied subjects), among which *IllegalState* is the most common one.

F3: The causes of such exceptions are 5-folds while the *inherent defect of APR tools* is the dominating one, accounting for nearly 60% of the total amount.

F4: The impact of such exceptions is widespread and significant on some certain APR tools under specific Fault Localization strategies.

Moreover, based on our in-depth analysis of the behind reasons of the existing exceptions in the logs, we distilled several implications for avoiding the influence, including being careful with the experimental setup of replication studies and evolving third party libraries continuously.

II. Background and Motivation

In this section, we first present background information about our study and then provide the motivation of our study that is other reasons rather than fixing abilities may bring threats to the validity of APR studies.

A. Background

Automated Program Repair (APR) has become a hot topic in Software Engineering community in recent years [1]. Given a buggy program and its corresponding test suite, APR first utilizes a Fault Localization (FL) technique which is usually GZoltar [12] to acquire a list of suspicious statements. After that, it tries to generate patches at these suspicious statements using either search-based technique [2], [5], [6], [11] (i.e., search within a predefined space) or synthesis-based technique [13] (i.e., use constraints solving strategy to synthesize a patch). At the end, APR uses specifications (test suites, under most conditions) to check the correctness of generated patches. If a generated patch passes all the test cases, it will be outputted as a *plausible patch*. Originally, APR techniques are designed for C language [2], [3]. However, recent years have witnessed the explosive growth of the number of Java techniques [6], [13]–[17]. As a result, this paper focuses on APR techniques for Java.

Over the years, researchers have created some defects benchmarks to evaluate the performances of APR techniques. **Defects4j** [4] contains 395 bugs from 6 open-source Java projects, namely *Chart*, *Closure*, *Lang*, *Math*, *Mockito*, and *Time*. With the help of bug tracking system, bugs are extracted by the identification of bug fixing commits and each buggy program possesses a fixed version which demonstrates how developers handle the bug (known as *ground-truth*). Upon created, this benchmark has been widely-used in APR studies [18]. Thus, in our study, we focus on this benchmark.

In this paper, to ease our presentation, we use **fixing attempt** to denote the execution of APR tool on a certain bug. Besides, we use **valid fixing attempt** to denote fixing attempt which ends with expected results. Normally, the result of a fixing attempt is either *Success* or *Fail*. The former represents that APR tools successfully find a patch that can pass all the test cases, while the latter indicates that APR tools fail to generate a test-suite-adequate patch in a predefined limitation (e.g., time budget) or traverse the whole search space but find nothing, both of which show the disability of the APR tools for fixing the bug (referred to as **repairability factors**). Note that there have already been several studies [8], [19] analyzing the repairability factors of APR tools. For instance, the authors of [19] attributed some failures of fixing bugs to *insufficient fix patterns*. However, there exists situations where the fixing attempt is terminated unexpectedly (referred to as **non-repairability factors**). To the best of our knowledge, we are the first to systematically investigate non-repairability factors.

B. Motivation

We observed that two recent studies (i.e., Liu et al. [9] and Durieux et al. [8]) both reran *Cardumen* [20] on the Defects4J benchmark. However, they reported different numbers of bugs which could be fixed by this tool (12 vs. 17). What surprises us is that the number is fewer when utilizing the latest fault localization framework (i.e., GZoltar-v1.7 in [9]), since previous study has demonstrated that better fault localization can enhance the repair performance [21]. We further checked the fixed bug lists of these two studies and searched for the differences. We observed that bug *Chart-1* can be fixed under GZoltar-v0.1 (in [8]) while cannot under the same tool with version 1.7. We then referred to the log of *Chart-1* in the study [9] and found that there exists an exception as shown in Listing 1 which causes the unexpected exception of this fixing attempt.

```
Exception in thread "main" java.lang.
      UnsupportedOperationException:
      PartialSourcePosition only contains a
      CompilationUnit at spoon.reflect.cu.position.
      NoSourcePosition.getLine(NoSourcePosition.java:49)
at fr.inria.astor.core.solutionsearch.population.
      ProgramVariantFactory.createModificationPoints(
      ProgramVariantFactory.java:257)
at fr.inria.astor.core.solutionsearch.population.
      ProgramVariantFactory.createProgramInstance(
      ProgramVariantFactory.java:121)
at .....
```

Listing 1: Exception from Cardumen on Chart-1 in study [9]

After carefully checking the stack trace, we found that *Cardumen* throws this exception when preparing to search plausible patches in the search space. When utilizing *GZoltar-v1.7*, the returned results are the statements whose suspicious values are larger than 0. While in *GZoltar-v0.1*, the suspicious values of the output statements should be larger than 0.1. This difference determines that the selected statements by *GZoltar-v1.7* are much more than its earlier version. *Cardumen* throws this exception when encountering a specific type of statement as we will analyze in Section IV-B.

This example is a vivid case demonstrating that the non-repairability factors may cause exceptions during execution and thus lead to the failure of fixing attempts.

III. Study Design

This section presents the design details of this study.

A. Research Questions

Overall, our investigation to the non-repairability factors of fixing attempts of different APR techniques aims to answer for the following research questions (RQs):

1) **RQ1. Number of occurrence of each type of exception**: We first investigated the question: *what is the number of occurrence of each type of exception?* To answer this question, we searched for the occurrences of all exceptions recorded in the logs generated by existing APR tools and classify them based on the thrown message.
2) **RQ2. Reasons behind the unexpected exceptions**: Upon finding out all the exceptions in existing experiments, we investigated the behind reasons for these unexpected results: *why these unexpected exceptions happen?* To seek the answer, we analyzed the in-depth reasons behind each kind of exception and further categorized them into different reasons. Answering such a question can provide practical guidance to avoid such exceptions in future studies.
3) **RQ3. Impact of unexpected exceptions**: Finally, given that a certain number of exceptions being generated in existing studies, we aim to investigate their impacts on the experimental results. Specifically, we question *what is the performance of APR tool when discarding those executions which are affected by non-repairability factors?* We re-calculated the repairability (i.e., effectiveness) of APR systems by only concerning those executions ended with normal results and make comparisons with previously reported results. By doing so, we can gain a clear insight on the impact of non-repairability factors.

B. Subject Selection

To investigate our research questions, we need to mine the execution logs in APR studies where the exception information is recorded. Note that there are mainly two kinds of papers in APR field: to introduce new APR techniques [5], [6], [13] or to empirically assess the performance of existing APR techniques [8], [9], [18]. In this study, to represent each scientific literature precisely, we use the tool name if the paper proposes an APR technique or the family name of the first author followed by the publication year if it is an empirical study.

