Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries

Shaikh Mostafa
shiakh.mostafa@utsa.edu
University of Texas at San Antonio
TX, USA

Rodney Rodriguez
kto128@my.utsa.edu
University of Texas at San Antonio
TX, USA

Xiaoyin Wang
xiaoyin.wang@utsa.edu
University of Texas at San Antonio
TX, USA

ABSTRACT

Nowadays, due to the frequent technological innovation and market changes, software libraries are evolving very quickly. Backward compatibility has always been one of the most important requirements during the evolution of software platforms and libraries. However, backward compatibility is seldom fully achieved in practice, and many relevant software failures are reported. Therefore, it is important to understand the status, major reasons, and impact of backward incompatibilities in real world software. This paper presents an empirical study to understand behavioral changes of APIs during evolution of software libraries. Specifically, we performed a large-scale cross-version regression testing on 68 consecutive version pairs from 15 popular Java software libraries. Furthermore, we collected and studied 126 real-world software bugs reports on backward incompatibilities of software libraries. Our major findings include: (1) 1,094 test failures / errors and 296 behavioral backward incompatibilities are detected from 52 of 68 consecutive version pairs; (2) there is a distribution mismatch between incompatibilities detected by library-side regression testing, and bug-inducing incompatibilities; (3) the majority of behavioral backward incompatibilities are not well documented in API documents or release notes; and (4) 67% of fixed client bugs caused by backward incompatibilities in software libraries are fixed by client developers, through several simple change patterns made to the backward incompatible API invocation.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**;

KEYWORDS

Behavior Backward Incompatibilities, Library Evolution

ACM Reference format:
Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
https://doi.org/10.1145/3092703.3092721

ISSTA'17, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092721

1 INTRODUCTION

Nowadays, as software products become larger and more complicated, software libraries have become a necessary part of almost any software. Since software libraries and their client software are typically maintained by different developers, the asynchronous evolution of software libraries and client software may result in incompatibilities. To avoid incompatibilities, for decades, "backward compatibility" has been a well known requirement in the evolution of software libraries. Each API method in an existing version of software library should exactly maintain its behavior in the following versions.

However, in reality, full backward compatibility is seldom achieved, and the resultant incompatibilities have been known to affect software users as well as the success of both software libraries and client software. For example, criticism on Windows Vista can be largely ascribed to its backward incompatibility with Windows XP [2]. In October 2014, the automatic system update to Android 5.0 caused a complete dysfunction of SogouInput (the top Android app for Chinese input on mobile phones, with more than 200 million users), which is not patched until 3 days later [3]. Also, a recent study [16] has shown that the usage of unstable Android APIs is an important factor affecting the success of Android apps. Therefore, we believe that it is necessary to conduct thorough studies to understand the status and reasons of backward incompatibilities, and the result of such studies may lead to more advanced techniques to support detection, documentation, and resolution of backward incompatibilities.

There have been many existing empirical studies [19, 27] on the stability of software libraries, and extensive research efforts on library migration [8, 34]. However, these studies mainly focus on the signature incompatibilities (i.e., added, revised, and removed API methods) between consecutive versions of software libraries. Although API signature changes form an important category of backward incompatibilities, they do not describe the whole picture. Even if the signature of an API method remains identical in a new version, it is possible that its behavior changes (i.e., it generates different output or side effect when certain input are fed in). We refer to such behavioral changes as *Behavioral Backward Incompatibilities* (BBIs). Actually, since signature incompatibilities can be detected by compilers, although they may cause extra efforts in library migration, they are less likely to cause real-world bugs, and thus less harmful compared to BBIs.

To acquire a deeper and more complete understanding of BBIs in real world software development, in this paper, we present an experimental study on 68 consecutive version pairs from 15 popular Java software libraries. For each pair, we performed cross-version testing to test the new version of software library with the test

code of the old version, and manually inspected and categorized the test errors / failures. We further collected and manually inspected 126 real-world bug reports related to BBIs in these libraries. We chose Java software libraries as our subjects because Java software typically extensively relies on libraries, and most popular Java software libraries are open source with test code available.

The three main contributions this paper makes include:

- A large scale experimental study on BBIs from 15 popular Java libraries and 68 version pairs, grouping 1094 detected test errors / failures to BBIs, and categorizing BBIs by incompatible behaviors, invocation conditions, and reasons.
- A qualitative study of 126 real world bugs caused by BBIs, including the categorization of their root-cause BBIs, and their resolutions.
- A data set including 296 BBIs detected in cross-version testing, and 126 real world bugs, serving as a basis for future research in this area.

2 RESEARCH SCOPE

In our study, we try to answer the five research questions as follows.

- **RQ1:** Are BBIs prevalent between consecutive version pairs of Java software libraries?
- **RQ2:** Are most BBIs distribute in the major version upgrades, so that minor version updates are safer?
- **RQ3:** What are the characteristics of BBIs and why library developers bring them in?
- **RQ4:** How are BBIs detected in regression testing compare with BBIs causing real-world bugs?
- **RQ5:** How are the BBI-related bugs fixed in software practice?

With the answers of these questions, we expect to understand: (1) whether BBIs are prevalent in the Java software libraries, and a problem that software developers need to face frequently, (2) whether BBIs are distributed most in major version upgrades, which are supposed to have some software interface changes, (3) whether it is possible to classify BBIs into several categories by their characteristics and reasons brought in, so that they can be avoided or detection and resolution techniques can be developed accordingly, (4) whether there are mismatches on the categories of BBIs being most detected and the categories of BBIs mostly likely to cause bugs, and (5) whether BBIs are fixed by library or client developers, and whether there exists certain fixing patterns on BBI related bugs.

3 CROSS-VERSION TESTING STUDY

In this section, we introduce our experimental study[1] on consecutive versions of Java libraries with cross-version testing.

3.1 Study Setup

3.1.1 Subject Libraries. In our experimental study, we used 15 popular Java libraries as our subjects as shown in Column 1 of Table 1. Specifically, we include in our subjects the two most widely used Java libraries: OpenJDK and the Android framework. Actually, since these two libraries have corresponding runtime platforms (i.e., JVM and Android OS), their BBIs are more likely to cause runtime errors, because the old version of client software may be executed in JVM or Android without recompile, and thus may result in runtime errors. Our subjects also include 7 libraries from Apache, and 6 other third-party libraries. All these libraries are from different domains and are the most popular software libraries in their domains according to a statistics [22] [23] of class imports among top 5,000 Java software projects in Github.

