Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level

Bo Yang, Jinqiu Yang
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
b_yang20, jinqiuy@encs.concordia.ca

***Abstract*—Test-based automated program repair techniques use test cases to validate the correctness of automatically-generated patches. However, insufficient test cases lead to the generation of incorrect patches, i.e., passing all the test cases, however are incorrect. In this work, we present an exploratory study to understand what are the runtime behaviours are being modified by automatically-generated plausible patches, and how such modifications of runtime behaviours are different from those by correct patches. We utilized an off-the-shelf invariant generation tool to infer an abstraction of runtime behaviours and computed the modified runtime behaviours at the abstraction level. Our exploratory study shows that majority of the studied plausible patches (92/96) expose different modifications of runtime behaviours (i.e., captured by the invariant generation tool), compared to correct patches.**

***Index Terms*—automated program repair, patch correctness, empirical study**

I. Introduction

Over the last decade, various automated program repair (APR) techniques [1, 2, 3, 4, 5, 6] are proposed to automatically generate bug fixes at source-code level. APR techniques aim to save tremendous developers' time spent in bug fixing activities, and the valuable efforts can be diverted to other important development tasks, such as new features and performance optimization. There has been an increasing attention in industry to to adopt APR techniques in practice [7, 8].

Among the various types of APR techniques, e.g., static analyzer-based [9, 10, 7] and bug-report-based [11, 12], test-based APR remains as one of the mainstream APR techniques. Test-based APRs utilize test cases to guide the repair process and also to validate the correctness of the machine-generated patches. If a machine-generated patch makes all the test cases pass, test-based APRs techniques will present the patch to developers, and terminate the repair process. However, due to insufficient test cases [13], test-based APR techniques may generate incorrect patches, i.e., patches that fail to fix the software bugs however make all the test cases pass. Following previous work, we refer to the incorrect machine-generated patches as *plausible* patches.

Plausible machine-generated patches hinder the performance of test-based APR techniques and negatively affect the adoption of APR techniques in practice. Previous efforts to identify plausible patches include applying ranking algorithms to de-prioritize the plausible patches [14], using new tests [15] and improved test oracles [16] to filter out plausible patches, and integrating improved validation in repair process [17]. Plausible machine-generated patches appear to have the "*desired*" impacts on program executions that are identical to correct patches, i.e., making all test cases pass. However, differences begin to surface if program executions are compared in detail. For example, a recent study performed by Xiong et al. [15] shows that simple heuristics based on test coverage, i.e., complete-path spectrum [18], can be used to differentiate plausible patches from correct patches.

In this work, we explore the direction of unveiling the differences between plausible and correct patches based on *their different impacts on runtime behaviours*. In particular, we utilize *program invariants* to abstract runtime behaviours and represent modified runtime behaviours using *changes on program invariants*. Program invariants are dynamically concluded to describe observed run-time behaviours during test case executions. Program invariants have been shown to reveal different aspects of program executions compared to coverage metrics. For example, Harder et al. [19] showed that utilizing dynamically-generated invariants is better for certain types of bugs compared to coverage metrics when being applied to minimize test suites. Recently, Cashin et al. [20] propose to use invariants to cluster machine-generated patches for better understandability since invariants capture semantic level of information.

In this study, we leverage an off-the-shelf invariant generation tool Daikon [21], to infer program invariants. We performed an empirical study to understand how dynamically-generated invariants (*invariants* for short) play a role in the context of automated program repair. We answer the following research questions (RQs) by analyzing a total of 73 bugs and their corresponding 169 distinct patches (96 plausible machine-generated patches and 73 correct patches). The bugs are from Defect4J dataset and the machine-generated patches are collected by prior work [15]

- **RQ1: What are the impacts on program invariants by a correct patch?**
  This RQ sets a baseline of how a correct patch may modify program runtime behaviours, i.e., represented by program invariants. One may expect that program runtime behaviours during passing test cases may not be affected too much by a correct patch since bug fixes should not affect correct program behaviours. On the contrary, runtime behaviours that are concluded based on failing

test cases, especially unexpected program behaviours, may be significantly modified by a correct patch.