In the end, APR studies considered for our study need to satisfy the criterion that *logs of all fixing attempts (not only those which generate plausible patches) should be released*. We consider the living review of APR by Monperrus [1] as our sources of information to identify Java APR studies. Table I enumerates all the Java-based APR studies and also provides the arguments for its inclusion/exclusion in this study.

In total, we included five studies: *SimFix* [6] is an APR technique that utilizes code change operations from existing

TABLE I: Included and excluded APR studies for our study

Selected	Reason	APR Studies
No	Not public	PAR [22], xPAR [14], JFix/S3 [23], ELIXIR [10], Hercules [24], SOFix [25], SketchFix [26], PraPR [27].
No	Do not release logs	HDRepair [14], ACS [17], kPAR [21], AVATAR [28], TBar [19], FixMiner [29], ssFix [30], CapGen [5], ARJA [11], GenProg-A [11], RSRepair-A [11], Kali-A [11], NPEFix [31], Nopol [13], DynaMoth [16], DeepRepair [15], LSRepair [32], jGenProg [33], jKali [33], jMutRepair [33], Cardumen [20], SequenceR [34], DLFix [35], CoCoNut [36].
No	Only release part of the logs	JAID [37].
Yes	Open-source & contain intended logs	SimFix [6], Wang-2019 [18], Liu-2020 [9], Durieux-2019 [8], Martinez-2016 [7]

TABLE II: Detailed information of included logs in this study

APR Study	#Tool	#Bug	#FL Strategy	#Logs
SimFix	1	357	1	357
Wang-2019	2	38	1	76
Liu-2020	10*	395	2⋆	6,368†
Durieux-2019	11	395	1	4,345
Martinez-2016	3	224◇	1	672

* *Liu-2020* releases logs from 10 tools. ⋆ *Liu-2020* configures each APR tools with two different FL strategies: utilizing off-the-shelf FL technique, GZoltar-v1.7 and feeding APR tools with ground-truth points. † This study does not release execution logs of *Nopol* and *DynaMoth* on *Closure* bugs. ◇ This study does not execute tools on bugs from *Closure* and *Mockito*.

patches and similar code to reduce search spaces; *Wang-2019* [18] reruns SimFix and CapGen [5] on *Mockito* bugs; *Liu-2020* [9] revisits the efficiency of test suite based program repair by performing a large-scale experiment on 16 APR tools; *Durieux-2019* [8] concerns benchmark overfitting problem through a comprehensive comparison of APR tools' performances among five benchmarks; *Martinez-2016* [7] reruns jKali, jGenProg, and Nopol on Defects4j to evaluate their abilities for fixing real-world defects. All these five studies have released their execution logs on their online repositories, which can be utilized for our study. Totally, we collected 11,818 logs in this study. The detailed information of the involved logs can be found in Table II. Other studies in the literature are excluded because 1) there is no online resource of their studies; 2) they do not release experiment logs; or 3) they only release logs where plausible patches are generated.

C. Experiment Settings

The following presents the experimental details of each research question.

1) RQ1: To answer RQ1, we need to find out all the fixing attempts that end with exceptions. To achieve so, we adopted a strategy, which is based on keyword mapping, to separate **valid fixing attempts** from the whole log benchmark. Specifically, **patch found** is used to search the *success* cases and **timeout** is used for searching the cases where APR tools fail to generate a patch within the time budget. For the cases where APR tools traverse the whole search space but find nothing, different types of APR tools throw diverse messages in the logs. For instance, **no angelic value** is used for constraint-based tools (e.g., `Nopol`) while **exhaustive navigated** is used for search-based tools (e.g., `jMutRepair`). Note that these three situations correspond to the three kinds of normal results of a fixing attempt as introduced in Section II-A. After filtering out those valid fixing attempts, we kept the rest as abnormal ones and categorized them based on the exception types in the thrown messages.

2) RQ2: To categorize the behind reasons of these unexpected exceptions, a process based on Thematic Analysis (TA) [38] was conducted. Following a previous study [39], we performed this process in six steps: 1) familiarizing with the log (by re-reading the logs for many times to understand its content); 2) identifying initial reasons (to understand the behind reasons for the exception in each log); 3) searching for themes (reasons re-appearing over many logs were identified); 4) reviewing themes (some themes with similar reasons were merged); 5) naming themes (some themes were renamed to better reflect the behind reasons); 6) producing the report (this paper reflects this step, compiling the main results of this analysis). This process was manually performed by the first author and another two authors further validated the result by reviewing the reason of each case from their comprehension.

3) RQ3: The evaluation of APR tools is usually assessed with respect to precision and recall [5], [17]. *Precision* measures how many bugs can be correctly fixed. On the contrary, *recall* measures the proportion of bugs for which a tool can generate plausible patches that can pass all the tests. The proportion is usually measured by the number of bugs with plausible patches generated over the total number of bugs. However, unexpected results are generated for certain bugs, in which case we actually cannot determine the capability of the tool to fix the bug since the fixing attempt terminated unexpectedly without results being generated. As a result, the recall such measured might be biased. In this RQ, we revisited the recall via removing those fixing attempts with unexpected results. Precision is not revisited since the non-repairability factors will not influence the precision value (i.e., the numbers of the generated patches and correct patches will not change). In our experiments, we also noted that our log benchmarks from different APR tools were generated under different FL strategies. For instance, *Durieux-2019* performs the study under GZoltar-v0.1 while *Liu-2020* utilizes two FL strategies including a new version of this framework which is GZoltar-v1.7 and directly feeding APR tools with ground-truth points. Another studied APR tool (i.e., *NPEFix*) does not require FL step since it only focuses on the potential crash point such as method call. We thus calculated the results of different APR tools under different FL strategies separately for constructing a baseline involving various FL configurations.

IV. Results and Analysis

In this section, we present experimental data as well as the key insights that are relevant to our research questions.

A. RQ1: Number of Occurrences

In total, we identified 3,035 exceptions from *Liu-2020* and *Durieux-2019* among our collected log benchmark which can be classified into 14 types based on the exception messages. Table III demonstrates the results aggregated by diverse APR tools as well as different projects in Defects4j.

• [***On the significance of this study***] We identified 3,035 logs ended with unexpected results, which account for 25.7% of the logs included by this study (i.e., 11,818 in total). This result suggests that a large proportion of fixing attempts suffered from the non-repairability factors during executions and thus it is of great significance to dissect the reasons for these non-repairability factors for avoiding identical problems in future studies.

