A Closer Look at Real-World Patches

Kui Liu*, Dongsun Kim*, Anil Koyuncu*, Li Li†, Tegawendé F. Bissyandé*, Yves Le Traon*
*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{kui.liu, dongsun.kim, koyuncu.anil, tegawende.bissyande, yves.letraon}@uni.lu
†Monash Software Force (MSF), Monash University, Melbourne, Australia
li.li@monash.edu

***Abstract*—Bug fixing is a time-consuming and tedious task. To reduce the manual efforts in bug fixing, researchers have presented automated approaches to software repair. Unfortunately, recent studies have shown that the state-of-the-art techniques in automated repair tend to generate patches only for a small number of bugs even with quality issues (e.g., incorrect behavior and nonsensical changes). To improve automated program repair (APR) techniques, the community should deepen its knowledge on repair actions from real-world patches since most of the techniques rely on patches written by human developers. Previous investigations on real-world patches are limited to statement level that is not sufficiently fine-grained to build this knowledge. In this work, we contribute to building this knowledge via a systematic and fine-grained study of 16,450 bug fix commits from seven Java open-source projects. We find that there are opportunities for APR techniques to improve their effectiveness by looking at code elements that have not yet been investigated. We also discuss nine insights into tuning automated repair tools. For example, a small number of statement and expression types are recurrently impacted by real-world patches, and expression-level granularity could reduce search space of finding fix ingredients, where previous studies never explored.**

***Keywords*-Program patch; fix pattern; abstract syntax tree.**

I. Introduction

In recent years, to reduce the cost of software bugs [1], the research community has invested substantial efforts into automated program repair (APR) approaches [2]–[15]. The first significant milestone in that direction is GenProg [16], an APR technique that uses genetic programming to apply a sequence of edits to a buggy source code until a test suite is satisfied. After GenProg, several follow-up techniques have been proposed: Nguyen et al. [17] relied on symbolic execution. Xiong et al. [18] focused on mining contextual information from documents and across projects. Kim et al. [19] proposed to leverage fix templates manually mined from human-written patches. Long and Rinard [20] built a systematic approach to leveraging past patches.

Although existing APR studies have achieved promising results, two issues mainly stand out: the applicability of APR techniques for a diverse set of bugs, and the low quality of patches generated by them [21], [22]. The existing APR techniques tend to generate patches for a few specific types of bugs [19] or to make nonsensical changes to programs [2].

In this study, our conjecture is that one potential issue with the (in)effectiveness of APR techniques is the limitation carried by the granularity at which APR techniques perform repair actions. Since *generate-and-validate* [23] approaches (such as GenProg) rely on fault localization techniques to identify buggy code, they generate patches at the statement level often based on stochastic mutations. Actually, as our study shows, most bugs are localized on specific code entities (e.g., the wrong infix-expression in Figure 1) within buggy statements. Therefore, mutating every part of a buggy statement is likely to lead, at best, to a very slow and resource-intensive fixing process, and at worst, to the generation of incorrect and nonsensical patches with high costs.

Real-world patches (i.e., written by human developers) can provide useful information (e.g., on repair actions) for efficient generation of correct patches. Previous work [11], [14], [19], [20] has already shown that patches from software repositories can be leveraged to improve software repair. Nevertheless, the prerequisite for further advancing state-of-the-art APR techniques is to acquire all-round and detailed understanding about real-world patches.

Several studies in the literature have attempted to build such knowledge, but all focused on characterizing changes at the *statement* level. Pan et al. [24] manually summarized 27 fix patterns from existing patches of Java projects. Their patterns are, however, in a high-level form (e.g. "If-related: Addition of Post-condition Check (`IF-APTC`)"). Martinez et al. [7] analyzed bug fix transactions at the AST statement level of code changes. Zhong et al. [21] also analyzed the repair actions of patches at the AST statement level to understand the nature of bugs and patches (see Section VII for more detailed comparison). Although these studies provide interesting insights into bug fix patterns at the coarse-grained level of statements, they can be misleading when implementing automated repair actions. Indeed, buggy parts can be localized in a more fine-grained way, leading to more accurate repair actions.

Consider the real-world code change illustrated in GNU Diff format in Figure 1. This change is committed to a software repository as a *patch*.
Most fault localization tools would identify Line 183 in file *MultivariateNormalDistribution.java* as a suspicious (i.e.,

```
Commit cedf0d27f9e9341a9e9fa8a192735a0c2e11be40
src/main/java/org/apache/commons/math3/distribution/MultivariateNormalDistribution.java
@@ −183, 3 +183, 3 @@
−     return FastMath.pow(2 * FastMath.PI, -dim / 2) *
+     return FastMath.pow(2 * FastMath.PI, -0.5 * dim) *
          FastMath.pow(covarianceMatrixDeterminant, −0.5) *
          getExponentTerm(vals);
```

Fig. 1: Patch of fixing bug MATH-929, a value-truncated bug.

The hierarchical repair actions of a patch parsed by GumTree:
UPD ReturnStatement@@"***buggy code***" to "***fixed code*** ".
UPD InfixExpression@@"***buggy code***" to "***fixed code***".
UPD MethodInvocation@@"***buggy code***" to "***fixed code***".
UPD InfixExpression@@"-dim / 2" to "-0.5 * dim".
UPD PrefixExpression@@ "-dim" to "-0.5".
DEL SimpleName@@ "dim" from "-dim" .
INS NumberLiteral@@"0.5" to "-dim".
UPD Operator@@"/" to "*".
DEL NumberLiteral@@ "2" from "2".
INS SimpleName@@"dim" to "2".

Fig. 2: A graphic representation of the hierarchical repair actions of a patch parsed by GumTree. Buggy code entities are marked with red and fixed code entities are marked with green. 'UPD' represents updating the buggy code with fixed code, 'DEL' represents deleting the buggy code, 'INS' represents inserting the missed code, and 'MOV' represents moving the buggy code to a correct position. Due to space limitation, the buggy and fixed code (See Figure 1) are not presented in this figure.

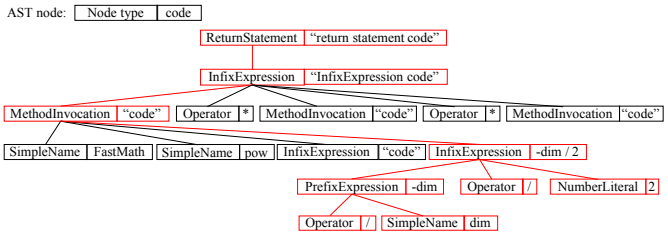

Fig. 3: Example of parsing buggy code in terms of AST. The exact buggy code entities in its AST are highlighted with red. For simplicity and readability of the AST of buggy code, the sub-trees of bug-free code nodes are not shown in this tree.

might be buggy) location before commit `cedf0d` of project `commons-math`. To generate a corresponding patch, APR tools will apply generic fix patterns for ReturnStatement, thus it may fail to properly fix the bug or take a long time even if it succeeds.

Based on the hierarchical and fine-grained view (see Figure 2) of repair actions of the patch in Figure 1 provided by GumTree (an AST based code changes differencing tool) [25], it is easy to find that the exact repair action of this patch occurs to "-dim / 2", an InfixExpresion node in the AST, a child of a MethodInvocation node (see Figure 3). This repair action is more precise than replacing a ReturnStatement as a whole with another statement. Our conjecture is that it is less probable to find or identify exactly the same repair action with statement replacement than expression replacement, from the search space of human-written patches.

