Noname manuscript No.
(will be inserted by the editor)

FixMiner: Mining Relevant Fix Patterns for Automated Program Repair

Anil Koyuncu · Kui Liu · Tegawendé F. Bissyandé · Dongsun Kim · Jacques Klein · Martin Monperrus · Yves Le Traon

Received: date / Accepted: date

Abstract Patching is a common activity in software development. It is generally performed on a source code base to address bugs or add new functionalities. In this context, given the recurrence of bugs across projects, the associated similar patches can be leveraged to extract generic fix actions. While the literature includes various approaches leveraging similarity among patches to guide program repair, these approaches often do not yield fix patterns that are tractable and reusable as actionable input to APR systems.

In this paper, we propose a systematic and automated approach to mining relevant and actionable fix patterns based on an iterative clustering strategy applied to atomic changes within patches. The goal of FixMiner is thus to infer separate and reusable fix patterns that can be leveraged in other patch generation systems. Our technique, `FixMiner`, leverages `Rich Edit Script` which is a specialized tree structure of the edit scripts that captures the AST-level context of the code changes. `FixMiner` uses different tree representations of `Rich Edit Scripts` for each round of clustering to identify similar changes. These are abstract syntax trees, edit actions trees, and code context trees.

We have evaluated `FixMiner` on thousands of software patches collected from open source projects. Preliminary results show that we are able to mine accurate patterns, efficiently exploiting change information in `Rich Edit Scripts`. We further integrated the mined patterns to an automated program repair prototype, $\texttt{PAR}_{\texttt{FixMiner}}$, with which we are able to correctly fix 26 bugs of the

A. Koyuncu, K. Liu, T. F. Bissyandé, J. Klein, and Y. Le Traon
SnT, University of Luxembourg
E-mail: {anil.koyuncu, kui.liu, tagewende.bissyande, jacques.klein, yves.le traon}@uni.lu

D. Kim
Furiosa.ai
E-mail: darkrsw@furiosa.ai

M. Monperrus
KTH Royal Institute of Technology
E-mail: martin.monperrus@csc.kth.se

Defects4J benchmark. Beyond this quantitative performance, we show that the mined fix patterns are sufficiently relevant to produce patches with a high probability of correctness: 81% of $\texttt{PAR}_{\texttt{FixMiner}}$'s generated plausible patches are correct.

1 Introduction

Code change patterns have various uses in the software engineering domain. They are notably used for labeling changes [77], triaging developer commits [87] or predicting changes [96]. In recent years, fix patterns have been heavily leveraged in the software maintenance community, notably for building patch generation systems, which now attract growing interest in the literature [68]. Automated Program Repair (APR) has indeed gained incredible momentum, and various approaches [10,12,27,30,32,33,41–44,50,51,56–58,64,72,88,90,94,95] have been proposed, aiming at reducing manual debugging efforts through automatically generating patches. A common and reliable strategy in automatic program repair is to generate concrete patches based on fix patterns [33] (also referred to as fix templates [54] or program transformation schemas [27]). Several APR systems [15,27,33,50,51,54,63,81] in the literature implement this strategy by using **diverse sets of fix patterns** obtained either via manual generation or automatic mining of bug fix datasets.

In PAR [33], the authors mined fix patterns by inspecting 60,000 developer patches manually. Similarly, for Relifix [84], a manual inspection of 73 real software regression bug fixes is performed to infer fix patterns. Manual mining is however tedious, error-prone, and cannot scale. Thus, in order to overcome the limitations of manual pattern inference, several research groups have initiated studies towards automatically inferring bug fix patterns. With Genesis [56], Long *et al.* proposed to automatically infer code transforms for patch generation. Genesis infers 108 code transforms, from a space of 577 sampled transforms, with specific code contexts. However, this work limits the search space to previously successful patches from only three classes of defects of Java programs: null pointer, out of bounds, and class cast related defects.

Liu and Zhong [54] proposed SOFix to explore fix patterns for Java programs from Q&A posts in Stack Overflow, which mines patterns based on GumTree [17] edit scripts, and builds different categories based on repair pattern isomorphism. SOFix then mines a repair pattern from each category. However, the authors note that most of the categories are redundant or even irrelevant, mainly due to two major issues: (1) a considerable portion of code samples are designed for purposes other than repairing bugs; (2) since the underlying GumTree tool relies on structural positions to extract modifications, these "modifications do not present the desirable semantic mappings". They relied on heuristics for manually filtering categories (e.g., categories that contain several modifications), and then after SOFIX mines repair patterns they have to manually select useful ones (e.g., merging some repair patterns due to their similar semantics).

Liu et al. [48] and Rolim et al. [80] proposed to mine fix patterns from static analysis violations from FindBugs and PMD respectively. Both approaches, leverage a similar methodology in the inference process. Rolim et al. [80] rely on the distance among edit scripts: edit scripts with low distances among them are grouped together according to a defined similarity threshold. Liu et al. [48], on the other hand, leverage deep learning to learn features of edit scripts, to find clusters of similar edit scripts. Eventually, both works do not consider code context in their edit scripts and manually derive the fix patterns from the clusters of similar edit scripts of patches.

In another vein, CapGen [90] and SimFix [30] propose to use frequency of code change actions. The former uses it to drive patch selection, while the latter uses it in computing donor code similarity for patch prioritization. In both cases, however, the notion of patterns is not an actionable artefact, but rather a supplementary information that guides their patch generation system. Although we concurrently[1] share with SimFix and CapGen the idea of adding more contextual information for patch generation, our objective is to infer actionable fix patterns that are tractable and reusable as input to other APR systems.

Table 1 presents an overview of different automated mining strategies implemented in literature to obtain diverse sets of fix patterns. Some of the strategies are directly presented as part of APR systems, while others are independent approaches. We characterize the different strategies by considering the diff representation format, the use of contextual information, the tractability of patterns (i.e., what extent they are separate and reusable components in patch generation systems), and the scope of mining (i.e., whether the scope is limited to specific code changes). Overall, although the literature approaches can come handy for discovering diverse sets of fix patterns, the reality is that the intractability of the fix patterns and the generalizability of the mining strategies remain a challenge for deriving relevant patterns for program repair.

Table 1: Comparison of fix pattern mining techniques in the literature.

	Genesis [56]	SOFix [54]	Liu et al. [48]	Rolim et al. [80]	CapGen [90]	SimFix [30]	FixMiner
Diff notation	Transform	Edit Script	Edit Script	Edit Script	Edit Script	Edit Script	Edit Script
Scope	Three defect classes	Any bug type	Static analysis violations	Static analysis violations	Any bug type	Insert and update changes only	Any bug type
Context information	✗	✗	✗	✗	✓	✓	✓
Tractability of Patterns*	Medium	High	High	High	Low	Low	High

* **High:** Patterns are part of output and reusable as input to APR systems
Medium: Patterns are not readily usable
Low: Patterns are not separate or available as output.

This paper. We propose to investigate the feasibility of mining relevant fix patterns that can be easily integrated into an automated pattern-based program repair system. To that end, we propose an iterative and three-fold clustering strategy, `FixMiner`, to discover relevant fix patterns automatically from atomic changes within real-world developer fixes. `FixMiner` is a pattern mining approach to produce fix patterns for program repair systems. We present

[1] The initial version of this paper was written concurrently to SimFix and CapGen.

in this paper the concept of `Rich Edit Script` which is a specialized tree data structure of the edit scripts that captures the AST-level context of code changes. To infer patterns, `FixMiner` leverages identical trees, which are computed based on the following information encoded in `Rich Edit Scripts` for each round of the iteration: abstract syntax tree, edit actions tree, and code context tree.

Contribution. We propose the `FixMiner` pattern mining tool as a separate and reusable component that can be leveraged in other patch generation systems.

Paper content. Our contributions are:

– We present the architecture of a pattern inference system, `FixMiner`, which builds on a three-fold clustering strategy where we iteratively discover similar changes based on different tree representations encoding contexts, change operations and code tokens.
– We assess the capability of `FixMiner` to discover patterns by mining fix patterns among 11 416 patches addressing user-reported bugs in 43 open source projects. We further relate the discovered patterns to those that can be found in a dataset used by the program repair community [31]. We assess the compatibility of `FixMiner` patterns with patterns in the literature.
– Finally, we investigate the relevance of the mined fix patterns by embedding them as part of an Automated Program Repair system. Our experimental results on the Defects4J benchmark show that our mined patterns are effective for fixing 26 bugs. We find that the `FixMiner` patterns are relevant as they lead to generating plausible patches that are mostly correct.

2 Motivation

Mining, enumerating and understanding code changes have been a key challenge of software maintenance in recent years. Ten years ago, Pan et al. have contributed with a manually-compiled catalog of 27 code change patterns related to bug fixing [77]. Such "bug fix patterns" however are generic patterns (e.g., IF-RMV: removal of an If Predicate) which represent the type of changes that are often fixing bugs. More recently, thanks to the availability of new AST differencing tools, researchers have proposed to automatically mine change patterns [47, 59, 74, 75]. Such patterns have been mostly leveraged for analysing and towards understanding characteristics of bug fixes. In practice, however, the inferred patterns may turn out to be irrelevant and intractable.

We argue however that mining fix patterns can help for guiding mutation operations for patch generation. In this case, there is a need to mine truly recurrent change patterns to which repair semantics can be attached, and to provide accurate, fine-grained patterns that can be actionable in practice, i.e., separate and reusable as inputs to other processes. Our intuition is that relevant patterns cannot be mined globally since bug fixes in the wild are subject to noisy details due to tangled changes [25]. There is thus a need to break patches into atomic units (contiguous code lines forming a hunk) and reason about the recurrences of the code changes among them. To mine changes, we propose to rely on the edit script format, which provides a fine-grained representation of code changes, where different layers of information are included:

- the context, i.e., AST node type of the code element being changed (e.g., a modifier in declaration statements, should not be generalized to other types of statements);
- the change operation (e.g., a "remove then add" sequence should not be confused with "add then remove" as it may have a distinct meaning in a hierarchical model such as the AST);
- and code tokens (e.g., changing calls to "*Log.warn*" should not be confused to any other API method).

Our idea is to iteratively find patterns within the contexts, and patterns of change operations for each context, and patterns of recurrently affected literals in these operations.

We now provide background information for understanding the execution as well as the information processed by `FixMiner`.

2.1 Abstract Syntax Tree

Code representation is an essential step in the analysis and verification of programs. Abstract syntax trees (ASTs), which are generally produced for program analysis and transformations, are data structures that provide an efficient form of representing program structures to reason about syntax and even semantics. An AST indeed represents all of the syntactical elements of the programming language and focuses on the rules rather than elements like braces or semicolons that terminate statements in some popular languages like Java or C. The AST is a hierarchical representation where the elements of each programming statement are broken down recursively into their parts. Each node in the tree thus denotes a construct occurring in the programming language.

Formally, let t be an AST and N be a set of AST nodes in t. An AST t has a root that is a node referred to as $root(t) \in N$. Each node $n \in N$ (and $n \neq root(t)$) has a parent denoted as $parent(n) = p \in N$. Note that there is no parent node of $root(t)$. Furthermore, each node n has a set of child nodes (denoted as $children(n) \subset N$). A label l (i.e., AST node type) is assigned to each node from a given alphabet L ($label(n) = l \in L$). Finally, each node has a string value v ($token(n) = v$ where $n \in N$ and v is an arbitrary string) representing the corresponding raw code token. Consider the AST representation in Figure 2 of the Java code in Figure 1. We note that the illustrated AST has nodes with labels matching structural elements of the Java language (e.g., `MethodDeclaration`, `IfStatement` or `StringLiteral`) and can be associated with values representing the raw tokens in the code (e.g., A node labelled `StringLiteral` from our AST is associated to value "Hi!").

```
public class Helloworld {
    public String hello(int i) {
        if (i == 0) return "Hi!";
    }
}
```

Fig. 1: Example Java class.

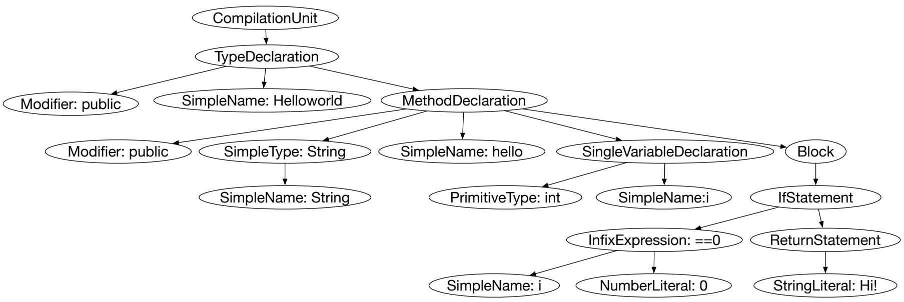


Fig. 2: AST representation of the `Helloworld` class.