- **RQ2: What are the impacts on program invariants by a plausible patch?**
  Previous studies [13, 16] show that a plausible patch may fail to fix the bug completely, and even break existing functionalities, i.e., passing test cases fail to identify the newly-introduced incorrect behaviours. This RQ investigates to what extent, runtime behaviours are being modified by plausible patches in both passing and failing test cases.
- **RQ3: What are the scopes of invariants impacted by test-passing patches?**
  Invariants are inferred at each program point. Among all the program points tracked and analyzed, only a portion of them are affected by test-passing patches, i.e., including both correct and plausible ones. Patches may only affect the invariants of the program points that are modified by the patches, or may spread out the entire execution (i.e., covered program points). A low impact scope indicates that we can be more selective in monitoring certain program points in invariant generation. Instrumenting and summarizing at fewer program points can significantly speed up the process.
- **RQ4: How often one plausible patch may affect the inferred invariants differently compared to the corresponding correct patch?**
  If plausible patches commonly modify program runtime behaviours in a different way compared to correct patches, it is promising to derive techniques to automatically identifying plausible machine-generated patches based on the inferred abstractions of runtime behaviours.

We position this paper as a preliminary study towards understanding plausible patches by APR techniques using program invariant. In this work, we present quantitative results to answer the four above-mentioned RQs.

II. An Illustration Example

We will use a real bug example from Chart to illustrate how a plausible patch will affect program behaviours and how such impacts are captured by program invariants.

Figure 1 shows a simplified class of the bug Chart-3 in Defect4J. The buggy method is `createCopy` (line 28-45), which will create a new object `TimeSeries` with part of `data` field (declared in line 3) with a specified range defined by `start` and `end`. The root cause is that *maxY* and *minY* which are the maximum and minimum values of the *data* field, may not be correctly updated in the clone process. First, calling the `clone` method in `java.lang.Object` (line 32) will also copy the original values of *maxY* and *minY* to the new object. Then, *minY* and *maxY* will be updated when calling `add` in line 41. The method `add` will update *minY* and *maxY* by calling `updateBoundsForAddedItem` (line 13). However, to make `updateBoundsForAddedItem` to function properly, the initial values of *minY* and *maxY* should be `Double.NaN`, as shown in the correct patch (line 33-34).

```
1  *public class TimeSeries {
2  ...//Below we show relevant class fields
3     protected List data;
4     private double minY;
5     private double maxY;
6  ...
7     public void add(TimeSeriesDataItem item, boolean
           notify) {
8        ...
9        boolean added = false;
10    //Add the item, if successful, added will be true
11    ...
12       if (added) {
13          updateBoundsForAddedItem(item);//see below
14          ...
15          removeAgedItems(false);//see below
16          ...
17       }
18    }
19    public void removeAgedItems(boolean notify) {
20       if (getItemCount() > 1) {
21          if (...) {//deciding whether or not removing
                aged items
22             removed = true;
23          a}
24 -        if (removed) {
25 +
        if(org.jfree.data.time.TimeSeries.this.data!=null)
        { // incorrect patch
26             findBoundsByIteration();
27          }
28    }
29    public TimeSeries createCopy(int start, int end)
           throws CloneNotSupportedException {
30       ...
31       TimeSeries copy = (TimeSeries) super.clone();
32 +     copy.minY = Double.NaN; //correct patch
33 +     copy.maxY = Double.NaN;
34       copy.data = new java.util.ArrayList();
35       if (this.data.size() > 0) {
36          for (int index = start; index <= end; index++) {
37             TimeSeriesDataItem item =
                    (TimeSeriesDataItem)
                    this.data.get(index);
38             TimeSeriesDataItem clone =
                    (TimeSeriesDataItem) item.clone();
39             try {
40                copy.add(clone);
41                ...
42             }
43          }
44          return copy;
45       }
46    }
47    private void
          updateBoundsForAddedItem(TimeSeriesDataItem
          item) {
48    ...//This method updates this.minY and this.maxY
          according to the parameter item. It does not
          reset this.minY and this.maxY the same way as
          line 52-53.
49    }
50    private void findBoundsByIteration() {
51    //The two lines below are identical to the correct
          patch in line 33-34.
52       this.minY = Double.NaN;
53       this.maxY = Double.NaN;
54
55       Iterator iterator = this.data.iterator();
56       while (iterator.hasNext()) {
57          TimeSeriesDataItem item = (TimeSeriesDataItem)
                iterator.next();
58          updateBoundsForAddedItem(item);
59       }
60    }
61 }
```