As shown in the motivating example above, previously existing studies [7], [21], [24] did not take a close look at existing patches using advanced tools such as GumTree and APR research may have been missing important and accurate insights for enhancing the state-of-the-art APR techniques. In particular, by mining patches beyond statement-level information, we can investigate which fine-grained buggy code entities are recurrent in repair changes, and which repair actions are successfully applied to them. Insights from such questions can allow tuning APR techniques towards faster completions (e.g., focus changes on more likely buggy entities) and more accurate generation of correct patches (e.g., make accurate changes). To that end, we investigate 16,450 bug fix commits collected in two distinct ways from seven Java open source project repositories, in a fine-grained way by leveraging GumTree.

Looking closely at real-world patches, we find that there are opportunities for APR techniques to be targeted at code elements that have not yet been investigated. We also find expression-level granularity could reduce search space of finding fix ingredients for similar bugs. We further discuss nine insights into tuning APR techniques towards being fast in their trials for patch generations, and also towards producing patches that have more probability to be correct.

II. Background

This section clarifies the notions of code entities related to AST representations and AST diffs of code changes.

A. Code entities

Code entities are basic units (i.e., nodes) comprising ASTs. Since our work investigates Java projects, this study collects code entities defined in the Eclipse JDT APIs [26]. The APIs describe code entities in the AST representation of Java source code. There are 22 statement [27] (e.g., ReturnStatement), declarations (e.g., TypeDeclaration, EnumDeclaration, MethodDeclaration and FieldDeclaration), and 35 expression [28] (e.g., InfixExpression) entity types. We collect these code entities from Java source code. Note that we refer direct children nodes of a statement or an expression in an AST as *code element*s in this study.

B. AST diffs

Our study analyzes patches in the form of AST diffs. In contrast with GNU diffs that represent code changes as a pure text-based and line-by-line edit script, AST diffs provide a hierarchical representation of the changes applied to the different code entities at different levels (statements, expressions, and elements). We leverage GumTree [25] to extract and describe repair actions implemented in patches since the tool is open source [29], allowing for replication, and is built on top of the Eclipse Java model [30].

Overall, in this study:

- A *Code entity* represents a *node* in ASTs. It can be a declaration, statement, expression, etc., or more specific element of a statement, an expression, etc.
- A *Change operator* is one of the following in GumTree specifications: UPDATE, DELETE, INSERT, and MOVE.
- A *Repair action* represents a combination of a change operator and a code entity (e.g., `UPD stmt` or `DEL expr`).

III. Research Questions

The objective of this study is to investigate the repair actions by closely looking at human-written patches, and to build knowledge on which/how code entities are commonly involved/impacted by them. Our study examines the finer-grained AST diff representations of patches than the existing studies in the literature [7], [21], [24], to implement a closer

look at code changes. In the study, we investigate the following research questions:

RQ1: Do patches impact some specific statement types?

We revisit a common research question in the literature of patch mining studies: common statements changed by patches. In the majority of APR processes, the initial task is locating the buggy line or statement. Specifically, in generate-and-validate approaches, a spectrum fault localization technique (such as Tarantula [31], Ochiai [32], Ochiai2 [33], Zoltar [34] and DStar [35]) is used to identify suspicious lines or statements that are then mutated by APR tools [2], [15], [16], [18]. It is thus essential, based on real-world patches, to investigate which types of statements are recurrently involved in bug fix patches, and what kinds of repair actions are regularly applied to them by human developers.

RQ2. Are there code elements in statements that are prone to be faulty?

A statement node in an AST representation can be decomposed into different children nodes whose types vary following the statement type. Consider the variable declaration statement "`private int id = 1;`", it can be decomposed into the modifier ("`private`"), the data type ("`int`"), the identifier ("`id`") and an initializer (the number literal "`1`"). Since common fault localization techniques can only point to suspicious lines, APR tools generally attempt to mutate the statements (often in a stochastic way) which can lead to nonsensical alien code [19]. With in-depth knowledge on fault-prone code elements of statements, APR tools can rapidly generate patch candidates that are more likely to be successful (with regards to the test cases), and which have more chances to be correct.

RQ3. Which expression types are most impacted by patches?

Statements in Java programs are generally built based on expressions whose values eventually determine the execution behavior, and are often associated with bugs. In a preliminary study, we have found that in most patches, expressions were the buggy elements where a patch actually modifies a program. Our conjecture is that only a small number of expression types could be responsible for the majority of bugs at the expression level.

RQ4. Which parts of buggy expressions are prone to be buggy?

Expressions in Java program can be composite entities: their AST nodes may have several children. For example, an `InfixExpression` consists of a left-hand expression, an infix operator, and a right-hand expression. We investigate the type of buggy elements within buggy expressions to further refine our understanding of recurring bug locations.

IV. Study Design

We describe our dataset and the methodology for identifying bug fix patches and the code entities impacted by patches. The data and the implemented tool are available at https://github.com/AutoProRepair/PatchParser.

A. Dataset

For this study, we focus on Java open source projects commonly used by the research community [5], [7], [21], [24].

TABLE I: Subjects used in our study. The number of bug fix commits actually used in the study is 16,450 as highlighted.

Projects	LOC	# Commits		
		All	Identified	**Selected**
commons-io	28,866	2,225	222	**191**
commons-lang	78,144	5,632	643	**522**
mahout	135,111	4,139	751	**717**
commons-math	178,84	7,228	1,021	**909**
derby	716,053	10,908	3,788	**3,356**
lucene-solr	943,117	51,927	11,408	**10,755**
Total	2,080,131	82,059	18,013	**16,450**

LOC: # lines of code. **All:** # of all available commits in a project. **Identified:** # of identified bug fix commits. **Selected:** # of selected bug fix commits actually used in this study.

Table I enumerates the subject projects[1], of varying sizes, collected from the Apache software foundation [36]. These projects have been leveraged in previous studies on software patches as they are reputed to provide commit messages that are clear and consistent with the associated code change diffs [21]. We further note that the Apache projects host an issue tracking system that is actively used, with a large number of commits keeping the link between reported bugs and the associated bug fix commits.

B. Identification of patches

We consider the following criteria of identifying bug fix commits in software repositories:

1) *Keyword matching*: we search commit messages for bug-related keywords (namely *bug*, *error*, *fault*, *fix*, *patch* or *repair*). This method was introduced by Mockus and Votta [37], and used in several studies [11], [21], [24].
2) *Bug linking*: we identify the reported and fixed bug IDs (e.g, **MATH-929**) in JIRA issue tracking system with two criteria: (1) *Issue Type* is '**bug**' and (2) *Resolution* is '**fixed**' [5]. We thus collect bug-fix commits by identifying such reported and fixed bug IDs in commit messages.

After applying the criteria above, we figure out that some selected commits are not actually bug fix commits; instead, they are commits regarding test cases, Javadoc and external documentation (e.g. xml files). These commits are out of scope and excluded from this study. Bug fix commits are collected by following the three criteria: (1) bug fix commits contains modified *.java* files, (2) these files do not have "*test*" in their names, and (3) these files can be parsed by GumTree to generate repair actions of buggy code fixing. Thus, 18,013 bug fix commits are collected from the seven projects.

To increase the confidence in our selected patches, we limit our study on patches with small-sized change hunks. The hunk size is defined as the number of lines of buggy code (respectively of fixed code) in a code change diff from a patch, where buggy hunk starts with '−' and fixed hunk starts with '+'. Figure 4 shows the distributions of sizes of buggy code hunks and fixed code hunks from all collected bug fix commits. In this study, the patches, whose buggy hunk size is up to 8

[1] `Lucene` and `Solr` share the same source code repository, so we put the results of the two projects in a single row.

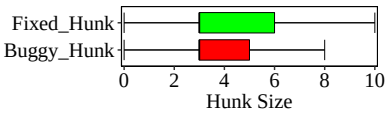

Fig. 4: Distributions of buggy and fixed hunk sizes of collected patches. *Fixed/Buggy_Hunk* refers to fixed/buggy lines in a code change hunk of collected patches.

lines and fixed hunk size is up to 10, are selected as our dataset. These threshold values are set based on the Upper Whisker values from the hunk size distribution Tukey boxplots [38] in Figure 4[2]. To that end, 16,450 bug fix commits are selected as the data of this study.