• [***On the occurrences of different types of exceptions***] As indicated by the last column, different types of exceptions possess diverse numbers of occurrences, ranging from 3 to 1,128. *IllegalState* is the most common exception type which has occurred for 1,128 times, accounting for 37.2% (1,128/3,035) of the total amount, followed by *ModelBuilding* and *UnsupportedOperation* which are 463 and 329, respectively. We also noted that there are three kinds of exceptions (i.e., *StackOverflow*, *NumberFormat*, and *IO*) that occur infrequently.

• [***On the distributions with respect to APR tools***] We found that 11 APR tools suffer from the non-repairability factors in their evaluations. Specifically, *Cardumen* witnesses the largest number of exceptions, which is 782 in total, accounting for 25.7% (782/3,035) of the entire set. The followings are *jMutRepair* and *jGenProg* whose values are 529 and 478, respectively. We also noted that there are some tools where exceptions happen infrequently. For instance, *DynaMoth* only contains 25 unexpected results of 2 types of exceptions which are *OutOfMemory* and *NullPointer*.

• [***On the distributions with respect to Defects4j projects***] We found that exceptions happen more frequently when repairing the *Closure* project which includes 1,158 instances while less frequently in the *Chart* project which includes only 65 cases. For the other projects, the numbers range from 207 to 608. We noted that different projects in the Defects4j benchmark contain diverse numbers of bugs. From the perspective of *how many exceptions happen for a single bug on average*, the *Mockito* project achieves the highest value since it only contains 38 bugs, but 608 exceptions were generated when repairing those bugs.

Overall, our results reveal that (1) the non-repairability factors affect a considerable number of fixing attempts (25.7% of our studied logs); (2) IllegalState is the most common exception type which accounts for 37.2% of the whole benchmark; (3) the distributions of these exceptions vary a lot with respect to different APR tools and Defects4j projects.

B. RQ2: Internal Reasons

We summarized our identified reasons in Table IV which can be divided into 5 aspects. **Library bugs** denotes the libraries integrated in the APR tools used for program analysis or fault localization contain certain defects or not being used appropriately; **Inherent defect of APR tools** means the APR tools themselves are defective and contain logic errors; **Improper operation of replication study** denotes the inappropriate operations or settings adopted by researchers lead to the unexpected results; **Low performance of machines** is caused by the limited performance of the utilized machines,

TABLE III: Number of Occurrences of Different Types of Exceptions

Exception Type	APR Tool											Project						Total
	Nopol	DynaMoth	NPEFix	jGenProg	jKali	jMutRepair	Cardumen	ARJA	GenProg-A	RSRepair-A	Kali-A	Chart	Closure	Lang	Math	Mockito	Time	
UnsupportedOperation	0	0	0	0	0	0	329	0	0	0	0	31	248	2	11	7	30	329
StackOverflow†	0	0	0	3	0	0	0	0	0	0	0	0	0	0	3	0	0	3
Spoon	0	0	133	43	0	0	0	0	0	0	0	7	135	24	9	0	1	176
OutOfMemory†	109	15	0	0	0	0	0	0	0	0	0	2	30	8	39	24	21	124
NumberFormat	0	0	0	1	1	1	1	1	1	1	1	0	8	0	0	0	0	8
NullPointer	62	10	0	37	17	5	64	14	16	15	15	0	74	30	26	105	20	255
ModelBuilding	0	0	155	91	79	65	73	0	0	0	0	0	0	167	133	136	27	463
IO	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	3	0	3
IndexOutOfBounds	0	0	0	0	0	0	0	0	75	74	0	0	57	0	0	92	0	149
IllegalState	0	0	1	257	254	342	274	0	0	0	0	25	510	155	308	22	108	1,128
IllegalArgument	68	0	0	0	23	23	23	1	0	0	0	0	43	21	0	74	0	138
FileNotFound	0	0	0	18	18	18	18	18	18	18	18	0	0	0	0	144	0	144
NoClassDefFound†	1	0	0	0	0	0	0	0	0	0	14	0	0	0	14	1	0	15
Cannot allocate memory†	0	0	0	25	0	75	0	0	0	0	0	0	53	43	4	0	0	100
Total	239	25	289	478	392	529	782	34	110	108	48	65	1,158	450	547	608	207	3,035

† In this study, we also consider *StackOverflow*,*OutOfMemory*,*NoClassDefFound* and *Cannot allocate memory* as exceptions since they also lead to the unexpected end of a fixing attempt.

TABLE IV: Reasons Behind Different Types of Exceptions

Reason	Exception Type	Number	Total
Library bugs	NullPointer#1	182	504 (16.6%)
	Spoon	176	
	IllegalArgument#1	69	
	ModelBuilding#1	77	
Inherent defect of APR tools	NullPointer#2	73	1,785 (58.8%)
	ModelBuilding#2	106	
	IndexOutOfBounds	149	
	IllegalState	1,128	
	UnsupportedOperation	329	
Improper operation of replication study	FileNotFound	144	504 (16.6%)
	NumberFormat	8	
	IO	3	
	IllegalArgument#2	69	
	ModelBuilding#3	280	
Low performance of machines	OutofMemory	124	227 (7.5%)
	StackOverflow	3	
	Cannot allocate memory	100	
Unstable environment	NoClassDefFound	15	15 (0.5%)

and **Unstable environment** denotes the influence from environment (e.g., different versions of jdk or Operating System). Note that our manual analysis shows that several exceptions (i.e., IllegalArgument, ModelBuilding, and NullPointer) are caused by diverse root causes. We thus use # to denote different root causes in the table. Generally speaking, **Inherent defect of APR tools** is the most common type, accounting for nearly 60% of the total amount (1,785/3,035). Due to page limit, we omit the analysis for three unpopular exception types (i.e., NumberFormat, IO, and NoClassDefFound) in this paper. Details can be found in our project page [1].

