This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

Mining Fix Patterns for FindBugs Violations

Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon

Abstract—Several static analysis tools, such as Splint or FindBugs, have been proposed to the software development community to help detect security vulnerabilities or bad programming practices. However, the adoption of these tools is hindered by their high false positive rates. If the false positive rate is too high, developers may get acclimated to violation reports from these tools, causing concrete and severe bugs being overlooked. Fortunately, some violations are actually addressed and resolved by developers. We claim that those violations that are recurrently fixed are likely to be true positives, and an automated approach can learn to repair similar unseen violations. However, there is lack of a systematic way to investigate the distributions on existing violations and fixed ones in the wild, that can provide insights into prioritizing violations for developers, and an effective way to mine code and fix patterns which can help developers easily understand the reasons of leading violations and how to fix them.

In this paper, we first collect and track a large number of fixed and unfixed violations across revisions of software. The empirical analyses reveal that there are discrepancies in the distributions of violations that are detected and those that are fixed, in terms of occurrences, spread and categories, which can provide insights into prioritizing violations. To automatically identify patterns in violations and their fixes, we propose an approach that utilizes convolutional neural networks to learn features and clustering to regroup similar instances. We then evaluate the usefulness of the identified fix patterns by applying them to unfixed violations. The results show that developers will accept and merge a majority (69/116) of fixes generated from the inferred fix patterns. It is also noteworthy that the yielded patterns are applicable to four real bugs in the Defects4J major benchmark for software testing and automated repair.

Index Terms—Fix pattern, pattern mining, program repair, findbugs violation, unsupervised learning.

1 Introduction

Modern software projects widely use static code analysis tools to assess software quality and identify potential defects. Several commercial [1], [2], [3] and open-source [4], [5], [6], [7] tools are integrated into many software projects, including operating system development projects [8]. For example, Java-based projects often adopt `FindBugs` [4] or `PMD` [5] while C projects use `Splint` [6], `cppcheck` [7], or `Clang Static Analyzer` [9], while Linux driver code are systematically assessed with a battery of static analyzers such as Sparse and the LDV toolkit. Developers may benefit from the tools before running a program in real environments even though those tools do not guarantee that all identified defects are real bugs [10].

Static analysis can detect several types of defects such as security vulnerabilities, performance issues, and bad programming practices (so-called code smells) [11]. Recent studies denote those defects as **static analysis violations** [12] or **alerts** [13]. In the remainder of this paper, we simply refer to them as *violations*. Fig. 1 shows a violation instance, detected by `FindBugs`, which is a violation tagged `BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS`, as it does not comply with the programming rule that the implementation of method `equals(Object obj)` should not make any assumption about the type of its `obj` argument [14].

- *Kiu Liu, Dongsun Kim, Tegawendé F. Bissyandé, and Yves Le Traon are with the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at University of Luxembourg, Luxembourg. E-mail: {kui.liu, dongsun.kim, tegawende.bissyande, yves.letraon}@uni.lu*
- *Shin Yoo is with the School of Computing, KAIST, Daejeon, Republic of Korea. E-mail: shin.yoo@kaist.ac.kr*


```
public boolean equals(Object obj) {
    // Violation Type:
    // BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS
    return getModule().equals(
         ((ModuleWrapper) obj).getModule());
}
```

Fig. 1: Example of a detected violation, taken from *PopulateRepositoryMojo.java* file at revision `bdf3fe` in project `nbm-maven-plugin`[1].

As later addressed by developers via a patch represented in Fig. 2, the method should return `false` if `obj` is not of the same type as the object being compared. In this case, when the type of `obj` argument is not the type of `ModuleWrapper`, a `java.lang.ClassCastException` should be thrown.

```
  public boolean equals(Object obj) {
-     return getModule().equals(
-          ((ModuleWrapper) obj).getModule());
+     return obj instanceof ModuleWrapper &&
+          getModule().equals(
+          ((ModuleWrapper) obj).getModule());
  }
```

Fig. 2: Example of fixing violation, taken from Commit `0fd11c` of project `nbm-maven-plugin`.

Despite wide adoption and popularity of static analysis tools (e.g., FindBugs has more than 270K downloads[2]), accepting the results of the tools is not yet guaranteed. Violations identified by static analysis tools are often ignored by developers [15], since static analysis tools may

[1]https://github.com/mojohaus/nbm-maven-plugin
[2]http://findbugs.sourceforge.net/users.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

yield high rates of **false positives**. Actually, a (false positive) violation might be (1) not a serious enough concern to fix, (2) less likely to occur in a runtime environment, or (3) just incorrectly identified due to the limitations of the tool. Depending on the context, developers may simply give up on the use of static analysis tools or they may try to prioritize violations based on their own criteria.

Nevertheless, we can regard a violation as **true positive** if it is recurrently removed by developers through source code changes as in the example of Fig. 2. Otherwise, a violation can be considered as ignored (i.e., not removed during revisions) or disappearing (a file or program entity is removed from a project) instead of being fixed. We investigate in this study different research questions regarding (**RQ1**) *to what extent do violations recur in projects?* (**RQ2**) *what types of violations are actually fixed by developers?*(i.e., true positives) (**RQ3**) *what are the patterns of violations code that are fixed or unfixed by developers?* From this question, we can identify common code patterns of violations that could help better understand static analysis rules. (**RQ4**) *how are the violations resolved when developers make changes?* Based on this question, for each violation type, we can derive fix patterns that may help summarize common violation (or real bug) resolutions and may be applied to fixing similar unfixed violations. (**RQ5**) *can fix patterns help systematize the resolution of similar violations?* This question may shed some light on the effectiveness of common fix patterns when applying them to potential defects.

To answer the above questions, we investigate violations and violation fixing changes collected from 730 open source Java projects. Although the approach is generic to any static bug detection tool, we focus on a single tool, namely `FindBugs`, applying it to every revision of each project. We thus identify violations in each revision and further enumerate cases where a pair of consecutive revisions involve the resolution of a violation through source code change (i.e., the violation is found in revision r_1 and is absent from r_2 after a code change can be mapped to the violation location): we refer to such recorded changes as *violation fixing changes*. We further conduct empirical analyses on identified violations and fixed violations to investigate their recurrences, their code patterns, etc.

After collecting violation fixing changes from a large number of projects using an AST differencing tool [16], we mine developer fix patterns for static analysis violations. The approach encodes a fixing change into a vector space using Word2Vec [17], extracts discriminating features using Convolutional Neural Networks (CNNs) [18] and regroups similar changes into a cluster using *X-means* clustering algorithm [19]. We then evaluate the suitability of the mined fix patterns by applying them to 1) a subset of unfixed violations in our subjects, to 2) a subset of faults in Defects4J [20] and to 3) a subset of violations in 10 open source Java projects.

Overall, this paper makes the following contributions:

1) **Large-scale dataset of static analysis violations:** we have carefully and systematically tracked static analysis violations across all revisions of a large set of projects. This dataset, which has required substantial effort to build, is available to the community in a labelled format, including the violation fixing change information.

> We release a dataset of 16,918,530 unique samples of `FindBugs` violations across revisions of 730 Java projects, along with 88,927 code changes addressing some of these violations.

2) **Empirical study on real-world management of `FindBugs`' violations:** our study explores the nature of violations that are widespread across projects and contrasts the recurrence of developer (non)fixes for specific categories, providing insights for prioritization research to limit deterrence due to overwhelming false positives, thus contributing towards improving tool adoption.

> Our analyses reveal cases of violations that appear to be systematically ignored by developers, and violation categories that are recurrently addressed. The pattern mining of violation code further provides insights into how violations can be prioritized towards enabling static bug detection tools to be more adopted.

3) **Violation fix pattern mining:** we propose an approach to infer common fix patterns of violations leveraging CNNs and *X-means* clustering algorithm. Such patterns can be leveraged in subsequent research directions such as automated refactoring tools (for complying with project rules as done by checkpatch[3,4] in the Linux kernel development), or automated program repair (by providing fix ingredients to existing tools such as PAR [21]).

> Mined fix patterns can be leveraged to help developers rapidly and systematically address high-priority cases of static violations. In our experiments, we showed that 40% of a sample set of 500 unfixed violations could be immediately addressed with the inferred fix patterns.

4) **Pattern-based violation patching:** we apply the fix patterns to unfixed violations and actual bugs in real-world programs. Our experiments demonstrate the potential of the approach to infer patterns that are effective which shows the potential of automated patch generation based on the fix patterns.

> Developers are ready to accept fixes generated based on mined fix patterns. Indeed out of 113 generated patches, 69 were merged in 10 open source projects. It is noteworthy that since static analysis can uncover important bugs, mined patterns can be leveraged for automated repair. Out of the 14 real-bugs in the Defects4J benchmark which can be detected with `FindBugs`, our mined fix patterns are immediately applicable to produce correct fixes for 4 bugs.

The remainder of this paper is organized as follows. We propose our study method in Section 2, describing the process of violation tracking, and the approach for mining

[3]http://tuxdiary.com/2015/03/22/check-kernel-code-checkpatch

[4]https://github.com/spotify/linux/blob/master/scripts/checkpatch.pl

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

code patterns based on CNNs. Section 3 presents the study results in response to the research questions. Limitations of our study are outlined in Section 4. Section 5 surveys related work. We conclude the paper in Section 6with discussions of future work. Several intermediary results, notably w.r.t. the statistics of violations are most detailed in the appendix.

2 Methodology

Our study aims at uncovering common code patterns related to static analysis violations and to developers' fixes. As shown in Figure 3, our study method unfolds in four steps: (1) applying a static analysis tool to collecting violations from programs, (2) tracking violations across the history of program revisions, (3) identifying fixed and unfixed violations, (4) mining common code patterns in each class of violations, and (5) mining common fix patterns in each class of fixed violations. We describe in details these steps as well as the techniques employed.

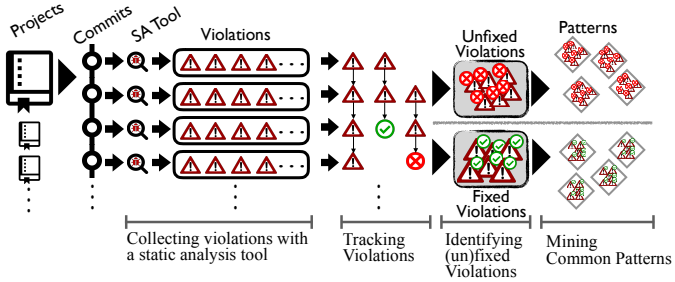

Fig. 3: Overview of our study method.

2.1 Collecting violations

To collect violations from a program, we apply a static analysis tool to every revision of the associated project's source code. Given the resource-intensive nature of this process, we focus in this study on the `FindBugs` [22] tool, although our method is applicable to other static analysis tools such as Facebook Infer[5], Google ErrorProne[6], etc. We use the most sensitive option to detect all types of violations defined in `FindBugs` violation descriptions [14]. For each individual violation instance, we record, as a six-tuple value, all information on the violation type, the enclosing program entity (e.g., project, class or method), the commit id, the file path, and the location (i.e., start and end line numbers) where the violation is detected. Figure 4 shows an example of a violation record in the collected dataset.

Since `FindBugs` requires Java bytecode rather than source code, and given that violations must be tracked across all revisions in a project, it is necessary to automate the compilation process. In this study, we accept projects that support the *Apache Maven* [23] build automation management tool. We apply maven build command (i.e., '*mvn package install*') to compiling each revision in 2014 projects that we have collected. Eventually, we were able to successfully build 730 automatically.

[5]http://fbinfer.com/
[6]https://errorprone.info/
[7]https://github.com/GWASpi/GWASpi

```
<ViolationInstance>
    <ViolationType>NP_NULL_ON_SOME_PATH</ViolationType>
    <ProjectName>GWASpi-GWASpi</ProjectName>
    <CommitVersionID>b0ed41</CommitVersionID>
    <FilePath>src/main/java/org/gwaspi/gui/reports/
            Report_AnalysisPanel.java</FilePath>
    <StartLineNumber>89</StartLineNumber>
    <EndLineNumber>89</EndLineNumber>
</ViolationInstance>
```

Fig. 4: Example record of a single-line violation of type `NP_NULL_ON_SOME_PATH` found in *ReportAnalysisPanel.java* file within Commit `b0ed41` in `GWASpi`[7] project.

2.2 Tracking violations

Violation tracking consists in identifying identical violation instances between consecutive revisions: after applying a static analysis tool to a specific revision of a project, one can obtain a set of violations. In the next version, another set of violations can be produced by the tool. If there is any change in the next revision, new violations can be introduced and existing ones may disappear. In many cases however, code changes can move violation positions, making this process a non-trivial task.

Static analysis tools often report violations with line numbers in source code files. Even when a commit modifies other lines in different source file than the location of a violation, it might be unable to use line numbers for matching identical violation pairs between two consecutive revisions. Yet, if the tracking is not precise, the identification of fixed violations may suffer from many false positives and negatives (i.e., identifying unfixed ones as fixed ones or vice versa). Thus, to match potential identical violations between revisions, our study follows the method proposed by Avgustinov et al. [24]. This method has three different violation matching heuristics when a file containing violations is changed. The first heuristic is (1) *location-based matching*: if a (potential) matching pair of violations is in code change diffs[8], it compares the offset of the corresponding violations in the code change diffs. If the difference of the offset is equal to or lower than 3, we regard the matching pair as an identical violation. When a matching pair is located in two different code snapshots, we use (2) *snippet-based matching*: if two text strings of the code snapshots (corresponding to the same type of violations in two revisions) are identical, we can match those violations. When the two previous heuristics are not successful, our study applies (3) *hash-based matching*, which is useful when a file containing a violation is moved or renamed. This matching heuristic first computes the hash value of adjacent tokens of a violation. It then compares the hash values between two revisions. We refer the reader to more details on the heuristics in [24].

There have been several other techniques developed to do this task. For example, Spacco et al. [25] proposed a fuzzy matcher. It can match violations in different source locations between revisions even when a source code file has been moved by package renaming. Other studies [26], [27] also provide violation matching heuristics based on software change histories. However, these are not precise enough to

[8]A "code change diff" consists of two code snapshots. One snapshot represents the code fragment that will be affected by a code change, and another one represents the code fragment after it has been affected by the code change.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

be automatically applied to a large number of violations in a long history of revisions [24].