Previous studies have indeed shown that large code change hunks usually address feature addition, refactoring, etc. [39], [40], and do not often contain meaningful bug fix patterns [24]. Pan et al. [24] further reported that most bug fix hunks (91-96%) are small ones and ignoring large hunks has minimal impact on patch analysis.

C. Identification of buggy code entities in ASTs

To identify the buggy code entities and their repair actions, all patches are parsed by feeding GumTree with the buggy and fixed versions of a buggy Java file. The buggy code entities and their repair actions are identified by retrieving GumTree output in terms of its hierarchical construct. In this study, all elements of *deleted* and *moved* statements are treated as buggy code entities and all elements of *inserted* statements are treated as fixed code entities. For *updated* buggy statements, we further identify their exact buggy elements to find out the exact buggy code entities. For a buggy expression, if it is deleted, moved, or replaced by another expression, it is considered as a whole buggy expression. Otherwise, the buggy expression is further parsed to identify its buggy element(s).

V. Analysis Results

In this study, we investigate patches found in the seven projects listed in Table I to identify the distributions of buggy code entities and their corresponding repair actions. The results would answer the RQs described in Section III. The distributions of the statistic data split by projects are similar to each other. Due to the space limitation, the statistic data of the seven projects are merged together. Project-split statistic data are available at aforementioned website.

A. RQ1: Buggy Statements and Associated Repair Actions

Root AST node types in patches: Declaration entities in source code can also be buggy. Figure 5 provides a statistical overview of the root AST node types impacted by repair actions in patches. While statement entities occupy a large proportion in buggy code, it is noteworthy that buggy declarations, and associated repair actions are seldomly mentioned in bug fix studies [7], [21], [24], and may thus be ignored by the APR community. Our study, however, finds that repair actions on declaration entities (i.e., *class* (TypeDeclaration), *enum*, *method* and *field* declarations) account for 26.7% of repair actions of

[2]The upper whisker value are determined by 1.5 IQR (interquartile ranges) where IQR = 3rd Quartile − 1st Quartile, as defined in [38].

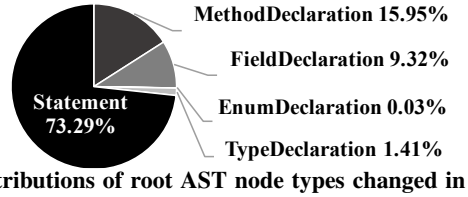

Fig. 5: Distributions of root AST node types changed in patches.

Commit 185e3033ef43527a729c9dda5d57ed0537921a27
src/main/java/org/apache/commons/math3/random/BitsStreamGenerator.java
@@ −29,2 +29,2 @@
public abstract class BitsStreamGenerator
− implements RandomGenerator {
+ implements RandomGenerator, Serializable {
Repair actions parsed by GumTree:
```
UPD TypeDeclaration@@"BitsStreamGenerator"
    INS Type@@"Serializable" to "TypeDeclaration"
```

Fig. 6: Patch of fixing bug MATH-927, a TypeDeclaration**-related bug, by adding the interface** Serializable**.**

patches, suggesting that the research community should take more efforts to investigate bugs related to declarations, as they may contribute to a significant portion of buggy code.

As shown in Figure 5, TypeDeclaration and EnumDeclaration only occupy 1.44% of repair actions of patches, that might be the reason why the state-of-the-art APR tools ignore the bugs relating these declaration entities and focus on fixing bugs at the statement level. However, buggy declaration entities indeed bother developers. For example, Figure 6 shows a patch of fixing bug MATH-927, a TypeDeclaration-related bug, which makes cloning broken and can cause java.io.NotSerializableException [41], thus it is fixed by adding the interface Serializable into its TypeDeclaration node. This bug is one bug in benchmark Defects4J [42], however, it has not been fixed by any state-of-the-art APR tools yet [43], since those tools focus on the statement level to fix bugs.

Insight 1: Declaration entities in source code can also be buggy, which constitutes a research opportunity for Automated Program Repair beyond statement level. To fix bugs related to declaration entities, such as the bug in Figure 6, mutation-based tools (e.g., GenProg) could generate mutations for the buggy TypeDeclaration by mutating common implementable interface types, pattern-based tools (e.g., PAR) could summarize NotSerializableException fix pattern from this kind of patches, or search-based tools could specify constraints with fine-granularity information (e.g., TypeDeclaration and NotSerializableException) to reduce search space and find fix ingredients from existing patches.

Repair actions for statements: Statements (73.3% shown in Figure 5) are the main buggy code entities, which motivates researchers to fix bugs at the statement level. Therefore, to build the knowledge on repair actions at the statement level, we investigate the statement types impacted by patches as well as repair actions (categorized in *Update*, *Delete*, *Move* and *Insert*) that are applied to them. Figure 7 shows the distribution of statement types impacted by patches as well as the distributions of repair actions. Due to space limitation, the figure only lists up the top-5 statement types and the remaining are summed in an "Others" category.

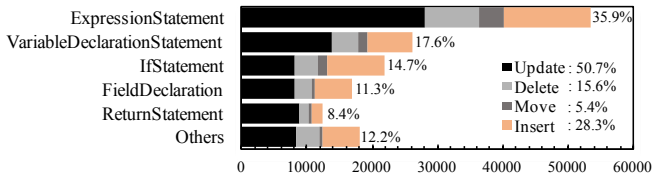

Fig. 7: Distributions of statement-level repair actions of patches.

1) Updating statements: As shown in Figure 7, a half of repair actions are statement updates, in which the entity types of buggy statements were not changed but their children entities were changed. This motivates researchers to fix bugs by mutating code at statement level (e.g., GenProg). However, coarse granularity is an important weak point for existing tools to fix bugs at the statement level.

Statements can be decomposed into several elements, which means that it would take a long time to generate patches by mutating each element even if it might succeed. For example, in Figure 1, the exact buggy code entity is the InfixExpression "-dim / 2", but other code entities can interfere with the mutating process of generating correct patches. Furthermore, the statement type can limit or noise the search space of finding fix ingredients. The buggy statement in Figure 1 is a ReturnStatement, which means this patch can only be a fix ingredient for buggy ReturnStatements at the statement level. However, similar bugs can locate in other statement types (such as the bug in Figure 8).

```
double d = FastMath.pow(2 * FastMath.PI, -dim / 2) *
           FastMath.pow(covarianceMatrixDeterminant, -0.5) *
           getExponentTerm(vals);
return d;
```

Fig. 8: A mutated bug of the bug in Figure 1.

Insight 2: The abundant real-world bugs fixed by updating buggy statements can support to tune the state-of-the-art APR tools by mining fine-grained characteristics with fine granularity (expression level) of patches. For example, if APR tools could extract fine-grained context information of fixing the bug in Figure 1 (such as the exact buggy InfixExpression "-dim / 2" and the detailed changes acted on the expression) to infer fix patterns (See Section V-C) or to constraint search space, they could fix more similar value-truncated bugs beyond the ReturnStatement code entity.

2) Adoption of deletion and replacement: Simply deleting buggy statement(s) is an effective way of fixing bugs, which can also be combined with replacement. Recent experiments with APR techniques and program test suites have shown that some programs may pass tests when suspicious statements are simply deleted [16]. Thus, dropping buggy code statements could be attractive as a fast and effective way of fixing bugs. Our study shows that only 15.6% cases of repair actions consist of deleting buggy statements. Additionally, 45% of them are the code entities of the dropped statements that are inserted in other statements to replace the dropped statements.