1) Library Bugs: • [***NullPointer exception#1***] An example of *NullPointer* exception is shown in Listing 2. APR tools rely on the *GZoltar* library to obtain the fault localization results and this type of exception happens at the beginning of the initialization of *GZoltar* (line 3). After carefully understanding the source code, we found that different APR tools invoke the library APIs in a unified way but this type of exception occurs when processing the bugs of specific projects (e.g., *Closure* and *Mockito*). Such finding indicates that it is indeed caused by the problem in the external library (i.e., unable to process certain bugs of certain projects normally) rather than the defect in the APR tools and we thus categorized this type into **Library bugs**. We noted that this problem has also been noticed and discussed by other practitioners.[2] We further confirmed that this case occurred for all the APR tools where *GZoltar* has been integrated for FL, which means this library has affected a wide range of APR tools.

```
1  Exception in thread "main" java.lang.
        NullPointerException
2  at com.gzoltar.core.GZoltar.run(GZoltar.java:51)
3  at us.msu.cse.repair.core.faultlocalizer.
        GZoltarFaultLocalizer.<init>(GZoltarFaultLocalizer.
        java:42)
4  at us.msu.cse.repair.core.AbstractRepairProblem.
        invokeFaultLocalizer(AbstractRepairProblem.java
        :311)
5  at ......
```

Listing 2: An Instance of First Type of NullPointer Exception

• [***Spoon exception***] *Spoon* [3] is a meta-programming library to analyze and transform Java source code designed by the same research group of *Astor*. Listing 3 gives an example of this type of exception. Some APR tools utilize this library for code transformation but will throw this exception under some circumstances. It is classified into **Library bugs** since it was caused by internal bugs of libraries or inappropriate usages of libraries. For instance, the exception as shown in Listing 3 occurred since the enforced code transformation (i.e., inserting a statement before a super invocation) is not supported by the library.

```
spoon.SpoonException: cannot insert a statement before a
     super or this invocation.
at spoon.support.reflect.code.CtStatementImpl.
     insertBefore(CtStatementImpl.java:79)
at spoon.support.reflect.code.CtStatementImpl.
     insertBefore(CtStatementImpl.java:66)
at spoon.support.reflect.code.CtInvocationImpl.
     insertBefore(CtInvocationImpl.java:108)
at fr.inria.astor.approaches.jgenprog.operators.
     InsertBeforeOp.applyChangesInModel(InsertBeforeOp.
     java:27)
at fr.inria.astor.core.entities.OperatorInstance.
     applyModification(OperatorInstance.java:177)
at ......
```

Listing 3: An Instance of Spoon Exception

• [***IllegalArgument exception#1 & ModelBuilding exception#1***] The *Astor* system executes a command "defects4j export -p cp.test" to obtain the *classpath* for compilation. However, this command will delete the built test files for some Mockito bugs (i.e., 12-17 and 22-38). As a result, exceptions as shown in Listing 4 will be thrown. This is a bug from the *Defects4j* benchmark and has not been resolved for several years.[4] We thus categorized it into **Library bugs**.

```
Exception in thread "main" java.lang.
     IllegalArgumentException: No suspicious gen for
     analyze
at fr.inria.astor.core.faultlocalization.gzoltar.
     GZoltarFaultLocalization.calculateSuspicious(
     GZoltarFaultLocalization.java:93)
```

[1]http://doi.org/10.5281/zenodo.3937198
[2]https://github.com/SpoonLabs/astor/issues/94#issuecomment-595231021
[3]http://spoon.gforge.inria.fr/
[4]https://github.com/rjust/defects4j/issues/39

```
at fr.inria.astor.core.faultlocalization.gzoltar.
    GZoltarFaultLocalization.searchSuspicious(
    GZoltarFaultLocalization.java:49)
at fr.inria.astor.core.solutionsearch.AstorCoreEngine.
    calculateSuspicious(AstorCoreEngine.java:898)
at ......
```

Listing 4: An Instance of First Type of IllegalArgument Exception

Similar phenomenon happens for *Astor* on bugs from Math-100 to 106. The returned *classpath* misses an imported package when executing the identical command, leading to the failure of the build process. As a result, *modelbuilding* exception is thrown, which is classified to the same categorization as the above case. Due to space limit, we do not provide a concrete example here.

2) Inherent defect of APR Tools: • [***ModelBuilding exception#2***] Listing 5 shows another instance of the *ModelBuilding* exception observed in *NPEFix*. *NPEFix* is a metaprogramming-based tool specially designed for the program crashed by null pointer exceptions. It does not require the FL process, instead, it encapsulates each method call and field access to assess whether the object is null. If so, it applies predefined strategies to find a way to avoid the null pointer exceptions. Listing 6 shows an instance of code transformed by *NPEFix* for bug Lang-14 that will throw a *ModelBuilding* exception as shown in Listing 5. This piece of code aims to detect whether the return value of the function *getKey* is null. Nevertheless, this code does not declare the type of the variable *npe_invocation_var*, which leads to the compile error in the model building process. This instance demonstrates that defects exist in the code transformation process of *NPEFix* and we subsequently labeled it as **Inherent defect of APR tools**.

```
spoon.compiler.ModelBuildingException: Syntax error on
    token "final", float expected at /tmp/
    NPEFix_Defects4J_Lang_15/Pair.java:133
at spoon.support.compiler.jdt.JDTBasedSpoonCompiler.
    report(JDTBasedSpoonCompiler.java:581)
at spoon.support.compiler.jdt.JDTBasedSpoonCompiler.
    reportProblems(JDTBasedSpoonCompiler.java:562)
at spoon.support.compiler.jdt.JDTBasedSpoonCompiler.
    compile(JDTBasedSpoonCompiler.java:157)
at ......
```

Listing 5: An Instance of Second Type of ModelBuilding Exception

```
final npe_invocation_var = getKey();
if (checkForNull(npe_invocation_var)) {
......
```

Listing 6: An Instance of Code Transformed by NPEFix

• [***IndexOutOfBounds exception***] This type of exception happens in two APR tools (*GenProg-A* and *RSRepair-A*) which are based on the same framework. We thus take *GenProg-A* as an example as shown in Listing 7. Upon obtaining the FL results, this tool tries to generate modification points and then extract some ingredients in the original buggy program as the seed to perform transformations to search for possible solutions (i.e., plausible patches). Specifically, *GenProg-A* sets a number of restrictions to filter invalid ingredients for every modification point, which means a portion of modification points may not get any valid ingredient. For instance, one of the restrictions is that the selected ingredients should not be in the subtree of the modification point from the perspective of Abstract Syntax Tree (AST). Therefore, it throws up an *IndexOutOfBounds* exception as shown in the list when trying to get ingredient elements but the ingredient list is empty. According to our analysis, this exception can be avoided by removing the modification points without any valid ingredient. Hence, we categorized this type of exception into **Inherent defect of APR tools**. We noted that the numbers of occurrences of this type of exception are different among *GenProg-A* and *RSRepair-A*, which is due to the different ability for generating a plausible patch at a certain modification point. For instance, for *Closure-33*, *GenProg-A* does not generate a patch passing all the tests at the first modification point while *RSRepair-A* does and stops searching. Therefore, *GenProg-A* keeps searching on the next modification point where it throws the exception.

```
Exception in thread "main" java.lang.
    IndexOutOfBoundsException: Index: 0, Size: 0
at java.util.ArrayList.rangeCheck(ArrayList.java:635)
at java.util.ArrayList.get(ArrayList.java:411)
at us.msu.cse.repair.ec.problems.GenProgProblem.evaluate
    (GenProgProblem.java:89)
at us.msu.cse.repair.ec.algorithms.GenProgGA.
    initPopulation(GenProgGA.java:107)
at ......
```