2.3 Identifying fixed violations

Once violation tracking is completed, we can figure out the resolution of an individual violation. Violation resolution can result in three different outcomes. (1) A violation can disappear due to deleting a file or a method enclosing the violation. (2) A violation exists at the latest revision after tracking (even some code is changed), which indicates that the violation has not been fixed so far. (3) A violation can be resolved by changing specific lines (including code line deletion) of source code. The literature refer to the first and second outcomes as *unactionable violations* [26], [27], [28] or *false positives* [25], [29], [30] while the third one is called *actionable violations* or *true positives*. In this study we inspect violation tracking results, focusing on the second outcome (which yields the set of **unfixed violations**) and the third outcome (which yield the set of **fixed violations**).

Starting from the earliest revision where a violation is seen, we follow subsequent revisions until a later revision has no matching violation (i.e., the violation is resolved by removal of the file/method or the code has been changed). If the violation location in the source code is in a diff pair, we classify it as a *fixed violation*. Otherwise, it is an *unfixed violation*.

2.4 Mining common code patterns

Our goal in this step is to understand how a violation is induced. To achieve this goal, we mine code fragments where violations are localized and identify common patterns, not only in fixed violations but also in unfixed violations. Before describing our approach of mining common code patterns, we formalize the definition of a code pattern, and provide justifications for the techniques selected in the approach (namely CNNs [18], [31], [32] and *X-means* clustering algorithm [19]).

2.4.1 Preliminaries

Definition of code patterns: In this study, a *code pattern* refers to a generic representation of similar source code fragments. Its definition is related to the definition of a *source code entity* and of a *code context*.

Definition 1. Source Code Entity (Sce): A *source code entity* (hereafter entity) is a pair of type and identifier, which denotes a node in an Abstract Syntax Tree (AST) representation, i.e.,

$$Sce = (Type, Identifier) \tag{1}$$

where *Type* is an AST node type and *Identifier* is a textual representation (i.e., raw token) of an AST node, respectively.

Definition 2. Code Context (Ctx): A *code context* is a three-element tuple, which is extracted from a fined-grained AST subtree (see Section 2.4.2) associated to a code block, i.e.,

$$Ctx = (Sce, Sce_p, cctx) \tag{2}$$

where *Sce* is an entity and Sce_p is the parent entity of *Sce* (with $Sce_p = \emptyset$ when *Sce* is a root entity). *cctx* is a list of code contexts that are the children of *Ctx*. When *Sce* is a leaf node entity, $cctx = \emptyset$.

Definition 3. Code Pattern (CP): A *code pattern* is a three-value tuple as following:

$$CP = (Sce_a, Sce_c, cctx) \tag{3}$$

where Sce_a is a set of abstract entities of which *identifiers* are abstracted from concrete representations of specific *identifiers* that will not affect the common semantic characteristics of the code pattern. Sce_c is a set of concrete entities, of which *identifiers* are concrete, that can represent the common semantic characteristics of the code pattern. Abstract entities represent that the entities of a code pattern can be specified in actual instances while concrete entities indicate characteristics of a code pattern and cannot be abstracted. Otherwise, the code pattern will be changed. $cctx$ is a set of code contexts (See Definition 2) that are used to explain the relationships among all entities in this code pattern.

Source Code:
return (String[]) list.toArray(new String[0]);

***A Code Pattern*:**
return (T[]) var.toArray(new T[#]);

Sce_a = {(ArrayType, T[]), (Variable, var), (NumberLiteral, #)}.
Sce_c = {(ReturnStatement, return), (Method, toArray)}.

cctx= {
c_1. ((ReturnStatement, return), (null, null), $[_1$
c_2. (CastExpression, (T[])), (ReturnStatement, return), $[_2$
c_3. ((ArrayType, T[]), (CastExpression, (T[])), ∅),
c_4. ((MethodInvocation, var.toArray), (CastExpression, (T[])), $[_4$
c_5. ((Variable, var), (MethodInvocation, var.toArray), ∅),
c_6. ((Method, toArray), (MethodInvocation, var.toArray), $[_6$
c_7. ((ArrayCreation, new T[]), (MethodInvocation, var.toArray), $[_7$
c_8. ((ArrayType, T[]), (ArrayCreation, new T[]), ∅),
c_9. ((NumberLiteral, #), (ArrayCreation, T), ∅)$]_7$)$]_6$)$]_4$)$]_2$)$]_1$)
}.

CP = (Sce_a, Sce_c, *cctx*).

Fig. 5: Example representation of a code pattern.

Figure 5 shows an example of a code pattern extracted from the source code. Sce_a contains an array type entity (`ArrayType, T[]`), a variable name entity (`Variable, var`), and a number literal entity (`NumberLiteral, #`), where `T[]` is abstracted from the identifier `String[]` of (`ArrayType, String[]`), `var` is abstracted from the identifier `list` in (`Variable, list`), and identifier `#` is abstracted from the number literal `0`. The three identifiers of the three entities can also be abstracted from other related similar entities, which will not change the attributes of this pattern. Sce_c consists of a (`ReturnStatement, return`) entity and a method invocation entity (`Method, toArray`). The identifiers of the two entities cannot be abstracted, otherwise, the attributes of this pattern will be changed. If extracting code pattern from the code at the level of violated source code expression (i.e., the code pattern is `(T[]) var.toArray(new T[#])`), the (`ReturnStatement, return`) node entity can be abstracted as a null entity because this node entity will not affect this code pattern.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

$cctx$ contains a code context that explains the relationships among these entities, of which code block is a `ReturnStatement`. c_1 is the code context of the root source code entity `ReturnStatement` and consists of three values. The first one is the current Sce that contains a *Type* and an *Identifier*. The second one is the Sce_p of the current Sce which is null as Sce is a root entity. The last one is a list of code contexts which are c_1's children. It is the same as others. c_2 is the direct child of c_1. c_3 and c_4 are the direct children of c_2. The source code entity of c_3 is a leaf node entity, as a result, its child set is null. It is the same for others.

Suitability of Convolutional Neural Networks: Grouping code requires the use of discriminating code features to compute reliable metrics of similarity. While the majority of feature extraction strategies perform well on fixed-length samples, it should be noted that code fragments often consist of multiple code entities with variable lengths. A single code entity such as a method call may embody some local features in a given code fragment, while several such features must be combined to reflect the overall features of the whole code fragment. It is thus necessary to adopt a technique which can enable the extraction of both local features and the synthesis of global features that will best characterize code fragments so that similar code fragments can be regrouped together by a classical clustering algorithm. Note that the objective is not to train a classifier whose output will be some classification label given a code fragment or the code change of a patch. Instead, we adopt the idea of unsupervised learning [33] and lazy learning [34] to extract discriminating features of code fragments and patch code changes.

Recently, a number of studies [35], [36], [37], [38], [39], [40], [41] have provided empirical evidence to support the *naturalness of software* [42], [43]. A recent work by Bui et al. [44] has provided preliminary results showing that some variants of Convolutional Neural Networks (CNNs) are even effective to capture code semantics so as to allow the accurate classification of code implementations across programming languages.

Inspired by the *naturalness* hypothesis, we treat source code of violations as documents written in natural language and to which we apply CNNs to addressing the objective of feature learning. CNNs are biologically-inspired variants of multi-layer artificial neural networks [31]. We leveraged the LeNet5 [45] model, which involves lower- and upper-layers. Lower-layers are composed of alternating convolutional and subsampling layers which are *local-connected* to capture the local features of input data, while upper-layers are *fully-connected* and correspond to traditional multi-layer perceptrons (a hidden layer and a logistic regression classifier), which can synthesize all local features captured by previous lower-layers.

Choice of X-means clustering algorithm: While K-Means is a classical algorithm that is widely used, it poses the challenge of a try-and-err protocol for specifying the number K of clusters. Given that we lack prior knowledge on the approximate number of clusters which can be inferred, we rely on X-Means [19], an extended version of K-Means, which effectively and automatically estimate the value of K based on Bayesian Information Criterion.

2.4.2 Refining the Abstract Syntax Tree

In our study, code patterns are inferred based on the tokens that are extracted from the AST of code fragments, i.e., the node types and identifiers. Preliminary observations reveal that some tokens generically tagged `SimpleName` in leaf nodes can interfere feature learning of code fragments. For example, in Figure 7, the variable node *list* is presented as (`SimpleName`, `list`), and the method node `toArray` is also presented as (`SimpleName`, `toArray`) at the leaf node in the generic AST tree. As a result, it may be challenging to distinguish the two nodes from each other. Hence, a method of refining the generic AST tree is necessary to reduce such confusions.

Algorithm 1 illustrates the algorithm of refining a generic AST tree. The refined AST tree keeps the basic construct of the generic AST tree. If the label of a current node can be specified as a `SimpleName` leaf node in generic AST tree, the node will be simplified as a single-node construct by combining its discriminating grammar type and its label (i.e., identifier), and its label-related children will be removed in the refined AST tree.

Algorithm 1: Refining a generic AST tree.

Input: A generic AST tree T.
Output: A refined AST tree T_{rf}.
Function *refineAST(T)*
  r ← $T.currentNode$;
  $T_{rf}.currentNode$ ← r;
  if *r's label can be a SimpleName node* **then**
    // r's label can be specified as a SimpleName leaf node;
    Remove SimpleName-related children from r;
    Update r to (r.Type, r.Label.identifier) in T_{rf};
  foreach *child* ∈ *r.children* **do**
    $children_{rf}$.add($refineAST(child)$);
    $T_{rf}.currentNode.children$ ← $children_{rf}$;
  return T_{rf};

Figure 7 shows the models respectively of the *generic* AST tree and of the *refined* AST tree of a code fragment containing a return statement. First, the refined tree presents a simplified architecture. Second, it becomes easier to distinguish some different nodes with the refined AST tree than the generic AST tree nodes. The node of array type `String[]` is simplified as (`ArrayType`, `String[]`), the variable (`SimpleName`, `list`) is simplified as (`Variable`, `a`), and the method invocation of `toArray` is simplified as (`Method`, `toArray`). Although the method node `toArray` can be identified by visiting its parent node (i.e., `MethodInvocation`), it requires more steps to obtain this information. In the refined AST tree, the two nodes are presented as (`Variable`, `list`) and (`Method`, `toArray`) respectively. Consequently, it becomes easier to distinguish the two nodes with the refined AST tree than the generic AST tree nodes.

To understand which implementations induce static analysis violations, we design an approach for mining common code patterns of detected violations. The patterns are expected to summarize the main ingredients of code violating a given static analysis rules. This approach involves two phases: data preprocessing and violation patterns mining, as illustrated in Figure 6.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

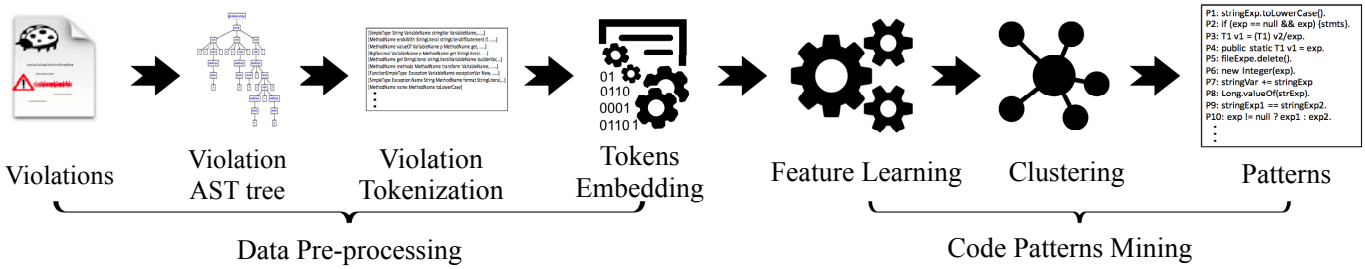

Fig. 6: Overview of our code patterns mining method.

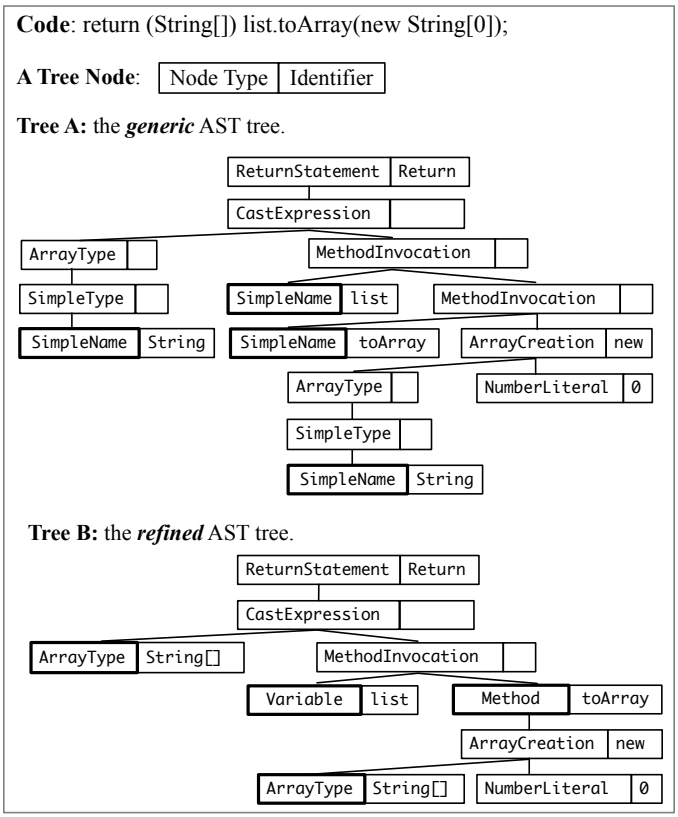

Fig. 7: Generic and Refined AST of an example code fragment.

2.4.3 Data preprocessing

`FindBugs`, reports violations by specifying the start and end lines of a code hunk which is relevant to the reported violation: this is considered as the location of the violation. It is challenging to mine common code patterns from these code hunks directly as they are just textual expression. A given violation code is therefore parsed into a refined AST tree and converted into a token vector. Token vectors are further embedded with Word2Vec [46] and converted into numeric vectors which can be fed to CNNs to learn discriminating features of violation code.