For example, Figure 9 shows that buggy code line is deleted but the buggy code expression is inserted in the new added If statement. Such patches are associated in the literature to the IF-related Addition of Post-condition Check (IF-APTC)

```
Commit e81ef196cd9dd3c7989b96f648f96ec138faa25b
src/java/org/apache/commons/math/optimization/general/AbstractLeast
SquaresOptimizer.java
@@ −187, 1 +188, 4 @@
−     ++objectiveEvaluations;
+     if (++objectiveEvaluations > maxEvaluations) {
+         throw new FunctionEvaluationException(new
+         MaxEvaluationsExceededException(maxEvaluations), point);
+     }
Repair actions parsed by GumTree:
DEL ExpressionStatement@@"++objectiveEvaluations;"
INS IfStatement@@"if"
    INS InfixExpression@@"code" to IfStatement
       MOV PrefixExpression@@"++objectiveEvaluations"
           to InfixExpression
    INS ......
```

Fig. 9: An infinite-loop bug taken from project `commons-math` is fixed by replacing buggy statement type.

```
Commit 9c5be23a3d00b4238ddb3794a1ffec463f2ceac9
solr/core/src/java/org/apache/solr/cloud/HashPartitioner.java
@@ −46, 5 +55, 5 @@
    while (end < Integer.MAX_VALUE) {
        end = start + srange;
−       start = end + 1L;
        ranges.add(new Range(start, end));
+       start = end + 1L;
    }
Repair actions parsed by GumTree:
MOV ExpressionStatement@@"start = end + 1L;"
    from 1 to 2 in WhileStatement_BodyBlock
```

Fig. 10: A bug taken from project `solr` is fixed by moving the buggy statement.

fix pattern defined by Pan *et al.* [24], which is, however, in a high-level form and has not been used in APR tools.

Note that buggy code statement in Figure 9 is an ExpressionStatement. With further parsing of this bug, the exact buggy code entity is the PrefixExpression "++objectiveEvaluation". There are only 1,362 cases related buggy PrefixExpressions (See Table III) that is a much smaller search space of fix ingredients for this bug compared against the statement level (more than 40,000 buggy ExpressionStatement cases, see the related value on the x-axis in Figure 7). If combining ExpressionStatement with PrefixExpression, we find that the search space can be further reduced to 28.

Insight 3: Expression-level granularity could improve the state-of-the-art APR tools by reducing search space, which could be further reduced if combining statement types with expression types.

3) Moving statements: Moving a buggy statement(s) to correct its position (without other changes) is another effective way of fixing bugs. We observe that 5.4% of repair actions involve moving statements across the program code. Figure 10 shows an example of fixing a bug by moving the buggy statement to the correct position. It is difficult to obtain valuable information from its simple repair action, but its context information, such as its parent statement (i.e. WhileStatement) and the dependency of three variables (i.e., start, end, and ranges), could be used to tune APR tools.

Insight 4: To generate fix patterns or create search space with patches involving *move* actions, more context information should be considered.

4) Recurrently impacted statements: A few statement types

are recurrently impacted by patches. From the distribution of statement types in Figure 7, we note that 5 (out of 22) statement types (namely ExpressionStatement, VariableDeclarationStatement, IfStatement, ReturnStatement and FieldDeclaration) represent ∼88% of statements impacted by patches. These statistics support the motivation of many researchers to focus on repairing a specific type of statements: ACS [18] is such an APR technique example that targets IfStatement-related bugs. Our study highlights other statement types which can benefit from targeted approaches. In particular, ExpressionStatement is impacted by a third (∼36%) of repair actions, suggesting that statements of this type are more likely to contain bugs than other types of statements.

B. RQ2: Fault-prone Parts in Statements

As discussed in Section V-A, if fine-granularity information can be extracted from existing patches, it could improve APR tools. Fortunately, statements can be decomposed into different sub elements, which supports us to further investigate exact buggy elements of statements. To the best of our knowledge, we are the first to take a close look at real-world patches in finer granularity than statement level in the literature. Our study may yield further insights into the code entities which are recurrently buggy and beyond the whole statements, thus they should be the focus of APR techniques.

A statement node in an AST representation can be decomposed into several children elements. In this study, all elements of statements are classified into four categories: *Modifier*, *Type*, *Identifier*, and *Expression* where *Modifier* denotes the modifiers of source code in its AST. *Type* refers to any type nodes, *Identifier* can be a name of *class*, *method*, or *variable*, and *Expression* includes the 35 kinds of expressions defined in the Eclipse JDT APIs. The statistic distributions of these elements impacted by patches are provided in Figure 11.

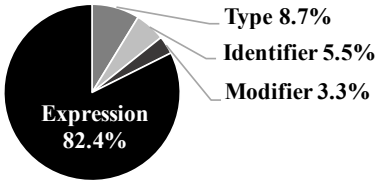

Fig. 11: Distributions of inner-statement elements impacted by patches.

Patches for "modifier" bugs: *Modifier* elements in statements can be buggy, and repair actions associated with them have simple instructions. Figure 11 presents that 3.3% cases of repair actions fixing bugs involve *Modifier* elements (i.e., qualifiers such as public, final, static). The bugs in such cases are often caused by missing a necessary modifier or assigning an inappropriate one. At best, these bugs can create style mismatch in the code, and at worst, can present semantic implications for program behavior. We can enumerate three ways repair actions that are applied:

1) *Add a missing modifier*, as in patch *A* of Figure 12, where the missed modifier "volatile" is inserted to make the variable "defaultStyle" thread-safe.

A: an example of adding a missed modifier:
Commit d55299a0554375f3f935a0ff85bd2002faa2ec55 (LANG−487)
src/java/org/apache/commons/lang/builder/ToStringBuilder.java
@@ −97, 1 +97, 1 @@
− private static ToStringStyle defaultStyle =
+ private static volatile ToStringStyle defaultStyle =
ToStringStyle.DEFAULT_STYLE;
Repair actions parsed by GumTree:
```
UPD FieldDeclaration
    INS Modifier@@"volatile" to FieldDeclaration
```

B: an example of deleting a redundant modifier:
Commit 60fe05eb7a32a4a178f4212da6a812c80cedce82 (LANG−334)
src/java/org/apache/commons/lang/enums/Enum.java
@@ −305, 1 +305, 1 @@
− private static final Map cEnumClasses = new WeakHashMap();
+ private static Map cEnumClasses = new WeakHashMap();
Repair actions parsed by GumTree:
```
UPD FieldDeclaration
    DEL Modifier@@"final" from FieldDeclaration
```

C: an example of replacing the inappropriate modifier:
Commit 0e07b8e599e0d59c259dcc8501167a80d030179b
src/java/org/apache/commons/lang/builder/EqualsBuilder.java
@@ −111, 1 +111, 1 @@
− protected boolean isEquals;
+ private boolean isEquals;
Repair actions parsed by GumTree:
```
UPD FieldDeclaration
    UPD Modifier@@"protected" to "private"
```

Fig. 12: Three bugs in project `commons-lang` are fixed by changing their modifiers.

2) *Delete an inappropriate modifier*, as in patch *B* of Figure 12, where the inappropriate modifier "final" is dropped to avoid exposing a mutating map.
3) *Replace an inappropriate modifier*, as in patch *C* of Figure 12, where modifier "protected" is changed into "private" to prevent potential vulnerability.

The Java language supports 12 *Modifier* types [44] whose inappropriate usage could lead to bugs and even vulnerabilities. The FindBugs [45] static analyzer even enumerates 17 bug types related to modifiers. Actually, four projects in our study integrate FindBugs in their development chain [46]–[49]. However, fixing those modifier-related bugs in these projects are still addressed manually. Additionally, all modifier-related bugs in benchmark Defects4J have not been fixed by any state-of-the-art APR tools [50]. The reason might be that APR tools cannot fix modifier-related bugs because of coarse granularity, so that it is necessary to tune APR tools to fix modifier related bugs with finer granularity.