2.2 Code Differencing

Differencing two versions of a program is the key pre-processing step of all studies on software evolution. The evolved parts must be captured in a way that makes it easy for developers to understand or analyze the changes. Developers generally deal well with text-based differencing tools, such as the GNU Diff represents changes as addition and removal of source code lines as shown in Figure 3. The main issue with this text-based differencing is that it does not provide a fine-grained representation of the change (i.e., `StringLiteral Replacement`) and thus it is poorly suited for systematically analysing the changes.

```
--- Helloworld_v1.java 2018-04-24 10:40:19.000000000 +0200
+++ Helloworld_v2.java 2018-04-24 11:43:24.000000000 +0200
@@ -1,5 +1,5 @@
 public class Helloworld {
     public String hello(int i) {
- if (i == 0) return "Hi!";
+ if (i == 0) return "Morning!";
     }
 }
```

Fig. 3: GNU diff format.

To address the challenges of code differencing, recent algorithms have been proposed based on tree structures (such as the AST). ChangeDistiller and GumTree are examples of such algorithms which produce *edit scripts* that detail the operations to be performed on the nodes of a given AST (as formalized in Section 2.1) to yield another AST corresponding to the new version of the code. In particular, in this work, we build on GumTree's core algorithms for preparing an edit script. An edit script is a sequence of edit actions describing the following code change actions:

– `UPD` where an $upd(n, v)$ action transforms the AST by replacing the old value of an AST node n with the new value v.
– `INS` where an $ins(n, n_p, i, l, v)$ action inserts a new node n with v as value and l as label. If the parent n_p is specified, n is inserted as the i^{th} child of n_p, otherwise n is the root node.
– `DEL` where a $del(n)$ action removes the leaf node n from the tree.
– `MOV` where a $mov(n, n_p, i)$ action moves the subtree having node n as root to make it the i^{th} child of a parent node n_p.

An edit action, embeds information about the node (i.e., the relevant node in the whole AST tree of the parsed program), the operator (i.e., `UPD`, `INS`, `DEL`, and `MOV`) which describes the action performed, and the raw tokens involved in the change.

2.3 Tangled Code Changes

Solving a single problem per patch is often considered as a best practice to facilitate maintenance tasks. However, often patches in real-world projects address multiple problems in a patch [37, 85]. Developers often commit bug fixing code changes together with changes unrelated to fix such as functionality enhancements, feature requests, refactorings, or documentation. Such patches are called tangled patches [25] or mixed-purpose fixing commits [71]. Nguyen et al. found that 11% to 39% of all the fixing commits used for mining archives were tangled [71].

Consider the example patch from GWT illustrated in Figure 4. The patch is intended to fix the issue[2] that reported a failure in some web browsers when the page is served with a certain mime type (i.e., application/xhtml+xml). The developer fixes the issue by showing a warning when such mime type is encountered. However, in addition to this change, a typo has been addressed in the commit. Since the typo is not related to the fix, the fixing commit is tangled. There is thus a need to separately consider single code hunks within a commit to allow the pattern inference to focus on finding recurrent atomic changes that are relevant to bug fixing operations.

3 Approach

`FixMiner` aims to discover relevant fix patterns from the atomic changes within bug fixing patches in software repositories. To that end, we mine code changes that are similar in terms of context, operations, and the programming tokens that are involved. Figure 5 illustrates an overview of the `FixMiner` approach.

[2] `https://github.com/gwtproject/gwt/issues/676`

```
--- a/dev/core/src/com/google/gwt/dev/shell/GWTShellServlet.java
+++ b/dev/core/src/com/google/gwt/dev/shell/GWTShellServlet.java
@@ -72,6 +72,8 @@

+ private static final String XHTML_MIME_TYPE = "application/xhtml+xml";
   private final Map loadedModulesByName = new HashMap();
   private final Map loadedServletsByClassName = new HashMap();
@@ -110,7 +112,7 @@
       writer.println("<html><body><basefont face='arial'>");
- writer.println("To launch an an application, specify a URL of the form <code>
     /<i>module</i>/<i>file.html</i></code>");
+ writer.println("To launch an application, specify a URL of the form <code>/<i>
     module</i>/<i>file.html</i></code>");
       writer.println("</body></html>");
     }
@@ -407,6 +409,8 @@
     }
+ maybeIssueXhtmlWarning(logger, mimeType, partialPath);

@@ -755,6 +759,25 @@

+ private void maybeIssueXhtmlWarning(TreeLogger logger, String mimeType,
+ String path) {
+ if (!XHTML_MIME_TYPE.equals(mimeType)) {
+ return;
+ }
+
+ String msg = "File was returned with content-type of \"" + mimeType
+ + "\". GWT requires browser features that are not available to "
+ + "documents with this content-type.";
+
+ int ix = path.lastIndexOf('.');
+ if (ix >= 0 && ix < path.length()) {
+ String base = path.substring(0, ix);
+ msg += " Consider renaming \"" + path + "\" to \"" + base + ".html\".";
+ }
+
+ logger.log(TreeLogger.WARN, msg, null);
+ }
```

Fig. 4: Tangled commit.

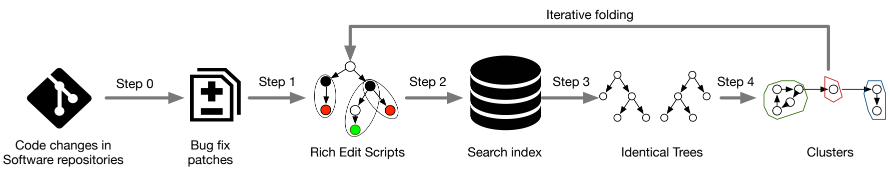


Fig. 5: The FixMiner Approach. At each iteration, the search index is refined, and the computation of tree similarity is specialized in specific AST information details.

3.1 Overview

In Step 0, as an initial step, we collect the relevant bug-fixing patches (cf. Definition 1) from project change tracking systems. Then, in Step 1, we compute a `Rich Edit Script` representation (cf. Section 3.3) to describe a code change in terms of the context, operations performed and tokens involved. Accordingly, we consider three specialized tree representations of the `Rich Edit Script` (cf. Definition 2) carrying information about either the impacted AST node types, or the repair actions performed, or the program tokens affected. `FixMiner` works in an iterative manner considering a single specialized tree representation in each pattern mining iteration, to discover similar changes: first, changes affecting the same code context (i.e., on identical abstract syntax trees) are identified; then among those identified changes, changes using the same actions (i.e., identical sequence of operations) are regrouped; and finally within each group, changes affecting the same tokens set are mined. Therefore, in `FixMiner`, we perform a three-fold strategy, carrying out the following steps in a pattern mining iteration:

- Step 2: We build a search index (cf. Definition 3) to identify the `Rich Edit Scripts` that must be compared.
- Step 3: We detect identical trees (cf. Definition 4) by computing the distance between two representations of `Rich Edit Scripts`.
- Step 4: We regroup identical trees into clusters (cf. Definition 5).

The initial pattern mining iteration uses `Rich Edit Scripts` computed in Step 1 as its input, where the following rounds use clusters of identical trees yielded in Step 4 as their input.

In the following sections, we present the details of Steps 1-4, considering that a dataset of bug fix patches is available.

3.2 Step 0 - Patch Collection

Definition 1 (**Patch**) A program patch is a transformation of a program into another program, usually to fix a defect. Let $\mathbb{P}$ being a set of programs, a patch is represented by a pair (p, p'), where $p, p' \in \mathbb{P}$ are programs before and after applying the patch, respectively. Concretely, a patch implements changes in code block(s) within source code file(s).

To identify bug fix patches in software repositories projects, we build on the bug linking strategies implemented in the Jira issue tracking software. We use a similar approach to the ones proposed by Fischer et al. [18] and Thomas et al. [86] in order to link commits to relevant bug reports. Concretely, we crawl the bug reports for a given project and assess the links with a two-step search strategy: (i) we check project commit logs to identify bug report IDs and associate the corresponding bug reports to commits; then (ii) we check for bug reports that are indeed considered as such (i.e., tagged as “BUG”) and are further marked as resolved (i.e., with tags “RESOLVED” or “FIXED”), and completed (i.e., with status “CLOSED”).

We further curate the patch set by considering bug reports that are fixed by a single commit. This provides more guarantees that the selected commits are indeed fixing the bugs in a single shot (i.e., the bug does not require supplementary patches [78]). Eventually, we consider only changes that are made on the source code files: changes on configuration, documentation, or test files are excluded.

3.3 Step 1 – `Rich Edit Script` Computation

> **Definition 2** (`Rich Edit Script`) A `Rich Edit Script` $r \in RE$ represents a patch as a specialized tree of changes. This tree describes which operations are made on a given AST, associated with the code block before patch application, to transform it into another AST, associated with the code block after patch application: i.e., $r : \mathbb{P} \to \mathbb{P}$. Each node in the tree is an AST node affected by the patch. Every node in `Rich Edit Script` has three different types of information: **Shape**, **Action**, and **Token**.

A bug-fix patch collected in open source change tracking systems is represented in the GNU diff format based on addition and removal of source code lines as shown in Figure 6. This representation is not suitable for fine-grained analysis of changes.

```
// modules. We need to move this code up to a common module.
- int indexOfDot = namespace.indexOf('.');
+ int indexOfDot = namespace.lastIndexOf('.');
          if (indexOfDot == -1) {
```

Fig. 6: Patch of fixing bug Closure-93 in Defects4J dataset.

To accurately reflect the change that has been performed, several algorithms have been proposed based on tree structures (such as the AST) [6, 9, 14, 17, 21, 24, 79]. ChangeDistiller [21] and GumTree [17] are state-of-the-art examples of such algorithms which produce edit scripts that detail the operations to be performed on the nodes of a given AST in order to yield another AST corresponding to the new version of the code. In particular, in this work, we selected the GumTree AST differencing tool which has seen recently a momentum in the literature for computing edit scripts. GumTree is claimed to build in a fast, scalable and accurate way the sequence of AST edit actions (a.k.a edit scripts) between the two associated AST representations (the buggy and fixed versions) of a given patch.

```
UPD SimpleName ''indexOf'' to ''lastIndexOf''
```

Fig. 7: GumTree edit script corresponding to Closure-93 bug fix patch represented in Figure 6.

Consider the example edit script computed by GumTree for the patch of Closure-93 bug from Defects4J illustrated in Figure 7. The intended behaviour of the patch is to fix the wrong variable declaration of *indexOfDot* due to a wrong method reference (*lastIndexOf* instead of *indexOf*) of java.lang.String object. GumTree edit script summarizes the change as an update operation on an AST node simple name (i.e., an identifier other than a keyword) that is modifying the identifier label (from *indexOf* to *lastIndexOf*).

Although GumTree edit script is accurate in describing the bug fix operation at fine-grained level, much of the contextual information describing the intended behaviour of the patch is missing. The information regarding method invocation, the method name (*java.lang.String*), the variable declaration fragment which assigns the value of the method invocation to *indexOfDot*, as well as the type information (*int* for *indexOfDot* - cf. Figure 6) that is implied in the variable declaration statement are all missing in the GumTree edit script. Since such contextual information is lost, the yielded edit script fails to convey the full syntactic and semantic meaning of the code change.

To address this limitation, we propose to enrich GumTree-yielded edit scripts by retaining more contextual information. To that end, we construct a specialized tree structure of the edit scripts which captures the AST-level context of the code change. We refer to this specialized tree structure as `Rich Edit Script`. A `Rich Edit Script` is computed as follows:

Given a patch, we start by computing the set of edit actions (edit script) using GumTree, where the set contains an edit action for each contiguous group of code lines (hunks) that are changed by a patch. In order to capture the context of the change, we re-organize edit actions under new AST (minimal) subtrees building an AST hierarchy. For each edit action in an edit script, we extract a minimal subtree from the original AST tree which has the GumTree edit action as its leaf node, and one of the following predefined node types as its root node: TypeDeclaration, FieldDeclaration, MethodDeclaration, SwitchCase, CatchClause, ConstructorInvocation, SuperConstructorInvocation or any Statement node. The objective is to limit the scope of context to the encompassing statement, instead of going backwards until the compilation unit (cf. Figure 2). We limit the scope of parent traversal mainly for two reasons: first, the pattern mining must focus on the program context that is relevant to the change; second, program repair approaches, which `FixMiner` is built for, generally target statement-level fault localization and patch generation.