Violation tokenization

In order to represent violations with numeric vectors, in this study, violations are tokenized into textual vectors in the first step. All code hunks of violations are parsed with the refined AST tree and are tokenized into textual vectors by traversing their refined AST trees with the depth-first search algorithm to obtain two kinds of tokens: one is the AST node type and another is the identifier (i.e., raw token) of this node. For example, the code "`int a`" is tokenized as a vector of four tokens (`PrimitiveType, int, Variable, a`). A given violation is thus represented as a vector of such tokens. Noisy information of nodes (e.g., meaningless variable names such as 'a', 'b', etc.) can interfere with identifying similar violations. Thus, all variable names are renamed as the combination of their data type and string '*Var*'. For example, variable `a` in "`int a`" is renamed as `intVar`.

Token embedding with Word2Vec

Widely adopted deep learning techniques require numeric vectors with the same size as the format of input data. Tokens embedding is performed with Word2Vec [46] which can yield a numeric vector for each unique token. Eventually, a violation is then embedded as a two-dimensional numeric vector (i.e., a vector of the vectors embedding the tokens). Since token vectors may have different sizes throughout violations, the corresponding numeric vectors must be padded to comply with deep learning algorithms requirements. We follow the workaround tested by Wang et al. [47] and append 0 to all vectors to make all vector sizes consistent with the size of the longest vector.

Word2Vec[9] [48] is a two-layer neural network, whose main purpose is to embed words, i.e., convert each word into a numeric vector.

Numerical representations of tokens can be fed to deep learning neural networks or simply queried to identify relationships among words. For example, relationships among words can be computed by measuring cosine similarity of vectors, given that Word2Vec strives to regroup similar words together in the vector space. Lack of similarity is expressed as a 90-degree angle, while complete similarity of 1 is expressed as a 0-degree angle. For example, in our experiment, 'true' and 'false' are `boolean` literal in Java. There is a cosine similarity of 0.9433768 between 'true' and 'false', the highest similarity between 'true' and any other token.

The left side of Figure 8 shows how a violation is vectorized. The $n \times k$ represents a two-dimensional numeric vector of an embedded and vectorized violation, where n is the number of rows and denotes the size of the token vector of a violation. A row represents a numeric vector of an embedded token. k is the number of columns and denotes the size of a one-dimensional numeric vector of an embedded token. The last two rows represent the appended 0 to make all numeric vector sizes consistent.

[9]https://code.google.com/archive/p/word2vec/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

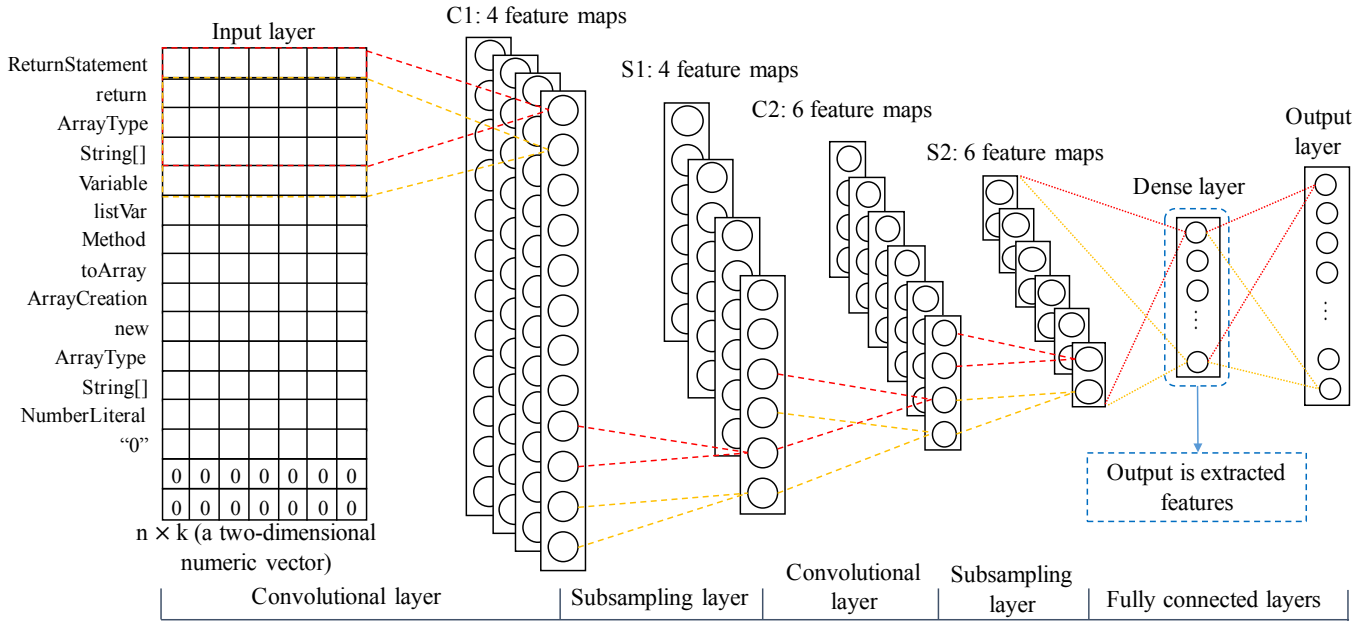

Fig. 8: CNN architecture for extracting clustering features. *C1* is the first convolutional layer, and *C2* is the second one. *S1* is the first subsampling layer, and *S2* is the second one. The output of dense layer is considered as extracted features of code fragments and will be used to do clustering.

2.4.4 Code Patterns Mining

Although violations can be parsed and converted into two-dimensional numeric vectors, it is still challenging to mine code patterns given that noisy information (e.g., specific meaningless identifiers) can interfere with identifying similar violations. Deep learning has recently been shown promising in various software engineering tasks [18], [47], [49]. In particular, it offers a major advantage of requiring less prior knowledge and human effort in feature design for machine learning applications. Consequently, our method is designed to deeply learn discriminating features for mining code patterns of violations. We leverage CNNs to perform deep learning of violation features with embedded violations, and also use *X-means* clustering algorithm to cluster violations with learned features.

Feature learning with CNNs

Figure 8 shows the CNNs architecture for learning violation features. The input is two-dimensional numeric vectors of preprocessed violations. The alternating local-connected convolutional and subsampling layers are used to capture the local features of violations. The dense layer compresses all local features captured by former layers. We select the output of the dense layer as the learned violation features to cluster violations. Note that our approach uses CNNs to extract features of violation code fragments, in contrast to normal supervised learning applications that classify labels with training process to show patterns clearly.

Violations Clustering and Patterns Labelling

With learned features of violations, cluster violations with *X-means* clustering algorithm. In this study, we use Weka's implementation [50] of *X-means* to cluster violations. Finally, we manually label each cluster with identified code patterns of violations from clustered similar code fragments of violations to show patterns clearly. Note that, the whole process of mining patterns is automated.

2.5 Mining Common Fix Patterns

Our goal in this step is to summarize how a violation is resolved by developers. To achieve this goal, we collect violation fixing changes and proceed to identify their common fix patterns. The approach of mining common fix patterns is similar to that of mining common code patterns. The differences lie in the data collection and tokenization process. Before describing our approach of mining common fix patterns, we formalize the definitions of patch and fix pattern.

2.5.1 Preliminaries

A patch represents a modification carried on a program source code to repair the program which was brought to an erroneous state at runtime. A patch thus captures some knowledge on modification behavior, and similar patches may be associated with similar behavioral changes.

Definition 4. Patch (P): A *patch* is a pair of source code fragments, one representing a buggy version and another as its updated (i.e., bug-fixing) version. In the traditional GNU diff representation of patches, the buggy version is represented by lines starting with −, while the fixed version is represented by lines starting with +. A patch is formalized as:

$$P = (Frag_b, Frag_f) \tag{4}$$

where $Frag_b$ and $Frag_f$ are fragments of buggy/fixing code, respectively; both are a set of text lines. Either of the two sets can be an empty set but cannot be empty simultaneously. If $Frag_b = \emptyset$, the patch purely adds a new line(s) to fix a bug. On the other hand, the patch only

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

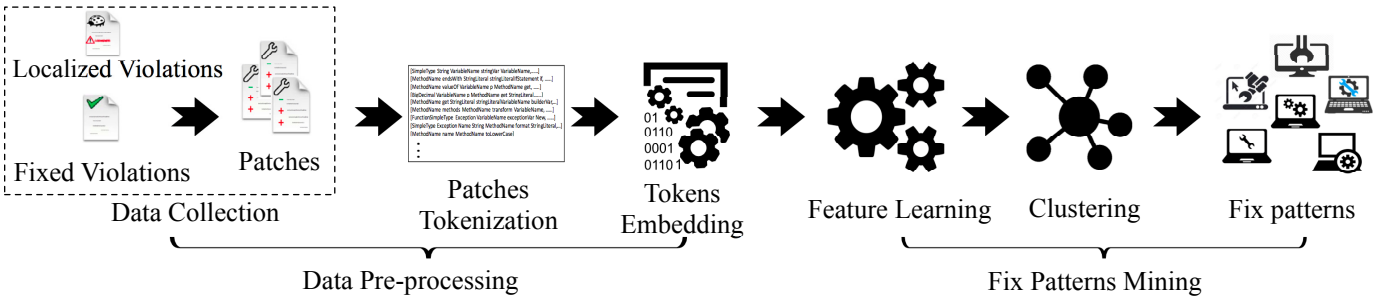

Fig. 9: Overview of our fix patterns mining method.

removes a line(s) if $Frag_f = \emptyset$. Otherwise (i.e., both sets are not empty), the patch replaces at least one line.

Figure 11 shows an example of a patch which fixes a bug of converting a `String List` into a `String Array`. $Frag_b$ is the line that starts with – while $Frag_f$ is the lines that start with +.

By analyzing the differences between the buggy code and the fixing code of the patch in Figure 11, the patch can be manually summarized as an abstract representation shown in Figure 12 which could be used to address similar bugs. Abstract representation indicates that specific identifiers and types are abstracted from concrete representation.

Abstract patch representations can be formally defined as *fix patterns*. Coccinelle [51] and its semantic patches provide a metavariable example of how fix patterns can be leveraged to systematically apply common patches, e.g., to address collateral evolution due to API changes [52]. Manually summarizing fix patterns from patches is however time-consuming. Thus, we are investigating an automated approach of mining fix patterns. To that end, we first provide a formal definition of a fix pattern.

Definition 5. Fix Pattern (FP): A *fix pattern* is a pair of a *code context* extracted from a buggy code block and a set of *change operations*, which can be applied to a given buggy code block to generate fixing code. This can be formalized as:

$$FP = (Ctx, CO) \tag{5}$$

where *Ctx* represents the code context that is an abstract representation of the buggy code block. CO is a set of change operations (See Definition 6) to be applied to modifying the buggy code block.

Definition 6. Change Operation (O): A *change operation* is a three-value tuple which contains a change action, a source code entity and a set of sub change operations. This can be formalized as:

$$O = (Action, Sce, CO) \tag{6}$$

where *Action* is an element of an action set (i.e., {`UPD`, `DEL`, `INS`, `MOV`}) working on the entity (*Sce*). `UPD` is an *update* action which means updating the target entity, `DEL` is a *delete* action which denotes deleting the target entity, `INS` is an *insert* action which represents inserting a new entity, and `MOV` is a *move* action which indicates moving the target entity. *CO* is a set of sub change operations working on the sub entities of the current action's entity. When an operation acts on a leaf node entity, $CO = \emptyset$.

***A Set of Change Operations*:**
o_1. (UPD, (ReturnStatement, return), [
o_2. (UPD, (CastExpression, (String[])), [
o_3. (UPD, (MethodInvocation, list.toArray), [
o_4. (UPD, (Method, toArray), [
o_5. (UPD, (ArrayCreation, String), [
o_6. (DEL, (ArraySize, NumberLiteral), ∅)
o_7. (INS, (MethodInvocation, list.size), [
o_8. (INS, (Variable, list), ∅),
o_9. (INS, (Method, size), ∅)])])])])])])

Fig. 10: A set of change operations of the patch in Figure 11.

For example, Figure 10 shows the set of change operations of the patch in Figure 11. o_1 is the change operation working on the root entity `ReturnStatement`. `UPD` is the *Action*, (`ReturnStatement`, `return`) is the root entity being acted, and o_2 is the sub change operation acting on the sub entity `CastExpression` of the root entity. It is the same as others. o_6, o_8, and o_9 are the change operations working on leaf node entities. So that, the sets of their sub change operations are null.

A fix pattern is used as a guide to fix a bug. The fixing process is defined as a *bug fix process* presented in Appendix A for interested readers.

2.5.2 Pattern mining process

Figure 11 shows a concrete patch that can only be used to fix related specific bugs as it limits the syntax and semantic structure of the buggy code. The statement is limited to be a `Return Statement` and the parameterized type of the `List` and the `Array` is also limited to `String`. Additionally, the variable name `list` can also interfere with the matching between this patch and similar bugs. However, the abstract patch in Figure 12 abstracts the aforementioned interferon, which can be matched with various mutations of the bug converting a `List` into an `Array`. Hence, it is necessary to mine common patch patterns from massive and various patches for specific bugs.

DiffEntry of a patch:

```
@@ -1246,1 +1246,1 @@
- return (String[]) list.toArray(new String[0]);
+ return (String[]) list.toArray(
+          new String[list.size()]);
```

Fig. 11: Example of a patch taken from *FilenameUtils.java* file within Commit `09a6cb` in project `commons-io`[10].

Our conjecture is that *common fix patterns* can be mined from large change sets. Exposed bugs are indeed generally

[10]https://commons.apache.org/proper/commons-io/

Abstract representation of a patch:

```
@
var: a list variable.
T: the parameterized type of a list.
@
- (T[]) var.toArray(new T[0]);
+ (T[]) var.toArray(new T[var.size()])
```

Fig. 12: Example of an abstract representation of the patch in Figure 11.

not new and common fix patterns may be an immediate and appropriate way to address them automatically. For example, when discussing the deluge of buggy mobile software, Andy Chou, a co-designer of the Coverity bug finding tool, reported that, based on his experience, the found bugs are nothing new and are "actually well-known and well-understood in the development community - the same *use after free* and *buffer overflow* defects we have seen for decades" [10]. In this vein, we design an approach to mine common fix patterns for static analysis violations by extracting changes that represent developers' manual corrections. Figure 9 illustrates our process for mining common fix patterns.

Data Preprocessing.