Listing 7: An Instance of IndexOutOfBounds Exception

• [***IllegalState & UnsupportedOperation exception***] *IllegalState* exception happens most frequently and it occurs majorly among four APR tools based on the *Astor* framework according to our investigation as shown in Table III. Listing 8 shows an instance of this type of exception. Similar to *GenProg-A* and *RSRepair-A*, the tools integrated in *Astor* system also seek to create modification points after the FL step. What is different is that the filter process is performed based on the type of the suspicious code while in *GenProg-A* and *RSRepair-A* it is conducted considering whether there exists valid ingredients for each point. Originally, these tools only support statements of certain types such as conditional statements and loop statements while do not support for the others (e.g., return statements and assignment statements). As a result, if all the identified suspicious statements are not supported, the illustrated exception will be thrown. Take *Chart-10* as an example. From the open access FL results,[5] the only suspicious line under GZoltar-v0.1 for this buggy program is a return statement. Then, when trying to generate modification points, the system throws this exception. Note that the four APR tools integrated in *Astor* generate different numbers of such exceptions as shown in Table III. The reason is that each tool under this framework is implemented independently but there are some common types of statements that they do not support. We acknowledge that it is extremely difficult to take every situation into consideration when implementing such a large project. Being that said, we still categorized this type of exception into **Inherent defect of APR tools** in that some unsupported statements are extremely common such as return statements.

```
Exception in thread "main" java.lang.
    IllegalStateException: Variant without any
    modification point. It must have at least one.
```

[5]https://github.com/SerVal-DTF/FL-VS-APR

```
at fr.inria.astor.core.solutionsearch.AstorCoreEngine.
      initializePopulation(AstorCoreEngine.java:826)
at fr.inria.astor.core.solutionsearch.AstorCoreEngine.
      initPopulation(AstorCoreEngine.java:694)
at fr.inria.main.evolution.AstorMain.createEngine(
      AstorMain.java:127)
at ......
```

Listing 8: An Instance of IllegalState Exception

We have demonstrated a case of *Unsupported* exception in Section II-B. This type of exception is an exclusive one of *Cardumen*. Further investigation reveals that it is due to the inappropriate implementation of this APR tool. Similar to *GenProg-A*, if the type of suspicious code is supported, the tool will retrieve program elements in the program to select potential ingredients. Specifically, it will record the position (i.e., the line number) of each program element and later perform a check to see whether any information is illegal. Nevertheless, the element's position attribute will be empty if the element is the `This` pointer in the class. Note that other tools in *Astor* will skip the check when processing a `This` pointer and that is why this exception only occurs in *Cardumen*. We thus also categorized this type of exception into **Inherent defect of APR tools**.

• [***NullPointer exception#2***] Listing 9 shows an example of this case that is exclusive to the *Astor* system. The tools try to record the number of valid repair attempts for each modification point to serve for further debugging. However, some modification points are unable to get any valid program element to be used as ingredients for repair due to the predefined restrictions, similar to the previous analysis of *GenProg-A*. In particular, the element should be extracted from the same file as the modification point. If the modification points contain no valid repair attempt, the returned object that records valid repair attempts will be *null*. Consequently, this exception will occur if the validity of the returned object is not checked. We categorized this type into **Inherent defect of APR tools**. The information from the log reveals that the experiment of [8] was performed in December 2018 while this bug was later fixed in January 2019 [6].

```
java.lang.NullPointerException
at fr.inria.astor.core.solutionsearch.spaces.ingredients
      .ingredientSearch.
      RandomSelectionTransformedIngredientStrategy.
      getFixIngredient(
      RandomSelectionTransformedIngredientStrategy.java
      :77)
at fr.inria.astor.core.ingredientbased.
      IngredientBasedEvolutionaryRepairApproachImpl.
      createOperatorInstanceForPoint(
      IngredientBasedEvolutionaryRepairApproachImpl.java
      :95)
at fr.inria.astor.core.solutionsearch.
      EvolutionarySearchEngine.modifyProgramVariant(
      EvolutionarySearchEngine.java:264)
at ......
```

Listing 9: An Instance of Second Type of NullPointer Exception

3) Improper Operation of Replication Study: • [***FileNotFound exception***] Listing 10 shows an instance of this kind of exception. We noted that the latest version of *GZoltar-v1.7* does not contain a package that can be called directly similar to *GZoltar-v0.1*. Therefore, researchers choose to first execute this tool and record the results in a file and then read the FL results from the corresponding files (we have confirmed this with the authors of [9]). If the researchers pass a wrong path to the program or there is no corresponding file under the patch, the repair tool will expose such an exception. We thus categorized this type into **Improper operation of replication study** since it is caused by the inappropriate operations from the performers of the study.

```
Exception in thread "main" java.io.FileNotFoundException
      : /data/RepairThemAll/location/Defects4J/Mockito_12
      .txt (No such file or directory)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java
      :146)
at us.msu.cse.repair.core.AbstractRepairProblem.
      invokeFaultLocalizer(AbstractRepairProblem.java
      :320)
at us.msu.cse.repair.core.AbstractRepairProblem.
      invokeModules(AbstractRepairProblem.java:282)
at ......
```

Listing 10: An Instance of FileNotFound Exception

• [***IllegalArgument exception#2***] This type of exception happens in several projects. We illustrate an example of *Nopol* on project *Closure* in Listing 11. *Closure* project needs a dependency named *rhino.jar* during execution. However, due to the evolution of this project, the name of this dependency changes to another one from bug *Closure-64* to *Closure-106*. To find a workaround for this problem, the authors updated the way of obtaining *classpath* for several times.[7] However, the execution time in the log is earlier than the submission time of this commit, which means the authors forgot to rerun the tool after update (we have confirmed this with the authors of [8]). Hence, we categorized this type of exception into **Improper operation of replication study**. Note that cases of this type of exception in other projects are related to other *classpath* problems. The instance of *Closure* project here is just an example. We also noted that the study *Martinez-2016* also ran *Nopol* while we did not identify exception from their logs. Our further check reveals that their study only included 224 bugs (*Closure* and *Mockito* were not included) as shown in Table II.

```
java.lang.IllegalArgumentException: File does not exist
      in: '/tmp/Nopol_Defects4J_Closure_100/build/lib/
      rhino.jar'
at xxl.java.library.FileLibrary.fail(FileLibrary.java
      :129)
at xxl.java.library.FileLibrary.openFrom(FileLibrary.
      java:29)
at xxl.java.library.FileLibrary.urlFrom(FileLibrary.java
      :100)
at xxl.java.library.JavaLibrary.classpathFrom(
      JavaLibrary.java:84)
at fr.inria.lille.repair.Main.parseArguments(Main.java
      :161)
at ......
```