Table 1: Basic Information of Studied Subjects and Versions

Subject	St. V.	End V.	# V.	St. Time	End Time
OpenJDK	7b157	8b13	2	2011-7	2014-3
Android	4.3.1	5.0.1	2	2013-10	2014-12
log4j	2.0.0	2.1	2	2014-7	2014-10
maven	3.0.0	3.2.5	4	2010-10	2014-12
bukkit	1.2.3	1.7.2	6	2011-12	2013-12
beanutils	1.9.0	1.9.2	1	2008-9	2013-12
codec	1.6	1.7	1	2011-11	2012-9
fileupload	1.2.0	1.3.1	3	2007-2	2014-2
commons-io	2.0	2.4	4	2007-7	2012-4
ela. Search	1.0.3	1.3.9	7	2014-4	2015-2
http-core	4.0.1	4.3.3	6	2009-2	2014-2
jodatime	2.0	2.7	7	2011-5	2015-1
jsoup	1.1.1	1.7.3	10	2010-6	2013-11
neo4j	1.8.3	2.0.3	5	2012-11	2015-2
snakeyaml	1.3	1.11	8	2009-7	2012-9

3.1.2 Selection of Version Pairs. Software developers use different levels of versions to mark different granularity of milestones in software evolution. In our study, we first rule out the alpha and beta versions which are typically immature versions and are not widely used by client software developers. Then, we also need to differentiate major versions and minor versions. According to Semantic Versioning [1][28], backward-incompatible API changes can be allowed in major versions (e.g., Java 6), but not minor versions (e.g., Java6u32). Also, major versions are typically developed in separate branches, while minor versions just corresponds to certain commits in the trunk or a branch.

To acquire a full picture, in our study, we study backward incompatibilities both between two consecutive major versions and within a major version. If a major version has more than two minor versions, we choose the first minor version and the last minor version to form an inner-major-version version pair. For example, ElasticSearch has four minor versions (1.0.0 through 1.0.3) for major version 1.0, and three minor versions (1.1.0 through 1.1.2) for major version 1.1. So, in our study, we choose four versions (1.0.0, 1.0.3, 1.1.0, 1.1.2), and form 1 major version pairs (1.0.3 to 1.1.0) and two minor version pairs (1.0.0 to 1.0.3, 1.1.0 to 1.1.2). We combine minor versions within a major version because they typically contain very small amount of changes (e.g., fixing a bug), and may be inverted in the later updates if bringing in bugs or BBIs. So the combination will remove temporary BBIs (brought in and fixed with in a major version) which may not affect client developers much. We also ruled out the versions that raise compilation errors, or unit test failures / errors[2]. Finally, we use only versions up to Jan. 2015, so that their status (e.g., documentation content) is relatively stable. Details of our selected versions as presented in Column 2-4 of Table 1 (Column 5-6 present the release time of the first and last version of the subject used in our study).

[1]Our data for this study and the following bug study are both available at https://sites.google.com/site/incompemp2017/.

[2]For JDK and Android, we used the versions despite test failures and errors because some test cases require hardware support that we do not have. For JDK and Android versions, we ignore the test cases that fail on their own version.

3.1.3 Detection of BBIs. Not all test cases in the old version compile with the source code of the new version (typically because they suffer from signature incompatibilities). Therefore, to detect behavioral incompatibilities, for each version pair, we automatically recompiled the test code of previous version with the source code of the new version, and iteratively remove the test cases that do not compile with the new version of source code, until all test cases can be compiled successfully. Finally, we executed all the remaining test cases and collected test failures and errors. Specifically, test compilation errors appear in 37 versions from 12 subjects (except for BeanUtils, FileUpload, and Codec), and in total 2,590 of 57,208 test cases (4.5%) are removed due to compilations errors.

Since one BBI may causes multiple test failures and errors, we further manually inspected these test failures and errors and grouped them into BBIs. For each test error/failure, we extracted the error messages, the version diff of the failed test code, and the version diff of the test class and the source class being tested. Then, we categorize multiple test errors/failures as one BBI if the test errors/failures are caused by the same API method and result in the same error message, exception, or wrong value of the same output/side effect. Note that, among 296 backward incompatibilities, library developers have revised test cases (e.g., changing test oracles or ways of invocation) for 267 incompatibilities, and deleted test cases for the other 11. For the rest 18 incompatibilities, change was made in other places of the test suite such as revising the setting up code or upgrading referred libraries. We found that revised test cases can help a lot in understanding the behaviors of BBIs. This categorization was done by first 2 authors separately, with the third author as a judge for conflicts.

3.2 BBIs in Popular Libraries

To answer **RQ1**, we present the detected test failures / errors from software-library consecutive version pairs in Table 2. The first column of the table presents the subject name. Columns 2-3 present the total number of test failures detected in all version pairs of a specific subject (abbreviated as $T.$), and the number of versions where test failures are detected (denoted as I) divided by all version pairs of the subject (denoted as A). Columns 4-5 and 6-7 present similar data for test errors and BBIs (after grouping). Note that a test failure is raised when an assertion in the test case fails, while a test error is raised when the test case throws an unhandled exception or fails to complete. We also carefully checked the corresponding release notes, API documents, and migration guides (for Android) of the corresponding version pairs, and present the results in Column 8 of Table 2.

From Table 2, we make our observation as follows. Considering that cross-version testing may generate an under approximation of the number of BBIs, the prevalence of BBIs may be much higher than what is shown in the table.

Observation 1: BBIs between version pairs are prevalent among Java software libraries. We detect 296 BBIs in 14 of 15 subjects (93.3%), and 52 of 68 version pairs (76.5%). Averagely each version pair suffers from 4.4 BBIs, and only 82 of the 296 BBIs are documented

Table 2: BBIs in Software-Library Version Pairs

Subject	Failure		Error		B-Incomp.		
	T.	I/A	T.	I/A	T.	I/A	Doc.
OpenJDK	203	2/2	15	2/2	35	2/2	13
Android	112	2/2	11	2/2	56	2/2	20
log4j	21	2/2	0	0/2	4	2/2	1
maven	14	3/4	226	4/4	19	4/4	3
bukkit	15	2/6	31	3/6	7	4/6	0
beanutils	0	0/1	0	0/1	0	0/1	0
codec	4	1/1	6	1/1	6	1/1	3
fileupload	0	0/3	12	2/3	2	2/3	0
commons-io	4	1/4	2	1/4	3	2/4	0
ela. Search	36	4/7	98	3/7	24	4/7	5
http-core	60	5/6	15	4/6	32	5/6	10
jodatime	15	5/7	6	2/7	17	5/7	7
jsoup	54	9/10	2	1/10	36	9/10	3
neo4j	3	2/5	7	1/5	6	2/5	0
snakeyaml	108	8/8	14	4/8	49	8/8	17
Tot.	**649**	**46/68**	**445**	**28/68**	**296**	**52/68**	**82**

Table 3: Distribution of BBIs in Different Version Pairs

Subject	Total		Average		Incomp. V / All V	
	Mj.	Mn.	Mj.	Mn.	Mj.	Mn.
JDK	23	12	23	12	1/1	1/1
Android	56	N/A	28	N/A	2/2	N/A
log4j	3	1	3	1	1/1	1/1
maven	10	9	5	4.5	2/2	2/2
bukkit	3	4	0.8	2	3/4	1/2
beanutils	N/A	0	N/A	0	N/A	0/1
codec	6	N/A	6	N/A	1/1	N/A
fileupload	0	2	0	1	0/1	2/2
commons-io	3	N/A	0.8	N/A	2/4	N/A
ela.Search	0	24	0	6	0/3	4/4
http-core	15	17	5	5.7	2/3	3/3
jodatime	17	N/A	2.4	N/A	5/7	N/A
jsoup	31	5	4.4	1.2	7/7	3/4
neo4j	6	0	2	0	2/3	0/2
snakeyaml	49	N/A	4.9	N/A	10/10	N/A
Total	**222**	**74**	**4.7**	**3.5**	**36/47**	**16/21**

Distribution of BBIs between / within major versions. Beyond the overall status of backward incompatibilities between consecutive version pairs of software libraries, we further studied the difference between major and minor version pairs. The results are presented in Table 3. From Table 3, we have the observation as follows.