Fig. 1: The buggy class `TimeSeries` of Chart-3 in Defect4J

```
1  *public void testCreateCopy3() {
2      TimeSeries s1 = new TimeSeries("S1");
3      s1.add(new Year(2009), 100.0);
4      s1.add(new Year(2010), 101.0);
5      s1.add(new Year(2011), 102.0);
6
7      TimeSeries s2 = s1.createCopy(0, 1); //Create a copy
           with the data added in line 3-4.
8      assertEquals(100.0, s2.getMinY(), EPSILON);
9      assertEquals(101.0, s2.getMaxY(), EPSILON); // This
           assertion fails, s2.getMaxY() equals to 102.0
10 }
```

Fig. 2: The failing test case of bug Chart-3.

In the buggy version, when being copied, *minY* and *maxY* are assigned with values in double.

The correct patch is to correctly assign initial values to *maxY* and *minY* (line 33-34). A plausible machine-generated patch (line 23-24) modifies a condition in `removeAgedItems` to be always true so that `findBoundsByIteration` will **always** be executed. Hence, the plausible patch will have *similar* effects as the correct patch in some cases. However, the plausible patch and correct patch still result three differences. We will use the test case shown in Figure 2 to explain the three differences. In the test case, three data instances are added (line 3-5). A copy of `TimeSeries` is created with the data added in line 3-4. *maxY* in line 9 should be 101.0 instead of 102., and this causes the assertion to fail.

The first difference is whether the values of *minY* and *maxY* are correct before the second execution of `add` in line 41. Although the plausible patch modifies the condition in line 24 to be always true, whether line 26 (i.e., the key to make the failing test case pass) will be executed or not still depends on the condition in line 20 (i.e., *false* when only zero or one data is added). Since there are totally two executions of `add` in this test case, the program invariants inferred at entering add can summarize the two executions. In fact, in both the buggy and incorrectly-patched version, the invariant "this.maxY == 102.0" is inferred at the program point of entering the method `add`. However, this invariant represents an incorrect specification since the correct value of *this.maxY* should be 101.0. As a result, from the correctly-patched version, a different invariant is inferred (i.e., "this.maxY is one of {100.0, 101.0, Double.NaN}) for the same program point.

The second difference is for the given test case in Figure 2, whether or not executing `removedAgedItems` will update *minY* and *maxY*. In both the buggy and correctly-patched versions, this update will *not* happen in `removedAgedItems`: there is no update in the buggy version, and the update will happen in `updateBoundsForAddedItem`. Differently, the plausible patch performs the updates in `removedAgedItems`. This difference is reflected by the different invariants generated at the program point of existing method `removeAgedItems`. In the buggy version and correctly-patched version, there are invariants such as "this.maxY == orig(this.maxY)". However in the incorrectly-patched version, this invariant is violated and a different but relevant invariant is inferred "this.maxY <= orig(this.maxY)". The pair of the violated and newly-inferred invariants indicate that *this.maxY* has been updated in the incorrectly-patched version of `removeAgedItems`.

The third difference is that extra computations of `findBoundsByIteration` (line 26) due to the change of the condition (line 25-26) by the plausible patch. However, this is not reflected by any modified invariants.

In summary, although the plausible patch appears to have similar behaviours to the correct patch (i.e., making the failing test case pass), their impacts on runtime behaviours are different. More importantly, such modified runtime behaviours can be captured by abstractions of runtime behaviours such as inferred program invariants.

TABLE I: Summary of the Studied Dataset

Project	Bugs	Correct Patches	Plausible Patches
Lang	12	12	13
Math	45	45	62
Chart	14	14	20
Mockito	2	2	1
Total	73	73	96

III. Experiment Method

In this section, we describe the dataset that we analyzed in this empirical study, the method we followed to conduct the experiments including applying Daikon [21] on the patched software and computing the differences in the generated invariants.

Data Collection. We conduct this study using bugs from Defect4J dataset [22]. Defect4J is widely used for benchmarking automated program repair techniques and mutation testing techniques. Note that our study does not include every bug in Defect4J, but only the bugs that there exists at least one machine-generated patch from one of the six test-based APR techniques. We use the set of machine-generated patches collected by Xiong et al. [15]. We excluded the bugs from *Closure* project because running Daikon on Closure test cases constantly produces time out errors.