Listing 11: An Instance of Second Type of IllegalArgument Exception

• [***ModelBuilding exception#3***] Another case of *ModelBuilding* exception is shown in Listing 12. We found that

[6]https://github.com/SpoonLabs/astor/commit/378776c63cf082d29156e3fcb02ef7a3a51fe693

[7]https://github.com/program-repair/RepairThemAll/commit/4a98bc108cb06b3f894bbb6e70dbd91e96c64601#diff-a0dddd360955b4f2a6ecb343397ea54e

the package name of 24 bugs (Lang-42 to 65) contains *enum* which can be identified as a keyword by the jdk version later than 1.5. Normally, these programs are compiled under jdk 1.4 which only throws a warning but does not influence the execution. However, the used version is 1.7 in the reproduction experiment (we got this information through the *compliance level* in the logs). Consequently, *enum* is identified as a keyword and the build process is failed. We thus categorized this type of cases into **Improper operation of replication study**. Bugs from Mockito-1 to 38 suffer the identical exception and our investigation reveals that those exceptions are also related with the utilized jdk version. We do not provide a case here due to space limit.

```
Exception in thread "main" spoon.compiler.
      ModelBuildingException: Syntax error on token "enum
      ", Identifier expected at /tmp/
      Cardumen_Defects4J_Lang_59/src/java/org/apache/
      commons/lang/enum/EnumUtils.java:17
at spoon.support.compiler.jdt.JDTBasedSpoonCompiler.
      report(JDTBasedSpoonCompiler.java:641)
at spoon.support.compiler.jdt.JDTBasedSpoonCompiler.
      reportProblems(JDTBasedSpoonCompiler.java:622)
at spoon.support.compiler.jdt.JDTBasedSpoonCompiler.
      build(JDTBasedSpoonCompiler.java:131)
at spoon.support.compiler.jdt.JDTBasedSpoonCompiler.
      build(JDTBasedSpoonCompiler.java:113)
at fr.inria.astor.core.manipulation.MutationSupporter.
      buildModel(MutationSupporter.java:81)
at ......
```

Listing 12: An Instance of Third Type of ModelBuilding Exception

4) Low Performance of Machines: • [***OutofMemory & StackOverflow & Cannot allocate memory exception***] Executing APR tools requires high performance servers since the search space is generally extremely large. If the computing performance is not high, some tools may run out of the heap space and thus cause an error. Listing 13 gives an example for this situation. Two similar exceptions (i.e., *StackOverflow* and *Cannot allocate memory*) are shown in Listings 14 and 15. The former happens when the length of stack exceeds a threshold while the latter occurs when there is not enough memory assigned. *StackOverflow* exception is not common and only occurs for *jGenProg*, a tool utilizes genetic programming (GP) for searching for potential patches. This result indicates that the search space of GP is generally larger than other techniques. If the experiment is performed under machines with better performance, these exceptions are of great possibility to be avoided. As a result, we categorized these three types of exceptions into **Low performance of machines**.

```
java.util.concurrent.ExecutionException: java.lang.
      OutOfMemoryError: Java heap space
at java.util.concurrent.FutureTask.report(FutureTask.
      java:122)
at java.util.concurrent.FutureTask.get(FutureTask.java
      :206)
at fr.inria.lille.repair.Main.main(Main.java:106)
```

Listing 13: An Instance of OutOfMemory Exception

```
java.lang.StackOverflowError
at java.lang.ThreadLocal.get(ThreadLocal.java:161)
at spoon.reflect.factory.FactoryImpl.dedup(FactoryImpl.
      java:420)
at spoon.support.reflect.reference.CtReferenceImpl.
      setSimpleName(CtReferenceImpl.java:62)
at ......
```

Listing 14: An Instance of StackOverflow Exception

```
Java HotSpot(TM) 64-Bit Server VM warning: INFO: os::
      commit_memory(0x00000000d5550000, 715849728, 0)
      failed; error='Cannot allocate memory' (errno=12)
```

Listing 15: An Instance of CannotAllocateMemory Exception

Our in-depth analysis reveals that (1) the causes of the non-repairability factors are diverse, which can be mainly classified into 5 categories containing divergent types of exceptions; (2) Inherent defect of APR tools *is the dominating one which accounts for nearly 60% of the total amount.*

C. RQ3: Unbiased Evaluation

We re-calculated the recalls of different APR tools under different Fault Localization strategies without considering the fixing attempts that ended with unexpected results as introduced in Section III-C3. Table V shows the results. The *FL Strategy* column denotes different FL strategies where *GZ-0.1* denotes utilizing FL results obtained from GZoltar-v0.1, *GZ-1.7* denotes utilizing FL results obtained from GZoltar-v1.7, *Perfect FL* denotes feeding APR tools with ground-truth points directly, and *No FL required* only denotes *NPEFix*.

TABLE V: Performances of APR Tools under Different FL Strategies when only Considering Valid Fixing Attempts.