Observation 2: Major version pairs and minor version pairs suffered from 4.7 and 3.5 backward incompatibilities on average, respectively, and 76% of both types of version pairs are backward incompatible. Since BBIs are still prevalent within a major version, continuous minor version updates are still not safe, although they are slightly safer than major version upgrades.

3.3 Categorization of BBIs

To answer **RQ3**, we categorize BBIs according to their incompatible behaviors, invocation conditions, and the reasons why library developers brought them in.

For the categorization of incompatibilities from 3 aspects (behaviors, invocation constraints, and reasons), it is exploratory so we do not have a criterion available beforehand. We predefined high-level categories (e.g. return value change as a high-level category of incompatible behaviors). Then, first 2 authors went through the incompatibilities separately and classify them to the categories. They also annotate each BBI with labels made up by themselves. Then, all authors discussed and merged labels with similar meanings, and removed too-narrow labels to get the final set of finer-grained categories. If we found a finer-grained category cannot be put into

predefined high-level categories (e.g., Environment in Figure 2), we made them separate high-level categories. Due to the complexity of BBIs, the categorization process is not easy, especially when the BBI involves multiple API methods.

Consider Example 1, which is a test code sample from Jsoup 1.7.1. In the test case, an HTML document object is generated from HTML text, and then the document is printed out using `doc.body().html()` after setting char set to ascii and turn on escape mode. In Jsoup 1.7.3, the translation of some special characters for escape under ascii char set becomes different, so the printed HTML text will be changed. After inspection, we found that the change is made in method `html()`. Therefore, until `html()` is called, the memory status remains the same for both versions. In such a scenario, we determine that `html()` is the API method for the BBI, and the four API methods called before it are invocation constraint of the BBI.

Example 1 Identification of BBI-Related API Method

```
Document doc =
    Jsoup.parse("<p title=p> & < > ...");
doc.outputSettings().charset("ascii");
doc.outputSettings().escapeMode();
assertEquals("...", doc.body().html());
```

3.3.1 Incompatible Behaviors. From the 296 BBIs, we identified the following major categories of incompatible behaviors.

Exceptions and Crashes indicates that, in the new version of the software library, an API method throws exceptions in a different way. This category contains 4 sub-categories. *New Exception* indicates that the API method throws an exception in the new version but not in the old version. *Different Exception* indicates that the API method throws exceptions both versions, but the exceptions are different. *No Exception* indicates that, the API method throws an exception in the old version, but not in the new version. *Infinite Loop* indicates infinite loop in the new version.

Return Variable Change indicates that, the return value of an API method changes in the new version under certain usage scenario and input. Specifically, we divide this category into four sub-categories. *Value Change* indicates that a primitive value (e.g., integer, boolean, String) is changed. The BBI in Example 1 belongs to this category. *Field Change* indicates a field of the return object is changed. *Type Change* indicates that the actual type of the return value is changed, although the signature itself remain unchanged. This typically happens when the return type in the API method signature has many subtypes (e.g., `java.lang.Object`). *Structure Change* indicates that, no primitive values in the return object is changed, but the object is organized differently (i.e., values of reference-type fields or sub-fields of the return object are changed).

Other Effects indicates that an API method causes a different side effect on other parts of the software itself or the operating system, such as value changes of other variables in *Memory*, changes of the *GUI*, and *File System*.

The distribution of 296 BBIs is presented in Figure 1. From the figure, we can see that the categories of *Return Variable Change* and *Exception and Crash* account for 162 and 105 BBIs, respectively, and they combined to account for more than 90% of the BBIs. The two major subcategories for *Return Variable Change* are changes of primitive return value or field value changes of object-type return values, which account for more than 90% of the category. Such a distribution implies either most BBIs cause exceptions / simple return value changes, or such BBIs are more likely to be detected by regression testing. We will try to answer this question to some extent with our field bug study, but in either case, the following observation is true.

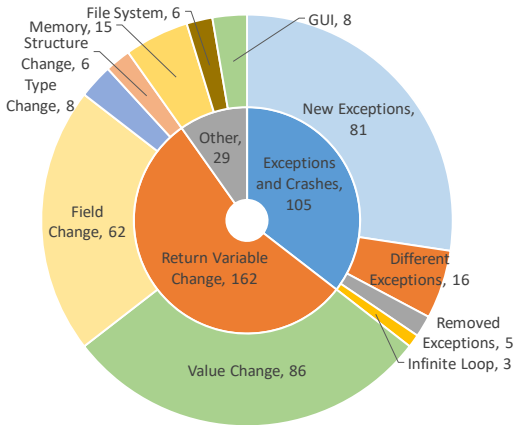

Figure 1: BBI Distribution on Incompatible Behaviors

Observation 3: Most BBIs detected by regression testing will cause either exceptions or value changes of the return variable or its fields, while side effects and environment effects are seldom detected.

3.3.2 Invocation Constraints. We further investigated the conditions under which BBIs can be invoked, and identified the following five major types of such conditions.

Always indicates that the BBI always happens as long as the corresponding API method is invoked.

Error indicates that the BBI happens only when an error happens. An example is the change of error message when a network error is invoked.

Environment indicates that the BBI happens only under certain environments of the application (e.g., operating systems, language settings).

Multiple APIs indicates that a number of other API methods must be invoked before the backward incompatible API method to invoke the BBI. Example 1 belongs to this category.

Input indicates that the BBI happens when a certain input value is fed into the corresponding API method. Specifically, we divide this category into five sub-categories. *Trivial Value* indicates a null pointer or an empty string / list as the input. *String Format* indicates that strings with specific structure as the input. *Special Field* indicates that objects with specific values at a certain field as the input. *Special Value* indicates that certain primitive values (not including strings) as the input. **Special Type** indicates the argument of the API method must be of a specific subtype of its parameter type.

The distribution of 296 backward incompatibilities in the above categories is presented in Figure 2. We can observe that, the top 3 categories of invocation conditions are *Always*, *Specific Value*, and

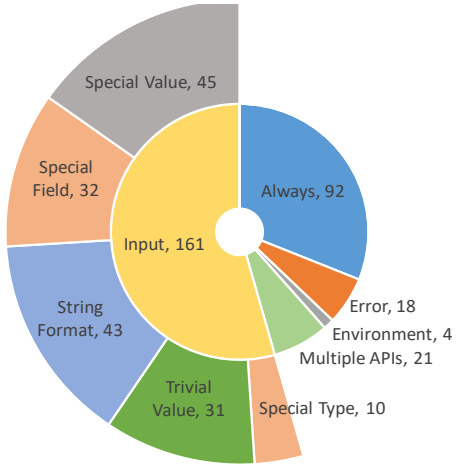

Figure 2: BBI Distribution on Invoking Conditions

String Format. The commonality among the 3 categories of BBIs is that they can be invoked with one API method invocation. By contrast, the BBIs requiring multiple API methods to invoke are not common in those detected by regression testing.