In total, our study includes 73 bugs from four open-source projects, 73 correct patches (either by APRs or by developers), 96 plausible patches from one of the six APR techniques, i.e., jGenProg [23], jKali [13], two versions of Nopol [24], HDRepair [25], ACS [3]. Our study focuses on comparing one plausible machine-generated patch and its corresponding correct patch of the same bug. Therefore we only perform the study on the bugs where at least one plausible patch is generated by one of six APR techniques. We include both the correct patches from developers and from APRs, therefore each bug in our study has at least one correct patch. We manually examine all the patches and remove the duplicate ones. The remaining patches are all semantically distinct. There exist some plausible patches that we are not able to

reproduce in our experiment, i.e., seven patches from APRs cannot make all test cases pass. In the current study we excluded them from this study.

Table I shows the details on the dataset in our study. In total our study contains 73 bugs from Defect4J, 73 correct patches from both APRs techniques and developers, and 96 distinct plausible patches from APRs techniques.

Invariant Generation. We utilize Daikon [21] to infer program invariants in our study, which is a widely adopted invariant generation engine. Daikon instruments a program, record runtime data when test cases are being executed, and infer program invariants at certain program points, such as entering and exiting methods. As we expect that patches (either incorrect or correct) may impact runtime behaviours of passing and failing test cases differently, we applied Daikon to generate invariants for passing and failing test cases separately. However, it is possible that passing test cases may contain erroneous program executions, and therefore should be grouped with failing test cases for invariant generation. The erroneous executions in passing test cases are not revealed due to insufficient test cases. As future work, we plan to examine the passing test cases and explore methods to identify erroneous program executions from passing test cases.

Running daikon can be extremely time-consuming. To speed up the process and reduce unnecessary computation, we leverage test coverage to select a subset of test cases that *likely* expose different invariants caused by patches. In particular, if a test case covers the code modified by a patch, this test case will be selected for inferring invariants.

Differencing the Inferred Invariants Between Versions. To quantify the affected runtime behaviours by a patch, we calculated the differences of the generated invariants between two versions, i.e., a buggy and a patched version. The patched version is either a correct patch or a plausible patch, both of which can make all the test cases pass. Differencing invariants is based on differencing the invariants of each identified program point. Examples of program points include procedure entries and exits, and accessing static variables of a class. In this study, we use a simple differencing method to compare the inferred invariants of two versions, i.e., buggy and correctly-patched versions, and buggy and incorrectly-patched versions. We calculate two sets: a set of invariants exist in a buggy version but not exist in a patched version (i.e., a set of violated invariants, *vio-inv* for short), and invariants that exist in a patched version but not in a buggy version (i.e., newly-inferred invariants, *new-inv* for short). Specifically, if one invariant of a program point exists in the buggy version, but the same program point in the patched version does not contain this invariant based on string comparison, this invariant is marked as one of the differences, i.e., part of *vio-inv*. On the contrary, a patched version may contain invariants that do not exist in the same program point of the buggy version, this invariant is new in the patched version, i.e., part of *new_inv*. We plan to incorporate advanced differencing methods on invariants such as implication distance and string distance metrics [20] to better quantify the differences between invariants.

IV. Results

In this section, we present the results to answer the four RQs. The conclusions are based on our analysis on 73 bugs and 169 patches to fix these bugs (73 correct ones and 96 plausible ones by six test-based APR techniques).

RQ1: Invariants Affected by Correct Patches

Motivation. Patches modify program semantics and therefore unavoidably affect the generated invariants. One may expect that invariants generated based on failing test cases would be affected significantly by correct patches. On the contrary, the affected invariants on passing test cases may be much less significant and quite minimal. In this RQ, we examine that to what extent, a correct patch may impact program semantics based on the affected invariants. Since the generated invariants highly depend on test cases, we study the impacts separately for passing and failing test cases.

Results. Table II summarizes how correct patches from the four studied open-source projects affect the generated invariants from both the passing and failing test cases. Each correct patch will be placed in one of the four categories, which come from the combinations of the two conditions: whether *vio-inv* is empty and whether *new-inv* is empty. How we calculate *vio-inv* and *new-inv* is described in Section III. Among the four categories, if both *new-inv* and *vio-inv* are empty, it means that a patch does not impact the executions of test cases (failing or passing) at the level of program invariants.