Insight 5: The small number of modifier types allows researchers to enumerate all possible mutations or change patterns for buggy modifier(s) at modifier level, and to reduce search space of fix ingredients for modifier-related bugs. Additionally, existing patches of fixing modifier-related bugs and the static analysis tools (e.g., FindBugs provides detailed definitions for specific modifier-related bugs) can help researchers tune APR tools to fix specific modifier-related bugs automatically, like the thread-safe bug of patch *A* in Figure 12.

Fixing modifier-related bugs can, however, have unsuspected impacts beyond the code base, and thus may constitute a new height that APR techniques can try to reach through

Commit 984a03d1ceb6e4b5d194e4d639d0b0fca46d92be
src/main/org/apache/tools/ant/types/Path.java
@@ −70, 2 +70, 2 @@
− public static Path systemClasspath =
+ public static final Path systemClasspath =
new Path(null,System.getProperty("java.class.path"));

Code @Line 1484 in InternalAntRunner.java in Eclipse project:
org.apache.tools.ant.types.Path.systemClasspath = systemClasspath;

Fig. 13: A modifier-related patch breaks the backward compatibility of project `Apache ant`.

Commit b032fb49f09c020174da1d5d865d878b8351d89d
lucene/codecs/src/java/org/apache/lucene/codecs/compressing/CompressingStoredFieldsIndex.java
@@ −366,1 +366,1 @@
− int startPointer = 0;
+ long startPointer = 0;
Repair actions parsed by GumTree:
```
UPD VariableDeclarationStatement
    UPD Type@@"int" to "long"
```

Fig. 14: An integer-overflow bug taken from project `lucene` is fixed by modifying the variable data type.

patch prioritization. For example, a fix may break the backward compatibility of client applications. Figure 13 illustrates an example of a modifier-related patch in the project `Apache ant` [51], which however leads to a new problem that the value of systemClasspath cannot be re-assigned in Eclipse, so that it breaks the Eclipse integration [52]. Therefore, *fixing modifier-related bugs should consider the backward compatibility of software*.

Patches for "Type" nodes: *Type* code entities can also be buggy, and their fix ingredients may be specific. In this study, *Type* refers to the data type in the code, such as "int" in Figure 14. Changes applied to *Type* nodes account for 8.7% cases of repair actions. The bug in Figure 14 is an integer-overflow bug taken from project `lucene` and fixed by replacing the data type "int" with "long".

Insight 6: Theoretically, repair of such traditional programming bugs can be performed readily by APR tools when key context information is available. For example, if the nature of the bug (e.g., "*integer-overflow*") is known, the APR tool could attempt, as one of its fix rules/templates, to replace the type "int" with "long" that has a bigger memory size. If GenProg or PAR could learn repair actions from this kind of patches to mutate the type nodes of bugs but not the whole statements, which could fix similar bugs like Math_30 and Math_57 in Defects4J that have not been fixed by these tools. *Nevertheless, the challenge arises for APR tools when the buggy type is a specific data type, which requires more precise context information.*

Patches for "Identifier" nodes: Identifiers are also impacted by patches, and naming an appropriate identifier is not an easy task. Similarly, changes applied to *identifiers* involve in 5.5% cases of repair actions. Changes on identifiers are generally about assigning appropriate names to identifiers to avoid confusion or inadequate usages which often complicate maintenance tasks or even lead to bugs [53]–[55]. Thus, we can enumerate two ways in which repair actions are applied:

1) *Modifying identifiers to satisfy naming convention*, as in patch A of Figure 15, where the old identifier of field 'random' is changed to 'RANDOM' by re-writing it with upper-case letters since constant names should be in upper-case letters recommended by Java naming conventions [56].
2) *Modifying inconsistent identifiers*, as in patch B of Figure 15, where the old variable name "result" is replaced with a new name "recommendations" which seems to be more consistent and easier to track during maintenance.

A: Example of incorrect naming convention identifier changes:
Commit ad2817beb235f8f24b7e73feac2ad717346bcd6f
core/src/main/java/org/apache/mahout/clustering/dirichlet/UncommonDistributions.java
@@ −31, 1 +31, 1 @@
− private static final Random random = RandomUtils.getRandom();
+ private static final Random RANDOM = RandomUtils.getRandom();
Repair actions parsed by GumTree:
```
UPD FieldDeclaration
    UPD SimpleName@@"random" to "RANDOM"
```

B: Example of inconsistent identifier changes:
Commit c3154b86dce55f8ca318c35f97751d3aeee415aa (MAHOUT−1151)
core/src/main/java/org/apache/mahout/cf/taste/hadoop/als/RecommenderJob.java
@@ −121, 1 +124,1 @@
− private RecommendedItemsWritable result =
+ private RecommendedItemsWritable recommendations =
new RecommendedItemsWritable();
Repair actions parsed by GumTree:
```
UPD FieldDeclaration
    UPD SimpleName@@"result" to "recommendations"
```

Fig. 15: Two identifier changes taken from project `mahout`.

This kind of changes may be questioned as bug fixes, but we find some of them are linked to bug reports (e.g., `MAHOUT−1151` in Figure 15). Naming things is the hardest task that programmers have to do [57], thus it is inevitable to generate bugs because of inconsistent identifiers [53], [54]. FindBugs also enumerates 10 bug types related to identifiers. So far, a number of research directions related to identifiers in code have been explored in the literature: Høst and Østvold [58] used name-specific implementation rules and certain semantic profiles of method implementations to find and fix method naming bugs, but limited to method names starting with "contain" or "find". Kim et al. [59] relied on a custom code dictionary to detect inconsistent identifiers. Allamanis et al. [60]–[62] leveraged deep learning techniques to suggest identifiers for variables, methods, and classes with sub-tokens extracted from code.

Although, current research contributions have shown promising results about identifier-related studies, identifying and fixing inconsistent identifiers remains an open challenge because of their short-comings, such as inadequate context information.

Insight 7: Identifiers are the basic knowledge of code understanding, thus, more context information (e.g., method implementation should be considered to name method identifiers) should be considered to address fixing inconsistent identifiers. Changing identifiers, however, is not a trivial endeavor: it may break the backward compatibility of applications, and developers' understanding of code might be impacted by identifier changes [63]. This challenge may thus be a relevant and worthy target for APR research.

Patches for "Expression" nodes: Expression is the main fault-prone element of statements. We observe that *Expressions* are concerned by 82% cases of repair actions. Statements in Java program are generally built based on various expressions whose values eventually determine the execution behavior. It is reasonable that most bugs are associated with expressions. Therefore, it does not come as a surprise that the majority of repair actions in patches are performed to mutate expressions. As 35 different expression types are defined in Eclipse JDT APIs, and many of them can be decomposed in several elements, we will take a close look at their repair actions in more details in following sections.

C. RQ3: Buggy Expressions and Associated Repair Actions

We further investigate which kinds of expressions are recurrently impacted by repair actions on code statements by retrieving the sub-trees of buggy statements to find the exact buggy expressions. For example, in the AST sub-tree (illustrated in Figure 3) of the buggy statement in Figure 1, the InfixExpression "FastMath.pow(2 * FastMath.PI, -dim / 2) * FastMath.pow(covarianceMatrixDeterminant, -0.5) * getExponentTerm(vals)" is impacted by this patch. With further parsing, the MethodInvocation "FastMath.pow(2 * FastMath.PI, -dim / 2)" is the more exact expression impacted by this patch than its parent infix-expression. Finally, we can find that the exact buggy expression is the InfixExpression "-dim / 2". All hierarchical expressions (i.e., InfixExpression → MethodInvocation → InfixExpression), eventually leading to the exact buggy code "-dim / 2", are obtained by looking closely into the AST sub-tree of the buggy statement.