As defined in Definition 5, a fix pattern contains a set of change operations, which can be inferred by comparing the buggy and fixed versions of source code files. In our study, code changes of a patch are described as a set of change operations in the form of Abstract Syntax Tree (AST) differences (i.e., AST diffs). In contrast with GNU diffs, which represent code changes as a pure text-based 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 the open source GumTree [16] tool to extract and describe change operations implemented in patches. GumTree, and its associated source code, is publicly available, allowing for replication and improvement, and is built on top of the Eclipse Java model[11].

All patches are tokenized into textual vectors by traversing their AST-level diff tree with the deep-first search algorithm and extracting the action string (e.g., `UPD`), the entity type (e.g., *ReturnStatement*) and the entity identifier (e.g., *return*) as tokens of a change action (e.g., *UPD ReturnStatement return*). A given patch is thus represented as a list of such tokens, further embedded and vectorized as a numeric vector using the same method described in Section 2.4.3.

Fix Patterns Mining.

Patches can be considered as a special kind of natural language text, which programmers leverage daily to request and communicate changes in their community. Currently available patch tools only perform directly the specified operations (e.g., remove and add lines for GNU diff) so far without the interpretation of what the changes are about. Although all patches can be parsed and converted into two-dimensional numeric vectors, it is still challenging to mine fix patterns given that noisy change information (e.g., specific changes) can interfere with identifying similar patches. Thus, our method is designed to effectively learn discriminating features of patches for mining fix patterns.

Similarly to the case of violation code pattern mining, we leverage CNNs to perform deep learning of patch features with preprocessed patches, and *X-means* clustering algorithm to automatically cluster similar patches together with learned features. Finally, we manually label each cluster with fix patterns of violations abstracted from clustered patches to show fix patterns clearly.

3 Empirical Study

3.1 Datasets

We consider project subjects based on a curated database of `Github.com` provided through `GHTorrent` [53]. We select projects satisfying three constraining criteria: (1) a project has, at least, 500[12] commits, (2) its main language is Java, and (3) it is unique, i.e., not a fork of another project. As a result, 2014 projects are initially collected. We then filter out projects which are not automatically built with *Apache Maven*. Subsequently, for each project, we execute `FindBugs` on the compiled[13] code of its revisions (i.e., committed version). If a project has at least two revisions in which `FindBugs` can successfully identify violations, we apply the tracking procedure described in Section 2.2 to collecting data.

Table 1 shows the number of projects and violations used in this study. There are 730 projects with 291,615 commits where 250,387,734 violations are detected; these violations are associated with 400 types defined by `FindBugs`. After applying our violation tracking method presented in Section 2.2 to these violations, as a result, 16,918,530 distinct violations are identified.

TABLE 1: Subjects used in this study.

# Projects	730
# Commits	291,615
# Violations (detected)	250,387,734
# Distinct violations	16,918,530
# Violations types	400

3.2 Statistics on detected violations

We start our study by quickly investigating **RQ1**: "*to what extent do violations recur in projects?*". We focus on three aspects of violations: number of occurrences, spread in projects and category distributions. Given that such statistics are merely confirming natural distributions of the phenomenon of defects, we provide all the details in the Appendix B of this paper. Interested readers can also directly refer to the replication package (including code and data) at :

https://github.com/FixPattern/findbugs-violations.

[11] http://www.vogella.com/tutorials/EclipseJDT/article.html

[12] A minimum number of commits is necessary to collect a sufficient number of violations, which will be used for violation tracking.

[13] `FindBugs` runs on compiled bytecode (cf. Section 2.1).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

Overall, we have found that around 10% of violation types are related to about 80% of violation occurrences. However, only 200 violation types are spread over more than 100 projects (i.e., 14% of the subjects), and some violation types which are the most widespread (i.e., top-50) actually have less occurrences than lesser widespread ones. Finally, although most violation types defined by `FindBugs` are related to *Correctness*, the clear majority (66%) of violation occurrences are associated with *Dodgy Code* and *Bad Practice*. *Security*-related violations account only for 0.5% of violation occurrences, although they are widespread across 30% of projects.

3.3 What types of violations are fixed?

Although overall statistics of violation detections show that there is variety in recurrence of violations, we must investigate *what types of violations are fixed by developers?* (**RQ2**). We provide in Appendix C more details on the following three sub-questions that are considered to thoroughly answer this question.

- RQ2-1: Which types of violations are developers most concerned about?
- RQ2-2: Are fixed violations per type proportional to all detected violation?
- RQ2-3: What is the distribution of fixed violations per category?

We refer the interested reader to this part for more statistics and detailed insights.

Overall, we have identified 88,927 violation instances which have been fixed by developer code changes. We note that we could not identify fixes for some 69 (i.e., 17%) types of violations, nor in 183 (i.e., 25%) projects. Given the significantly low proportion of violations that eventually get fixed, we postulate that some violation types must represent programming issues that are neglected by the large majority of developers. Another plausible explanation is the limited use of violation checkers such as `FindBugs` in the first place since 36% (273) of the projects associated with `FindBugs` include at least one commit referring to the `FindBugs` tool, and 1,944 (2% of 88,927) cases where the associated commit messages refer to `FindBugs`.

Only a small fraction of violations are fixed by developers. This suggests these violations are related to a potentially high false positive ratio in the static analysis tool, or lack developer interest due to their minor severity. There is thus a necessity to implement a practical prioritization of violations.

With respect to RQ2-1, we find that only 50 violation types, i.e., 15% of the fixed violation types, are associated with 80% of the fixed violations, and only 63 (19%) fixed violation types are appearing in at least 10% of the projects.

Developers appear to be concerned about only a few number of violation types. The top-2 fixed violation types (`SIC_INNER_SHOULD_BE_STATIC_ANON`[14] and `DLS_DEAD_LOCAL_STORE`[15]) are respectively performance and Dodgy code issues.

With respect to RQ2-2, we compute a fluctuation ratio metric which, for a given violation type, assesses the differences of ranking in terms of detection and in terms of fixes. Indeed a given violation type may account for a very high x% of all violation detections, but account for only a low y% (i.e., $y \ll x$). Or vice versa. This metric allows to better perceive how violations can be prioritized: for example, we identified 4 violation types, including `NM_CLASS_NAMING_CONVENTION`[16], have fluctuation ratio values higher than 10, suggesting that, although they have high occurrence rates, they have lower fix rates by developers. On the other hand, violation type `NP_NONNULL_RETURN_VIOLATION`[17] has an inversed fluctuation ratio of over 20, suggesting that although it has low occurrences in detection, it has a high priority to be fixed by developers.

Our detailed study of the differences between detection and fix ratios provides data and insights to build detection report and fix prioritization strategies of violations.

Finally, with respect RQ2-3, our investigations revealed that the top-50 fixed violation types are largely dominated by *Dodgy code*, *Performance* and *Bad Practice* categories. Although *Correctness* overall regroups the largest number (33%) of fixed violation types, its types have, each, a low number of fix occurrences. Interestingly, *Internationalization* is also a common fixed category, with 6,719 fixed instances across 347 (63.3%) projects, with only two types (`DM_CONVERT_CASE`[18] and `DM_DEFAULT_ENCODING`[19]) which are among top-5 most occurring violation types and among top-10 most widespread throughout projects).

Overall, *Dodgy code*, *Performance*, and *Bad Practice* issues are the most addressed by developers. *Correctness* issues, however, although they are with to the majority of fixed types, developers fail to address a large portion of them. Compared to *Internationalization*, which are straightforward and resolved uniformly, the statistics suggest that developers could accept to fix *Correctness* issues if there were tool support.

3.4 Comparison against other empirical studies on FindBugs violations

The literature includes a number of studies related to `FindBugs` violations. While our work includes such a study, it is substantially more comprehensive and is based on more representative subjects. As presented in Table 2, our study collects data from 730 real-world projects (i.e., in the wild) where 400 violation types (of 9 categories) can be found. Other studies have only considered overall only 3 real-world projects. Vetro et al. [54] collect data from 301 projects, but they are in-the-lab projects which may not

[14]Inner class could be refactored into a named static inner class.
[15]Dead store to local variable.
[16]Class names should start with an upper case letter.
[17]Method may return null, but is declared @Nonnull.
[18]Consider using Locale parameterized version of invoked method.
[19]Reliance on default encoding.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

be representative of real-world development. Ayewah et al. [15] only investigated some ($<$ 100) *Correctness*-related violations. Fixit [55] studied violations at the category level and limited violations into six categories. Vetro et al. [54] studied 77 violation types but ignored violation categories.

TABLE 2: Comparison of empirical studies on FindBugs violations.

	Our study	Ayewah et al. [15]	Fixit [55]	Vetro et al. [54]
Projects	730 projects in the wild	Two projects in the wild	One student project, One project in the wild	301 projects in the lab
# types	400	$<$ 100	-	77
# categories	9 (all of them)	1 (Correctness)	6	-
# detected cases	16,918,530	1,506	10,479	1,692
# fixed cases	88,927	518	640	-
Objective	Fix pattern mining	Evaluating static analysis warnings	Look into the value of static analysis	Assess percentage and type of violations

Additionally, our study investigates detected violation distributions from three aspects: occurrences, spread, and categories, which provides three different metrics to prioritize violations. Nevertheless, it should be noted that the false positives of `FindBugs` could threaten the reliability of violation prioritization based on the statistics of detected violations. Previous studies [15], [54], [55] do not discuss this aspect. To reduce this threat, we further investigate distributions of fixed violations, which represent violations that attract developer attention for resolution, thus suggesting higher probabilities for true positives. Our results provide more reliable prioritization metrics for violations reporting.

We further note that these studies focused on objectives that are different from ours. Ayewah et al. [15] focused on evaluating the importance of static analysis warnings in production software. In Fixit [55], the authors looked into the value of FindBugs on finding program issues. Vetro et al. [54] aimed at assessing the percentage and type of issues of FindBugs that are actual defects. After going through their research tracks, our work could be applied to their research questions, but our eventual goal is to mine fix patterns for FindBugs violations.

3.5 Code Patterns Mining

Empirical findings on violation tracking across the projects showed that only a small fraction of violations are fixed by developers. Thus, overall, the distribution of unfixed violations follow that of detection violations. We now investigate the research question *what kinds of patterns do unfixed and fixed violations have respectively?* (**RQ3**), focusing on the following sub-questions:

- RQ3-1: What are the common code patterns for unfixed violations and fixed ones respectively?
- RQ3-2: What is the relationship or difference between the common source code patterns of unfixed violations and fixed ones?
- RQ3-3: What are possible reasons for some violations to remain unfixed?

To avoid noise in the dataset due to varying distributions, we focus on instances the instances of violations where the violation types are among the top-50 types that developers are concerned about (i.e., the most fixed ones). Then, we apply the approach of mining code patterns presented in Section 2.4 to identify common code patterns of unfixed violations and fixed ones respectively.

Disclaimer: Note that `FindBugs` produces a large number of false positives in two ways: 1) locations of detected violations can be incorrectly reported by `FindBugs`, or 2) the detected violations are correctly located, but developers may still treat it as a false positive warning since it could not be considered as a serious enough concern to fix. While the second kind of false positives does not threaten patterns mining, but the first kind does. To reduce the threat to validity due to false positives related to incorrect localization, we focus on the pattern mining process on the recurrent fixed violations: their locations are most likely correct given that developers manually checked and addressed the issue.

3.5.1 Experiment Setup

`FindBugs` reports violations by specifying the start line and the end line of the code hunk that is relevant to the violation. Since it is challenging (and error-prone) to mine code patterns by considering big code hunks, we limit our experiments on small hunks. Figure 13 illustrates the distribution of sizes (i.e., the code line numbers of hunks) of the code hunks associated with all violations.

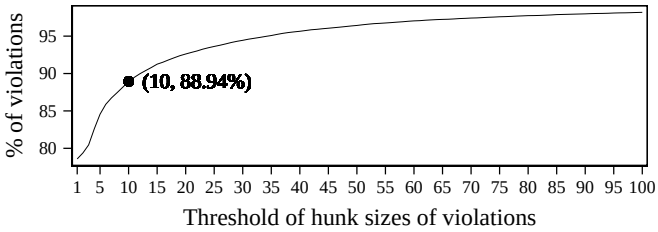

Fig. 13: Hunk sizes' distribution of all violations.

For 89% of the violations, the relevant code hunk is limited to 10 code lines or less. We have further manually observed that a line-based calculation of hunk size is not reliable due to the presence of noise caused by comments, annotations and unnecessary blank lines, so we select violations by their tokens. Figure 14 provides the distribution of numbers of code tokens by violations. We discard outliers and thus focus on violations where the code includes at most 40 tokens extracted based on their refined AST trees (cf. tree B in Figure 7).

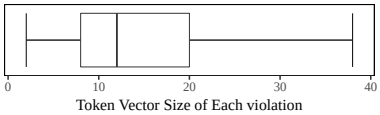

Fig. 14: Sizes' distribution of all violation token vectors.

Following the methodology described in Section 2.4, violations are represented with numeric vectors using Word2Vec with the following parameters (Size of vector = 300; Window size = 4; Min word frequency = 1)

Feature extraction is then implemented based on CNNs whose parameters are listed in Table 3. The literature has consistently reported that effective models for Word2Vec and deep learning applications require well-tuned parameters [17], [56], [57], [58], [59]. In this study, all parameters

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

of the two models are tuned through a visualizing network training UI[20] provided by DeepLearning4J.

TABLE 3: Parameters setting of CNNs.

Parameters	Values
# nodes in hidden layers	1000
learning rate	1e-3
Optimization algorithm	stochastic gradient descent
pooling type	max pool
activation (output layer)	softmax
activation (other layers)	leakrelu
loss function	mean squared logarithmic error

Finally, Weka's [50] implementation of *X-means* clustering algorithm uses the extracted features to find similar code for each violation type. Parameter settings for the clustering are enumerated in Table 4.

TABLE 4: Parameters setting of X-means.

Parameters	Values
Distance Function	Euclidean Distance
KD Tree	true
# max iterations	1000
# max K-means	500
# max K-means of children	500
# seed	500
# max clusters	500
# min clusters	1

3.5.2 Code Patterns

Given that violation code fragments are represented in the generic form of an AST, we can automatically mine patterns by simply considering the most recurring fragment in a cluster yielded by our approach as the pattern. We then manually assess each pattern to assign a label to it. We investigate code patterns on fixed violations and unfixed ones respectively. Overall, while unfixed violations yield a few more patterns than fixed violations, we find that most patterns are shared by both unfixed and fixed sets. Table 5 shows some examples of identified common code patterns of 10 violation types.