3.3.3 Reasons for Bringing in BBIs. Since the library developers revised the test cases to accommodate the changes, we can find out the reasons and the purposes of the behavior change from the revised test code. We identified the following 3 major categories of reasons, which contains 13 sub-categories.

Usage Change indicates that the library developers expect to keep the normal behavior of the library, but also expect client developers to change their usage pattern. This category has 4 sub-categories, which are *API Pattern Change* indicating changes on expected API sequences, *Enforce Rules* indicating rejecting of some poor input, *Enable Poor Inputs* indicating the support to some poor input, and *Input Format* indicating the change on the format of string type inputs.

Better Output indicates that the library developers expect to change the normal behavior of the API, while not expecting client developers to change their input. This category also has 5 sub-categories, which are *More Reasonable Output/Effect* indicating behavior change of an API method under a normal input to make it more reasonable[3], *Change Default Setting* indicating changing of the default setting (often a constant such as the default screen size), *Error Message* indicating change of reported errors, *Explicit Report* indicating explicitly throwing exceptions for errors, and *Output Format* indicating format change of string output.

Other indicates reasons not in the above 2 categories, including *Exposure of Internal Structure Change*, *Signature Change exposed with Reflection*, and *Upgrade Library*, in which the first two categories are regression faults.

The distribution of 296 backward incompatibilities in the above categories is presented in Figure 3. From the table, we can see the top 3 reasons for bringing incompatible behaviors are *More Reasonable Output/Effect*, *Output Format*, and *Enforce Rules*. We also find that, *Enable Poor Inputs*, the opposite of *Enforce Rules*, is the 4th popular

[3]For example, in log4j 2.0.2, the time stamp on the log is changed from when the log is initialized to when the time stamp line is printed.

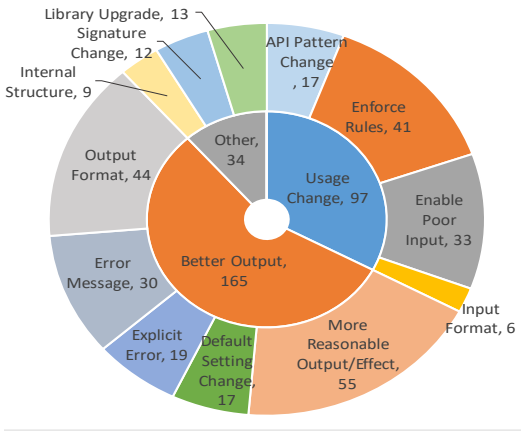

Figure 3: BBI Distribution on Reasons

reasons. Such contradiction reflects hesitation of library developers on whether responsibilities (e.g., input validation checking) should be at the library side or client side. Also, we did observe library developers moving back-and-forth on some incompatibilities. For example, in SnakeYaml, whether the dump output should include class name of the data value is changed 3 times through versions 1.3 to 1.6. Furthermore, although *More Reasonable Output/Effect* is the largest sub-category in *Better Output*, 93 of 165 BBIs in the category are not semantic changes, but presentation changes (e.g., error message, explicit exceptions, output formats). The 17 BBIs on change of default setting also related to client developers' preference. To sum up, we have the following observation.

Observation 4: Developers bring in a large portion of BBIs because they want to allow more or less inputs, or change the output presentation or option. This implies that the root cause of most BBIs is the different requirement of client developers, so the BBIs can be avoided if library developers understand requirement better (e.g., through surveys or code statistics on client projects), or have design for dual support.

4 REAL-WORLD BUG STUDY

In this section, to answer the **RQ4** and **RQ5**, we present the study result on real world bug reports related to BBIs, and explore how BBIs are affecting the client software developers. The basic information of the collected bug reports are presented in Table 4.

4.1 Collection of Bug Reports

To collect bug reports caused by BBIs. We searched two large on-line open bug repositories: JIRA and GitHub, on their project-specific bug report search engine. Specifically, we used as keywords the combination of terms related to software upgrading (e.g., "upgrade", "update", "version"), and the names of the software libraries listed in Table 1. From the collected bug reports, we *randomly* selected 500 bug reports, and carefully inspected these bug reports. During the inspection, we read the developers' comments and other references from the bug report to check whether they are confirmed by the developers to be caused by BBIs, and retain all the bug reports that are caused by BBIs.

Table 4: Basic Information of Bugs

Subject	Library Bugs	Client Bugs	Total
Java SDK	8	10	18
Android	13	64	77
Other	29	2	31
Total	50	76	126

We collected only bugs that are closed before Jan. 1st 2015 and never re-opened after that. We believe that these bugs are not likely to be re-opened so their status should be confirmed. We collect both bugs that are closed and fixed and bugs that are closed but not fixed due to developers' decision.

The reason is that, BBIs bugs are related to both software libraries and client software, so they can be fixed either at the library side or at the client side. Also, there are cases that a BBI bug is never fixed because the software library developers refuse to revert their changes, and the client developers did not find a way to work around it. In such cases, the developers may choose to not to support the new version of software library (not likely for runtime libraries such as OpenJDK and Android because not supporting the updated platform will cause dysfunction in client software), or have the users to tolerate the bug if the bug is relatively minor.

With the process above, we collected 126 bugs, and divided these bugs into two groups: *library bugs* that are submitted to software library projects that have BBIs, and *client bugs* that are submitted to software client projects because they triggers BBIs of their software libraries. The breakdown of collected bugs is shown in Table 4.

From the table, we can observe that, as we used a random selection of bugs, the majority of selected bug reports are from Android and Java SE. The reasons are two fold. First, Java SE and Android are much popular than other software libraries studied. Second, Java SE and Android are both runtime platforms, so that client software developers do not have control on which version of JVM and Android system their software will be executed on. Therefore, BBIs may be revealed after a Java or Android update at the users' side, and get reported to the client software developers. This also explains another observation that, the majority of bugs of Android and Java SE are client bugs, while the majority of bugs of other libraries are library bugs, because Android and Java BBIs are more likely to be reported by end users, while BBIs in other libraries are more likely to be reported by client software developers to library software as library bugs. Sampling from unbalanced bug sets is difficult. Random sampling will be ruled by dominating classes (e.g., JVM and Android). Giving quota to classes, samples will not reflect the actual data distribution, and proper quota size needs to be determined. We chose random sampling in our study to see the actual impact of BBIs in the field.

4.2 Study on Bug-Inducing BBIs

In this subsection, we categorize the bug-inducing BBIs and compare them to the BBI distribution in our library study. Specifically, among the 126 bugs, we find 13 bugs (12 client bugs from Android and 1 library bug from http-core) that can be directly mapped to the BBIs found in our library study. This shows that the test cases in regression testing are far from sufficient for identifying all bug-inducing BBIs, but it also implies that, if the regression testing results can be well documented or conveyed to client developers in better ways, some bugs can be avoided.