The distributions of expression types impacted by patches are presented in Figure 16. Due to space limitation, Figure 16 only lists up top-5 expression types. The remaining are summed in an "Others" category.

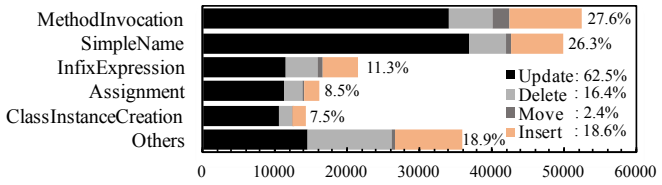

Fig. 16: Distributions of repair actions at the expression level.

Repair actions of recurrently impacted expressions: A small number of expression types are recurrently impacted by patches. It is noteworthy that the 5 out of 35 expression types (namely MethodInvocation, SimlpeName, InfixExpression, Assignment, and ClassInstanceCreation) account for ∼81% cases of repair actions at the expression level. In particular, repair actions on MethodInvocation and SimpleName account for more than half of repair action cases. In this study, MethodInvocation expressions are method references, and SimpleName expressions denote variable names and method names. Their presence indicates that incorrect references to methods and variables are the main cause of many bugs.

Insight 8: A small number of expression types are recurrently impacted by real-world patches, which can motivate to generate mutations with mutation-based APR tools (e.g., GenProg) and to mine fix patterns for a specific type of expressions with pattern-based APR tools (e.g., PAR).

For example, in Figure 1, the exact buggy expression is an InfixExpression: "-dim / 2", and is fixed by replacing it with another InfixExpression: "-0.5 * dim". It is known that "1.0 / 2 = 0.5" can represent the relationship between the deleted NumbeLiteral "2" and the inserted NumberLiteral "0.5", further inferred that "-1.0 / 2 * dim" is a function-identical mutation of the patch code "-0.5 * dim". With the following inferring process, it is easy to extract a fix pattern for value-truncated bugs at the expression level beyond the limitation of statement types.

$$dim/2 \;\rightarrow\; 0.5 * dim \;\rightarrow\; 1.0/2 * dim$$

$$\Rightarrow Pattern: a/b \rightarrow 1.0/b * a, (a: dividend, b: divisor)$$

However, it is difficult to mine fix patterns only with the buggy SimpleName expressions since they capture less useful characteristics. For example, Figure 17 shows a bug is fixed by modifying the buggy SimpleName is and SimpleName os that are meaningless or could be any identifiers, so that it is difficult to extract distinguishing characteristics from them. Therefore, *if mining fix patterns from patches involving* SimpleName *expression changes, more context information (such as its method reference "*copyBytes*") should be considered.*

Commit 37ecb1c20f0ed36e7c438d265b0c30a282e4fff5
lucene/core/src/java/org/apache/lucene/store/Directory.java
@@ −200, 1 +200, 1 @@
− is.copyBytes(os, is.length());
+ os.copyBytes(is, is.length());
Repair actions parsed by GumTree:

```
UPD ExpressionStatement@@"is.copyBytes(os,is.length());"
    UPD MethodInvocation@@"is.copyBytes()"
        UPD SimpleName@@"is" to "os"
            UPD SimpleName@@"os" to "is"
```

Fig. 17: Bug LUCENE-4377 is fixed by modifying the wrong SimpleName **expressions "is" and "os".**

Rarely impacted expressions: There are expression entities rarely changed by patches. It is also noteworthy that there are very few cases (less than 0.05%, 100 cases) of repair actions involving LambdaExpression, CharacterLiteral, TypeLiteral, Annotation and SuperFieldAccess expressions. Our data also includes no repair action case impacting MethodReference (i.e., CreationReference, ExpressionMethodReference, SuperMethodReference, and TypeMethodReference). In the case of LambdaExpression (1,138 cases) and MethodReference (120 cases), we understand that they have been introduced in Java 8 [64], thus they are not yet involved in bugs from our dataset. *It implies that APR tools could ignore such expressions when fixing bugs.*

Repair actions of literal expressions: Literal expressions can also lead to bugs, and their repair actions could be specific. Table II presents the distributions of repair actions on buggy

TABLE II: Distributions of repair actions on buggy literal expressions.

Expressions	Updated	Deleted or replaced by other expressions
BooleanLiteral	12%	88%
CharacterLiteral	46%	54%
NumberLiteral	65%	35%
StringLiteral	62%	38%

TABLE III: Distribution of *whole* vs. *sub-element* changes in buggy expressions.

Expression	Quantity	% whole expression	% each sub-element		
ArrayAccess	1,127	47.7%	ArrayExp(35.4%)	ArrayIndex(20.6%)	
ArrayCreation	740	27.3%	ArrayType(14.2%)	Initializer(60.9%)	
Assignment	13,762	18.1%	Left_Hand_Expression(13.3%)	Operator(0.8%)	Right_Hand_Expression(73.5%)
CastExpression	2,192	45.8%	Type(11.9%)	Expression(42.9%)	
ClassInstanceCreation	12,385	15.5%	Expression(10.2%)	ClassType(19.7%)	Arguments(63.0%)
ConditionalExpression	882	22.9%	Condition_Expression(24.1%)	Then_Expression(33.0%)	Else_Expression(49.5%)
FieldAccess	568	57.2%	Expression(9.2%)	Field(35.9%)	
InfixExpression	15,896	27.3%	Left_Hand_Expression (35.0%)	Operator(5.6%)	Right_Hand_Expression(68.7%)
InstanceofExpression	371	55.5%	Expression(16.7%)	Type(30.5%)	
MethodInvocation	40,054	14.7%	MethodName(22.1%)	Arguments(79.8%)	
PostfixExpression	512	85.2%	Expression (14.6%)	Operator(0.8%)	
PrefixExpression	1,362	50.0%	Operator (0.1%)	Expression (49.9%)	
QualifiedName	4,567	48.7%	QualifiedName (10.0%)	Identifier(48.9%)	
VariableDeclarationExpression	676	67.3%	Modifier (32.7%)		

† "% whole expression" indicates the percentage in which the whole buggy expression is replaced by another expression or removed directly. "% sub-elements" represents the percentage in which one or more sub-elements of an expression are changed instead of the whole expression. For each expression type, the sum of percentages may not be 100% since sub-expressions in the third column can be overlapped among each other. For example, for the *ArrayAccess* expression, the sum percentage of *ArrayExp* and *ArrayIndex* is 81.8% in linked patches that is over 74.7% (100%−35.6%), which indicates that both *ArrayExp* and *ArrayIndex* of some buggy *ArrayAccess* expressions are changed simultaneously in same bug fixes. The same as other expressions.

Commit ae4734f4cfc17453f8d5889a08ae90bb6d3601b7
solr/core/src/java/org/apache/solr/util/SimplePostTool.java
@@ −778, 1 +778, 1 @@
− if (type.equals("text/xml")
+ if (type.equals("application/xml")
|| type.equals("text/csv") || type.equals("application/json")) {
Repair actions parsed by GumTree:
```
UPD IfStatement@@"If"
    UPD InfixExpression@@"InfixExp_code"
        UPD MethodInvocation@@"type.equals()"
            UPD StringLiteral@@"text/xml" to
                                "application/xml"
```

Fig. 18: Patch of fixing bug SOLR-6959 (StringLiteral-related bug).

literal expressions. Recurrent repair actions on BooleanLiteral expressions are mostly deleting them or replacing them with other types of expressions. We note that in only a few cases, the repair action switches "true" and "false" booleans. In the case of buggy CharacterLiteral expressions, the related repair actions are balanced on updating the literal values and deleting them or replacing them with other types of expressions. Both buggy NumberLiteral and StringLiteral expressions have similar distributions of repair actions. Ratios of updating the buggy values are slightly higher than other repair actions. StringLiteral related bugs can be very specific, such as the bug in Figure 18, thus, *more specific context information or fix ingredients are needed to fix literal expression related bugs, which arises a new height challenge for APR tools.*

D. RQ4: Fault-prone Parts in Expressions

In this section, we further investigate the distributions of buggy sub-elements of expressions. Our investigation results are provided in Table III. The first column of Table III enumerates different expressions types. The second column represents the percentage in which each expression is replaced or deleted as a whole. An expression can be further decomposed into several sub-elements. For example, an InfixExpression consists of a left-hand expression, an infix operator, and a right-hand expression. The third column shows the percentage in which each sub-element is changed.