Consider the AST differencing tree presented in Figure 8. From this diff tree, GumTree yields the leaf nodes (gray) of edit actions as the final edit script. To build the `Rich Edit Script`, we follow these steps:

i) For each GumTree-produced edit action, we remap it to the relevant node in the program AST;
ii) Then, starting from the *GumTree edit action* nodes, we traverse the AST tree of the parsed program from bottom to top until we reach a node of *predefined root node* type.
iii) For every *predefined root node* that is reached, we extract the AST subtree between the discovered *predefined root node* down to the leaf nodes mapped to the *GumTree edit actions*.

iv) Finally, we create an ordered[3] sequence of these extracted AST subtrees and store it as `Rich Edit Script`.

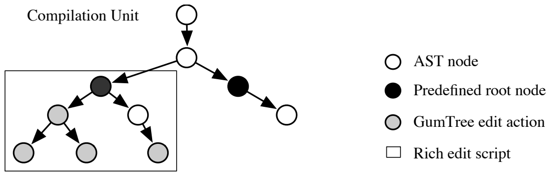


Fig. 8: Illustration of subtree extraction.

Concretely, with respect to our running example, consider the case of Closure-93 illustrated in Figure 6. The construction of the `Rich Edit Script` starts by generating the GumTree edit script (cf. Figure 7) of the patch. The patch consists of a single hunk, thus we expect to extract a single AST subtree, which is illustrated by Figure 9. To extract this AST subtree, first, we identify the node of the edit action "SimpleName" at position 4 in the AST Tree of program. Then, starting from this node, we traverse backward the AST tree until we reach the node "VariableDeclarationStatement" at position 1. We extract the AST subtree, by creating a new tree, setting "VariableDeclarationStatement" as root node of the new tree, and adding the intermediate nodes at positions 2,3 until we reach the corresponding node of the edit action "`UPD SimpleName`" at position 4. We create a sequence, and add the extracted AST subtree to the sequence.

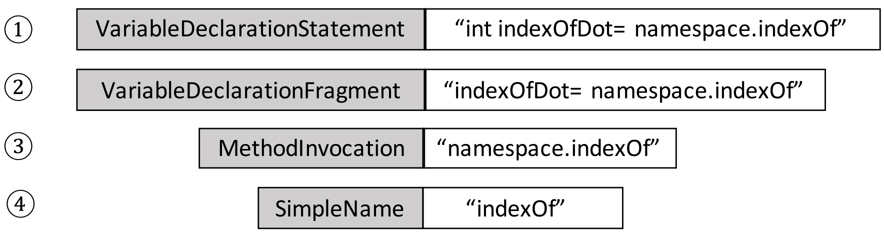


Fig. 9: Excerpt AST of buggy code (Closure-93).

`Rich Edit Scripts` are tree data structures. They are used to represent changes. In order to provide tractable and reusable patterns as input to other APR systems, we define the following string notation (cf. Grammar 1) based on syntactic rules governing the formation of correct `Rich Edit Script`.

Figure 10 illustrates the computed `Rich Edit Script`. The first line indicates the root node (no dashed line). 'UPD ' indicates the action type of the node, VariableDeclarationStatement corresponds to ast node type of the node, tokens between '@@' and '@TO@' contains the corresponding code tokens before the change, where as tokens between '@TO@' and '@AT' corresponding to new code tokens with the change. The three dashed (`---`) lines indicate a child node. Immediate children nodes contain three dashes while their children add another three dashes (`------`) preserving the parent-child relation.

[3] The order of AST subtrees follows the order of hunks of the GNU diff format.

$\langle richEditScript \rangle$ ::= $\langle node \rangle$+

$\langle node \rangle$ ::= '---'* $\langle move \rangle$
| '---'* $\langle delete \rangle$
| '---'* $\langle insert \rangle$
| '---'* $\langle update \rangle$

$\langle move \rangle$ ::= 'MOV ' $\langle astNodeType \rangle$ '@@' $\langle tokens \rangle$ '@TO@' $\langle astNodeType \rangle$ '@@' $\langle tokens \rangle$ '@AT@'

$\langle delete \rangle$ ::= 'DEL ' $\langle astNodeType \rangle$ '@@' $\langle tokens \rangle$ '@AT@'

$\langle insert \rangle$ ::= 'INS ' $\langle astNodeType \rangle$ '@@' $\langle tokens \rangle$ '@TO@' $\langle astNodeType \rangle$ '@@' $\langle tokens \rangle$ '@AT@'

$\langle update \rangle$ ::= 'UPD ' $\langle astNodeType \rangle$ '@@' $\langle tokens \rangle$ '@TO@' $\langle tokens \rangle$ '@AT@'

Grammar 1: Notation of `Rich Edit Script`

```
UPD VariableDeclarationStatement@@int indexOfDot = namespace.indexOf('.'); ↩
     @TO@ int indexOfDot = namespace.lastIndexOf('.'); @AT@
---UPD VariableDeclarationFragment@@indexOfDot = namespace.indexOf('.') @TO@ ↩
     indexOfDot = namespace.lastIndexOf('.') @AT@
-----UPD MethodInvocation@@namespace.indexOf('.') @TO@ ↩
     namespace.lastIndexOf('.') @AT@
--------UPD SimpleName@@MethodName:indexOf:['.'] @TO@ ↩
     MethodName:lastIndexOf:['.'] @AT@
```

Fig. 10: `Rich Edit Script` for Closure-93 patch in Defects4J. ↩ represents the carriage return character which is necessary for presentation reasons.

An edit action node carries the following three types of information: the AST node type (Shape), the repair action (Action), the raw tokens (Token) in the patch. For each of these three information types, we create separate tree representations from the `Rich Edit Script`, named as ShapeTree, ActionTree and TokenTree, each carrying respectively the type of information indicated by its name. Figures 11, 12, and 13 show ShapeTree, ActionTree, and TokenTree, respectively, generated for Closure-93.

```
VariableDeclarationStatement
---VariableDeclarationFragment
------MethodInvocation
---------SimpleName
```

Fig. 11: ShapeTree of Closure-93.

```
UPD root
---UPD child1
------UPD child1_1
---------UPD child1_1_1
```

Fig. 12: ActionTree of Closure-93.

```
@@int indexOfDot = namespace.indexOf('.'); @TO@ int indexOfDot = namespace.la...
---@@indexOfDot = namespace.indexOf('.') @TO@ indexOfDot = namespace.lastInde...
------@@namespace.indexOf('.') @TO@ namespace.lastIndexOf('.')
---------@@MethodName:indexOf:['.'] @TO@ MethodName:lastIndexOf:['.']
```

Fig. 13: TokenTree of Closure-93.

3.4 Step 2 – Search Index Construction

Definition 3 (Search Index) To reduce the effort of matching similar patches, a search index (SI) is used to confine the comparison space. Each fold ({Shape, Action, Token}) defines a search index: SI_{Shape}, SI_{Action}, and SI_{Token}, respectively. Each is defined as $SI_* : Q_* \rightarrow 2^{RE}$, where Q is a query set specific to each fold and $* \in \{Shape, Action, Token\}$.

Given that `Rich Edit Scripts` are computed for each hunk in a patch, they are spread inside and across different patches. A direct pairwise comparison of these `Rich Edit Scripts` would lead to a combinatorial explosion of the comparison space. In order to reduce this comparison space and enable a fast identification of `Rich Edit Scripts` to compare, we build search indices. A search index is a set of comparison sub-spaces created by grouping the `Rich Edit Scripts` with criteria that depend on the information embedded the used tree representation (Shape, Action, Token) for the different iterations.

The search indices are built as follows:

"Shape" search index. The construction process takes the ShapeTree representations of the `Rich Edit Scripts` produced by Step 1 as input, and groups them based on their tree structure in terms of AST node types. Concretely, `Rich Edit Scripts` having the same root node (e.g., IfStatement, MethodDeclaration, ReturnStatement) and same depth are grouped together. For each group, we create a comparison space by enumerating the pairwise combinations of the group members. Eventually, the "Shape" search index is built by storing an identifier per group, denoted as root node/depth (e.g., IfStatement/2, IfStatement/3, MethodDeclaration/4), and a pointer to its comparison space (i.e., the pairwise combinations of its members).

"Action" search index. The construction process follows the same principle as for "Shape" search index, except that the regrouping is based on the clustering output of ShapeTrees. Thus, the input is formed by ActionTree representations of the `Rich Edit Scripts` and the group identifier for each comparison space is generated as node/depth/ShapeTreeClusterId (e.g., IfStatement/2/1,MethodDeclaration/2/2) where ShapeTreeClusterId represents the id of the cluster yielded by the clustering (Steps 3-4) based on the ShapeTree information. Concretely, this means that the "Action" search index is built on groups of trees having the same shape.

"Token" search index. The construction process follows the same principle as for "Action" search index, using this time the clustering output of ActionTrees. Thus, the input is formed by TokenTree representations of the `Rich Edit Scripts` and the group identifier for each comparison space is generated as node/depth/ShapeTreeClusterId/ActionTreeClusterId (e.g., IfStatement/2/1/3,MethodDeclaration/2/2/1) where ActionTreeClusterId represents the id of the cluster yielded by the clustering (Steps 3-4) based on the ActionTree information.

3.5 Step 3 – Tree Comparison

Definition 4 (Pair of identical trees) Let $a = (r_i, r_j) \in R_{\text{identical}}$ be a pair of `Rich Edit Script` specialized tree representations if $d(r_i, r_j) = 0$, where $r_i, r_j \in RE$ and d is a distance function. $R_{\text{identical}}$ is a subset of $RE \times RE$.

The goal of tree comparison is to find identical tree representations of `Rich Edit Scripts` for a given fold. There are several straightforward approaches for checking whether two `Rich Edit Scripts` are identical. For example, syntactical equality could be used. However, we aim at making `FixMiner` a flexible and extensible framework where future research may tune threshold values for defining similar trees. Thus, we propose a generic approach for comparing `Rich Edit Scripts`, taking into account the diversity of information to compare for each specialized tree representation. To that end, we compute tree edit distances for the three representations of `Rich Edit Scripts` separately. The tree edit distance is defined as the sequence of edit actions that transform one tree into another. When the edit distance is zero (i.e., no operation is necessary to transform one tree to another) the trees are considered as identical. In Algorithm 1 we define the steps to compare `Rich Edit Scripts`.

The algorithm starts by retrieving the identifiers from the search index SI corresponding to the $fold$. An identifier is a pointer to a comparison subspace that contains pairwise combinations of tree representation of `Rich Edit Scripts` to compare (cf. Section 3.4). Concretely, we restore the `Rich Edit Scripts` of a given pair from the cache, and their corresponding specialized tree representation according to the $fold$: At the first iteration, we consider only trees denoted ShapeTrees, whereas in the second iteration we focus on ActionTrees, and TokenTrees for the third iteration. We compute the edit distance between the restored trees in two distinct ways.

- In the first two iterations (i.e, Shape and Action) we leverage again the edit script algorithm of GumTree [16, Section 3]. We compute the edit distance by simply invoking GumTree on restored trees as input, given that `Rich Edit Scripts` are indeed AST subtrees that are compatible with GumTree. Concretely, GumTree takes the two AST trees as input, and generates a sequence of edit actions (a.k.a edit script) that transform one tree into another, where the size of the edit script represents the edit distance between the two trees.
- For the third iteration (i.e., Token), since the relevant information in the tree is text, we use a text distance algorithm (Jaro-Winkler [29, 92]) to compute the edit distance between two tokens extracted from the trees. We use the

Algorithm 1: `Rich Edit Script` Comparison.

```
input  : SI: Search Index where SI ∈ {SI_Shape, SI_Action, SI_Token}
input  : fold ∈ {Shape, Action, Token}
input  : threshold: Set to 0 to retrieve identical trees.
output: R_identical: A set of pairs tagged to be identical

Function main (SI, fold)
    R_identical ← ∅
    I ← SI.getIdentifiers()                /* I: list of identifiers in the index */
    foreach i ∈ I do
        R ← SI.getPairs(i)  /* R: list of tree pairs to compare for identifier i */
        foreach a ∈ R do
            if compareTree(a, fold) then
                R_identical.add(a)         /* add if a is a pair of identical trees */
    return R_identical
Function compareTree(a, fold)
    (sTree1, sTree2) ← specializedTree(a, fold)
    if Fold != Token then
        editActions ← GumTree(sTree1, sTree2)
        editDistance ← size(editActions)
    else
        tokens1, tokens2 ← getTokens(sTree1, sTree2)
        editDistance ← d_w(tokens1, tokens2)      /* d_w: Jaro-Winkler distance */
    if  editDistance <= threshold then
        return true
    else
        return false
Function specializedTree(a, fold)
    (eTree1, eTree2) ← getRichEditScripts(a)     /* restore Rich Edit Scripts of a
      given pair from the cache */
    if fold == Shape then
        sTree1, sTree2 ← getASTNodeTrees(eTree1, eTree2)
    else if fold == Action then
        sTree1, sTree2 ← getActionTrees(eTree1, eTree2)
    else
        sTree1, sTree2 ← getTokenTrees(eTree1, eTree2)      /* fold == Token */
    return (sTree1, sTree2)
```

implementation of Jaro-Winkler edit distance from Apache Commons Text library[4], which computes the Jaro-Winkler edit distance of two strings d_w as defined in Equation 1. The equation consists of two components; Jaro's original algorithm (j_{sim}) and Winkler's extension(w_{sim}). The Jaro similarity is the weighted sum of percentage of matched characters c from each file and transposed characters t. Winkler increased this measure for matching initial characters, by using a prefix scale p that is is set to 0.1 by default, which gives more favorable ratings to strings that match from the beginning for a set prefix length l. The algorithm produces a similarity score (w_{sim}) between 0.0 to 1.0, where 0.0 is the least likely and 1.0 is a positive match.