Table 5: Bug-Causing BBIs

Subject	Cause Library Bugs	Cause Client Bugs	Total
JDK	8	10	18
Android	13	51	64
Other	29	1	30
Total	50	62	112

Table 6: Categorization of Incompatible Behaviors

Behavior		Android		JDK		Other		All
		L	C	L	C	L	C	
Exception and Crash	New Exception	2	21	5	5	13	1	47
	Infinite Loop	0	0	1	0	0	0	1
Ret. Var. Change	Value Change	1	0	0	1	2	0	4
	Field Change	1	2	1	2	7	0	13
	Type Change	1	2	0	0	1	0	4
	Structure Change	0	0	0	1	3	0	4
Other Effects	Memory	3	4	0	0	1	0	8
	GUI	5	19	0	1	1	0	26
	File Sys.	0	2	0	0	1	0	3
	Sys. Event	0	1	1	0	0	0	2

Before we perform more in-depth investigation, we first manually scanned the bug reports to detect *cross-software duplicate bug reports*. Duplicate bug reports inside a project are typically labeled and we did not select them when collecting our bug report set. However, different client software may fail due to a same BBI of a same library, so these client bug reports are "duplicate" with each other. After the duplicate-bug-report detection, we identified 112 BBIs as presented in Table 5.

4.2.1 Incompatible Behaviors. The breakdown of bug-inducing BBIs per incompatible behaviors is presented in Table 6. Note that, in our bug study, we use the same categories as defined in Section 3.3. For incompatible behaviors, we found 2 BBIs that cannot be put into any defined categories, so we add a new category *Sys. Event*, which indicates changes in system events (Example 2).

Example 2 Bug-408: Be able to configure as a default SMS app in KitKat (from WhisperSystems/TextSecure)

```
In Android 4.4, SMS apps are no longer able
to send SMS to the SMS provider (rejected
silently) in the Android system, unless they
are reset to receive a broadcast
SMS_DELIVER_ACTION.
```

In the table, Columns 2-3 present the number of library bugs (denoted as L) and client bugs (denoted as C) from Android. Columns 4-7 present similar data for Java and Other subjects. Column 8 presents the total of each line. Since Android and JDK dominates our bug report study set, in the rest of the paper, when comparing bug study results and library study results, we provide both overall library study results, and library study results for JDK and Android combined. From Table 6, we have the following observations.

First, *New Exception* and *Infinite Loop* account for 48 of the 112 (43%) BBIs, which is higher than their proportion in the library study (28% overall, 30% for JDK+Android). These behaviors may be more likely to be reported as bugs due to their severity.

Second, *Value Change* accounts for 4 BBIs (3%), which is much smaller than its share in library study (29% overall, 18% for JDK+ Android). But *Field Change* accounts for 13 BBIs, which is much more than *Value Change* despite its lower share in library study.

Table 7: Categorization of Invocation Conditions

Conditions		Android		Java		Other		All
		L	C	L	C	L	C	
Always		5	14	2	1	2	0	24
Environment		0	1	0	0	1	0	2
Special Type		0	2	0	0	2	0	4
Multiple APIs		4	21	1	5	7	1	39
Input	Trivial Value	0	0	1	0	2	0	3
	String Format	2	2	3	3	6	0	16
	Specific Field	0	4	0	0	0	0	4
	Specific Value	2	7	1	1	9	0	20

Third, categories *Different Exception* and *No Exception* do not cause any bugs in our data set. This may be because client developers tend to avoid exceptions in their code so they are not affected. This also indicates that such changes are safer at the library side.

Example 3 Bug-46:Bold ZeroTopPaddingTextView displays cut off on 4.4 (from derekbrameyer/android-betterpickers)

```
In Android 4.4, a method that update the
padding setting must be invoked before
showing the date information, otherwise,
part of the date information can not be seen.
```

Fourth, GUI changes accounts for 26 BBIs (22.3%), which is the second largest category in bug study, but its share in library study is only 3% overall. The large number of GUI-change BBIs may be related to the large proportion of Android-related bugs in our bug set. Although these BBIs may be specific to Android framework, they are still important and representative because of the popularity of Android apps, and the existence of similar frameworks. We discovered that, most bug-inducing GUI BBIs are changing settings of UI controls and thus affecting presentation. Examples of the settings include the position to put notification bars (top or bottom), box widths, etc. Example 3 presents a GUI-related bug. The bug report is caused by a BBI between Android 4.3 and Android 4.4 about the changed value of padding settings. It should be noted that, user interface bugs are not just decoration problems, and they may largely affect software usages (i.e., information cannot be seen, or buttons go outside the screen and cannot be clicked). In general, we have the observation as follows.

Observation 5: Distribution of BBIs in library study and field bug study is mismatched on the incompatibility behavior categories. New Exception, GUI Changes, and other side-effects account for a higher proportion in field bug study, while other categories of BBIs account for a lower proportion.

4.2.2 Invocation Constraints. The breakdown of bug reports according to BBI-invoking conditions is presented in Table 7. From the table, we have the following observations.

First of all, 24 of 112 (21%) BBIs always happen (compared to 31% overall and 22% in JDK+Android in the library study), causing at least 15 client-side bugs. Second, only 2 of the BBIs are related to the environment. This may be largely due to the platform independence of Java, so we doubt whether this conclusion can be generalized to other programming languages. Third, 39 BBIs (35%) occur only after certain other API methods are invoked and thus belong to the category of *Multiple API*, compared to 7% overall and 11% JDK+Android in the library study. This actually implies that a lot of BBIs happen under special usage scenarios which are not covered by the library-side test code. In such cases, client code may be a good source to extract suitable test cases for detecting BBIs in a software library. Fourth, no BBIs fall into the *Error* category of invocation-condition, which shows that BBIs in this category are less likely to cause client bugs or the client bugs they cause are difficult to detect. In general, we have the observation as follows.

Table 8: Documentation Status of BBIs

Behavior		Android		Java		Other		All
		L	C	L	C	L	C	
No Doc.		12	43	7	6	29	1	98
Doc.	Release Notes	1	1	1	3	0	0	6
	JavaDoc	0	3	0	1	0	0	4
	Migration Guide	0	4	0	0	0	0	4

Observation 6: Distribution of BBIs in library study and field bug study is mismatched on invocation constraints. Multiple APIs account for a much higher proportion in field bug study, compared to its share in library study. This implies that regression testing may need to be strengthened for usage patterns involving multiple API methods.

4.3 Documentation Study

To answer the third research question, we further studied the documentation status of the bug-inducing BBIs. Since these BBIs are bug inducing, we predict that they may be more poorly documented then the BBIs detected from cross-version testing (34% documented), and the results shown in Table 8 confirm our guess. In Table 8, the first column presents the documentation status (and the place if documented). The rest columns are organized similar to Table 6.