We manually checked the patterns yielded for the top-50 violation types and assessed these patterns with respect to `FindBugs`' documentation. For example, `DM_NUMBER_CTOR` violation refers to the use of a number constructor to create a number object, which is inefficient [14]. For instance, using `new Integer(...)` is guaranteed to always result in a new `Integer` object whereas `Integer.valueOf(...)` allows caching of values to be done by the compiler, class library, or JVM. Using cached values can avoid object allocation and the code will be faster. Our mined patterns are the five types of number creations with number constructors. `DM_FP_NUMBER_CTRO` has the similar patterns with it. This example shows how violation code patterns mined with our approach are consistent with the static analysis tool documentation. We have carefully checked the patterns for the top-50 violation types, and found that for 76%, the patterns are adequate with respect to the documentation. Appendix D provides details on 10 example violation types.

> Our code pattern mining approach yields patterns that are consistent with the violation descriptions in documentation of the static analysis tool.

We focused our investigations on some of the patterns that are yielded only from unfixed violation code, and found that in some cases, there are inconsistencies between the pattern and the bug description provided by `FindBugs`.

First, we consider a case where the number of patterns discovered for a given violation type exceeds the number of cases enumerated by `FindBugs` in its documentation. `MS_SHOULD_BE_FINAL` is a violation type raised when the analyzer encounters a static field that is public but not final: such a field could be changed by malicious code or accidentally from another package [14]. Besides public static field declarations, the identified patterns on violation code of this type include protected static field declarations, which is inconsistent with the description by `FindBugs`. Figure 15 shows an example of such inconsistent detection by `FindBugs` in project `BroadleafCommerce`. When developers confront `FindBugs`' warning message against their code, they may decide not to address such an undocumented bug.

Violation Type:
`MS_SHOULD_BE_FINAL`

Violation Code:
```
protected static String FACETS_ATTRIBUTE_NAME =
    "facets";
```

Fig. 15: Example of a detected `MS_SHOULD_BE_FINAL` violation, taken from project `BroadleafCommerce`[21].

Second, we consider a case where the mined pattern is inconsistent with the documentation of the violation. `RI_REDUNDANT_INTERFACES` is a warning on a class which implements an interface that has already been implemented by one of the class' super classes [14]. Its mined common code pattern is associated to a super constructor invocation. However, the violation location is positioned on the class declaration line. After manually checking some `RI_REDUNDANT_INTERFACES` cases, we find that the Java classes with `RI_REDUNDANT_INTERFACES` violations indeed have a redundant interface(s) in their class declaration code part. However, some detected `RI_REDUNDANT_INTERFACES` violations locate on the super constructor invocations but not the class declaration code, which could confuse developers and increase the perception of high false positives rates. For example, in Figure 16, the exact position of the `RI_REDUNDANT_INTERFACES` violation should be the "implements Serializable" part (L-33). `FindBugs` however reports the position at L-49 (highlighted with red background) which is not precise and can even confuse developers on *why the code is a violation and how to resolve it*.

[20]https://deeplearning4j.org/visualization

[21]https://github.com/BroadleafCommerce/BroadleafCommerce

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

TABLE 5: Common code pattern examples of violations.

Violation Type	Common Source Code Pattern(s)
DM_CONVERT_CASE	① stringExp.toLowerCase(), ② stringExp.toUpperCase().
RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	① if (exp == null ...) {...}, ② if (exp != null ...) {...}, ③ exp == null ? exp1 : exp2, ④ exp != null ? exp1 : exp2.
BC_UNCONFIRMED_CAST	① T1 v1 = (T1) v2/exp, ② v1 = (T1) v2/exp, ③ ((T1) v2).exp.
MS_SHOULD_BE_FINAL	public/protected static T1 v1 = exp.
RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	① fileExpe.mkdirs(), ② fileExpe.mkdir(), ③ fileExpe.delete(), ④ fileExpe.createNewFile(), ⑤ other exp.method_invoation() returns a value.
DM_NUMBER_CTOR	① new Long(...), ② new Integer(...), ③ new Short(...), ④ new Byte(...), ⑤ new Char(...).
SBSC_USE_STRINGBUFFER_CONCATENATION	① stringVariable += stringExp, ② stringVariable = stringExp1 + stringExp2.
DM_BOXED_PRIMITIVE_FOR_PARSING	① Integer.valueOf(str), ② Long.valueOf(str).
PZLA_PREFER_ZERO_LENGTH_ARRAYS	return null.
ES_COMPARING_STRINGS_WITH_EQ	① stringExp1 == stringExp2, ② stringExp1 != stringExp2.

Violation Type:
`RI_REDUNDANT_INTERFACES`

Violation Code:
```
L-33   public abstract class AbstractFormat extends
           NumberFormat implements Serializable {
           ......
L-48       protected AbstractFormat() {
L-49           this(getDefaultNumberFormat());
           }
```

Fig. 16: Example of a miss-located `RI_REDUNDANT_INTERFACES` violation, taken from `commit 84a642` in project `commons-math`.

> Some violations remain unfixed as a result of their imprecise detection. False positives in `FindBugs` can be improved by addressing some issues with accurate reporting of violation locations, as well as updating the documentation.

Finally, we note that it is challenging to identify common code patterns for some violation types for two main reasons.

First, some clusters are too small, indicating that the violation instances, despite the abstraction with AST, are too specific. For example, `DLS_DEAD_LOCAL_STORE` violations are about variable assignments which are specific operators in source code. It is challenging to identify any common code pattern except for the pattern, *variable assignment statement*, identified at the level of AST node types. With this information alone, it is practically impossible to figure out why a code fragment is related to a `DLS_DEAD_LOCAL_STORE` violation. This is a potential reason why some `DLS_DEAD_LOCAL_STORE` violations remain unfixed.

Second, again, `FindBugs` cannot locate some violations accurately. We enumerate three scenarios:

- The detected violation code is the method body but not the method name. For example, `NM_METHOD_NAMING_CONVENTION` violations violate the method naming convention but not method bodies, however the source code of these violations tracked with their position provided by `FindBugs` is the method bodies. Similar source code can be clustered into the same cluster to identify some patterns which cannot explain how the violation is induced, but could help interpret the behavior of these methods. Actually, the method name is the abstract description of method body, so we think that it is inefficient to identify the violation of method names by their naming convention without considering the behavior of method bodies.
- The second case is that the source code of violations is irrelevant source code. For instance, `UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR` indicates that a field is never initialized within any constructor, loaded and referenced without a null check [14]. According to observing the instances of this violation type, the source code of these violations is the statements of one method body in these violated Java class, which is irrelevant to the violation type. Some similar source code can be clustered together to obtain some patterns which still cannot explain the violation type. Therefore, it is inconsistent with the bug description of this type.
- The third case is that the violation locates on class body rather the declaration of class name. `SE_NO_SERIALVERSIONID` means the current violated Java class implements the `Serializable` interface, but does not define a `serialVersionUID` field [14]. The positions of this kind of violations provided by `FindBugs` are located in the class body. It is impossible to identify the common code patterns of this violation type which can interpret why the source code makes the violations.

These inaccurate localized violations could mislead or confuse developers, which may cause that developers do not prefer to fix these kinds of violations. In this study, we re-locate the violations of `serialVersionUID` and `RI_REDUNDANT_INTERFACES` to class declarations. Combining the results with source code changes of type-related fixed violations, it is easy to follow why the source code fragment is a violation. Figure 17 shows an example of fixing a `RI_REDUNDANT_INTERFACES` violation. Interface `java.util.Map` has been implemented in the super class `AbstractMap` of the current class `Map`. Thus, it is fixed by removing the redundent `java.util.Map` interface.

[22]https://github.com/datanucleus/datanucleus-core

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

Violation Type: `RI_REDUNDANT_INTERFACES`

Fixing Patch:

```
   public class Map extends AbstractMap implements
-      java.util.Map, SCOMap<java.util.Map>,
+      SCOMap<java.util.Map>,
       Cloneable, java.io.Serializable {
```

Fig. 17: Example of a fixed `RI_REDUNDANT_INTERFACES` violation, taken from `commit ea876b` in `datanucleus-core`[22] project.

> Many violation types are associated with code from which patterns can be inferred. Such patterns are relevant for immediately understanding how violations are induced. For some other violations code however it is difficult to mine patterns, partly due to the limitation of `FindBugs` and the fact that the code fragment is too specific.

3.6 Fix Patterns Mining

We now investigate our ultimate research question on *how are the violations resolved if fixed?* (**RQ4**). To that end, we first dissect the violation fixing changes and propose to cluster relevant fixes to infer common fix patterns following the CNN-based approach described in Section 2.5.

We curate our dataset of 88,927 violation fixing changes by filtering out changes related to:

- 4,682 violations localized in test files. Indeed, we focus on mining patterns related to developer changes on actual program code.
- 7,010 violations whose fix do not involve a modification in the violation location file. This constraint, which excludes cases where long data flow may require a fixing change in other files, is dictated by our automation strategy for computing the AST edit script, which is simplified by focusing on the violation location file.
- 7,121 violations where the associated fix changes are not local to the method body of the violation.
- 25,464 violations where the fixing changes are applied relatively far away from the violation location. We consider that the corresponding AST edit script matches if the change actions are performed within ± 3 lines of the violation location. This constraint conservatively helps to further remove false positive cases of violations which are actually not fixed but are identified as fixed violations due to limitations in violation tracking.
- 9,060 violations whose code or whose fix code contain a large number of tokens. In previous works, Herzig et al. [60] and Kawrykow et al. [61] have found that large source code change hunks generally address feature additions, refactoring needs, etc., rather than bug fixes. Pan et al. [62] also showed that large bug fix hunk pairs do not contain meaningful bug fix patterns, and most bug fix hunk pairs (91-96%) are small ones. Ignoring large hunk pairs has minimal impact on analysis. Consequently, we use the same threshold (i.e., 40, presented in Section 3.5) of tokens to select fixed violations.

Overall, our fix pattern mining approach is applied to 35,590 violation fixing changes, which are associated with 288 violation types. Parameter values of Word2Vec, CNNs and *X-means* are identical to those used for common code patterns mining (cf. Section 3.5). In this study, once a cluster of similar changes, for a given violation type, are found, we can automatically mine the patterns based on the AST diffs. Although approaches such as the computation of longest common subsequence of repair actions could be used to mine fix patterns, we observe that they do not always produce semantically meaningful patterns. Thus, we consider a naive but empirically effective approach of inferring fix patterns by considering the most recurring AST edit script in a given cluster, i.e., the code change that occurs identically the most. Finally, labels to each change pattern are assigned manually after a careful assessment of the pattern relevance.

For the experiments, we focus on the top-50 fixed violation types for the mining of fix patterns. Table 6 summarizes 10 example cases of violation types with details, in natural language, on the fix patterns.

Figure 18 presents an inferred pattern in terms of AST edit script for violation type `RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE` described in Table 6. For AST-level representation of patterns of other violations, we refer the reader to the replication package.

Overall, the pattern presented in AST edit script format, which should be translated into fix changes to "*delete the null check expression*" requires some code context to be concretized. When the `var`[23] `!= null` expression is the *null-checking* conditional expression of an `IfStatement`, the concrete patch must delete the violated expression. Similarly, when the `exp == null` expression is the condition expression of an `IfStatement`, the patch also removes the *null-checking* expression. When `exp == null` or `exp != null` expression is one of the condition expressions of an `IfStatement`, the patch is deleting the violated expression. This example shows the complexity of automatically generating patches from abstract fix patterns, an entire research direction which is left for future work. For now, we generate the patches manually based on the mined fix patterns.

> Our proposed fix pattern mining approach can effectively cluster similar changes of fixing violations together. And the fix pattern mining protocol is applicable to derive meaningful patterns.

Listing 1: Violation types failed to be identified fix pattern

```
1. UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR
2. SF_SWITCH_NO_DEFAULT
3. UWF_UNWRITTEN_FIELD
4. IS2_INCONSISTENT_SYNC
5. VA_FORMAT_STRING_USES_NEWLINE
6. SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING
7. OBL_UNSATISFIED_OBLIGATION
8. OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE
9. OS_OPEN_STREAM
10.OS_OPEN_STREAM_EXCEPTION_PATH
11.ODR_OPEN_DATABASE_RESOURCE
12.NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE
```

Listing 1 enumerates 12 violation types for which our mining approach could not yield patterns, given that the number of samples per cluster was small, or that within a

[23] `var` represents any variable being checked.
[24] https://github.com/apache/pdfbox

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

TABLE 6: Common fix pattern examples of fixed violations.

Violation Type	Fix Pattern(s)
DM_CONVERT_CASE	ADD a rule of Locale.ENGLISH into toLowerCase()/toUpperCase().
RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	① Delete the null check expression. ② Delete the null check IfStatement.
BC_UNCONFIRMED_CAST	① Delete the violated statement, ② Delete the cast type, ③ Replace CastExpression with a null value.
MS_SHOULD_BE_FINAL	Add a "final" modifier.
RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	① Add an IfStatement to check the return value of violated source code. ② Replace violated expression with a new method invocation.
DM_NUMBER_CTOR	Replace the number constructor with a static number.valueOf() method.
SBSC_USE_STRINGBUFFER_CONCATENATION	Replace the String type with the StringBuilder, and replace plus operator of StringVarialbe with the append method of StringBuilder.
DM_BOXED_PRIMITIVE_FOR_PARSING	Replace Number.valueOf() with Number.parseXXX() method.
PZLA_PREFER_ZERO_LENGTH_ARRAYS	① Delete the buggy statement, ② Replace the null value with an empty array.
ES_COMPARING_STRINGS_WITH_EQ	Replace the "==" or "!=" InfixExpression with a equals() method invocation.