[4] `https://commons.apache.org/proper/commons-text/`

Finally, this similarity score is transformed to distance (d_w).

$$
\begin{aligned}
&d_w(s_1, s_2) = 1 - w_{sim}(s_1, s_2)\\
&w_{sim}(s_1, s_2) = j_{sim}(s_1, s_2) + l * p(1 - j_{sim}(s_1, s_2))\\
&j_{sim}(s_1, s_2) = \begin{cases} 0 & \text{if } c = 0;\\ \frac{1}{3}\left(\frac{c}{|s_1|} + \frac{c}{|s_2|} + \frac{c-t}{c}\right) & otherwise. \end{cases}
\end{aligned}
\tag{1}
$$

l: The number of agreed characters at the beginning of two strings.
p: is a constant scaling factor for how much the score is adjusted upwards for having common prefixes, which is set to 0.1 in Winkler's work [92].

As the last step of comparison, we check the edit distance of the tree pair and tag the pairs having the distance zero as identical pairs, since the distance zero implies that no operation is necessary to transform one tree to another, or for the third fold ($Token$) the tokens in the tree are the same. Eventually, we store and save the set of identical tree pairs produced in each iteration, which would be used in Step 4.

3.6 Step 4 – Pattern Inference

Definition 5 (Pattern) Let g be a graph in which nodes are elements of RE and edges are defined by $R_{\text{identical}}$.
g consists of a set of connected subgraphs SG (i.e., clusters of specialized tree representations of `Rich Edit Scripts`) where sg_i and sg_j are disjoint $\forall sg_i, sg_j \in SG$. A pattern is defined by $sg_i \in SG$ if sg_i has at least two nodes (i.e., there are recurrent trees).

Finally, to infer patterns, we resort to clustering of the specialized tree representations of `Rich Edit Scripts`. First, we start by retrieving the set of identical tree pairs produced in Step 3 for each iteration. Following Algorithm 2, we extract the corresponding specialized tree representations according to the fold (i.e., ShapeTrees, ActionTrees, TokenTrees) since the trees are identical only in a given fold. In order to find groups of trees that are identical among themselves (i.e., clusters) we leverage graphs. Concretely, we implement a clustering process based on the theory of connected components (i.e., subgraph) identification in a graph [82]. We create an undirected graph from the list of tree pairs, where the nodes of the graph are the trees and the edges represent trees that are associated (i.e., identical tree pairs). From this graph, we identify clusters as the subgraphs, where each subgraph contains a group of trees that are identical among themselves and disjoint from others.

A cluster contains a list of `Rich Edit Scripts` sharing a common specialized tree representations according to the $fold$. Finally, a cluster is qualified as a pattern, when it has at least two members. The patterns for each $fold$ are defined as follows:

Algorithm 2: Clustering based on subgraph identification.

```
input : R_identical: A list of identical Rich Edit Script pairs
input : fold ∈ {Shape,Action,Token}
output: C: A list of clusters
Function main(R_identical,fold)
    C ← ∅
    TP ← getTreePairs(R_identical,fold)
    E ← transformPairsToEdges(TP)      /* E: edges created from tree pairs TP */
    g ← createGraph(E)
    SG ← g.connectedComponents()   /* SG: list of subgraphs found in graph g */
    foreach sg in SG do
        c ← s.nodes()          /* c: cluster formed from the nodes of subgraph sg */
        C.add(c)
    return C
Function getTreePairs(R_identical,fold)
    P ← ∅                                          /* P: list of tree pairs */
    foreach a in R_identical do
        (eTree1, eTree2) ← getRichEditScripts(a)  /* restore Rich Edit Scripts of a
          given pair from the cache */
        if fold == Shape then
            sTree1,sTree2 ← getASTNodeTrees(eTree1, eTree2)
        else if fold == Action then
            sTree1,sTree2 ← getActionTrees(eTree1, eTree2)
        else
            sTree1,sTree2 ← getTokenTrees(eTree1, eTree2)    /* fold == Token */
        P.add(sTree1,sTree2)
    return P
```

Shape patterns. The first iteration attempts to find patterns in the ShapeTrees associated to developer patches. We refer to them as Shape patterns, since they represent the shape of the changed code in a structure of the tree in terms of node types. Thus, they are not fix patterns per se, but rather the context in which the changes are recurrent.

Action patterns. The second iteration considers samples associated to each shape pattern and attempts to identify reoccurring repair actions from their ActionTrees. This step produces patterns that are relevant to program repair as they refer to recurrent code change actions. Such patterns can indeed be matched to dissection studies performed in the literature [83]. We will refer to Action patterns as the sought fix patterns. Nevertheless, it is noteworthy that, in contrast with literature fix patterns, which can be generically applied to any matching code context, our Action patterns are specifically mapped to a code shape (i.e., a shape pattern) and is thus applicable to specific code contexts. This constrains the mutations to relevant code contexts, thus yielding more likely precise fix operations.

Token patterns. The third iteration finally considers samples associated to each action pattern and attempts to identify more specific patterns with respect to the tokens available. Such token-specific patterns, which include specific tokens, are not suitable for implementation into pattern-based automated program repair systems from the literature. We discuss however their use in the context of deriving collateral evolutions (cf. Section 5.2).

4 Experimental Evaluation

We now provide details on the experiments that we carry out for `FixMiner`. Notably, we discuss the dataset, and present the implementation details. Then, we overview the statistics on the mining steps, and eventually enumerate the research questions for the assessment of `FixMiner`.

4.1 Dataset

We collect code changes from 44 large and popular open-source projects from Apache-Commons, JBoss, Spring and Wildfly communities with the following selection criteria: we focused on projects (1) written in Java, (2) with publicly available bug reports, (3) having at least 20 source code files in at least one of its version; finally, to reduce selection bias, (4) we choose projects from a wide range of categories - middleware, databases, data warehouses, utilities, infrastructure. This is a process similar to Bench4bl [46]. Table 2 details the number of bug fixing patches that we considered in each project. Eventually, our dataset includes 11 416 patches.

Table 2: Dataset.

Community	Project	# Patches	Project	# Patches
Apache	camel	1366	commons codec	11
	commons collections	56	commons compress	73
	commons configuration	89	commons crypto	9
	commons csv	18	common io	58
	hbase	2169	hive	2641
JBoss	entesb	15	jbmeta	14
Spring	amqp	89	android	5
	batch	224	batchadm	11
	datacmns	151	datagraph	19
	datajpa	112	datamongo	190
	dataredis	65	datarest	91
	ldap	26	mobile	11
	roo	414	sec	304
	secoauth	66	sgf	35
	shdp	35	shl	11
	social	14	socialfb	12
	socialli	2	socialtw	9
	spr	1098	swf	84
	sws	101		
Wildfly	ely	217	swarm	131
	wfarq	8	wfcore	547
	wfly	802	wfmp	13
Total				**11416**

4.2 Implementation Choices

We recall that we have made the following parameter choices in the `FixMiner` workflow:

– The "Shape" search index considers only `Rich Edit Scripts` having a depth greater than 1 (i.e., the AST sub-tree should include at least one parent and one child).
– Comparison of `Rich Edit Scripts` is designed to retrieve identical trees (i.e., tree edit distance is 0).

4.3 Statistics

`FixMiner` is a pattern mining approach to produce fix patterns for program repair systems. Its evaluation (cf. Section 5) will focus on evaluating the relevance of the yielded patterns. Nevertheless, we provide statistics on the mining process to provide a basis of discussion on the implications of `FixMiner`'s design choices.

Search Indices. `FixMiner` mines fix patterns through comparison of hunks (i.e., contiguous groups of code lines). 11 416 patches in our database are eventually associated to 41 823 hunks. A direct pairwise comparison of these hunks would lead to 874 560 753 tree comparison computations. The combinatorial explosion of the comparison space is overcome by building search indices as previously described in Section 3.4. Table 3 shows the details on the search indices built for each fold in the `FixMiner` iterations. From the 874+ million tree pairs to be compared (i.e., C^2_{41823}), the construction of the Shape index (implements criteria on the tree structure to focus on comparable trees) lead to 670 relevant comparison sub-spaces yielding a total of only 12+ million tree comparison pairs. This represents a reduction of 98% of the comparison space. Similarly, the Action index and the Token index reduce the associated comparison spaces by 88% and 72% respectively.

Table 3: Comparison space reduction.

Search Index	# of hunks (in fold)	# Comparison sub-spaces	# Tree comparison pairs
Shape	41 823	670	12 601 712
Action	25 290	2 457	1 427 504
Token	6759	411	401 980

Clusters. We infer patterns by considering recurrence of trees: the clustering process groups together only tree pairs that are identical among themselves. Table 4 overviews the statistics of clusters yielded for the different iterations: Shape patterns (which represent code contexts) are the most diverse. Action patterns (which represent fix patterns that are suitable as inputs for program repair systems) are substantially less numerous. Finally, Token patterns (which may be codebase-specific) are significantly fewer. We recall that we consider all possible clusters as long as it includes at least 2 elements. A practitioner may however decide to select only large clusters (i.e., based on a threshold).

Because `FixMiner` considers code hunks as the unit for building `Rich Edit Scripts`, a given pattern may represent a repeating context (i.e., Shape pattern) or change (i.e., Action or Token pattern) that is only *part* of the patch

Table 4: Statistics on clusters.

Pattern	# Trees (clustering input)	# Corresponding change hunks	# Clusters
Shape	1 370 406	25 290	2947
Action	628 531	6 759	428
Token	18 471	1 562	326

(i.e., this patch includes other change patterns) or that is the *full* patch (i.e., the whole patch is made of this change pattern). Table 5 provides statistics on partial and full patterns. The numbers represent the disjoint sets of patterns that can be identified as always full or as always partial. Patterns that may be *full* for a given patch but *partial* for another patch are not considered.Overall, the statistics indicate that, from our dataset of over 40 thousand code hunks, only a few (e.g., respectively 278 and 7 120 hunks) are associated with patterns that are always *full* or always *partial* respectively. In the remaining cases, the pattern is associated to a code hunk that may form alone the patch or may be tangled with other code. This suggests that `FixMiner` is able to cope with tangled changes during pattern mining.

Table 5: Statistics on Full vs Partial patterns.

	Partial patterns			Full patterns		
	# Patterns	# Patch	# Hunk	# Patterns	# Patch	# Hunk
Shape	1358	3140	7120	120	223	278
Action	124	554	1153	10	20	25
Token	75	148	277	14	22	32

Similarly, we investigate how the patterns are spread among patches. Indeed, a pattern may be found because a given patch has made the same change in several code hunks. We refer to such pattern as *vertical*. In contrast, a pattern may be found because the same code change is spread across several patches. We refer to such pattern as *horizontal*. Table 6 shows that vertical and horizontal patterns occur in similar proportions for Shape and Action patterns. However, Token patterns are significantly more vertical than horizontal (65 vs 224). This is in line with studies of collateral evolutions in Linux, which highlight large patches making repetitive changes in several locations at once [76] (i.e., collateral evolutions are applied through vertical patches).

Table 6: Statistics on Pattern Spread.