From Table 8, we can see that bug-inducing behavioral BBIs are very poorly documented. Only 14 (13%) BBIs are documented. Also, the documented changes are relatively scattered, especially for Android (in release notes, JavaDocs, and Migration guides). Also, 8 client bugs of Android and 4 client bugs of Java are related to documented behavioral changes. This implies that we may need a better way than documentation to convey the information of behavioral change and remind client software developers about such changes.

4.4 Bug Resolution Study

To answer **RQ5**, we further studied how the real world bugs related to BBIs are resolved (note that they may be not fixed). Cross-software duplicate bugs may be fixed differently in different client software, so we view them as separate bugs in this subsection.

4.4.1 Resolution of Library Bugs. The breakdown of bugs according to how they are resolved is shown in Table 9. The first two columns present the types of resolution. If a library bug is fixed, we check whether it is fixed by a simple revert of the previous change, a patch of the previous change, or library developers decided to support both the previous behavior and the new behavior (typically by adding a parameter, and set either the previous behavior or the new behavior as default). If a library bug is not fixed, we study how library developers response to the bug report, and check whether it is intended behavior, or the developer is reporting a

Table 9: Resolution of Library Bugs

Resolution		Android	Java	Other	All
Fixed	Reverted	1	0	2	3
	Patched	6	4	16	26
	Double Support	0	0	1	1
Not Fixed	Intended	5	2	10	17
	Discouraged	1	2	0	3

behavioral change on internal APIs which should not be used by client developers.

From Table 9, we have the following observations. First, 20 of 50 library bugs are not fixed. The major reason is that the behavior change described in the bug report is intentional. It should be noted that, since these behaviors are reported as library bugs, they may already cause some bugs or at least test failures at the client side, although the client bug may not be reported. Second, among the 30 bugs that are fixed, most of them are patched, which shows that the many bug-inducing BBIs are caused by side effect of other productive changes.

4.4.2 Resolution of Client Bugs. The breakdown of client bugs according to how they are resolved is shown in Table 10. The first two columns present the types of resolution. If a client bug is fixed, we check whether it is fixed by (1) changing the incompatible API method to another one; (2) changing the arguments of the incompatible API method to other constants, variable, or expressions; (3) adding an API invocation to set a certain internal-state field before or after the invocation of the incompatible API method; (4) converting the return value of the incompatible API invocation to the original value; (5) a global structural code change; (6) updating libraries; (7) changing configuration of software; or (8) bypassing the incompatibility behavior by skipping software features (see Example 4).

Example 4 Bug-969: Android 5.0 crash when trying to open the app (from open-keychain/open-keychain)

```
The cause of the bug is that, "ResourceNotFound
Exception" is thrown in Android 5.0 when an
"overall scroll glow drawer" is requested.
In the fix, the client developers simply catch
the exception without doing anything with it.
Therefore, the request of "overall scroll glow
drawer" is actually bypassed.
```

For the client bugs that are not fixed, we discovered two resolutions. The first resolution is that the client developer simply decided to wait until a new version library is released. One reason of such resolution is that, the BBI is caused by a regression bug, so the client developer waits for the library developers to release a bug-free version. Another reason (and the major reason in our study) is that, the BBI affects a third-party library that the client developers are relying on. Since the client developers cannot change the code of the third-party library (sometime they even do not have access to the source code), they are not able to resolve the BBI, and have to wait for the new version of the third party library. The second solution is that, the client developer simply tolerate the behavior change (if the BBI does not cause crashes). They may simply ask their users to get used to the new behavior such as a UI change, or transfer the BBI to downstream developers.

Table 10: Resolution of Client Bugs

Resolution		Android	Java	Other	All
Fixed	Change API	1	2	0	3
	Change Input	13	0	1	14
	Add Set Field	17	1	0	18
	Return Convert	6	0	0	6
	Structural	8	5	0	13
	Config	2	0	0	2
	Lib. Update	2	0	0	2
	Bypass	4	0	0	4
Not Fixed	Wait Lib. Fix	4	2	0	6
	Tolerate	7	0	1	8

Table 9 shows that 14 client bugs are not fixed. Note that, we find that most developers are willing to and have tried to fix the bugs, but BBI-related bugs are more difficult to fix, because they typically involve code written by other people. Regarding the fixed client bugs, we have the observation as follows.

Observation 7: 41 of the 62 fixed client bugs are fixed through small changes including changing API, changing input value, add an API to set field, or convert the return value to the original value.

We present an example of client bug fixing with "Add an API to set field" in Example 5. The corresponding BBI is that, for Android 5.0, if the developer wants to start a service with an intent, the type of the intent must be explicitly set with the method `setClass(...)`, and the method `startService(...)` will check the `class` field of the intent and throw exceptions if the value is null.

Example 5 Fix: Bug-812: Lollipop notification settings won't work (from klassm/andFHEM)

```
 Intent intent = new Intent(
 Actions.NOTIFICATION_SET_FOR_DEVICE);
+intent.setClass(context,
     NotificationIntentService.class);
 intent.putExtra(BundleExtraKeys.DEVICE_NAME
     , deviceName);
 ...
 context.startService(intent);
```

4.4.3 Reporting to Library Developers. In our study, we further studied whether client developers would like to report their bugs to the library developers. Among the 76 client bug reports, we find that the symptom is reported to library developers in only 6 bug reports. In most of the cases, the developers simply search through the Internet to find a workaround. Also, for the 6 reported bugs, only 3 are fixed by the library developers, while the other 3 are rejected because the corresponding BBIs are intended behaviors.

5 DISCUSSION

In this section, we discuss the lessons learned, limitations and the threats to our study, and regression faults.

5.1 Lessons Learned

The final goal of our study is to find actionable goals for library / client developers to avoid or resolve bugs caused by behavioral backward incompatibilities.

5.1.1 Avoidance of BBI Bugs. **Enforcing Old Tests on Release.** In our study, we are surprised by the large number of BBIs found with simple cross-version testing. Note that all tests come with the project and developers are supposed to run them every time they build the code. Our study shows that, most tests detecting BBIs are changed accordingly or deleted when BBIs were brought in, so they can never detect the BBIs. Therefore, we suggest to enforce testing with old tests when releasing a new version. Such a feature can be added to IDEs or version control systems. It is unnecessary that all old tests pass but developers should provide document or workaround for failed tests, especially on minor version releases.

Augmenting Regression Tests. Our results in Figure 1 shows that, although 105 BBIs cause different exception / crash status and 86 BBIs cause primitive return value changes, the remaining 105 of 296 BBIs (36%) cause memory state change (e.g., certain field of the returned object) or other side effects (e.g., GUI, file systems). Such BBIs can be revealed only with extra API calls or side-effect checking. Furthermore, Table 1 shows that such BBIs cause 60 of 112 (54%) studied BBI-related bugs. Some side effects, such as file and UI change can be difficult to detect using normal assertions. Augmented regression tests with more advanced memory revealing assertions will detect more bug-inducing BBIs, and automatic test augmentation techniques such as Ostra [37] may be helpful.