Faulty parts of expressions: Not all parts of the expressions are completely faulty, but some specific sub-elements are the exact buggy parts. As shown in Table III, there are

Commit 44854912194177d67cdfa1dc765ba684eb013a4c
src/main/java/org/apache/commons/lang3/time/FastDateParser.java
@@ −895, 1 +895, 1 @@
− final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase());
+ final TimeZone tz = TimeZone.getTimeZone(value.toUpperCase(
Locale.ROOT));
Repair actions parsed by GumTree:
```
UPD VariableDeclarationStatement@@"code"
   UPD VariableDeclarationFragment@@"tz"
      UPD MethodInvocation@@"TimeZone.getTimeZone()"
         UPD MethodInvocation@@"value.toUpperCase()"
            INS QualifiedName@@"Locale.ROOT" to
                MethodInvocation
```

Fig. 19: Patch of bug LANG-1357 fixed by adding the parameter "Locale.ROOT" into the MethodInvocation: "toUpperCase()".

different percentages of fault-prone parts for each expression type, which provides an abundant resource of learning fix behavior for various specific bugs. For example, the whole buggy expressions could improve APR tools by reducing search space to find fix ingredients by combining their parent statement types, such as the bug fix shown in Figure 9 and Insight 3.

Insight 9: The statistics can support to categorize bug fixes, mine fix patterns mining, or reduce search space at expression level with common distinguishing characteristics (such as non-faulty parts of expressions) to tune APR tools.

For example, the exact buggy entity in Figure 19 is the MethodInvocation "value.toUpperCase()", which can cause i18n issues [65] because of the missing parameter "Locale.ROOT". With the corresponding repair actions, an executable fix pattern (as below) can be extracted. Method name "toUpperCase" can also be the specific constraints to search fix ingredients for i18n issues [65].

$$str.toUpperCase() \rightarrow str.toUpperCase(Locale.ROOT)$$

Non-recurrent faulty operators: Faulty operators are not recurrent in real-world bugs. It is noteworthy that repair actions on operators only account for 0.7% of all repair actions for expressions. Specifically, fixing operators only account for 0.8% cases in buggy Assignment expressions, 5.6% cases in InfixExpressions, 0.8% cases in PostfixExpressions, and 0.1% cases in PrefixExpressions. *It implies that when APR tools*

generate mutations to fix bugs, they should focus on non-operator code entities of potential buggy code.

VI. Threats to validity

A threat to validity is the complexity of patches. Patches could involve updating MethodDeclarations, and most repair actions on MethodDeclarations (except for repair actions on its *Modifier* and *Identifier*) lead to the changes of method bodies, which further complicates accurate modeling or learning of the repair actions. Patches about adding new methods or code files, multi-hunk changes or several files would challenge fix behavior learning and pattern mining. To reduce this threat, we select patches with small size hunks. Threats to validity also include the limitation of identifying bug fix commits. To reduce this threat, our study collects bug-fixing commits in two different ways.

VII. Related Work

Bug fix commits study: Various studies have mined software repositories to analyze commits [66]–[69]. Purushothaman and Perry [70] studied patch-related commits in terms of sizes of bug fix hunks and repair action types to investigate the impact of small source code changes. German [71] analyzed the characteristics of *modification records* (i.e., source code changes in the version control system of software) from three aspects: authorship, the number of files, and modification coupling of files. Alali et al. [72] analyzed the relationships among three size metrics (# of files, # of lines, and # of hunks) for commits to infer the characteristics of commits from years of historical information. Yin et al. [73] presented a comprehensive characteristic study on incorrect bug-fixes which are figured out by tracking the revision history of each patch, and showed that bug fixes could further cause new bugs. Thung et al. [74] performed a study on real faults to investigate whether bugs are localizable by extracting faults from code changes manually. Their results showed that most faults are not within small code hunks. Nguyen et al. [75] studied the recurrent code changes and found that repetitiveness is common in bug fix hunks with small size. Eyolfson et al. [76] investigated the relationship between time-based characteristics of commits and their bugginess, of which results showed that the bugginess of a commit is correlated with the commit time. However, these studies did not investigate the links between the nature of bug fixes and automatic program repair, which is analyzed in this study.

Patches study: Pan et al. [24] manually explored 27 common bug fix patterns in Java programs to understand how developers change code to fix bugs. Martinez et al. [7] and Zhong et al. [21] analyzed the repair actions of patches at the statement level to understand the nature of bugs and patches. Although these studies provide interesting insights into program repair, they could be misleading for implementing automated repair actions because of the coarse-grained level of statements. As listed in Table IV, the three studies focus on statement level to investigate patches. Indeed, as investigated in this study, buggy parts can be localized in a more fine-grained way, which could lead to more accurate repair actions. Last but not least, moving buggy statement is also an effective way of fixing bugs, which is, however, ignored by them.

TABLE IV: Comparison of our work with other previous real-world patch studies.

Patch study	Granularity of code entities	Granularity of change operators
Pan et al. [24]	Statement level.	Abstract patterns.
Martinez et al. [7]	Statement level and method invocations.	Update, delete, and insert.
Zhong et al. [21]	Statement level.	Modify, add, and delete.
Our work	All AST node code entities impacted by patches.	Update, delete, move, and insert.

Program repair with real-world patches: Kim et al. [39] proposed PAR which utilizes common fix patterns to automatically fix bugs. Le et al. [11] extended PAR by automatically mining bug fixes across projects in their commit history to guide and drive a program repair. Bissyande [77] considered also investigating fix hints for reported bugs. Tan et al. [78] analyzed anti-patterns that may interfere with the process of automated program repair. Koyuncu et al. [79] investigated the practice of patch construction to study the impact of different patch generation techniques in Linux kernel development. Long et al. [14] proposed a new system, Genesis, that processes patches to automatically infer code transforms for automated patch generation. These studies obtained promising results, but they have a common limitation that focuses on statement level but not as the finer granularity at expression level investigated in this study.

VIII. Conclusion

Real-world patches can provide useful information (e.g., on repair actions) for learning-based and template-driven automated program repair techniques, allowing for fast generation of correct patches. In general, we argue that towards boosting the performance of automated program repair techniques, the community needs to deepen its knowledge on bug fix code transformations from real-world (i.e., human-written) patches. In this study, we engaged in this endeavor through a systematic and fine-grained investigation of 16,450 bug fix-related commits collected from seven open source Java projects. We find that there are opportunities for APR techniques to be targeted at code elements that have not yet been investigated. We also find that a small number of statement and expression types are recurrently impacted by real-world patches, and expression-level granularity could reduce search space of finding fix ingredients for similar bugs. We further discuss nine insights into tuning APR tools, challenges and possible resolves through investigating research questions around the actual locations of buggy code and repair actions at the AST level.

Acknowledgements This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOMMEND C15/IS/10449467, FIXPATTERN C15/IS/9964569.