	Vertical			Horizontal		
	# Patterns	# Patch	# Hunk	# Patterns	# Patch	# Hunk
Shape	881	881	2432	1194	3808	3808
Action	148	148	488	132	574	574
Token	224	224	709	65	170	170

* A pattern can simultaneously be vertical (when it is associated to several changes in hunks of the same patch) and horizontal (when it appears as well within other patches).

4.4 Research Questions

The assessment experiments are performed with the objective of investigating the usefulness of the patterns mined by `FixMiner`. To that end, we focus on the following research questions (RQs):

RQ-1 Is automated patch clustering of `FixMiner` consistent with human manual dissection?
RQ-2 Are patterns inferred by `FixMiner` compatible with known fix patterns?
RQ-3 Are the mined patterns effective for automated program repair?

5 Results

5.1 RQ1: Comparison of `FixMiner` Clustering against Manual Dissection

Objective. We propose to assess relevance of the clusters yielded by `FixMiner` in terms of whether they represent patterns which practitioners would view as recurrent changes that are indeed relevant to the patch behaviour. In previous section, the statistics showed that several changes are recurrent and are mapped to `FixMiner`'s clusters. In this RQ, we validate whether they are relevant to the practitioner's viewpoint. For example, if `FixMiner` was not leveraging AST information, removal of blank lines would have been seen as a recurrent change (hence a pattern); however, a practitioner would not consider it as relevant.
Protocol. We consider an oracle dataset of patches with change patterns that are labelled by humans. Then we associate each of these patches to the relevant clusters mined by `FixMiner` on our combined study datasets. This way, we ensure that the clustering does not overfit to the oracle dataset labelled by humans. Eventually, we check whether each set of patches (from the oracle dataset) that are associated to a given `FixMiner` cluster, consists of patches having the same labels (from the oracle).
Oracle. For our experiments, we leverage the manual dissection of Defects4J [31] provided by Sobreira et. al [83]. This oracle dataset associates the developer patches of 395 bugs in the Defects4J datasets with 26 repair pattern labels (one of which is being "Not classified").
Results. Table 7 provides statistics that describe the proportion[5] of `FixMiner`'s patterns that can be associated to change patterns in the Defects4J patches.

Table 7: Proportion of shared patterns between our study dataset and Defects4J.

	Study dataset		Defects4J	
	# corresponding hunks	# Patterns	# corresponding hunks	# Patterns
Shape	25272	2947	479	214
Action	6755	428	103	37
Token	1562	326	23	13

[5] In this experiment, we excluded 34 patches from Defects4J dataset which affect more than 1 file.

Diversity. We check the number of patterns that can be found in our study dataset and Defects4J. In absolute numbers, Defects4J patches include a limited set of change patterns (i.e., ~ 7% = $\frac{214}{2947}$) in comparison to what can be found in our study dataset.

Consistency. We check for consistency of `FixMiner`'s pattern mining by assessing whether all Defects4J patches associated to a `FixMiner` cluster are indeed sharing a common dissection pattern label. We have found that the clustering to be consistent for ~ 78% = $\frac{166}{214}$, ~ 73% = $\frac{27}{37}$ and ~ 92% = $\frac{12}{13}$ of Shape, Action and Token clusters respectively.

RQ1-Consistency ▸ *`FixMiner` can produce patterns that are matching patches that are labeled similarly by humans. The patterns are thus largely consistent with manual dissection.*

Granularity. The human dissection provides repair pattern labels for a given patch. Nonetheless, the label is not specifically associated to any of the various changes in the patch. `FixMiner` however yields patterns for code hunks. Thus, while `FixMiner` links a given hunk to a single pattern, the dissection data associates several patterns to a given patch. We investigate the granularity level with respect to human-provided patterns. Concretely, several patterns of `FixMiner` can actually be associated (based on the corresponding Defects4J patches) to a single human dissection pattern. Consider the example cases in Table 8. Both patches consist of nested InfixExpression under the IfStatement. The first `FixMiner` pattern indicates that the change operation (i.e., update operator) should be performed on the children InfixExpression. On the other hand, the second pattern implies a change operation in the parent InfixExpression. Thus, eventually, `FixMiner` patterns are finer-grained and associates the example patches to two distinct patterns each pointing the precise node to update, while manual dissection considers them under the same coarse-grained repair pattern.

Table 8: Granularity example to `FixMiner` mined patterns.

	Pattern	Example patch from `FixMiner` dataset

`FixMiner`	`UPD IfStatement`
	`---UPD InfixExpression`
	`------UPD InfixExpression`
	`---------UPD Operator`
		`@@ -83,7 +83,7 @@ public BoundedInputStrea ...`
		`    @Override`
		`    public int read() throws IOException {`
		`- if (max >= 0 && pos == max) {`
		`+ if (max >= 0 && pos >= max) {`
		`        return -1;`

Dissection [83]	Logic expression modification Single Line	

`FixMiner`	`UPD IfStatement`
	`---UPD InfixExpression`
	`------UPD Operator`
		`@@ -145,7 +145,7 @@ private void moveFile(Path s ...`
		`  private Path createTargetPath(Path targetPath ...`
		`    Path deletePath = null;`
		`    Path mkDirPath = targetPath.getParent();`
		`- if (mkDirPath != null & !fs.exists(mkDirPath)) {`
		`+ if (mkDirPath != null && !fs.exists(mkDirPath)) {`
		`      Path actualPath = mkDirPath;`

Dissection [83]	Logic expression modification Single Line	

We have investigated the differences between `FixMiner` patterns and dissection labels and found several granularity mismatches similar to the previous example: `condBlockRetAdd` (*condition block addition with return statement*) from manual dissection is associated to 14 fine-grained Shape patterns of `FixMiner`: this suggests that the repair-potential of this pattern could be further refined depending on the code context. Similarly, `expLogicMod` (*logic expression modification*), is associated to 2 separate Action patterns (see Table 8) of `FixMiner`: this suggests that the application of this repair pattern can be further specialized to reduce the repair search space and the false positives.

Overall, we found in total 37, 3 and 1 dissection repair patterns are further refined into several `FixMiner`'s Shape, Action and Token patterns respectively.

> **RQ1-Granularity** ▸ *We observe that manually-dissected patterns are more coarse-grained compared to* `FixMiner`*'s patterns.*

Assessment of `FixMiner`*'s patterns with respect to associated bug reports.* Beyond assessing the consistency of `FixMiner`'s patterns based on human-built oracle dataset of labels, we further propose to investigate the relevance of the patterns in terms of the semantics that can be associated to the intention of the changes. To that end, we consider bug reports associated to patches as a proxy to characterize the intention of the code changes. We expect bug reports sharing textual similarity to be addressed by patches that are syntactically similar. This hypothesis drives the entire research direction on Information retrieval-based bug localization [46].

Figure 14 provides the distribution of pairwise bug report (textual) similarity values for the bug reports corresponding to patches associated to each cluster. For clear presentation, we focus on the top-20 clusters (in terms of size). We use TF-IDF to represent each bug report as a vector, and leverage Cosine similarity to compute similarity scores among vectors. The represented boxplots display all pairwise bug report similarity values, including outliers. Although for Shape and Action patterns the similarities are near 0 for all clusters, we note that there are fewer outliers for Action patterns. This suggests a relative increase in the similarity among bug reports. As expected, similarity among bug reports is the highest with Token patterns.

5.2 RQ2: Compatibility between `FixMiner`'s patterns and APR literature patterns

Objective. Given that `FixMiner` aims to automatically produce fix patterns that can be used by automated program systems, we propose to assess whether the yielded patterns are compatible with patterns in the literature.

Protocol. We consider the set of patterns used by literature APR systems and compare them against `FixMiner`'s patterns. Concretely, we systematically try to map `FixMiner`'s patterns with patterns in the literature. To that end, we rely on the comprehensive taxonomy of fix patterns proposed by Liu et al. [52]: if a given `FixMiner` pattern can be mapped to a type of change in

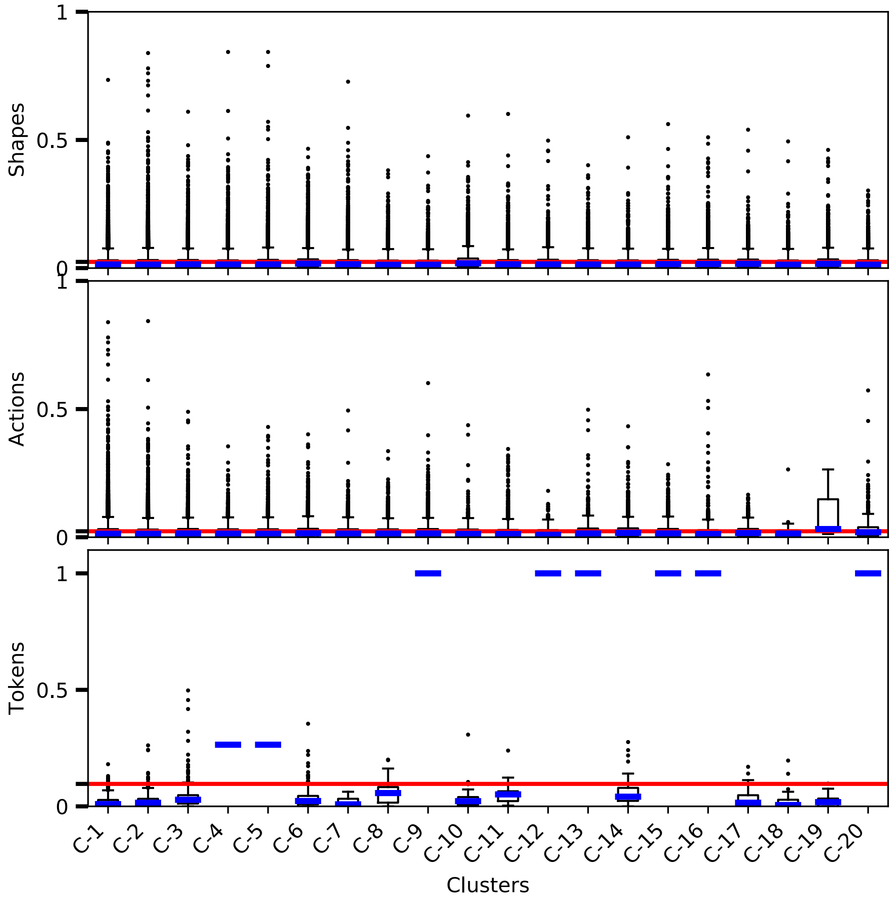


Fig. 14: Distribution of pairwise bug report similarity. Note: A red line represents an average similarity for all bug reports in fold, and blue line represents average similarity bug reports within a cluster.

the taxonomy, then this pattern is marked as *compatible* with patterns in the literature.

Recall that, as described earlier, fix patterns used by APR tools abstract changes at the form of `FixMiner`'s Action patterns (Section 3 - Step 4). In the absence of common language for specifying patterns, the comparison is performed manually. For the comparison, we do not conduct exact mapping between literature patterns and the ones yielded by `FixMiner` as fix patterns yielded by `FixMiner` have more context information. We rather consider whether the context information yielded by `FixMiner` patterns matches with the context of literature patterns. We discuss the related threats to validity in Section 6. Given that the assessment is manual and thus time-consuming, we limit the comparisons to the top 50 patterns (i.e., Action patterns) yielded by FixMiner.

Oracle. We build on the patterns enumerated by Liu et al. [52] who systematically reviewed fix patterns used by Java APR systems in the literature. They summarised 35 fix patterns in GNU format, which we refer to for comparing against `FixMiner` patterns.

Results. Overall, among the 35 fix patterns used by the total of 11 studied APR systems, 16 patterns are also included in the fix patterns (i.e., Action patterns) yielded by `FixMiner` when mining our study dataset. We recall that these patterns are often manually inferred and specified by researchers for their APR tools. Table 9 illustrates examples of `FixMiner`'s fix patterns associated to some of the patterns used in literature. We note that fix patterns identified by `FixMiner` are specific (e.g., for `FP4: Insert Missed Statement`, the corresponding `FixMiner`'s fix pattern specifies which type of statement must be inserted).

Table 9: Example `FixMiner` fix-patterns associated to APR literature patterns.