BBI Recommendation. Our study reveals mismatches between the distribution of BBIs in various categories and the corresponding distribution of BBI-related bugs, which shows that certain categories of BBIs are more likely to result in bugs. For example, GUI changes as incompatible behaviors and multiple APIs as invocation conditions has a much higher population in the studied bug-related BBIs, and none of studied bugs are caused by BBIs related to different exceptions or changed error messages. Also, APIs that are frequently used, once have BBIs, are more likely to result in multiple client-side bugs, which is also observed in our study. Furthermore, our study also observed contradictory reasons of behavioral changes (e.g., allowing lousy input and enforce input rules) and several reverted or double-supported BBIs, which shows that developers are not always clear about the consequences of involving BBIs. Therefore, based on the category and API-usage frequency information (together with other factors such as major or minor version released, development status, etc.), a recommendation system can be helpful for developers when they make decisions on involving a BBI in a release.

Test Change Tracking. Our study shows that, for 267 of 296 BBIs (90.2%), developers change their test code to accommodate the behavior change. Therefore, change of test code on public APIs can be a sign of BBIs, and tracking test code changes may provide more information about the happening and evolution of BBIs.

5.1.2 Detection and Resolution of BBI-Related Bugs. **BBI Notification.** Our study shows that the documentation status of behavioral incompatibilities is very poor. Even when a behavioral change is documented, there are still many relevant client bugs. We believe that, advanced techniques on documentation of behavioral incompatibilities is in a great need and will help reduce many bugs related to backward incompatibilities. The technique should be able to directly check the client code (e.g., finding code clones of a failed library-side test case) and raise warnings about potential relevant behavioral backward incompatibilities.

Advanced GUI Testing. We find that, GUI behavior change is one of the major cause of BBI-related client bugs. Many of such bugs cannot be easily detected by normal assertions, such as the bugs on text invisibility due to color and size change of UI controls (Example 3). Some BBI bugs can be detected only with human eyes. Automatic oracle checking is straightforward for unit regression testing, but hard for GUI regression testing. This calls for more advanced user-interface checking techniques to support automatic regression testing of GUI applications.

Test Code Reference. Since developers change their test code to accommodate BBIs, the test code change can be used as examples of how to work around BBIs. When using certain API methods, client developers may consider watching the library-side test-code changes, or searching for relevant test-code changes for workarounds when resolving the BBIs from client side. The resource of test code changes can also be used in documentation, BBI notifications, or automatic BBI resolution tools for the client side.

Automatic Fixing of Client Bugs. Our study shows that, 67% bugs (41 of 62) fixed from client are based on simple changes such as replacing the input arguments, converting the return values, and add an invocation to a certain field-setting API method before or after the BBI API invocation. Consider Example 5, where an Intent objectâĂŹs field needs to be set before it is used. In many such BBIs, a validation is added in library code to check the field (e.g., checking class field of Intent for accessible classes) and an exception is thrown there. The existence of such patterns shows possibilities that many BBI-related bugs can be fixed automatically by adding new change rules to automatic bug fixing tools.

5.2 Limitations and Threats

Limitations. First, we use cross-version testing to detect BBIs in software libraries. This result in an under-estimation of the number of BBIs between version pairs. Also, since we require all test cases pass in their original versions, the detected BBIs are biased to the intended behavioral changes (since the relevant test cases are already fixed). However, the major goal of our study on regression testing is to show the prevalence of BBIs, and we believe the above mentioned limitations do not affect our conclusion. Second, when studying BBI-related bugs, we searched the bug repositories with keywords such as "update". Bugs related to BBIs do not have obvious keywords, such as "deadlock" for concurrency bugs. In particular, some BBI-related bugs may stay in the software for a long time, and the client developers may not realize the root cause of the bug even after it is fixed. Thus, our selected bugs may be biased to those bugs that are easily found to be relevant to BBIs.

Threats to Validity. The major threats to internal validity of our study is the potential errors and mistakes in the process of building software and performing regression testing, studying the bugs, and doing the statistics. To reduce this threat, we carefully wrote all the tools we used, and manually checked the results for correctness. The major threats to external validity is that, our conclusion may hold for only Java software libraries, and the libraries under study. Furthermore, our conclusion may hold for only the bugs studied. Since our bug dataset is dominated by Android and Java, some conclusions (e.g., about GUI) may be specific to these libraries. To reduce this threat, we chose the most popular Java software libraries, as well as randomly chose the bugs to be studied.

5.3 Detailed Classification of BBIs

Intention of Behavior Changes. In our paper, we do not differentiate regression bugs from other BBIs and treat them exactly the same. Our second study shows that, both regression bugs and intentional BBIs cause real-world client bugs. Also, it is very hard to tell whether a behavioral change is intentional because an intentional behavioral change can have unexpected side effects.

Contract-based Classification of Behaviors. In our paper, we manually categorized incompatible behaviors and invocation constraints in an intuitive way. In future, we plan to apply analysis tools to categorize incompatibilities in a more formal manner. Specifically, we plan to categorize incompatibilities to precondition violations where the new upgraded function requires more from its inputs than the old one(e.g., an non-null input is required) and post-condition violations where the return values of the new function do not subsume the old ones, or the new function throws different exceptions or has different side effects.

6 RELATED WORK

Our work is related to studies on software library evolution, summarization of library changes, and migration for library evolution.

6.1 Studies on Software Libraries Evolution

Researchers have noticed that software libraries are evolving frequently for various reasons such as bug fixing [24], adding features [11], and better UIăĂĂsupport [32] [31]. A number of studies have been conducted on the evolution of software libraries. Dig and Johnson [9] carried out an empirical study on how developers refactor API methods. Raemaekers et al. [27] proposed a measurement of software-library stability which considers API method difference and code difference, and studied the stability of 140 industrial Java systems based on the measurement. McDonnell et al. [19] studied the stability of Android APIs (in terms of added and removed classes and methods), and the time lag between the release of API changes and the corresponding adaptation at the client software side. Espinha et al. [10] interviewed 6 web client software developers and conducted an empirical study on four widely used web services to understand their API evolution trends, including the frequency of API changes, and the time given client developers to upgrade to the new version of services. Bavota et al. [4, 5], studied the evolution of software dependency upgrades in the apache software ecosystem. Robbes et al. [29] studied the reaction of developers to deprecated APIs in SmallTalk ecosystem. Wu et al. [35, 36] studied the evolution of API usages in large software ecosystem. Brito et al. [7]'s empirical study shows that deprecation message is not helpful for developers. The existing research efforts mainly focus on signature-level API changes (Raemaekers et al.'s work further considers the amount of code difference) to measure API changes and stability. By contrast, our study focuses on behavioral changes of software libraries, which are more difficult to be detected and may cause more severe consequences.

There have also been a number of research efforts on the impact of software-library to client software. Linares-VĂąsquez et al. [16] studied the relationship between change proneness of APIs methods and the successfulness of client software that uses those API methods. Bavota et al. [6] further extends the work with more detailed experimental results. These research efforts shows that backward incompatibilities have much effect on the successfulness of client software, and thus motivate the study in our paper.