On one hand, we find that for **72**/73 of the studied correct patches, correct patches do have impact on the generated invariants from failing test cases. Some invariants on a buggy version are violated due to program changes. A patched version may have additional invariants that cannot be concluded in the buggy version, such as newly executed code in the patched version. Since a correct patch makes the failing test cases pass, it is likely to have impacts on runtime behaviours, and such modified runtime behaviours are mostly captured by the abstraction level invariants.

On the other hand, many of the correct patches also have impacts on the generated invariants (**43**/73) from passing test cases. It is likely that correct patches may not affect too much on the inferred invariants from the passing test cases. However it is also possible that there exist erroneous program executions in passing test cases that should be identified and grouped into failing test cases instead. Erroneous program executions do not lead to test case failures due to weak oracles. Such erroneous program executions could be the reasons behind that we find many correct patches affect passing test cases. Also, some of the generated invariants may be related to runtime dependencies and are likely to change for a different run.

As a future extension, we will perform a qualitative analysis to understand the reasons behind and design techniques to identify the more "*relevant*" invariant changes.

TABLE II: Comparing the invariants generated on a buggy version and a correctly-patched version.

	Passing Test Cases		**Failing Test Cases**	
	new-inv is empty	*new-inv* is **not** empty	*new-inv* is empty	*new-inv* is **not** empty
Lang (12 patches)				
vio-inv is empty	4	0	0	3
vio-inv is **not** empty	0	8	0	9
Math (45 patches)				
vio-inv is empty	20	3	1	2
vio-inv is **not** empty	0	22	0	42
Chart (14 patches)				
vio-inv is empty	6	0	0	2
vio-inv is **not** empty	0	8	0	12
Mockito (2 patches)	0	0	0	0
vio-inv is **not** empty	0	2	0	2

TABLE III: Comparing the invariants generated on a buggy version and the version with one plausible patch by APRs.

	Passing Test Cases		**Failing Test Cases**	
	new-inv is empty	*new-inv* is **not** empty	*new-inv* is empty	*new-inv* is **not** empty
Lang (13 patches)				
vio-inv is empty	3	0	1	5
vio-inv is **not** empty	0	10	1	6
Math (62 patches)				
vio-inv is empty	28	1	0	0
vio-inv is **not** empty	2	31	0	62
Chart (20 patches)				
vio-inv is empty	4	0	0	1
vio-inv is **not** empty	0	16	0	19
Mockito (1 patches)	0	0	0	0
vio-inv is **not** empty	0	1	0	1

RQ2: Invariants Affected by Incorrect Patches by APR techniques

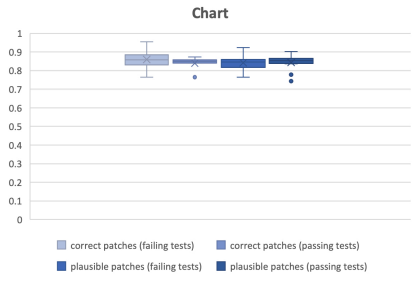
Motivation. In this RQ, we investigate the impacted invariants by each plausible patch. Plausible patches are essentially incorrect patches, e.g., making the failure symptom disappear instead of actually fixing a bug. As such, it is expected that a plausible patch should impact the invariants generated from failing test cases since the failure symptom is removed. Different from a correct patch, a plausible patch may also modify correct program behaviours, which may be reflected by invariants from passing test cases.

Results. Table III shows a summary of to what extent, plausible patches may impact the generated invariants from either passing and failing test cases. In short, among all the **96** plausible patches in our study, all but one show impacts on the generated invariants from failing test cases. From the side of passing test cases, 60 of the plausible patches show impacts on the generated invariants. By comparing with the results of RQ1, we find that more of the plausible patches have impacts on the executions of passing test cases and such impacts are reflected by the inferred invariants. It is possible to utilize the different impacts to identify plausible patches. In the future, we will qualitatively analyze and uncover case-by-case and propose techniques to automatically identify plausible patches.