FL Strategy	APR tool	Chart	Closure	Lang	Math	Mockito	Time	Total	Recall
GZ-0.1	Nopol	7(26)	70(71)	6(38)	23(68)	0(7)	1(6)	107(216)	↑ **22.44**%
	DynaMoth	7(25)	45(122)	2(64)	15(105)	1(33)	1(27)	71(376)	↑ 0.9%
	jGenProg	7(26)	4(129)	0(40)	20(92)	0(0)	0(27)	31(314)	↑ 2.02%
	jKali	5(26)	12(133)	0(53)	10(99)	0(8)	0(27)	27(346)	↑ 0.96%
	jMutRepair	3(26)	7(128)	0(53)	10(90)	0(10)	0(27)	20(334)	↑ 0.92%
	Cardumen	5(13)	0(8)	0(36)	12(84)	0(0)	0(4)	17(145)	↑ 7.42%
	Arja	9(26)	37(133)	16(65)	23(106)	1(33)	0(27)	86(390)	↑ 0.27%
	GenProg	6(26)	22(116)	2(65)	15(105)	0(28)	0(27)	45(367)	↑ 0.86%
	RSRepair	8(26)	27(116)	4(65)	23(106)	0(28)	0(27)	62(368)	↑ 1.15%
	Kali	6(26)	51(133)	2(65)	12(92)	1(33)	0(27)	72(376)	↑ 0.92%
GZ-1.7	Nopol	6(26)	0(78)	6(65)	18(106)	0(38)	1(27)	31(340)	↑ 1.26%
	DynaMoth	6(26)	0(133)	2(65)	13(106)	0(38)	1(27)	22(395)	→
	jGenProg	5(20)	2(131)	2(18)	11(84)	0(18)	0(26)	20(297)	↑ 1.67%
	jKali	4(26)	8(122)	4(53)	13(98)	0(18)	0(27)	29(344)	↑ 1.08%
	jMutRepair	4(25)	5(79)	2(44)	11(96)	0(18)	0(27)	22(289)	↑ 2.04%
	Cardumen	4(9)	2(6)	0(51)	6(99)	0(9)	0(0)	12(174)	↑ 3.85%
	Arja	10(26)	29(133)	3(65)	15(106)	1(15)	0(27)	58(372)	↑ 0.9%
	GenProg	5(26)	15(129)	1(65)	9(106)	0(0)	0(27)	30(353)	↑ 0.9%
	RSRepair	5(26)	22(130)	3(65)	12(106)	0(0)	0(27)	42(354)	↑ 1.23%
	Kali	6(26)	48(133)	0(65)	10(106)	1(15)	0(27)	65(372)	↑ 1.01%
Perfect FL	Nopol	1(26)	0(133)	3(64)	5(106)	0(33)	0(27)	9(389)	↑ 0.03%
	DynaMoth	1(25)	0(133)	4(65)	8(106)	0(33)	0(27)	13(389)	↑ 0.05%
	jGenProg	2(23)	2(4)	0(17)	12(35)	0(17)	0(0)	16(96)	↑ **12.61**%
	jKali	1(24)	2(13)	0(17)	5(29)	0(20)	0(0)	8(103)	↑ 5.74%
	jMutRepair	1(7)	4(6)	0(6)	6(13)	0(1)	0(0)	11(33)	↑ **30.54**%
	Cardumen	6(24)	0(1)	1(18)	9(23)	0(18)	0(0)	16(84)	↑ **14.99**%
	Arja	2(26)	16(132)	5(65)	13(106)	0(33)	0(27)	36(389)	↑ 0.14%
	GenProg	3(26)	16(124)	3(65)	7(106)	0(7)	0(27)	29(355)	↑ 0.82%
	RSRepair	2(26)	16(124)	4(65)	12(106)	0(7)	0(27)	34(355)	↑ 0.96%
	Kali	1(26)	30(132)	3(65)	8(106)	0(33)	0(27)	42(389)	↑ 0.16%
No FL required	NPEFix	5(26)	0(0)	0(18)	3(43)	1(19)	0(0)	9(106)	↑ 6.21%

x(y) means that under this project and FL strategy, the APR tool performs y valid fixing attempts and generates patches that can pass all the tests for x bugs. The column **Recall** demonstrates the variations of recall w.r.t results from previous studies where the bold ones indicate the change of the recall value is larger than 10%. → means the recall value does not change.

• [***On the broad impact of non-repairability factors***] Table V lists the results for 31 experiments via considering different APR tools and FL strategies. We found that the results of 30 of them have been affected by the non-repairability factors (i.e., the recall value is changed when revisited). The only result remains unchanged is obtained when executing *DynaMoth* under GZoltar-v1.7 where the 395 original fixing attempts are all valid. Such results reveal that non-repairability factors bring broad impact on various APR tools under diverse FL configurations.

• [***On the changes of recalls***] We found that the change of recall value under most of the experiments is less than 2% (21/31) which means the impact of non-repairability factors is limited. However, there are also 4 experiments in which the values changed significantly (i.e., more than 10%) as emphasized in the table. Generally speaking, the changes under *GZoltar-v1.7* are limited where the maximum variance only reaches 3.85%. As a comparison, the changes under the setting of *Perfect FL* are generally more significant since the number of valid fixing attempts of a tool is sometimes even less than 100. For instance, the number of valid fixing attempts of *jMutRepair* is 33, much less than the total number of bugs in Defects4j benchmark which is 395. As a result, its recall has increased by 30.54%.

Our revisiting reveals that (1) non-repairability factors are widespread among evaluations on diverse combinations of APR tools and FL strategies; (2) the changes of recalls vary for different experiments. Although the changes are limited under most of the experiments, the maximum value can reach over 30%.

V. Discussion

A. Implications

To recap, we have gained insights into the behind reasons of the existing exceptions in the execution logs of APR tools, which can be mainly divided into 5 categories. Based on these insights, we provide several implications for avoiding such exceptions as follows.

Execution logs of APR tools should be made publicly available to the community.

We are grateful for the authors of [6]–[9], [18] to release their execution logs. We believe that considerable efforts have been made by the original authors to ensure the quality of their experiments. Nevertheless, we still found over a quarter of these fixing attempts ended with unexpected results. Our findings here (e.g., the types of the exceptions and the behind reasons) may not generalize to the logs which have not been released publicly. Therefore, we encourage future APR studies to release their data for public inspection. The community can then gain more insights on the exceptions occurred during the experiments and to better avoid the impacts of such non-repairability factors.

Developers of APR tools should ensure the quality of their products.

From our analysis, several types of exceptions occurred due to the inappropriate implementation of the APR tools. Our in-depth analysis revealed that the majority of such exceptions could be avoided. For instance, the example of *IllegalState* exception as shown in Listing 8 can be resolved by considering more situations and the example of *Unsupported* exception in Listing 1 can be avoided by carefully checking the code. We thus recommend the developers of APR tools to carefully check the repair process and fix bugs before releasing the tool publicly.

Third party libraries should be evolved continuously.

Our results revealed that several integrated libraries may affect the evaluation results of APR tools (e.g., the widely-used GZoltar and Spoon). We thus encourage the developers of these third party libraries to pay more attention to the exposed problems. We believe it is difficult to ensure the quality of such a large library under undetermined execution environments. Nonetheless, the library should be evolved if a certain defect has been reported. From this perspective, we thank the efforts of authors of *Astor* [33] for recently fixing a situation of the *IllegalState exception*[8] as we introduced in Section IV-B.

Replication studies should make great efforts to ensure the reliability of the performed experiments.