6.2 Summarizing Software Library Changes

To provide more information about the changes on API methods, there have also been research efforts trying to summarize changes between two consecutive versions of a software library. On the signature level, Wu et al. [33] proposed AUCA, an auditor for API changes, that reports a large variety of signature-level changes of APIs. Tang et al. [30] proposed to use tree adjoining language to summarize dependency relationships in libraries. Moreno et al. [21] proposed ARENA, an automatic tool to summarize software-library changes and generate release notes.

On the behavior level, McCamant and Ernst [17, 18] proposed to represent behavior API methods with program invariants generated with Daikon. Person et al. [25, 26] proposed differential symbolic execution to summarize as symbolic expressions of inputs the semantic difference between two versions of a method. Lahiri et. al [15] proposed SymDiff, a tool that leverages a modularized approach to check semantic equivalence of different code versions, and calculate program paths that can reveal code behavioral difference.

6.3 Support for Library Migration

Godfrey and Zou [12] proposed a number of heuristics based on text similarity, call dependency, and other code metrics, to infer evolution rules of software libraries. Later on, S. Kim et al. [14] further improved their approach to achieve fully automation. M. Kim et al. [13] inspected existing framework evolution process to gather a number name-changing patterns and used these patterns to infer rules of framework evolution. Dagenais and Robillard [8] proposed *SemDiff*, which infers rules of framework evolution via analyzing and mining the code changes in the software library itself. Wu et al. developed *AURA* [34], which further involves multiple rounds of iteration applying call-dependency and text-similarity based heuristics on the code of software library itself. Most recently, Meng et al. [20] proposed *Hima*, which further enhances *AURA* by involving information from comments of code commits between two consecutive versions of software libraries.

7 CONCLUSION

In this paper, we present a study on behavioral backward incompatibilities based on regression testing of 68 version pairs of 15 Java software libraries, and inspection of 126 real world bugs. From our study, we find that behavioral backward incompatibilities are prevalent among Java software libraries, and caused most of real-world backward-incompatibility bugs. Furthermore, many of the behavioral backward incompatibilities are intentional, but are rarely well documented. We also categorized behavioral backward incompatibilities according to incompatible behaviors and invocation conditions, and found category mismatches between the the BBIs detected in regression testing and the BBIs causing bugs.

ACKNOWLEDGMENT

The research presented in this paper is supported in part by NSF grant CCF-1464425, and DHS grant DHS-14-ST-062-001.

REFERENCES

[1] Semantic Versioning, http://semver.org/. Accessed: 2016-08-22.

[2] 2017. Criticism of Windows Vista. (2017). https://en.wikipedia.org/wiki/Criticism_of_Windows_Vista

[3] 2017. Sougou. (2017). https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en

[4] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. 280–289.

[5] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. 2014. How the Apache Community Upgrades Dependencies: an Evolutionary Study. *Empirical Software Engineering* (2014), 1–43.

[6] G. Bavota, M. Linares-Vasquez, C. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 2014. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *Software Engineering, IEEE Transactions on* 99 (2014), 1–1.

[7] G. Brito, A. Hora, M. T. Valente, and R. Robbes. 2016. Do Developers Deprecate APIs with Replacement Messages? A Large-Scale Analysis on Java Systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 360–369.

[8] B. Dagenais and M. P. Robillard. 2008. Recommending Adaptive Changes for Framework Evolution. In *Proceedings of International Conference on Software Engineering*. 481–490.

[9] D. Dig and R. Johnson. 2006. How Do APIs Evolve&Quest; A Story of Refactoring: Research Articles. *J. Softw. Maint. Evol.* 18, 2 (March 2006), 83–107.

[10] T. Espinha, A. Zaidman, and H.-G. Gross. 2014. Web API Growing Pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*. 84–93.

[11] M. W. Godfrey and Q. Tu. 2000. Evolution in Open Source Software: a Case Study. In *Proceedings 2000 International Conference on Software Maintenance*. 131–142.

[12] M. W. Godfrey and L. Zou. 2005. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering* 31, 2 (February 2005), 166–181.

[13] M. Kim, D. Notkin, and D. Grossman. 2007. Automatic Inference of Structural Changes for Matching across Program Versions. In *Proceedings of International Conference on Software Engineering*. 333–343.

[14] S. Kim, K. Pan, and E. J. Whitehead, Jr. 2005. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *Proceedings of Working Conference on Requirement Engineering*. 143–152.

[15] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. RebÃłlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification*. 712–717.

[16] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. 477–487.

[17] S. McCamant and M. D. Ernst. 2003. Predicting Problems Caused by Component Upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 287–296.

[18] S. McCamant and M. D. Ernst. 2004. Early Identification of Incompatibilities in Multi-component Upgrades. In *European Conference on Object-Oriented Programming*. 440–464.

[19] T. McDonnell, B. Ray, and M. Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. 70–79.

[20] S. Meng, X. Wang, L. Zhang, and H. Mei. 2012. A History-based Matching Approach to Identification of Framework Evolution. In *Proceedings of the 34th International Conference on Software Engineering*. 353–363.

[21] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora. 2014. Automatic Generation of Release Notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 484–495.

[22] S. Mostafa and X. Wang. 2014. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *2014 14th International Conference on Quality Software*. 127–132.

[23] S. Mostafa and X. Wang. 2014. A Statistics on Usage of Java Libraries. In *Technical Report*. http://xywang.100871.net/TechReport_LibStats.pdf

[24] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. 2002. Evolution Patterns of Open-source Software Systems and Communities. In *Proceedings of the International Workshop on Principles of Software Evolution*. 76–85.

[25] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. 2008. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 226–237.

[26] S. Person, G. Yang, N. Rungta, and S. Khurshid. 2011. Directed Incremental Symbolic Execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 504–515.

[27] S. Raemaekers, A. van Deursen, and J. Visser. 2012. Measuring Software Library Stability through Historical Version Analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. 378–387.

[28] S. Raemaekers, A. van Deursen, and J. Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. 215–224.

[29] R. Robbes, M. Lungu, and D. Röthlisberger. 2012. How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 56:1–56:11.

[30] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. 2015. Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 83–95.

[31] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. 2010. Locating Need-to-translate Constant Strings in Web Applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 87–96.

[32] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. 2012. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 16:1–16:11.

[33] W. Wu, B. Adams, Y.-G. Gueheneuc, and G. Antoniol. 2014. ACUA: API Change and Usage Auditor. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. 89–94.

[34] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim. 2010. AURA: A Hybrid Approach to Identify Framework Evolution. In *Proceedings of International Conference on Software Engineering*. 325–334.

[35] W. Wu, F. Khomh, B. Adams, Y. Guéhéneuc, and G. Antoniol. 2016. An Exploratory Study of API Changes and Usages based on Apache and Eclipse Ecosystems. *Empirical Software Engineering* 21, 6 (2016), 2366–2412.

[36] W. Wu, A. Serveaux, Y. Guéhéneuc, and G. Antoniol. 2015. The Impact of Imperfect Change Rules on Framework API Evolution Identification: an Empirical Study. *Empirical Software Engineering* 20, 4 (2015), 1126–1158.

[37] T. Xie. 2006. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In *Proc. 20th European Conference on Object-Oriented Programming (ECOOP 2006)*. 380–403.