REFERENCES

[1] NIST, "Software errors cost u.s. economy $59.5 billion annually," http://www.abeacha.com/NIST_press_release_bugs_cost.htm, Last Accessed: Mar. 2018.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[3] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze, "Automatic recovery from runtime failures," in *Proceedings of the International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, 2013, pp. 782–791.

[4] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *Proceedings of the International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, 2013, pp. 792–801.

[5] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 306–317.

[6] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Lincoln, NE, USA: IEEE, 2015, pp. 295–306.

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

[8] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. Florence, Italy: IEEE, 2015, pp. 448–458.

[9] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo, Italy: ACM, 2015, pp. 166–178.

[10] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA: IEEE, 2016, pp. 428–432.

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

[12] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. Urbana, IL, USA: IEEE, 2017, pp. 637–647.

[13] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. Paderborn, Germany: ACM, 2017, pp. 593–604.

[14] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. Paderborn, Germany: ACM, 2017, pp. 727–739.

[15] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le 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.

[16] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering, May 16-24,*. Vancouver, Canada: IEEE, 2009, pp. 364–374.

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, 2013, pp. 772–781.

[18] 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*. Buenos Aires, Argentina: IEEE, 2017, pp. 416–426.

[19] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, 2013, pp. 802–811.

[20] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg, FL, USA: ACM, 2016, pp. 298–312.

[21] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, ICSE*, vol. 1. Florence, Italy: IEEE, 2015, pp. 913–923.

[22] M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India - May 31 - June 07: ACM, 2014, pp. 234–242.

[23] ——, "Automatic software repair: a bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 17:1–17:24, 2017.

[24] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[25] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*. Vasteras, Sweden: ACM, 2014, pp. 313–324.

[26] Eclipse, "Eclipse jdt api," http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/overview-summary.html, Last Access: Mar. 2018.

[27] ——, "Statement," http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Statement.html, Last Access: Mar. 2018.

[28] ——, "Expression," http://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Expression.html, Last Access: Mar. 2018.

[29] J.-R. Falleri, "Gumtree," https://github.com/GumTreeDiff/gumtree, Last Access: Mar. 2018.

[30] Eclipse, "Java model," http://www.vogella.com/tutorials/EclipseJDT/article.html, Last Access: Mar. 2018.

[31] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Long Beach, CA, USA: ACM, 2005, pp. 273–282.

[32] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, PRDC'06.* University of California, Riverside, USA: IEEE, 2006, pp. 39–46.

[33] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, p. 11, 2011.

[34] T. Janssen, R. Abreu, and A. J. Van Gemund, "Zoltar: a spectrum-based fault localization tool," in *Proceedings of the ESEC/FSE workshop on Software integration and evolution@ runtime*. Amsterdam, The Netherlands: ACM, 2009, pp. 23–30.

[35] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[36] Apache, "Apache projects," http://www.apache.org/index.html#projects-list, Last Access: Mar. 2018.

[37] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance*. San Jose, California, USA: IEEE, 2000, pp. 120–130.

[38] M. Frigge, D. C. Hoaglin, and B. Iglewicz, "Some implementations of the boxplot," *The American Statistician*, vol. 43, no. 1, pp. 50–54, 1989.

[39] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*. San Francisco, CA, USA: IEEE, 2013, pp. 121–130.

[40] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*. Waikiki, Honolulu , HI, USA: ACM, 2011, pp. 351–360.

[41] JIRA, "Math-927," https://issues.apache.org/jira/browse/MATH-927, Last Access: Mar. 2018.

[42] 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*. San Jose, CA, USA: ACM, 2014, pp. 437–440.

[43] Defects4J, "Math-12," http://program-repair.org/defects4j-dissection/\#!/bug/Math/12, Last Access: Mar. 2018.

[44] Eclipse, "Modifier," https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Modifier.html, Last Access: Mar. 2018.

[45] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[46] Apache, "Commons-io," https://github.com/apache/commons-io/blob/master/pom.xml\#L373, Last Accessed: Mar. 2018.

[47] ——, "Commons-lang," https://github.com/apache/commons-lang/blob/master/pom.xml\#L685, Last Accessed: Mar. 2018.

[48] ——, "Commons-math," https://github.com/apache/commons-math/blob/master/pom.xml\#L717, Last Accessed: Mar. 2018.

[49] ——, "mahout," https://github.com/apache/mahout/blob/master/pom.xml\#L771, Last Accessed: Mar. 2018.

[50] Defects4J, "Defects4j dissection," http://program-repair.org/defects4j-dissection/\#!/, Last Access: Mar. 2018.

[51] Apache, "Ant," http://ant.apache.org/, Last Accessed: Mar. 2018.

[52] Bugzilla, "Bug 60582 - change to systemclasspath breaks eclipse integration," https://bz.apache.org/bugzilla/show_bug.cgi?id=60582, Last Access: Mar. 2018.

[53] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *Proceedings of 16th Working Conference on Reverse Engineering, WCRE'09*. Lille, France: IEEE, 2009, pp. 31–35.

[54] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc, "Can lexicon bad smells improve fault prediction?" in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*. Kingston, Ontario, Canada: IEEE, 2012, pp. 235–244.

[55] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Name suggestions during feature identification: the variclouds approach," in *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 2016, pp. 119–123.

[56] Oracle, "Java naming convention," http://www.oracle.com/technetwork/java/codeconventions-135099.html, Last Access: Mar. 2018.

[57] P. Johnson, "Don't go into programming if you don't have a good thesaurus," http://www.itworld.com/article/2823759/enterprise-software/124383-Arg-The-9-hardest-things-programmers-have-to-do.html, Last Accessed: Dec. 2017.

[58] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP*. Genoa, Italy: Springer, 2009, pp. 294–317.

[59] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Empirical Software Engineering*, vol. 21, no. 2, pp. 565–604, 2016.

[60] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 281–293.

[61] ——, "Suggesting accurate method and class names," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. Bergamo, Italy: ACM, 2015, pp. 38–49.

[62] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the International Conference on Machine Learning*. New York City, NY, USA: JMLR.org, 2016, pp. 2091–2100.

[63] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018.

[64] Oracle, "What's new in jdk 8," http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html, Last Access: Mar. 2018.

[65] G. Gregory, "The java lowercase conversion surprise in turkey," https://garygregory.wordpress.com/2015/11/03/java-lowercase-conversion-turkey/, Last Access: Mar. 2018.

[66] L. Li, T. F. Bissyandé, Y. Le Traon, and J. Klein, "Accessing inaccessible android apis: An empirical study," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 411–422.

[67] D. Li, L. Li, D. Kim, T. F. Bissyandé, D. Lo, and Y. L. Traon, "Watch out for this commit! a study of influential software changes," *arXiv preprint arXiv:1606.03266*, 2016.

[68] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. Le Traon, "Facoy - a code-to-code search engine," in *The 40th International Conference on Software Engineering (ICSE 2018)*, 2018.

[69] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android apis," in *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[70] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.

[71] D. M. German, "An empirical study of fine-grained software modifications," *Empirical Software Engineering*, vol. 11, no. 3, pp. 369–393, 2006.

[72] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC*. Amsterdam, The Netherlands: IEEE, 2008, pp. 182–191.

[73] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Szeged, Hungary: ACM, 2011, pp. 26–36.

[74] F. Thung, D. Lo, L. Jiang *et al.*, "Are faults localizable?" in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. Zurich, Switzerland: IEEE, 2012, pp. 74–77.

[75] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. Silicon Valley, CA, USA: IEEE, 2013, pp. 180–190.

[76] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1009–1039, 2014.

[77] T. F. Bissyandé, "Harvesting fix hints in the history of bugs," *arXiv preprint arXiv:1507.05742*, 2015.

[78] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle, WA, USA: ACM, 2016, pp. 727–738.

[79] A. Koyuncu, T. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Impact of Tool Support in Patch Construction," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2017, pp. 237–248.