Patterns enumerated by Liu et al. [52]	Example fix pattern from `FixMiner` (*)
FP2. Insert Null Pointer Checker	INS IfStatement — INS InfixExpression —— INS SimpleName —— INS Operator —— INS NullLiteral — INS ReturnStatement —— INS NullLiteral
FP4. Insert Missed Statement	INS ExpressionStatement —INS MethodInvocation ——INS SimpleName
FP7. Mutate Data Type	UPD CatchClause — UPD SingleVariableDeclaration —— UPD SimpleType
FP9. Mutate Literal Expression	UPD FieldDeclaration — UPD VariableDeclarationFragment —— UPD StringLiteral
FP10. Mutate Method Invocation Expression	UPD ExpressionStatement — UPD MethodInvocation —— UPD SimpleName ——— INS SimpleName
FP11. Mutate Operators	UPD IfStatement — UPD InfixExpression —— UPD Operator
FP12. Mutate Return Statement	UPD ReturnStatement — UPD MethodInvocation —— UPD SimpleName

* **Complete list of 16 Fix Patterns from literature that match `FixMiner`'s patterns:** FP2. Insert Null Pointer Checker (i.e., 2.1, 2.2 and 2.5), FP3. Insert Range Checker, FP4. Insert Missed Statement (i.e., 4.1), FP7. Mutate Data Type (i.e., 7.1), FP9. Mutate Literal Expression (i.e., 9.1), FP10. Mutate Method Invocation Expression (i.e., 10.1, 10.2, 10.3, and 10.4), FP11. Mutate Operators (i.e., 11.1), FP12. Mutate Return Statement, FP13. Mutate Variable (i.e., 13.1), FP14. Move Statement and FP15. Remove Buggy Statement (i.e., 15.1).

Table 10 illustrates the proportion of `FixMiner`'s patterns that are compatible with patterns in the literature. In this comparison, we select the top-50 fix patterns yielded by `FixMiner` and verify their presence within the fix patterns used in the APR systems.

Table 10: Compatibility of Patterns: `FixMiner` vs Literature Patterns.

PAR	HDRepair	ssFix	ELIXIR	S3	NPEfix	SketchFix	SOFix	Genesis	CapGen	SimFix	AVATAR
7/16	7/12	6/34	8/11	3/4	1/9	5/6	9/12	1/108	12/30	8/16	6/13

We provide x/y numbers: x is the number of fix patterns in the corresponding APR tool that are mined by `FixMiner`; y is the number of fix patterns used by the corresponding APR tool.

We observed that

- 7 patterns are compatible with fix patterns that are mined manually from bug fix patches (i.e., fix patterns in PAR [33]).
- between 1 and 8 patterns are compatible with researcher-predefined fix patterns used in ssFix [93], ELIXIR [81], S3 [41], NEPfix [15], and SketchFix [27], respectively.
- 7 patterns are compatible with fix pattern mined from history bug fixes by HDRepair [43], 9 patterns are compatible with fix patterns mined from StackOverflow by SOFix [54], and 1 fix pattern is compatible with 1 fix pattern mined by Genesis [56] that focuses on mining fix patterns for three kinds of bugs.
- 12 and 8 patterns are compatible with the patterns used by CapGen [90] and SimFix [30], respectively, which extract patterns in a statistic way similar to the empirical studies of bug fixes [49, 61].
- 6 patterns are compatible with the fix patterns used in AVATAR [51], which are presented in a study for inferring fix patterns from FindBugs [26] static analysis violations [48].

RQ2▸ *`FixMiner` effectively yields Action patterns that are compatible for 16 over 35 patterns used in the literature of pattern-based program repair.*

Manual (but Systematic) Assessment of Token patterns. Action and Token patterns are the two types of patterns that relate to code changes. In the assessment scenario above, we only considered Action patterns since they are the most appropriate for comparison with the literature patterns. We now focus on Token patterns to assess whether our hypothesis on their usefulness for deriving collateral evolutions holds (cf. Section 3 - Step 4). To that end, we consider the various Token clusters yielded by `FixMiner` and manually verify whether the recurrent change (i.e., the pattern) is relevant (i.e., a human can explain whether the intentions of the changes are the same). Eventually, if the pattern is validated, it should be presented as a generic/semantic patch [3, 76] written in SmPL[6].

In Table 11, we list some of the patches that we found to be relevant. Among the top 50 Token patterns investigated, 12 patterns correspond to a modifier change, 4 patterns target changes in logging methods, and 1 pattern is about fixing the infix operator (e.g., `>` → `>=`). The remaining cases mainly focus on changes that complete the implementation of code *finally* block logic (e.g., missing call to *closeAll* for opened files), changes in Exception handling, updates to wrong parameters passed to method invocations, as well as wrong

[6] Semantic Patch Language

method invocations. As mentioned earlier, these patterns are spread mostly vertically (i.e. change is recurrent in several code hunks of a given patch) and the semantic behaviour of these patterns are specific to project nature.

Overall, our manual investigations on the top 50 Token patterns confirm that many of the recurrent changes associated to specific tokens are indeed relevant. We even found several cases where collateral evolution changes are regrouped to form a pattern as exhibited by the corresponding pattern example presented in Figure 15. In this example, we illustrate the pattern using the SmPL specification language, which was designed for specifying collateral evolutions. This finding suggests that `FixMiner` can be leveraged to systematically mined collateral evolutions in the form of Token patterns which could be automatically rewritten as semantic patches in SmPL format. This endeavour is however out of the scope of this paper, and will be investigated in future work.

Table 11: Example changes associated to `FixMiner` mined patterns.

Semantic Behaviour of Pattern	Example change in developer patch
Missing field modifier	`- private boolean closed = true;` `+ private volatile boolean closed = true;`

Wrong Log level	`} catch (Exception e) {`
	`- LOG.fatal("Could not append. Requesting close of wal", e);`
	`+ LOG.warn("Could not append. Requesting close of wal", e);`
	`requestLogRoll();`

```
// [caption=Wrong Log level]
@@
Logger log;
@@
...
- log.fatal(...);
+ log.warn(...);
```

Fig. 15: Example SmPL patch corresponding to generic representation of the pattern associated to `FixMiner` pattern.

5.3 RQ3: Evaluation of Fix Patterns' Relevance for APR

Objective. We propose to assess whether fix patterns yielded by `FixMiner` are effective for automated program repair.

Protocol. We implement a prototype APR system that uses the fix patterns mined by `FixMiner` to generate patches for bugs by following the principles of the PAR [33], which is referred to as $\texttt{PAR}_{\texttt{FixMiner}}$ in the remainder of this paper. In contrast with PAR where the templates were engineered by a manual investigation of example bug fixes, in $\texttt{PAR}_{\texttt{FixMiner}}$, the templates for repair are engineered based on fix patterns mined by `FixMiner`. Figure 16 overviews the workflow of $\texttt{PAR}_{\texttt{FixMiner}}$.

Fault Localization. $\texttt{PAR}_{\texttt{FixMiner}}$ uses spectrum-based fault localization. We use the GZoltar[7] [8] dynamic testing framework and leverage Ochiai [1] ranking

[7] We used GZoltar version 0.1.1

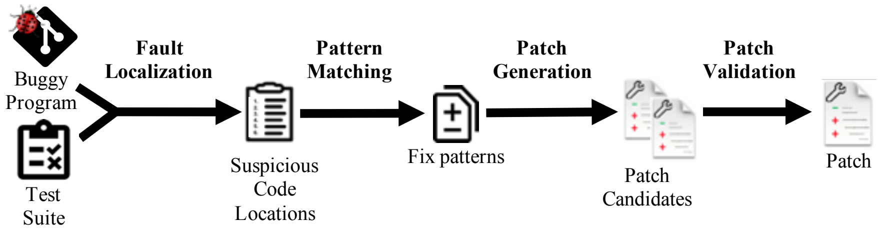


Fig. 16: The overall workflow of $\texttt{PAR}_{\texttt{FixMiner}}$ program repair pipeline.

metric to predict buggy statements based on execution coverage information of passing and failing test cases. This setting is widely used in the repair community [53,62,90,93,94], allowing for comparable assessment of $\texttt{PAR}_{\texttt{FixMiner}}$ against the state-of-the-art.

Pattern Matching and Patch Generation. Once the spectrum-based fault localization (or ir-based fault localization [38, 91]) process yields a list of suspicious code locations, $\texttt{PAR}_{\texttt{FixMiner}}$ attempts to select fix patterns for each statement in the list. The selection of fix patterns is conducted by matching the context information of suspicious code locations and fix patterns mined by `FixMiner`. Concretely, first, we parse the suspicious statement and traverse each node of its AST from its first child node to its last leaf node and form an AST subtree to represent its context. Then, we try to match the context (i.e., shape) of the AST subtree (from a suspicious statement) to the fix patterns' shapes.

If a matching fix pattern is found, we proceed with the generation of a patch candidate. Some fix patterns require donor code (i.e., source code extracted from the buggy program) to generate patch candidates with fix patterns. These are also often referred to as part of fix ingredients. Recall that, to integrate with repair tools, we leverage `FixMiner` Action patterns, which do not contain any code token information: they have "holes". Thus we search the donor code locally from the file which contains the suspicious statement. We select relevant donor code among the ones that are applicable to the fix pattern and the suspicious statement (i.e., data type(s) of variable(s), expression types, etc. that are matching to the context) to reduce the search space of donor code and further limit the generation of nonsensical patch candidates. For example, the fix pattern in Figure 17 can only be matched to a suspicious return statement that has a method invocation expression: thus, the suspicious return statement will be patched by replacing its method name with another one (i.e., donor code). The donor code is searched by identifying all method names from the suspicious file having the same return type and parameters with the suspicious statement. Finally, a patch candidate is generated by mutating suspicious statements with identified donor code following the actions indicated in the matched fix pattern. We generate as many patches as the number of identified pieces of donor code. Patches are generated consecutively in the order of matching within the AST.

Note: We remind the reader that in this study, we do not perform a specific patch prioritization strategy. We search donor code from the AST tree of the local file that contains the suspicious statement by traversing each node of the AST of the local file from its first child node to its last leaf node in a breadth-first strategy (i.e., left-to-right and top-to-bottom). In case of multiple donor code options for a given fix pattern, the candidate patches are generated (each with a specific donor code) following the positions of donor codes in the AST tree (of the local file which contains the suspicious statement).

```
UPD ReturnStatement
---UPD MethodInvocation
------UPD Simple@MethodName
```

Fig. 17: Example of fix patterns yielded by `FixMiner`.

Pattern Validation. Once a patch candidate is generated, it is applied to buggy program and will be validated against the test suite. If it can make the buggy program pass all test cases successfully, the patch candidate will be considered as a plausible patch and $\texttt{PAR}_{\texttt{FixMiner}}$ stops trying other patch candidates for this bug. Otherwise, the pattern matching and patch generation steps are repeated until the entire suspicious code locations list is processed. Specifically, we consider only the first generated plausible patch for each bug to evaluate its correctness. For all plausible patches generated by $\texttt{PAR}_{\texttt{FixMiner}}$, we further manually check the equivalence between these patches and the oracle patch provided in Defects4J. If they are semantically similar to the developer-provided fix, we consider they as correct patches, otherwise remain as plausible.

Oracle. We use Defects4J[8] [31] dataset which is widely used as a benchmark for Java-targeted APR research [10, 43, 60, 62]. The dataset contains 357 bugs with their corresponding developer fixes and test cases covering the bugs. Table 12 details statistics on the benchmark.

Table 12: Details of the benchmark.

Project	Bugs	LOC	Tests
JFreechart (Chart, C)	26	96K	2,205
Apache commons-lang (Lang, L)	65	22K	2,245
Apache commons-math (Math, M)	106	85K	3,602
Joda-Time (Time, T)	27	28K	4,130
Closure compiler (Closure, Cl)	133	90K	7,927
Total	357	321K	20,109

† In the table, column "Bugs" denotes the total number of bugs in Defects4J benchmark, column "LOC" denotes the number of thousands of lines of code, and column "Tests" denotes the total number of test cases for each project.

Results. Overall, we implemented the 31 fix patterns (i.e., Action patterns) mined by `FixMiner`, focusing only on the top-50 clusters (in terms of size).

We compare the performance of $\texttt{PAR}_{\texttt{FixMiner}}$ against 13 state-of-the-art APR tools which have also used Defects4J benchmark for evaluating their repair performance. Table 13 illustrates the comparative results in terms of numbers of

[8] Version 1.2.0 - `https://github.com/rjust/defects4j/releases/tag/v1.2.0`

plausible (i.e., that passes all the test cases) and *correct* (i.e., that is eventually manually validated as semantically similar to the developer-provided fix) patches. Note that although HDRepair manuscript counts 23 bugs for which "correct" fixes are generated (and among which a correct fix is ranked number one for 13 bugs), the authors labeled fixes as "verified ok" for only 6 bugs (see artefact page [9]). We consider these 6 bugs in our comparison.

Overall, we find that $\texttt{PAR}_{\texttt{FixMiner}}$ successfully repaired 26 bugs from the Defects4J benchmark by generating correct patches. This performance is only surpassed to date by SimFix [30] that was concurrently developed with $\texttt{PAR}_{\texttt{FixMiner}}$.