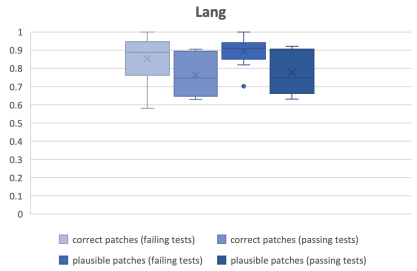
RQ3: The Scope of Invariants Affected by Test-Passing Patches

Motivation. Invariant generation can be very expensive, especially when there are many test cases. In this RQ, we study for how many program points, patches (either correct or plausible) modify the inferred invariants. If the scope of impact is a small percentage, this indicates that we can further speed up the process by limiting the program points being monitored. If the scope is a large percentage, meaning that to reduce the overhead of invariant generation, it requires to select some program points which may be most helpful in identifying plausible patches.

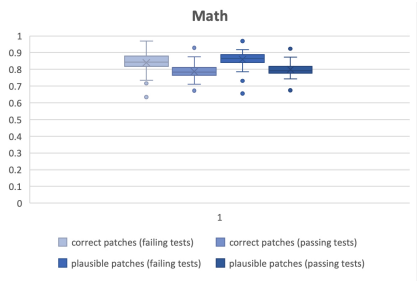
Results. We study the scope of the affected invariants, whether they are limited to the buggy class and method (i.e., the methods and the class modified by the patches) or are expanded to other related methods. In particular, for each version and different sets of test cases (i.e., passing and failing) we calculate a ratio of impacted program points out of the total number of covered program points. Note that we already exclude the test cases that do not cover the modified code in invariant generation. Figure 3 shows the ratio of program points where the generated invariants show differences for each of the studied projects. In short, we find that on average, the impact ratio is between 0.7 and 0.8 under different settings, i.e., passing or failing test cases, correct or plausible patches. This shows that a significant portion of program points are



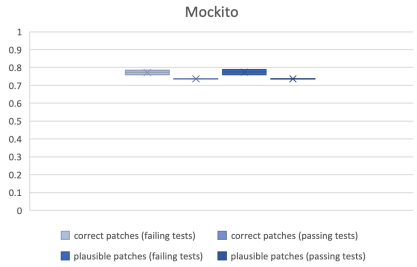
(a) Chart



(b) Lang



(c) Math



(d) Mockito

Fig. 3: The ratio of the program points where the invariants are affected by test-passing patches.

affected by test-passing patches even though the patches may only modify one method. This also indicates that there is still some space but not too many to further speed up the process by limiting the instrumentation to certain program points. In addition, limiting to only the program points that are modified by patches may miss a significant portion of invariants that are impacted differently between versions.

RQ4: Comparing Correct and Plausible Patches on the Affected Invariants

Motivation. Since a plausible patch shows similar behaviours with correct patches at coarse-grained level, i.e., making all the test cases pass, it is possible that the affected invariants by a correct patch would be similar to those by a plausible patch. If they are identical, it means that we cannot distinguish one from the other. However, if they are often not identical, which means that a plausible patch often impacts invariants differently compared to a correct patch, it is potential to utilize the affected invariants to correctly identify plausible patches.

Results. For the 96 studied plausible patches, we compare *each* of them with a corresponding correct patch with respect to whether the affected invariants by the two patches are identical. Similar to RQ1 and RQ2, for each plausible patch, the comparison is performed separately for both passing and failing test cases. For example, one plausible patch for a bug may have the same set of affected invariants with the correct patch, but have a different set of affected invariants with the correct patch. In short, if a plausible patch shows no difference in affected invariants with the correct patch under both settings, i.e., passing and failing test cases, then we conclude there is no way to tell the differences between them based on affected invariants. This sets an upper limit of how many plausible patches can be identified using invariants.

Our study finds that among the 96 plausible patches, only **four** *patches shows no distinction in affected invariants with their corresponding correct patch. This indicates that invariants have great potential to be leveraged in filtering out plausible patches and improving APR techniques.*

In addition, we examine the percentage of *useful* affected invariants among all the affected invariants by each plausible patch. A portion of affected invariants are useful if they are not affected by the correct patch. If the percentage of *useful* invariants is higher, the plausible patch modifies runtime behaviours differently compared to the correct patch. Figure 4 shows the percentage of useful invariants of 96 plausible patches. In addition to separating failing and passing test cases, we also separate the two types of affected invariants: *vio-inv* and *new-inv*. The results show that the portion of useful invariants based on passing tests are constantly higher than failing tests. In most of the cases (except for CHART on failing tests), *vio-inv* typically has a similar portion of useful affected invariants compared to that of *new-inv*. Overall both *vio-inv* and *new-inv* show promising results to distinguish plausible patches from correct ones, i.e., the median portion of useful invariants is at least 50% across the four projects.