DiffEntry of a patch:

```
-   if (dictionaryObject!=null && !onlyEmptyFields){
+   if (!onlyEmptyFields) {
          signatures.put(new COSObjectKey(dict),
                (COSDictionary)dict.getObject());
     }
```

Repair actions parsed by GumTree at AST level:

```
UPD IfStatement@@if(dictionaryObject!=null &&
                !onlyEmptyFields)
---UPD InfixExpression@@dictionaryObject!=null &&
                !onlyEmptyFields
------DEL InfixExpression@@dictionaryObject!=null
---------DEL VariableName@@dictionaryObject
---------DEL Operator@@!=
---------DEL NullLiteral@@null
------DEL Operator@@&&
```

Inferred Fix Pattern:

```
UPD IfStatement@@if(Null-Check-Expression
                Operator Other-Expression)
---UPD InfixExpression@@Null-Check-Expression
                Operator Other-Expression
------DEL Null-Check-Expression
------DEL Operator
```

Fig. 18: Example of a fix pattern for `RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE` violation inferred from a violation fix instance taken from `commit a41eb9` in project `apache-pdfbox`[24].

cluster we could not find strictly redundant change actions sequences. Our observations of such cases revealed the following causes of failure in fix pattern mining:

- violations can be fixed by adding completely new node types. For example, one fix pattern of `RV_RETURN_VALUE_IGNORED_BAD_PRACTICE` violations is replacing the violated expression with a method invocation which encapsulates the detailed source code changes.
- violations can occur on specific source code fragments from which it is even difficult to mine patterns. Fixes for such violations generally do not share commonalities.
- violations can have fix changes applied in separate region than the violation code location. Since we did not consider such cases for the mining, we systematically miss bottom-7 violation types of Listing 1 which are in this case.
- violations can be associated to a `String` literal. For example, we observe that the fixing changes of `VA_FORMAT_STRING_USES_NEWLINE` violations are replacing "\n" with "%n" within strings. Unfortunately, our AST nodes are focused on compilable code tokens, and thus changes in `String` literal are ignored to guarantee sufficient abstraction from concrete patches.

3.7 Usage and effectiveness of fix patterns

We finally investigate whether *fix patterns can actually help resolve violations in practice?* (**RQ5**). To that end, we consider the following sub-questions:

- RQ5-1: Can fix patterns be applied to automate the management of some *unfixed* violations?
- RQ5-2: Can fix patterns be leveraged as ingredients for automated repair of *buggy* programs?
- RQ5-3: Can fix patterns be effective in systematizing the resolution of `FindBugs` violations *in the wild*?

We recall that our work automates the generation of fix patterns. Patch generation is out of scope, and thus will be performed manually (based on the mined fix patterns), taking into account the code context.

3.7.1 Resolving unfixed violations

We apply fix patterns to a subset of unfixed violations in our subjects following the process illustrated in Figure 19. For a given unfix violation, we search for the top-k[25] most suitable fix patterns to generate patches. To that end, we consider cosine similarity between the violation code features vector (built with CNNs in Section 2.4.3) and the features vector of the centroid fixed violation in the cluster associated to each fix pattern.

A fix pattern is regarded as a true positive fix pattern for an unfixed violation, if a patch candidate derived from this pattern is addressing the violation. We check this by ensuring that the resulting program after applying the patch candidates passes compilation and all tests, FindBugs no longer raises a warning at this location, and manual checking by the authors has not revealed any inappropriate change of semantics in program behaviour.

Test data: We collect a subset of unfixed violations in the top-50 fixed violation types (described in Section 3.5) as the testing data of this experiment to evaluate the effectiveness of fixed patterns. For each violation type, at most 10 unfixed violation instances, which are the most similar to the centroids of the corresponding fixed violations clusters, are selected as the evaluation subjects.

Results: Table 7 presents summary statistics on unfixed violations resolved by our mined fix patterns. Overall, among the selected 500 unfixed violations in the test

[25] $k = 10$ in our experiments

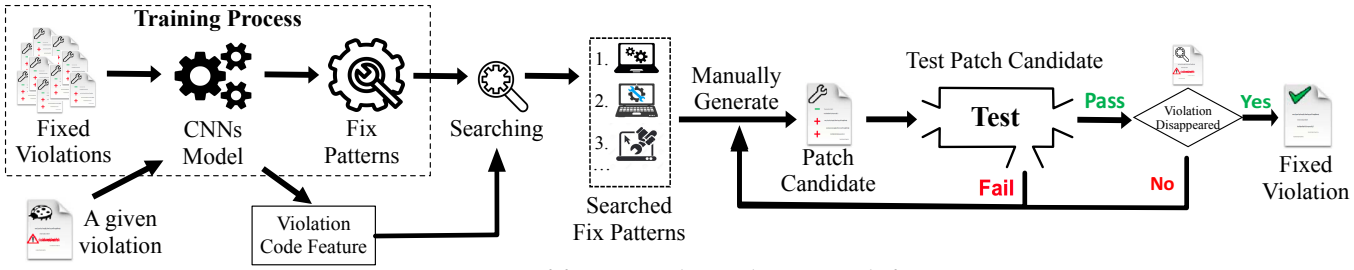

Fig. 19: Overview of fixing similar violations with fix patterns.

TABLE 7: Unfixed-violations resolved by fix patterns.

Violation types	Top 1	Top 5	Top 10	Total
RI_REDUNDANT_INTERFACES	10	10	10	10
SE_NO_SERIALVERSIONID	10	10	10	10
UPM_UNCALLED_PRIVATE_METHOD	10	10	10	10
DM_NUMBER_CTOR	9	10	10	10
DM_FP_NUMBER_CTOR	9	10	10	10
DM_BOXED_PRIMITIVE_FOR_PARSING	8	9	10	10
DM_CONVERT_CASE	7	9	10	10
MS_SHOULD_BE_FINAL	7	9	9	10
PZLA_PREFER_ZERO_LENGTH_ARRAYS	7	7	8	10
RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE	6	8	8	10
RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	6	7	8	10
SBSC_USE_STRINGBUFFER_CONCATENATION	4	10	10	10
MS_PKGPROTECT	4	9	9	10
EI_EXPOSE_REP2	4	4	5	10
DM_DEFAULT_ENCODING	4	5	5	10
WMI_WRONG_MAP_ITERATOR	3	7	9	10
UC_USELESS_CONDITION	3	6	6	10
ES_COMPARING_STRINGS_WITH_EQ	2	8	10	10
RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	3	4	4	10
SIC_INNER_SHOULD_BE_STATIC_ANON	3	3	3	10
UCF_USELESS_CONTROL_FLOW	2	9	10	10
BC_UNCONFIRMED_CAST_OF_RETURN_VALUE	2	4	4	10
DLS_DEAD_LOCAL_STORE	2	3	4	10
NP_NULL_ON_SOME_PATH	1	5	7	10
BC_UNCONFIRMED_CAST	1	1	1	10
UC_USELESS_OBJECT	0	8	8	10
NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE	0	3	5	10
VA_FORMAT_STRING_USES_NEWLINE	0	0	0	10
UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	0	0	0	10
DE_MIGHT_IGNORE	0	0	0	10
EI_EXPOSE_REP	0	0	0	10
IS2_INCONSISTENT_SYNC	0	0	0	10
NM_METHOD_NAMING_CONVENTION	0	0	0	10
NP_LOAD_OF_KNOWN_NULL_VALUE	0	0	0	10
NP_NONNULL_RETURN_VIOLATION	0	0	0	10
NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE	0	0	0	10
OBL_UNSATISFIED_OBLIGATION	0	0	0	10
OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE	0	0	0	10
ODR_OPEN_DATABASE_RESOURCE	0	0	0	10
OS_OPEN_STREAM	0	0	0	10
OS_OPEN_STREAM_EXCEPTION_PATH	0	0	0	10
REC_CATCH_EXCEPTION	0	0	0	10
RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT	0	0	0	10
SF_SWITCH_NO_DEFAULT	0	0	0	10
SIC_INNER_SHOULD_BE_STATIC	0	0	0	10
SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING	0	0	0	10
ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	0	0	0	10
URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD	0	0	0	10
URF_UNREAD_FIELD	0	0	0	10
UWF_UNWRITTEN_FIELD	0	0	0	10
Total	127(25.4%)	188(37.6%)	203(40.6%)	500

Identified fix patterns are applied to fixing a subset of unfixed violations in our subjects.

data, 127 (25.4%) are fixed by the most similar matched fix patterns (i.e., top-1), 188 (37.6%) are fixed by a pattern among the top-5, and 203 (40.6%) are fixed within the top-10. The matched positive fix patterns mainly cluster on top-5 fix pattern candidates, which are a few less than the top-10 range. This suggests that enlarging the search space of fix pattern candidates cannot effectively find positive fix patterns for more target violations.

Among the 203 fixed unfixed-violations, only 3 of them are fixed by matched fix patterns collected across violation types. We observe that `DM_NUMBER_CTOR` and `DM_FP_NUMBER_CTOR` have similar fix patterns. We use the fix patterns of `DM_FP_NUMBER_CTOR` to match fix pattern candidates for `DM_NUMBER_CTOR` violations. The fix patterns of `DM_FP_NUMBER_CTOR` can fix the `DM_NUMBER_CTOR` violations, and vice versa.

> Almost half of the unfixed violations in a sampled dataset can be systematically resolved with mined fix patterns from similar violations fixed by developers. 1 out of 4 of these unfixed violations are immediately and successfully fixed by the first selected fix pattern.

We note that fix patterns for 23 violation types are effective in resolving any of the related unfixed violations. There are various reasons for this situation, notably related to the specificity of some violation types and code, the imprecision in `FindBugs` violation report, or the lack of patterns. We provide detailed examples in Appendix E.

3.7.2 *Fixing real bugs*

We attempt to apply fix patterns to relevant faults documented in the Defects4J [20] collection of real-world defects in Java programs. This dataset is largely used in studies of program repair [63], [64], [65].

Test data: We run `FindBugs` on the 395 buggy versions of the 6 Java projects used to establish Defects4J. As a result, it turns out that 14 bugs can be detected as static analysis violations detectable `FindBugs`. This is a reasonable number since most of the bugs in Defects4J are functional bugs which fail under specific test cases rather than programming rule violations.

For each relevant bug, we consider the fix patterns associated to their violation types, and manually generate the patches. When the generated patch candidate can (1) pass the failed test cases of the corresponding bug and (2) `FindBugs` cannot identify any violation at the same position, then the matched fix pattern is regarded as a positive fix pattern for this bug.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

Results: Table 8 shows the results of this experiment. 4 out of the 14 bugs are fixed with the mined fix patterns and the generated patches by fix patterns are semantically equivalent to the patches provided by developers for these bugs. The violations of 2 bugs are indeed eliminated by fix patterns, but the generated patches lead to new bugs (in terms of test suite pass). There are 2 bugs that can be matched with fix patterns, but more context information was necessary to fix them. For example, bug `Lang23` is identified as a `EQ_DOESNT_OVERRIDE_EQUALS` violation and matched with a fix pattern: overriding the `equals(Obj o)` method. It is difficult to generate a patch of the bug with this fix pattern without knowing the property values of the object being compared. The remaining 6 (out of 14 bugs) occurred on specific code, which is challenging to match plausible fix patterns for them without any context.

TABLE 8: Fixed bugs in Defects4J with fix patterns.

Classification	# bugs
Fixed bugs	4
Violations are removed but generates new bugs	2
Need more contexts	2
Failed to match plausible fix patterns	6
Total	14

Static analysis violations can represent real bugs that make programs fail functional test cases. Our mined fix patterns can contribute to automating the fix of such bugs as experimented on the Defects4J dataset.

3.7.3 *Systematically fixing `FindBugs` violations in the wild*

We conduct a live study to evaluate the effectiveness of fix patterns to systematize the resolutions of violations in open source projects. We consider 10 open source Java projects collected from `Github.com` on 30th September 2017 and presented in Table 9. `FindBugs` is then run on compiled versions of the associated programs to localize static analysis violations.

TABLE 9: Ten open source Java projects.

Project Name	# files	# lines of code
json-simple	12	2,505
commons-io	117	28,541
commons-lang	148	77,577
commons-math	841	186,425
ant	859	219,506
cassandra	1,625	216,192
mahout	1,145	222,345
aries	1,570	216,646
poi	4,562	894,514
camel	8,119	1,079,671

Test data: We focus on violation instances in the top-50 fixed violation types (presented in Section 3.3) are randomly selected as our evaluating data. For each violation, patches are generated manually in a similar process than the previous experiments: a patch must lead to a program that compiles, passes the test cases, and the previous violation location should not be flagged by `FindBugs` anymore. For each of such patch, we create a pull request and submit the patch to the project developers.

Results: Overall, we managed to push 116 patches to the developers of the 10 projects (cf. Table 10). 30 patches have been ignored while 15 have been rejected. Nevertheless, 2 patches have been improved by developers and 67 have been immediately merged. 1 of our pull requests to the `json-simple` project was not merged, but an identical patch has been applied later by the developers to fix the violation. Finally, the last patch (out the 116) has not been applied yet, but was attached to the issue tracking system, probably for later replacement.

TABLE 10: Results of live study.

Project Name	# Patches				
	pushed	ignored	rejected	improved	merged
json-simple	2	1	0	0	0
commons-io	2	0	2	0	0
commons-lang	7	5	1	1	0
commons-math	6	6	0	0	0
ant	16	2	4	1	9
cassandra	9	9	0	0	0
mahout	3	2	0	0	0
aries	5	5	0	0	0
poi	44	0	0	0	44
camel	22	0	8	0	14
Total†	116	30	15	2	67

†One patch of `json-simple` is the same as a patch of the same violation which has been fixed by its developer in another version. One patch of `mahout` is attached to its bug report system but has not yet been merged.

Table 11 presents the distribution of delays before acceptance for the 69 accepted (merged + improved) patches. 67% of the patches are accepted within 1 day, while 97% (67% +30%) are accepted within 2 days. Only 2 patches took a longer time to get accepted. We note that this acceptance delay is much shorter than the median distributions of the three kinds of patches submitted for the Linux kernel [8].

TABLE 11: Delays until acceptance.

Delay	less than 1 day	1 to 2 days	17 days
Number of Patches	46 (67%)	21 (30%)	2 (3%)

Acceptance indicates one of improved or merged patches.

As summarized in Table 12, we note that 21 accepted patches were verified by at least two developers. Although 48 accepted patches were verified by only one developer, we argue that this does not bias the results: first, the common source code patterns of these accepted fixed violation types are consistent with the descriptions documented by `FindBugs`; second, the matched fix patterns are likely acceptable by developers since the patterns are common in fixing violations as mined in the revision histories of real-world projects.

TABLE 12: Verification of accepted patches.