Table 13: Number of bugs fixed by different APR tools.

Proj.	$\texttt{PAR}_{\texttt{FixMiner}}$	kPAR	jGenProg	jKali	jMutRepair	Nopol	HDRepair	ACS	ssFix	ELIXIR	JAID	SketchFix	CapGen	SimFix
Chart	**5/8**	3/10	0/7	0/6	1/4	1/6	0/2	2/2	3/7	4/7	2/4	6/8	4/4	4/8
Lang	**2/3**	1/8	0/0	0/	0/1	3/7	2/6	3/4	5/12	8/12	1/8	3/4	5/5	9/13
Math	**13/15**	7/18	5/18	1/14	2/11	1/21	4/7	12/16	10/26	12/19	1/8	7/8	12/16	14/26
Time	**1/1**	1/2	0/2	0/2	0/1	0/1	0/1	1/1	0/4	2/3	0/0	0/1	0/0	1/1
Closure	**5/5**	5/9	0/0	0/0	0/0	0/0	0/7	0/0	2/11	0/0	5/11	3/5	0/0	6/8
Total	**26/32**	17/47	5/27	1/22	3/17	5/35	6/23	18/23	20/60	**26/41**	9/31	19/26	21/25	**34/56**
P(%)	**81.3**	36.2	18.5	4.5	17.7	14.3	26.1	78.3	33.3	63.4	29.0	73.1	**84.0**	60.7

† In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool (i.e., a patch that makes the program pass all test cases). Precision (P) means the precision of correctly fixed bugs in bugs fixed by each APR tool. kPAR [50] is the Java implementation of PAR. The data about jGenProg, jKali and Nopol are extracted from the experimental results reported by Martinez et al. [60]. The data of HDRepair [43] is collected from its author's reply. And the results of other tools are obtained from their papers in the literature (jMutRepair [62], ACS [94], ssFix [93], ELIXIR [81], JAID [10], SketchFix(SF) [27], CapGen [90] and SimFix [30]). The same for the data presented in Table 14.

Nevertheless, while these tools generate more correct patches than $\texttt{PAR}_{\texttt{FixMiner}}$, they also generate many more plausible patches which are however not correct. In order to comparatively assess the different tools, we resort to a Precision metric (P), which is the probability of correctness of the generated patches. P(%) is defined as the ratio of the number of bugs for which a correct fix is generated first (i.e., before any other plausible patch) against the number of bugs for which a plausible (but incorrect) patch is generated first. For example, 81% of $\texttt{PAR}_{\texttt{FixMiner}}$'s plausible patches are actually correct, while it is the case for 63% and 60% of respectively ELIXIR and SimFix plausible patches are correct. To date only CapGen [90] achieves similar performance at yielding patches with slighter higher probability (at 84%) to be correct. The high performance of CapGen confirms their intuition that context-awareness, which we provide with `Rich Edit Script`, is essential for improving patch correctness.

Table 14 enumerates 128 bugs that are currently fixed (both correct and plausible) in the literature. 89 of them can be correctly fixed by at least one APR tool. $\texttt{PAR}_{\texttt{FixMiner}}$ generates correct patches for 26 bugs. Among the bugs in the used version of Defects4J benchmark, 267 bugs have not yet been fixed by any tools in the literature, which still is a big challenge for automated program repair research.

Finally, we find that, thanks to its automatically mined patterns, $\texttt{PAR}_{\texttt{FixMiner}}$ is able to fix six (6) bugs which have not been fixed by any state-of-the-art APR tools (cf. Figure 18).

[9] https://github.com/xuanbachle/bugfixes/blob/master/fixed.txt

Table 14: Defects4J bugs fixed by different APR tools.

"✓" indicates that the bug is correctly fixed, "✗" indicates the produced patch is plausible but not correct. "(✓)" indicates that a correct patch is generated by JAID, but is not the first plausible patch to be generated)".

Proj.	FixMiner	kPAR	SimFix	CapGen	SketchFix	JAID	ssFix	ACS	ELIXIR	HDRepair	jGenProg	jKali	jMutRepair	Nopol
C-1	✓	✓	✓	✓	✓	(✓)	✓		✓	✗	✗	✗	✓	
C-3		✗	✓						✗		✗			✗
C-4	✓	✓												
C-5		✗									✗	✗		✓
C-7		✓	✓								✗		✗	
C-8				✓	✓				✓	✗				
C-9					✓	(✓)			✓					
C-11	✓			✓	✓				✓					
C-12	✗	✗	✗											
C-13	✗	✗			✗				✗		✗	✗		✗
C-14			✗					✓						
C-15		✗									✗	✗		
C-17									✗					
C-18			✗											
C-19								✓						
C-20			✓		✓		✓							
C-21														✗
C-22			✗											
C-24	✓			✓	✓	✓	✓							
C-25	✗	✗									✗	✗	✗	✗
C-26	✓	✗			✗	✓						✗	✗	✗
Cl-2		✓												
Cl-5						✗								
Cl-10	✓									✗				
Cl-14			✓		✓		✓			✗				
Cl-18						✓								
Cl-21		✗												
Cl-22		✗												
Cl-31						(✓)								
Cl-33						✓								
Cl-38	✓	✓												
Cl-40						✓								
Cl-51										✗				
Cl-57			✓											
Cl-62	✓	✓	✓		✓	(✓)				✗				
Cl-63	✓	✓	✓			(✓)								
Cl-70					✗	✓				✗				
Cl-73	✓	✓	✓		✗	✓				✗				
Cl-79			✗											
Cl-106			✗											
Cl-109		✗												
Cl-115			✓				✓							
Cl-125						✗								
Cl-126		✗			✓	(✓)				✗				
L-6				✓	✓		✓		✓	✓				
L-7								✓						
L-10			✗							✗				
L-16			✓											
L-21							✓							
L-24						✗		✓	✓					
L-26				✓					✓					
L-27			✓										✗	
L-33			✓			✓	✓		✓					
L-35								✓						
L-38						(✓)			✓					
L-39			✓			✗		✗	✗					✗
L-41			✓											
L-43	✗	✗	✓	✓			✓		✓	✗				
L-44		✗	✗						✗					✓
L-45		✗	✗			(✓)								
L-46														✗
L-50			✓											
L-51		✗			✗	(✓)			✗	✓				✗
L-53		✗												✗
L-55					✓	(✓)								✓
L-57	✓	✗		✓					✓	✗				

Proj.	FixMiner	kPAR	SimFix	CapGen	SketchFix	JAID	ssFix	ACS	ELIXIR	HDRepair	jGenProg	jKali	jMutRepair	Nopol
L-58		✗	✓						✗					✓
L-59	✓	✓		✓	✓		✓		✓	✗				
L-60			✓											
L-61						✗								
L-63			✗											
M-1		✗	✗											
M-2									✗		✗	✗	✗	
M-3								✓						
M-4								✓						
M-5			✓	✓	✓	✓		✓	✓	✓	✓			
M-6			✗											
M-8		✗	✗								✗	✗		
M-10	✓													
M-15		✓												
M-20			✗						✗					
M-22	✓									✓				
M-25								✓						
M-28		✗	✗					✗			✗	✗	✗	
M-30	✓			✓			✓		✓					
M-32						(✓)			✗			✗		✗
M-33	✓	✓	✓	✓	✓		✓		✓					✗
M-34	✓								✓	✗				
M-35	✓		✓					✓						
M-40											✗	✗	✗	✗
M-41			✓				✓							
M-42														✗
M-49		✗									✗	✗		✗
M-50		✗	✓		✓	(✓)	✓		✓	✓	✓	✓	✗	✓
M-51		✗												
M-53			✓	✓		(✓)	✓			✓	✓			
M-57	✓		✓	✓			✓		✓				✗	✗
M-58	✓	✓		✓					✓				✗	✗
M-59			✓	✓	✓		✓		✓					
M-61								✓						
M-62		✗												
M-63		✗	✓	✓					✗					
M-65				✓										
M-69														✗
M-70	✓	✓	✓	✓	✓		✓		✓	✗	✓			
M-71			✓								✗			✗
M-72			✗											
M-73			✗		✗			✗	✗		✓			✗
M-75	✓	✓	✓	✓					✓					
M-78											✗	✗		✗
M-79	✓		✓	✗			✓							
M-80			✗	✗		(✓)	✓		✗		✗	✗		✗
M-81	✗	✗	✗	✗				✗			✗	✗	✗	✗
M-82	✓	✗	✗	✗	✓	(✓)		✓	✓	✗	✗	✗	✓	✗
M-84	✗	✗									✗	✗	✗	
M-85	✓	✓	✗	✓	✓	(✓)		✓	✓		✗	✗	✓	✗
M-87														✗
M-88			✗										✗	✗
M-89		✓						✓						
M-90								✓						
M-93								✓						
M-95											✗	✗		
M-97								✗						✗
M-98			✓											
M-99								✓						
M-104									✗					✗
M-105						✗								✗
T-4					✗				✓		✗	✗		
T-7		✓	✓											
T-11		✗							✗		✗	✗	✗	✗
T-15								✓	✓					
T-19	✓									✗				

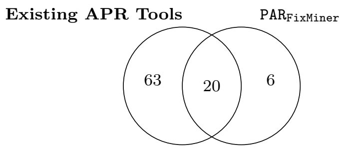


Fig. 18: Overlap of the correct patches by $\texttt{PAR}_{\texttt{FixMiner}}$ and other APR tools.

> **RQ3▸** *Fix patterns (i.e., Action Patterns) yielded by* `FixMiner` *can be directly used in automated program repair pipelines and generates correct patches for buggy programs effectively. Additionally, the repair performance of* $\texttt{PAR}_{\texttt{FixMiner}}$*, which uses fix patterns yielded by* `FixMiner`*, is comparable to the state-of-the-art APR tools.*

6 Discussions and Threats to Validity

Runtime performance. To run the experiments with `FixMiner`, we leveraged a computing system with 24 Intel Xeon E5-2680 v3 cores with 2.GHz per core and 3TB RAM. The construction of the `Rich Edit Scripts` took about 17 minutes. `Rich Edit Scripts` are cached in memory to reduce disk access during the computation of identical trees. Nevertheless, we recorded that comparing 1 108 060 pairs of trees took about 18 minutes.

Threats to external validity. The selection of our bug-fix datasets carries some threats to external validity that we have limited by considering known projects, and heuristics used in previous studies. We also make our best effort to link commits with bug reports as tagged by developers. Some false positives may be included if one considers a strict and formal definition of what constitutes a bug.

Threats to construct validity arise when checking the compatibility of `FixMiner`'s patterns against the patterns used by literature APR systems. Indeed, for the comparison, we do not conduct exact mapping where the elements should be the same, given that literature patterns can be more abstract than the ones yielded by `FixMiner`. For example, *Modify Method Name* (i.e., FP10.1) is a sub-fix pattern of *Mutate Method Invocation Expression* (i.e., FP10), which is about replacing the method name of a method invocation expression with another appropriate method name [52]. This fix pattern can be matched to any statement that contains a method name under method invocation expression. However, in this paper, the similar fix patterns yielded by `FixMiner` have more context information. Therefore, we consider context information to check the compatibility of `FixMiner`'s patterns against the patterns used by literature APR systems. For example, the fix pattern shown in Figure 17 is to modify the buggy method name of a method invocation expression with another appropriate method name which is inside a `Return-Statement`. As the context information refers to a `Return-Statement` the fix pattern shown in Figure 17 considered as compatible with *Mutate Return Statement* (i.e., FP12.). Nevertheless, the mapping is conservative in the sense that we consider that a `FixMiner` pattern matches a pattern from the literature as long as it can fit with the literature pattern.

7 Related Work

Automated Program Repair. Patch generation is one of the key tasks in software maintenance since it is time-consuming and tedious. If this task is automated, the cost and time of developers for maintenance will be dramatically reduced. To address the issue, many automated techniques have been proposed for program repair [68]. GenProg [45], which leverages genetic programming, is a pioneering work on program repair. It relies on mutation operators that insert, replace, or delete code elements. Although these mutations can create a limited number of variants, GenProg could fix several bugs (in their evaluation, test cases were passed for 55 out of 105 real program bugs) automatically, although most of them have been found to be incorrect patches later. PACHIKA [13] leverages object behavior models. SYDIT [65] and LASE [66] automatically extracts an edit script from a program change. While several techniques have focused on fixability, Kim et al. [33] pointed out that patch acceptability should be considered as well in program repair. Automatically generated patches often have nonsensical structures and logic even though those patches can fix program bugs with respect to program behavior (i.e., w.r.t. test cases). To address this issue, they proposed PAR, which leverages manually-crafted fix patterns. Similarly Long and Rinard proposed Prophet [58] and Genesis [56] which generates patches by leveraging fix patterns extracted from the history of changes in repositories. Recently, several approaches [5, 22] leveraging deep learning have been proposed for learning to fix bugs. Even recent APR approaches that target bug reports rely on fix templates to generate patches. iFixR [39] is such an example which builds on top of the templates built TBar [52] templates. Overall, we note that the community is going in the direction of implementing repair strategies based on fix patterns or templates. Our work is thus essential in this direction as it provides a scalable, accurate and actionable tool to mine such relevant patterns for automated program repair.