We also pointed out that several types of exceptions are related to the inappropriate settings of the experiments such as not providing corresponding input files (*FileNotFound exception*) or not preparing the correct content (*NumberFormat exception*). We thus encourage future replication studies to ensure the reliability of the performed experiments via adopting appropriate settings. Besides, we also recommend researchers to pay attention to the configurations of their machines to avoid exceptions such as *OutOfMemory*. According to the project page of *RepairThemAll*, their studies were performed on a server whose maximum memory is 4G but we still detected *OutOfMemory exception* in their logs. This implicates that high performance machines might be needed when running certain APR tools.

B. Threats to Validity

External validity. The main lack of this study is that we only select 5 existing researches as our study subjects, which is a small part compared with existing studies on Java repair systems in the literature. Nevertheless, this is mitigated by our literature review that these studies are the only ones that release execution logs of all fixing attempts. Furthermore, as demonstrated in IV-C, our study subjects include 11 APR tools under diverse FL strategies, which strengthens the generality of our findings.

Internal validity. The taxonomy of internal reasons of the studied exceptions is the result from our manual analysis of logs. As any manual work, it is a difficult and time-consuming process and extremely error-prone. Nonetheless, we mitigate this by stopping the annotation process only if the three involved authors reached a consensus. We have also reported some of our findings to the authors of [8], [9] and got positive feedback. Besides, we have mined evolution information from the repositories of studied subjects to back up our analysis (shown as the links in the footnote). Furthermore, we make our data publicly available to ease future investigation and check for the whole repair community.

VI. Related Work

Each time an APR tool was proposed, its developers performed considerable experiments to evaluate its effectiveness, from which the **repairability factors** of the tool were discussed. To name a list, *TBar* suffers from insufficient fix patterns and ineffective search of fix ingredients [19], *SimFix* can be improved by considering the dependency between variables and statements [6], and incorrect fault localization and multiple fault locations significantly hamper the state-of-the-art APR tools [8]. On the contrary, our study is the first to

[8]https://github.com/SpoonLabs/astor/commit/5f9b69c7d5bef0814a0f7a9182e1de03de102815

deeply understand the **non-repairability factors** of APR tools, aiming to eliminate the potential biases in the evaluations.

There are many other studies concerning the biases in APR tasks. Durieux et al. [8] and Wang et al. [18] focused on the benchmark overfitting problem. Long et al. [40] concerned the patch overfitting problem while a number of studies followed [41], [42]. Liu et al. [21] first pointed out that different APR tools integrate different FL strategies. They thus designed a baseline tool kPAR and demonstrated that different FL results can significantly influence the number of generated patches, which is further confirmed by [9] where 16 APR tools are systematically considered.

VII. Conclusion

This paper reports a large-scale empirical study on the exceptions in the execution logs of APR techniques. We dissected the frequencies of occurrence, behind reasons of such occurrences, as well as the impact on the repairability of APR techniques of non-repairability factors. We further discussed several implications for eliminating biases caused by such exceptions. All data in this study are publicly available at: **http://doi.org/10.5281/zenodo.3937198**.

References

[1] M. Monperrus, "The living review on automated program repair," 2018.

[2] C. Le Goues and et al., "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, 2012.

[3] H. D. T. Nguyen and et al., "Semfix: Program repair via semantic analysis," in *International Conference on Software Engineering*, 2013.

[4] R. Just and et al., "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM.

[5] M. Wen and et al., "Context-aware patch generation for better automated program repair," in *International Conference on Software Engineering*, 2018.

[6] J. Jiang and et al., "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018.

[7] M. Martinez and et al., "Automatic repair of real bugs in java," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[8] T. Durieux and et al., "Empirical review of Java program repair tools," in *Proceedings of the Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 302–313.

[9] K. Liu and et al., "On the efficiency of test suite based program repair," in *International Conference on Software Engineering*, 2020.

[10] R. K. Saha and et al., "Elixir: Effective object-oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 648–659.

[11] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, 2018.

[12] J. Campos and et al., "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 378–381.

[13] J. Xuan and et al., "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[14] X. B. D. Le and et al., "History driven program repair," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 213–224.

[15] M. White and et al., "Sorting and transforming program repair ingredients via deep learning code similarities," in *International Conference on Software Analysis, Evolution and Reengineering*, 2019.

[16] T. Durieux and M. Monperrus, "Dynamoth: dynamic code synthesis for automatic program repair," in *Proceedings of the 11th IEEE/ACM International Workshop in Automation of Software Test*. IEEE, 2016.

[17] Y. Xiong and et al., "Precise condition synthesis for program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 2017, pp. 416–426.

[18] S. Wang and et al., "Attention please: Consider mockito when evaluating newly proposed automated program repair techniques," in *Proceedings of the 23rd Evaluation and Assessment on Software Engineering*. ACM.

[19] K. Liu and et al., "TBar: Revisiting template-based automated program repair," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 31–42.

[20] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 65–86.

[21] K. Liu and et al., "You Cannot Fix What You Cannot Find!" in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*, 2019, pp. 102–113.

[22] D. Kim and et al., "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013, pp. 802–811.

[23] X.-B. D. Le and et al., "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 593–604.

[24] S. Saha and et al., "Harnessing evolution for multi-hunk program repair," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 13–24.

[25] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 118–129.

[26] J. Hua and et al., "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 12–23.

[27] A. Ghanbari and et al., "Practical program repair via bytecode mutation," in *International Symposium on Software Testing and Analysis*, 2019.

[28] K. Liu and et al., "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019.

[29] A. Koyuncu and et al., "Fixminer: Mining relevant fix patterns for automated program repair," *arXiv preprint arXiv:1810.01791*, 2018.

[30] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2017, pp. 660–670.

[31] T. Durieux and et al., "Dynamic patch generation for null pointer exceptions using metaprogramming," in *the International Conference on Software Analysis, Evolution and Reengineering*, 2017.

[32] K. Liu and et al., "LSRepair: Live Search of Fix Ingredients for Automated Program Repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, 2018, pp. 658–662.

[33] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *International Symposium on Software Testing and Analysis*, 2016.

[34] Z. Chen and et al., "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*.

[35] Y. Li and et al., "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 2020.

[36] T. Lutellier and et al., "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2020.

[37] L. Chen and et al., "Contract-based program repair without the contracts," in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2017, pp. 637–647.

[38] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[39] V. Sobreira and et al., "Dissection of a bug dataset," in *the International Conference on Software Analysis, Evolution and Reengineering*, 2018.

[40] F. Long and M. Rinard, "An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016.

[41] X.-B. D. Le and et al., "On reliability of patch correctness assessment," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2019, pp. 524–535.

[42] S. Wang and et al., "How different is it between machine-generated and developer-provided patches?" in *the International Symposium on Empirical Software Engineering and Measurement*, 2019.