Verified by	1 developer	2 developers	3 developers
Number of Patches	48	19	2

Our mined fix patterns are effective to fix violations in the wild. Furthermore, the generated patches are eventually quickly accepted by developers.

The live study further yields a number of insights related to static analysis violations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

Insight 1. Well-maintained projects are not prone to violating commonly-addressed violation types. We note that 8 violation types (presented in Listing 2) do not appear at the current revisions of the selected 10 projects. Type `RI_REDUNDANT_INTERFACES` occurs only one time in `json-simple` project. This finding suggests that violation recurrences may be time-varying, so that, there is a time-variant issue of violation recurrences in revision histories of software projects, which may help to prioritize violations. It is included in our future work.

Listing 2: Violation types not seen in the selected 10 projects.

```
1. SIC_INNER_SHOULD_BE_STATIC
2. NM_METHOD_NAMING_CONVENTION
3. SIC_INNER_SHOULD_BE_STATIC_ANON
4. NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE
5. NP_NONNULL_RETURN_VIOLATION
6. UPM_UNCALLED_PRIVATE_METHOD
7. ODR_OPEN_DATABASE_RESOURCE
8. SE_NO_SERIALVERSIONID
```

Insight 2. Developers can write positive patches to fix bugs existing in their projects based on the fix patterns inferred with our method. For example, the developers of `commons-lang` [26] project fixed a bug[27] reported as a `DM_CONVERT_CASE` violation by `FindBugs` by improving the patch that was proposed using our method (cf. Figure 20). Our method cannot generate the patch they wanted because there is no fix pattern that is related to adding a rule of `Locale.ROOT` in our dataset, so that there might be a limitation of existing patches in revision histories.

The patch generated by fix patterns:

```
- final TimeZone tz = TimeZone.getTimeZone(
-         value.toUpperCase());
+ final TimeZone tz = TimeZone.getTimeZone(
+         value.toUpperCase(Locale.ENGLISH));
```

The patch improved by developers:

```
- final TimeZone tz = TimeZone.getTimeZone(
-         value.toUpperCase());
+ final TimeZone tz = TimeZone.getTimeZone(
+         value.toUpperCase(Locale.ROOT));
```

Fig. 20: Example of an improved patch in real project.

Insight 3. Developers will not accept plausible patches that appear unnecessary even if those are likely to be useful. For example, Figure 21 shows a rejected patch that adds an `instanceof` test to the implementation of `equals(Object obj)`. The developers want to accept this patch at the first glimpse, but they reject to change the source code after reading the context of these violations since the implementation of `equals(Object obj)` belongs to an inner static class which is only used in a generic type that will not compare against other Object types.

Rejected Patch:

```
- return Arrays.equals(keys,((MultipartKey)obj).keys);
+ return obj instanceof MultipartKey &&
+       Arrays.equals(keys, ((MultipartKey)obj).keys);
```

Fig. 21: Example of a rejected patch in real projects.

Insight 4. Some violations fixed based on the mined fix patterns may break the backward compatibility of other applications, leading developers to reject patches for such violations. For example, Figure 22 shows a rejected patch of a `MS_SHOULD_BE_FINAL` violation in *Path.java* file of the `ant` project, which breaks the backward compatibility of `systemClasspath` in *InternalAntRunner* class [28] of Eclipse project.

Rejected Patch breaks the backward compatibility:

```
-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. 22: Example of a rejected patch breaking the backward compatibility.

A Patch makes program fail to checkstyle:

```
-public static String defaultCharset =
+public static final String defaultCharset =
     ObjectHelper.getSystemProperty(
        Exchange.DEFAULT_CHARSET_PROPERTY, "UTF-8");
```

Fig. 23: Example of a patch making program fail to checkstyle.

Insight 5. Some violation types have low impact. For example, `PZLA_PREFER_ZERO_LENGTH_ARRAYS` refers to the `FindBugs`' rule that an array-return method should consider returning a zero-length array rather than null. Its fix pattern is replacing the null reference with a corresponding zero-length array. Developers ignored or rejected patches for this type of violations because they do have null-check to prevent these violations. If there is no null-check for these violations, the invocations of these methods would be identified as `NP_NULL_ON_SOME_PATH` violations. Thus, `PZLA_PREFER_ZERO_LENGTH_ARRAYS` might not be useful in practice.

Insight 6. Some fix patterns make programs fail to compile. For example, the common fix pattern of `RV_RETURN_VALUE_IGNORED_BAD_PRACTICE` violations is adding an `if` statement to check the return `boolean` value of the violated source code. We note that return values of some violated source code of this violation type is not `boolean` type. Copying the change behavior of the fix pattern directly to this kind of violations will lead to compilation errors.

Insight 7. Some fix patterns make programs fail to checkstyle. Figure 23 presents an example of a patch generated by our method for a `MS_SHOULD_BE_FINAL` violation in *XmlConverter.java* file of `camel` [29] project, which makes the project fail to checkstyle.

Insight 8. Some fix patterns of some violations are controversial. For example, the fix patterns of `DM_NUMBER_CTOR` violations are replacing the Number constructor with static Number `valueOf` method. It has been found that changing `new Integer()` to `Integer.valueOf()` and changing `Integer.valueOf()` to `new Integer()` were reverted repeatedly. Some developers find that `new Integer()` outperforms `Integer.valueOf()`, and some other developers find that `Integer.valueOf()` outperforms `new`

[26]https://github.com/apache/commons-lang

[27]https://garygregory.wordpress.com/2015/11/03/java-lowercase-conversion-turkey/

[28]https://github.com/eclipse/eclipse.platform/blob/R4_6_maintenance/ant/org.eclipse.ant.core/src_ant/org/eclipse/ant/internal/core/ant/InternalAntRunner.java#L1484

[29]https://github.com/apache/camel

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

`Integer()`. Additionally, some developers report that `Double.doubleToLongBits()` could be more efficient than `new Double()` and `Double.valueOf()` when comparing two double values with `equals()` method. We infer that the `DM_NUMBER_CTOR` or `DM_FP_NUMBER_CTOR` violations should be identified and revised based on the specific function, otherwise, developers may be prone to ignoring these kinds of violations.

4 Discussion

4.1 Threats to validity

A major threat to external validity of our study is the focus on `FindBugs` as the static analysis tool, with specific violation types and names. Fortunately, the code problems described by `FindBugs` violations are similar to the violations described by other static analysis tools. For example, `NP_NULL_ON_SOME_PATH` violations in FindBugs, `Null dereference` violations in Facebook Infer, and `ThrowNull` violations in Google ErrorProne are about the same issue: A NULL pointer is dereferenced and will lead to a NullPointerException when the code is executed. With the fix pattern of `NP_NULL_ON_SOME_PATH` of FindBugs mined in this study, we fixed 9 out of 10 different cases (each is from a distinct project in our subjects) of `Null dereference` violations detected by Facebook Infer and 8 out of 10 different cases of `ThrowNull` violations detected by Google ErrorProne, respectively. It shows the potential generalizability of the inferred fix patterns. We acknowledge, however, that there are some differences between `FindBugs` violations and other static analysis violations. Another threat to external validity of our study is that the fix patterns of violations are mined from open-source projects. Our findings might not applicable to industry projects that could have specific policies related to code quality.

Threats to internal validity include the limitations of the underlying tools used in this study (i.e., `FindBugs` and `GumTree`). `GumTree` may produce unfeasible edit scripts. To reduce this threat, we have added extra labels into `GumTree`. `FindBugs` may produce some violations with inaccurate positions. To reduce this threat, we re-locate and re-visit the violated source code with the bug descriptions of some violation types by `FindBugs`. `FindBugs` may yield high false positives. In order to reduce this threat, we focus on the common fixed violations in this study since common fixed violations are really concerned by developers. If the common fixed violations were addressed by common fix patterns, the common fixed violations are highly possible to be true positives and the common fix patterns are highly possible to be effective resolutions. These threats could be further reduced by developing more advanced tools.

Threats to internal validity also involve limitations in our method. Violation tracking may produce false positive fixed violations. We combine the commit `DiffEntry` and diffs parsed by `GumTree` to reduce this threat. Irrelevant code contexts can interfere with patterns mining. For example, one statement contains complex expressions, which may lead to a high number of irrelevant tokens. If this kind of violations were not filtered out in this study, it would increase the interference of noise. To reduce this validity, our study should be replicated in future work by extracting and analyzing the key violated source code with relevant code contexts identified using system dependency graphs. In this study, we also find that some violations are replaced by method invocations which encapsulate the detailed source code changes of fixing the corresponding violations. The method we proposed extracts source code changes from source code changing positions of violations. It is challenging to extract source code changes from these kinds of fixed violations. In order to reduce this validity, we are planning to integrate static analysis technique into our method to get more detailed source code changes.

4.2 Insights on unfixed violations

Given the high proportion of violations that were found to remain unfixed in software projects, we investigate the potential reasons for this situation. By comparing, in Section 3.3, the code patterns of unfixed violations against those of fixed patterns, we note that they are commonly shared, suggesting that the reasons are not mainly due to the violation code characteristics. Instead, we can enumerate other implicit reasons based on the observation of statistical data as well as the comments received during our live study to fix violations in ten open source projects.

- Actually, many developers do not use `FindBugs` as part of their development tool chain. For example, we found that only 36% of projects in our study include a commit mentioning `FindBugs`. Also, interestingly, in the cases of projects where we found that only 2% (1,944/88,927) of fixed `FindBugs` violations explicitly refer to the `FindBugs` tool in commit messages.
- As a static analysis tool, `FindBugs` yields a significant number of false positives: i.e., violations that developers do not consider as being true violations. We indeed highlighted some code patterns of detected violations that they are inconsistent with the descriptions provided by `FindBugs` (cf. Section 3.5).
- Our interaction with developers helped us confirm that developers do not consider most `FindBugs` violations as being severe enough to deserve attention in their development process.
- Some violations identified by `FindBugs` might be controversial because we find that some fix patterns of some violations are controversial (cf. Insight 8 in Section 3.7.3).
- Finally, with our live study, we note that some developers may be willing to fix violations if they had in hand some fix patterns. Unfortunately, `FindBugs` only reports the violations, and does not provide in many cases any hint on how to deal with them. Our work is towards filling this gap systematically based on harvested knowledge from developer fixes.

5 Related work

5.1 Static analysis

Classification of Actionable and Unactionable Violations: Static analysis violations are studied and investigated from different aspects. Several studies attempted to classify actionable (likely to be true positive) and unactionable (false positive) violations by using machine learning techniques [13], [27], [29]. Classifying new and recurring alarms

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2018.2884955, IEEE Transactions on Software Engineering

is necessary to prune identical alarms between subsequent releases. Hash code matching [25] and coding pattern analysis [12] can be used for identifying recurring violations. Model checking techniques [66], [67] and constraint solvers [68], [69] can also verify true violations and prune false positive. As discussed in Section 3.5, trivial violations reported by FindBugs can be treated as false positives by developers, but they cannot be identified by previous work since they are negligible issues and too trivial to be addressed by developers. Investigating the violations recurrently addressed by developers like this study could reduce this threat to identify true positive violations.

Violation Prioritization: Violation prioritization can provide a ranked list so that developers focus on important ones first. Z-ranking [70] prioritizes violations based on observations of real error trends. Jung et al. leveraged Bayesian statistics to rank violations [71]. History-based prioritization [72], [73], [74] utilizes history of program changes to prioritize violations. In addition, several studies attempted to leverage user feedback to rank violations [22], [26], [75]. However, these works did not investigate violations with the big number of violations as our work, from multiple aspects as we done. Thus, our work can provide more reliable insights for violation ranking than these works.

5.2 Change pattern mining

Empirical Studies on Change Patterns: Common change patterns are useful for various purposes. Pan et al. [62] explored common bug fix patterns in Java programs to understand how developers change programs to fix a bug. Their fix patterns are, however, in a high-level schema (e.g. "If-related: Addition of Post-condition Check (IF-APTC)"). Based on the insight, PAR [21] leveraged common predefined fix patterns for automated program repair, that only contain six fix patterns which can only be used to fix a small number of bugs. Martinez and Monperrus further investigated repair models that can be utilized in program fixing while Zhong and Su [76] conducted a large-scale study on bug fixing changes in open source projects. Tan et al. [77] analyzed anti-patterns that may interfere with the process of automated program repair. However, all of them studied code changes at the statement level, which is not as fine-grained as our work that extracts fine-grained code changes with an extended version of GumTree [16].

Pattern Mining for Code Change: SYDIT [78] and Lase [79] generate code changes to other code snippets with the extracted edit scripts from examples in the same application. RASE [80] focuses on refactoring code clones with Lase edit scripts [79]. FixMeUp [81] extracts and applies access control templates to protect sensitive operations. Their objectives are not to address issues caused by faulty code in program, such as the static analysis bugs studied in this study. REFAZER implements an algorithm for learning syntactic program transformations for C# programs from examples [82] to correct defects in student submissions, which however are mostly useless across assignments [83] and are not really defects in the wild as the violations in our study. Genesis [83] heuristically infers application-independent code transform patterns from multiple applications to fix bugs, but its code transform patterns are tightly coupled with the nature and syntax of three kinds of bugs (i.e., null pointer, out of bounds, and class cast defects). Koyuncu et al. [84] have generalized this approach with FixMiner to mining fix patterns for all types of bugs given a large dataset. Our work tries to mine the common fix patterns for general static analysis violations which are not application-independent. Closely related to our work is the concurrent work of Reudismam et al. [85] who try to learn quick fixes by mining code changes to fix PMD violations [5]. Their approach aims at learning code change templates to be systematically applied to refactor code. Our approach can be used for a similar scenario, and scales to a huge variety of violation types.

5.3 Bug datasets

Several datasets of real-world bugs have been proposed in the literature to evaluate approaches in software testing, software repair, and defect prediction approaches. Do et al. [86] have thus contributed to testing techniques with a controlled experimentation platform. The associated dataset was added to the SIR database, which provides a widely-used test bed for debugging and test suite optimization. Lu et al. [87] and Cifuentes et al. [88] have respectively proposed BugBench and BegBunch as benchmarks for bug detection tools. Similarly, Dallmeier et al. [89] have proposed iBugs, a benchmark for bug localization. Similarly to our process, their benchmark was obtained by extracting historical bug data. Bug data can also be found in the PROMISE repository [90] which includes a large variety of datasets for software engineering research. Le Goues et al. [91] have designed the GenProg benchmark with C bugs. Just et al. [20] have proposed Defects4J to evaluate software testing and repair approaches. Their dataset was collected from the recent history of five widely-used Java bugs, for which they could include the associated test suites. To ensure the reliability of our experiments, we also collect subjects to identify violations and corresponding patches from real-world projects. The existing bug datasets focus on the bugs that make programs fail to pass some test case(s), but our data is about static analysis violations which may not fail to pass test cases.

5.4 Program repair

Recent studies of program repair have presented several achievements. There are mainly two lines of research: (1) fully automated repair and (2) patch hint suggestion. The former focuses on automatically generating patches that can be integrated into a program without human intervention. The patch generation process often includes patch verification to figure out whether the patch does not break the original functionality when it is applied to the program. The verification is often achieved by running a given test suite. Automatize violation repair is included in our future work. The latter techniques suggest code fragments that can help create a patch rather than generating a patch ready to integrate. Developers may use the suggestions to write patches and verify them manually, that is similar to the patch generation of our work.

Fully Automated Repair: Automated program repair is pioneered by GenProg [92], [93]. This approach leverages genetic programming to create a patch for a given buggy program. It is followed by an acceptability study [94] and systematic evaluation [95]. Regarding the acceptability issue, Kim et al. [21] advocated GenProg may generate nonsensical patches and proposed PAR to deal with the issue. PAR leverages human-written patches to define fix templates and can generate more acceptable patches. HDRepair [65] leverages bug fixing history of many projects to provide better patch candidates to the random search process. Recently, LSRrepair [96] proposes a live search approach to the ingredients of automated repair using code search techniques. While GenProg relies on randomness, utilizing program synthesis techniques [97], [98], [99] can directly generate patches even though they are limited to a certain subset of bugs. Other notable approaches include contract-based fixing [100], program repair based on behavior models [101], and conditional statement repair [102]. This study does not focus on the fully automated program repair but the automated fix pattern mining for violations.

Patch Hint Suggestion: Patch suggestion studies explored diverse dimensions. MintHint [103] generates repair hints based on statistical analysis. Tao et al. [104] investigated how automatically generated patches can be used as debugging aids. Bissyandé suggests patches for bug reports based on the history of patches [105]. Caramel [106] focuses on potential performance defects and suggests specific types of patches to fix those defects. Our study is closely related to patch hint suggestion since we can suggest top-10 most similar fix patterns for targeting violations. The difference is that fix patterns in this work are mined from developers' patch submissions of static analysis violations.

Empirical Studies on Program Repair: Many studies have explored properties of program repair. Monperrus [107] criticized issues of patch generation learned from human-written patches [21]. Barr et al. discussed the plastic surgery hypothesis [108] that theoretically illustrates graftibility of bugs from a given program. Long and Rinard analyzed the search space issues for population-based patch generation [109]. Smith et al. presented an argument of overfitting issues of program repair techniques [110]. Koyuncu et al. [8] compared the impact of different patch generation techniques in Linux kernel development. Benchmarks for program repair are proposed for different programming languages [20], [91]. Based on a benchmark, a large-scale replication study was conducted [63]. More recently, Liu et al. [111] investigated the distribution of code entities impacted by bug fixes with fine-grained granularity, and found that some static analysis tools (e.g., FindBugs [14] and PMD [5]) are involved in some bug fixes.

6 Conclusion

In this study, we investigate recurrences of violations as well as their fixing changes, collected from open source Java projects. The yielded findings provide a number of insights into prioritization of violations for developers, as well as for researchers to improve violation reporting.

In this paper, we propose an approach to mine code patterns and fix patterns of static analysis violations by leveraging CNNs and *X-means*. The identified fix patterns are evaluated through three experiments. They are first applied to fixing many unfixed violations in our subjects. Second, we manage to get 67 of 116 generated patches accepted by the developer community and eventually merged into 10 open source Java projects. Third, interestingly, the mined fix patterns were effective for addressing 4 real bugs in the Defects4J benchmark.

As further work, we plan to combine fix pattern mining with automated program repair techniques to generate violation fixes more automatically. In the live study, we find that some common violations never occurred in latest versions of those projects. We postulate that violation recurrences may be time-varying. Our future work also includes studies on the time-variant issue of violation recurrences to further figure out the historic changes of fixed violations and the latest trend of violations, which may help new directions of violation prioritization.

References

[1] Synopsys, "Coverity," http://www.coverity.com/, Last Accessed: Nov.2017.

[2] ObjectLab, http://www.objectlab.co.uk/, Last Accessed: Nov. 2017.

[3] Github, "semmle," https://semmle.com/, Last Accessed: Nov. 2017.

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

[5] Github, "Pmd: an extensible cross-language static code analyzer," https://pmd.github.io/, Last Accessed: Nov. 2017.

[6] I. P. A. Group, "Splint," http://splint.org/, Last Accessed: Nov. 2017.

[7] D. Marjamäki, "Cppcheck," http://cppcheck.sourceforge.net/, Last Accessed: Nov. 2017.

[8] A. Koyuncu, T. F. Bissyand, 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.

[9] LLVM, "Clang static analyzer," https://clang-analyzer.llvm.org/, Last Access: Nov. 2017.

[10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 57–72.

[12] R. D. Venkatasubramanyam and S. Gupta, "An Automated Approach to Detect Violations with High Confidence in Incremental Code Using a Learning System," in *Companion Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 472–475.

[13] S. Heckman and L. Williams, "A Model Building Process for Identifying Actionable Static Analysis Alerts," in *International Conference on Software Testing Verification and Validation, 2009. ICST '09*. IEEE, Apr. 2009, pp. 161–170.

[14] F. 3.0.1, "Findbugs bug descriptions," http://findbugs.sourceforge.net/bugDescriptions.html, Last Accessed: Nov. 2017.

[15] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 1–8.

[16] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering*. Vasteras, Sweden - September 15 - 19: ACM, 2014, pp. 313–324.

[17] Y. Goldberg and O. Levy, "word2vec explained: Deriving mikolov et al.s negative-sampling word-embedding method," *CoRR*, 2014.

[18] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing." in *AAAI*, 2016, pp. 1287–1293.

[19] D. Pelleg, A. W. Moore *et al.*, "X-means: Extending k-means with efficient estimation of the number of clusters." in *ICML*, vol. 1, 2000, pp. 727–734.

[20] 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*. New York, NY, USA: ACM, 2014.

[21] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[22] H. Shen, J. Fang, and J. Zhao, "EFindBugs: Effective Error Ranking for FindBugs," in *Verification and Validation 2011 Fourth IEEE International Conference on Software Testing*, Mar. 2011, pp. 299–308.

[23] Apache, "Maven," https://maven.apache.org/, Last Accessed: Nov. 2017.

[24] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. de Moor, M. Schfer, and J. Tibble, "Tracking Static Analysis Violations over Time to Capture Developer Characteristics," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 437–447.

[25] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking Defect Warnings Across Versions," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2006, pp. 133–136.

[26] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: ACM, 2008, pp. 41–50.

[27] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014, pp. 152–161.

[28] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, Apr. 2011.

[29] J. Yoon, M. Jin, and Y. Jung, "Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM," in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 2, Dec. 2014, pp. 3–6.

[30] K. Yi, H. Choi, J. Kim, and Y. Kim, "An empirical study on classification methods for alarms from a bug-finding static C analyzer," *Information Processing Letters*, vol. 102, no. 2–3, pp. 118–123, Apr. 2007.

[31] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, "Subject independent facial expression recognition with robust face detection using a convolutional neural network," *Neural Networks*, vol. 16, no. 5-6, pp. 555–559, 2003.

[32] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *International Conference on Knowledge Science, Engineering and Management*. Springer, 2015, pp. 547–553.

[33] T. Hastie, R. Tibshirani, and J. Friedman, "Unsupervised learning," in *The elements of statistical learning*. Springer, 2009, pp. 485–585.

[34] D. W. Aha, *Lazy learning*. Washington, DC: Springer, 1997.

[35] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management*. Chongqing, China: Springer, 2015, pp. 547–553.

[36] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proceedings of the 32nd International Conference on Machine Learning*. Lille, France: JMLR.org, 2015, pp. 2123–2132.

[37] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing." in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. Phoenix, Arizona, USA.: AAAI, 2016, pp. 1287–1293.

[38] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle, WA, USA: ACM, 2016, pp. 631–642.

[39] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. Urbana, IL, USA: IEEE, 2017, pp. 135–146.

[40] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *Proceedings of the 39th International Conference on Software Engineering*. Buenos Aires, Argentina: IEEE, 2017, pp. 438–449.

[41] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 933–944.

[42] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[43] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[44] N. D. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," *arXiv preprint arXiv:1710.06159*, 2017.

[45] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, 1998.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *International Conference on Learning Representations: Workshop*, 2013.

[47] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. Austin, TX, USA, May 14-22: ACM, 2016, pp. 297–308.

[48] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[49] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[50] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[51] LIP6, "Coccinelle," http://coccinelle.lip6.fr/, Last Accessed: Nov. 2017.

[52] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 247–260.

[53] G. Georgios and S. Diomidis, "Ghtorrent," http://ghtorrent.org/halloffame.html, Last Accessed: Nov. 2017.

[54] A. Vetro, M. Morisio, and M. Torchiano, "An empirical validation of findbugs issues related to defects," in *Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*. IET, 2011, pp. 144–153.

[55] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 241–252.

[56] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.

[57] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems 26 (NIPS)*, pp. 3111–3119, 2013.

[58] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[59] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer Vision and Pattern Recognition (CVPR), IEEE Conference on*. IEEE, 2012, pp. 3642–3649.

[60] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Mining Software Repositories (MSR), 10th IEEE Working Conference on*. San Francisco, CA, USA, May 18-19: IEEE, 2013, pp. 121–130.

[61] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu , HI, USA, May 21-28: ACM, 2011, pp. 351–360.

[62] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, Aug. 2008.

[63] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, pp. 1–29, 2016.

[64] 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*. IEEE Press, 2017, pp. 416–426.

[65] 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, Mar. 2016, pp. 213–224.

[66] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh, "Precise Analysis of Large Industry Code," in *2012 19th Asia-Pacific Software Engineering Conference*, vol. 1, Dec. 2012, pp. 306–309.

[67] T. Muske and U. P. Khedker, "Efficient elimination of false positives using static analysis," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2015, pp. 270–280.

[68] Y. Kim, J. Lee, H. Han, and K.-M. Choe, "Filtering false alarms of buffer overflow analysis using SMT solvers," *Information and Software Technology*, vol. 52, no. 2, pp. 210–219, Feb. 2010.

[69] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "SMT-Based False Positive Elimination in Static Program Analysis," in *SpringerLink*. Springer, Berlin, Heidelberg, Nov. 2012, pp. 316–331.

[70] T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," in *SpringerLink*. Springer, Berlin, Heidelberg, Jun. 2003, pp. 295–315.

[71] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis," in *SpringerLink*. Springer, Berlin, Heidelberg, Sep. 2005, pp. 203–217.

[72] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, Jun. 2005.

[73] S. Kim and M. D. Ernst, "Prioritizing Warning Categories by Analyzing Software History," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, May 2007, pp. 27–27.

[74] ——, "Which Warnings Should I Fix First?" in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, NY, USA: ACM, 2007, pp. 45–54.

[75] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2004, pp. 83–93.

[76] H. Zhong and Z. Su, "An Empirical Study on Real Bug Fixes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 913–923.

[77] 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*. New York, NY, USA: ACM, 2016, pp. 727–738.

[78] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.

[79] ——, "LASE: locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 502–511.

[80] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 392–402.

[81] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: finding missing security checks when you do not know what checks are," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 1069–1084.

[82] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 404–415.

[83] 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*. ACM, 2017, pp. 727–739.

[84] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *arXiv preprint arXiv:1810.01791*, 2018.

[85] R. Rolim, G. Soares, R. Gheyi, and L. D'Antoni, "Learning quick fixes from code repositories," *arXiv preprint arXiv:1803.03806*, 2018.

[86] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[87] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5, 2005.

[88] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "Begbunch: Benchmarking for c bug detection tools," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, 2009, pp. 16–20.

[89] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 433–436.

[90] N. S. U. Computer Science, "tera-promise repository," http://openscience.us/repo/, Last Accessed: Nov. 2017.

[91] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[92] 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*, ser. ICSE '09. IEEE Computer Society, 2009, pp. 364–374.

[93] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, Feb. 2012.

[94] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 177–187.

[95] 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 *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 3–13.

[96] K. Liu, K. Anil, K. Kim, D. Kim, and T. F. Bissyandé, "Live search of fix ingredients for automated program repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, Nara, Japan, 2018.

[97] 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 '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781.

[98] 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*. New York, NY, USA: ACM, 2016, pp. 691–701.

[99] ——, "DirectFix: Looking for Simple Program Repairs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 448–458.

[100] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in

Proceedings of the 19th international symposium on Software testing and analysis. Trento, Italy: ACM, 2010, pp. 61–72.

[101] V. Dallmeier, A. Zeller, and B. Meyer, "Generating Fixes from Object Behavior Anomalies," in *24th IEEE/ACM International Conference on Automated Software Engineering, 2009. ASE '09*. IEEE, Nov. 2009, pp. 550–554.

[102] J. Xuan, M. Martinez, F. DeMarco, M. Clment, 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, Jan. 2017.

[103] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: Automated Synthesis of Repair Hints," in *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 266–276.

[104] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically Generated Patches As Debugging Aids: A Human Study," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2014, pp. 64–74.

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

[106] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 902–912.

[107] 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*. New York, NY, USA: ACM, 2014, pp. 234–242.

[108] 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*. New York, NY, USA: ACM, 2014, pp. 306–317.

[109] F. Long and M. Rinard, "An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems," in *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016, pp. 702–713.

[110] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2015, pp. 532–543.

[111] K. Liu, D. Kim, L. Li, K. Anil, T. F. Bissyandé, and Y. L. Traon, "A closer look at real-world patches," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME 2018, Madrid, Spain, 2018, pp. 304–315.

[112] S. Root, "Evaluation of findbugs," https://www.cs.cmu.edu/~aldrich/courses/654/tools/square-root-FindBugs-2009.pdf, Last Accessed: Aug. 2017.