Despite the potential, there exists a significant portion of affected invariants that are shared between a correct patch and a plausible patch. This poses challenges to correctly distinguish the two types: correct and plausible. Our extension will include an in-depth analysis to reveal more comprehensive information on the shared and non-shared affected invariants between a correct and a plausible patch.

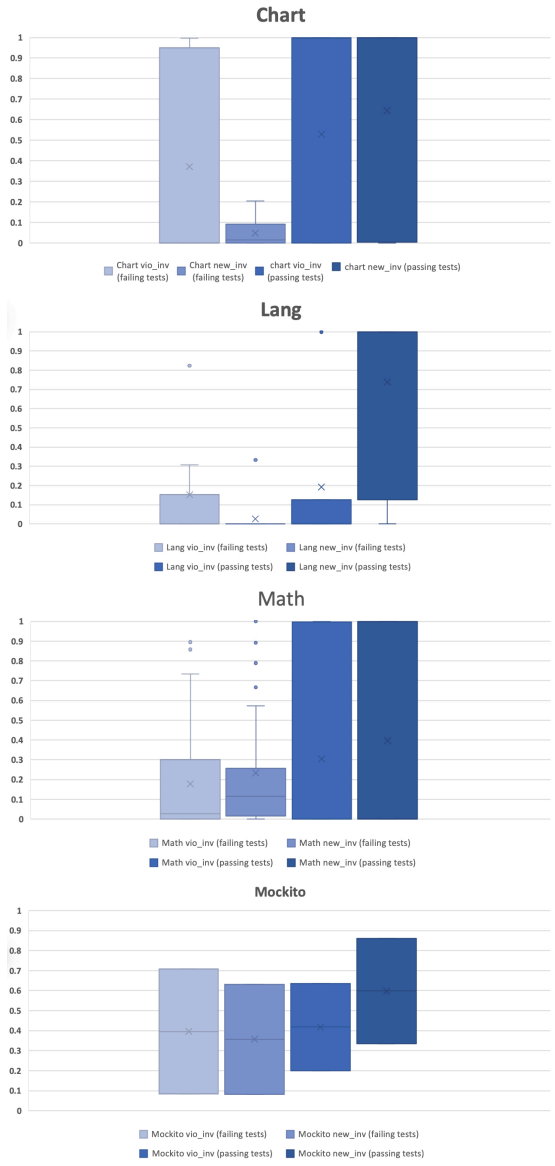


Fig. 4: The portion of useful invariants under different settings. Useful invariants are the affected invariants that are exposed by plausible patches, however **not** by correct patches.

V. Related Work

Many APR techniques [2, 1, 13, 25, 26, 27, 11, 28, 29] are proposed in the last decade. Such test-based APR techniques rely on readily-available test cases for patch validation. However, test cases are largely insufficient [13], hence APR techniques may generate plausible patches. A recent study by Cambronero et al. [30] shows that plausible patches does not help developers in bug fixing.

Various techniques are proposed to improve test-based APRs by identifying plausible patches [31, 17, 16, 32, 15]. Such techniques either rely on building repair models (i.e., manually-crafted patch antipatterns, or prioritizing likely correct patches in the search spaces), or enhancing test cases (i.e., new test input, improved test oracles). In this paper, we present a preliminary study that sheds light on a different direction, i.e., utilizing dynamically-generated invariants from the readily-available test cases. Our preliminary results show that assesing the affected invariants for the purpose of distinguishing plausible patches from correct ones has great potential. Despite the great potential, our study also shows that it might be challenging to achieve this objective since the affected invariants by plausible patches often share a common part with those by correct patches. Our future extension will include a qualitative analysis on the affected invariants to reveal more inspirations for designing an automated solution.

Program invariants are abstraction of runtime behaviours and can be utilized to benefit many software engineering tasks. Prior work on utilizing invariants includes to improve test suites [19], to check the upgrades of software components [33], to support refactoring [34], and to improve mutation testing [35]. Different from previous studies, we apply invariant generation in a new context – automated program repair. Compared to a recent study by Cashin et al. [20], which uses invariants for understanding automatically-generated patches, our study focuses on studying how the affected invariants differ between versions.

VI. Threats

Internal Threats. Our computation of *new-inv* and *vio-inv* is coarse-grained and does not consider the relation between two invariants. For example, the two invariants "this.maxY == orig(this.maxY)" and "this.maxY <= orig.(this.maxY)" from the illustration example are viewed as two distinct invariants instead of relevant ones.

External Threats. Our study is limited to Defect4J dataset and six test-based APR techniques. Therefore, the conclusions may not be generalizable to new bug dataset and other APR techniques. Our study is limited to Java bugs in Defect4J.

Construct Threats. Our study is based on comparing inferred invariants during test case executions. Inferred invariants can be viewed as an abstraction of runtime behaviours, however it does not reflect every aspect of runtime behaviours, e.g., local variables are not recorded for invariant generation.

VII. Conclusions

Test-based APR techniques may generate plausible patches due to inadequate test cases. In this work, we examine and explore the differences between a plausible and a correct patch for the same bug from the perspective of abstractions of runtime behaviours, i.e., represented as invariants. Our preliminary study shows that invariants from failing test cases are very often affected by both correct and plausible patches. Our study highlights the potential to utilize dynamically-generated invariants in identifying plausible patches.

References

[1] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE, 2012, pp. 3–13.

[2] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013*

International Conference on Software Engineering, ser. ICSE, 2013, pp. 772–781.

[3] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE, 2017, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[4] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *In proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering*, ser. FSE, 2015, pp. 166–178.

[5] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, 2017, pp. 727–739.

[6] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, ser. ICSE, vol. 1, 2015, pp. 459–470.

[7] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 159:1–159:27, Oct. 2019. [Online]. Available: http://doi.acm.org/10.1145/3360585

[8] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, p. 269–278. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00039

[9] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 613–624. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338952

[10] J. Yang, L. Tan, J. Peyton, and K. A. Duer, "Towards better utilizing static application security testing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 51–60. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00014

[11] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2Fix: Automatically generating bug fixes from bug reports," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ser. ICST, 2013, pp. 282–291.

[12] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "iFixR: Bug report driven program repair," in *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019.

[13] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA, 2015, pp. 24–36.

[14] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER, vol. 1, 2016, pp. 213–224.

[15] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 789–799. [Online]. Available: https://doi.org/10.1145/3180155.3180182

[16] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, 2017, pp. 831–841.

[17] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 8–18. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330558

[18] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," *SIGPLAN Not.*, vol. 33, no. 7, pp. 83–90, Jul. 1998. [Online]. Available: http://doi.acm.org/10.1145/277633.277647

[19] M. Harder, M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 60–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776824

[20] P. Cashin, C. Martinez, W. Weimer, and S. Forrest, "Understanding automatically-generated patches through symbolic invariant differences," in *ASE 2019 NIER*.

[21] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/302405.302467

[22] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[23] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 441–444. [Online]. Available: http://doi.acm.org/10.1145/2931037.2948705

[24] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[25] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 213–224.

[26] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *Proceedings of the 2015 Internaltional Conference on Software Engineering*, ser. ICSE, 2015, pp. 471–482.

[27] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE, 2016, pp. 691–701.

[28] J. Y. Sergey Mechtaev and A. Roychoudhury, "DirectFix: Looking for Simple Program Repairs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE, 2015, pp. 448–458.

[29] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2012, pp. 221–236.

[30] J. P. Cambronero, J. Shen, J. Cito, E. Glassman, and M. Rinard, "Characterizing developer use of automatically generated patches," 2019.

[31] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE, 2016, pp. 727–738. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950295

[32] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Journal of Empirical Software Engineering*, vol. 20, no. 1, pp. 176–205, Feb. 2015.

[33] S. McCamant and M. D. Ernst, "Predicting problems caused by component upgrades," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 287–296, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/949952.940110

[34] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold, "Automated support for program refactoring using invariants," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 736–. [Online]. Available: https://doi.org/10.1109/ICSM.2001.972794

[35] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572282