Code differencing. Code differencing is an important research and practice concern in software engineering. Although commonly used by human developers in manual tasks, differencing at the text line level granularity [69] is generally unsuitable for automated analysis of changes and the associated semantics. AST differencing work has benefited in the last decade for the extensive investigations that the research community has performed for general tree differencing [2, 6, 9, 11]. ChangeDistiller [21] and GumTree [17] constitute the current state-of-the-art for AST differencing in Java. In this work, we have selected GumTree as the base tool for the computation of edit scripts as its results have been validated by humans, and it has been shown to be more accurate and fine-grained edit scripts. Nevertheless, we have further enhanced the edit script yielding an algorithm that keeps track of contextual information. Our approach echoes a recently published work by Huang et al. [28]: their CLDIFF tool similarly enriches the AST produced by GumTree to enable the generation of concise code differences. The tool however was not available at the time of our experiments. Thus, to satisfy the input requirements of our fix pattern mining approach, we implement `Rich Edit Script`, to enrich GumTree-yielded edit scripts by retaining more contextual information.

Change patterns. The literature includes a large body of work on mining change patterns.

Mining-based approaches. In recent years, several approaches have built upon the idea of mining patterns or leveraging templates. Fluri et al., based on edit scripts computed by their ChangeDistiller AST difference, have used hierarchical clustering to discover unknown change types in three Java applications [20]. They have limited themselves however to considering only changes implementing the 41 basic change types that they had previously identified [19]. Kreutzer et al. have developed C3 to automatically detect groups of similar code changes in code repositories with the help of clustering algorithms [40]. Martinez and Monperrus [61] assessed the relationship between the types of bug fixes and automatic program repair. They perform extensive large scale empirical investigations on the nature of human bug fixes based on fine-grained abstract syntax tree differences by ChangeDistiller. Their experiments show that the mined models are more effective for driving the search compared to random search. Their models however remain at a high level and may not carry any actionable patterns to be used by other template-based APR. Our work however also targets systematizing and automating the "mining of actionable fix patterns" to feed pattern-based program repair tools.

An example application is related to work by Livshits and Zimmermann [55] who discovered application-specific repair templates by using association rule mining on two Java projects. More recently, Hanam et al. [23] have developed the BugAID technique for discovering most prevalent repair templates in JavaScript. They use AST differencing and unsupervised learning algorithms. Our objective is similar to theirs, focusing on Java programs with different abstraction levels of the patterns. FixMiner builds on a three-fold clustering strategy where we iteratively discover recurrent changes preserving surrounding code context.

Studies on code change redundancies. A number of empirical studies have confirmed that code changes are repeatedly performed in software code bases [34, 36, 67, 97]. Same changes are prevalent because multiple occurrences of the same bug require the same change. Similarly, when an API evolves, or when migrating to a new library/framework, all calling code must be adapted by same collateral changes [76]. Finally, code refactoring or routine code cleaning can lead to similar changes. In a manual investigation, Pan et al. [77] have identified 27 extractable repair templates for Java software. Among other findings, they observed that if-condition changes are the most frequently applied to fix bugs. Their study, however, does not discuss whether most bugs are related to If-condition or not. This is important as it clarifies the context to perform if-related changes. Recently, Nguyen et al. [73] have empirically found that 17-45% of bug fixes are recurring. Our focus in this paper is to provide tool-support automated approach to inferring change patterns in a dataset to drive repair patterns to guide APR mutation. Moreover, our patterns are less generic than the ones in previous works (e.g., as in [73, 77]).

Concurrently to our work, Jiang et al. have proposed SimFix [30], and Wen et al. CapGen [90] which implements a similar idea of leveraging code redundancies using contextual information for shaping the program repair space. In `FixMiner` however, the pattern mining phase is independent from the patch generation phase, and the resulting patterns are tractable and reusable as input to other APR systems.

Generic and semantic patch inference. Ideally, `FixMiner` is a tool that aims at performing towards finding a generic patch that can be leveraged by automated program repair to correctly update a collection of buggy code fragments. This problem has been recently studied by approaches such as `spdiff` [3, 4] which work on the inference of generic and semantic patches. This approach, however, is known to be poorly scalable and has constraints of producing ready-to-use semantic patches that can be used by the Coccinelle matching and transformation engine [7]. There have however a number of prior work that tries to detect and summarize program changes. A seminal work by Chawathe et al. describes a method to detect changes to structured information based on an ordered tree and its updated version [9]. The goal was to derive a compact description of the changes with the notion of minimum cost edit script which has been used in the recent ChangeDistiller and GumTree tools. The representations of edit operations, however, are either often too overfit to a particular code change or abstract very loosely the change so that it cannot be easily instantiated. Neamtiu et al. [70] proposed an approach for identifying changes, additions and deletions of C program elements based on structural matching of syntax trees. Two trees that are structurally identical but have differences in their nodes are considered to represent matching program fragments. Kim et al. [35] have later proposed a method to infer "change-rules" that capture many changes. They generally express changes related to program headers (method headers, class names, package names, etc.). Weissgerber et al. [89] have also proposed a technique to identify likely refactorings in the changes that have been performed in Java programs. Overall, these generic patch inference approaches address the challenges of how the patterns that will be leveraged in practice. Our work goes in that direction by yielding different kinds of patterns for different purposes: shape-based patterns reduce the context of code to match;

action patterns are the ones that correspond to fix patterns used in the repair community; token patterns are used for inferring collateral evolutions.

8 Conclusion

We have presented `FixMiner`, a systematic and automated approach to mine relevant and actionable fix patterns for automated program repair. The approach builds on an iterative and three-fold clustering strategy, where in each round forming clusters of identical trees representing recurrent patterns.

We assess the consistency of the mined patterns with the patterns in the literature. We further demonstrate with the implementation of an automated repair pipeline that the patterns mined by our approach are relevant for generating correct patches for 26 bugs in the Defects4J benchmark. These correct patches correspond to 81% of all plausible patches generated by the tool.

Availability All the data and tool support is available at :

`https://github.com/SerVal-DTF/fixminer-core`

.

Acknowledgements This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, through RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), pp. 89–98. IEEE (2007)
2. Al-Ekram, R., Adma, A., Baysal, O.: diffx: an algorithm to detect changes in multi-version xml documents. In: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research, pp. 1–11. IBM Press (2005)
3. Andersen, J., Lawall, J.L.: Generic patch inference. Automated software engineering **17**(2), 119–148 (2010)
4. Andersen, J., Nguyen, A.C., Lo, D., Lawall, J.L., Khoo, S.C.: Semantic patch inference. In: Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on, pp. 382–385. IEEE (2012)
5. Bhatia, S., Singh, R.: Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv preprint arXiv:1603.06129 (2016)
6. Bille, P.: A survey on tree edit distance and related problems. Theoretical computer science **337**(1-3), 217–239 (2005)
7. Brunel, J., Doligez, D., Hansen, R.R., Lawall, J.L., Muller, G.: A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09, pp. 114–126. ACM, New York, NY, USA (2009). DOI 10.1145/1480881.1480897
8. Campos, J., Riboira, A., Perez, A., Abreu, R.: Gzoltar: an eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 378–381. ACM (2012)
9. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change Detection in Hierarchically Structured Information. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD '96, pp. 493–504. ACM, New York, NY, USA (1996). DOI 10.1145/233269.233366

10. Chen, L., Pei, Y., Furia, C.A.: Contract-based program repair without the contracts. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 637–647. IEEE, Urbana, IL, USA (2017)
11. Chilowicz, M., Duris, E., Roussel, G.: Syntax tree fingerprinting for source code similarity detection. In: Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on, pp. 243–247. IEEE (2009)
12. Coker, Z., Hafiz, M.: Program transformations to fix c integers. In: Proceedings of the International Conference on Software Engineering, pp. 792–801. IEEE, San Francisco, CA, USA (2013)
13. Dallmeier, V., Zeller, A., Meyer, B.: Generating fixes from object behavior anomalies. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 550–554. IEEE Computer Society (2009)
14. Duley, A., Spandikow, C., Kim, M.: Vdiff: a program differencing algorithm for verilog hardware description language. Automated Software Engineering **19**(4), 459–490 (2012)
15. Durieux, T., Cornu, B., Seinturier, L., Monperrus, M.: Dynamic patch generation for null pointer exceptions using metaprogramming. In: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering, pp. 349–358. IEEE (2017)
16. Falleri, J.R.: GumTree. `https://github.com/GumTreeDiff/gumtree` (Last Access: Mar. 2018.)
17. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of ACM/IEEE International Conference on Automated Software Engineering, pp. 313–324. ACM, Vasteras, Sweden (2014)
18. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceeding of the 19th ICSM, pp. 23–32. IEEE (2003)
19. Fluri, B., Gall, H.C.: Classifying change types for qualifying change couplings. In: Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on, pp. 35–45. IEEE (2006)
20. Fluri, B., Giger, E., Gall, H.C.: Discovering patterns of change types. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 463–466. IEEE, L'Aquila, Italy (2008)
21. Fluri, B., Wuersch, M., PInzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on software engineering **33**(11) (2007)
22. Gupta, R., Pal, S., Kanade, A., Shevade, S.: Deepfix: Fixing common c language errors by deep learning. In: AAAI, pp. 1345–1351 (2017)
23. Hanam, Q., Brito, F.S.d.M., Mesbah, A.: Discovering bug patterns in javascript. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 144–156. ACM (2016)
24. Hashimoto, M., Mori, A.: Diff/ts: A tool for fine-grained structural change analysis. In: 2008 15th Working Conference on Reverse Engineering, pp. 279–288. IEEE (2008)
25. Herzig, K., Zeller, A.: The impact of tangled code changes. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp. 121–130. IEEE, San Francisco, CA, USA (2013)
26. Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM Sigplan Notices **39**(12), 92–106 (2004)
27. Hua, J., Zhang, M., Wang, K., Khurshid, S.: Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th International Conference on Software Engineering, pp. 12–23. ACM (2018)
28. Huang, K., Chen, B., Peng, X., Zhou, D., Wang, Y., Liu, Y., Zhao, W.: Cldiff: generating concise linked code differences. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 679–690. ACM (2018)
29. Jaro, M.A.: Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. Journal of the American Statistical Association **84**(406), 414–420 (1989)
30. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 298–309. ACM (2018)

31. Just, R., Jalali, D., Ernst, M.D.: 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, pp. 437–440. ACM, San Jose, CA, USA (2014)
32. Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: Repairing programs with semantic code search. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 295–306. IEEE, Lincoln, NE, USA (2015)
33. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 802–811. IEEE Press (2013)
34. Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: Proceedings of the 31st International Conference on Software Engineering, pp. 309–319. IEEE Computer Society (2009)
35. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: ICSE, vol. 7, pp. 333–343. Citeseer (2007)
36. Kim, S., Pan, K., Whitehead Jr, E.: Memories of bug fixes. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 35–45. ACM (2006)
37. Koyuncu, A., Bissyandé, T., Kim, D., Klein, J., Monperrus, M., Le Traon, Y.: Impact of Tool Support in Patch Construction. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 237–248. ACM, New York, NY, USA (2017)
38. Koyuncu, A., Bissyandé, T.F., Kim, D., Liu, K., Klein, J., Monperrus, M., Traon, Y.L.: D&c: A divide-and-conquer approach to ir-based bug localization. arXiv preprint arXiv:1902.02703 (2019)
39. Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Monperrus, M., Klein, J., Le Traon, Y.: ifixr: bug report driven program repair. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 314–325. ACM (2019)
40. Kreutzer, P., Dotzler, G., Ring, M., Eskofier, B.M., Philippsen, M.: Automatic clustering of code changes. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, pp. 61–72. ACM, New York, NY, USA (2016). DOI 10.1145/2901739.2901749. URL http://doi.acm.org.proxy.bnl.lu/10.1145/2901739.2901749
41. Le, X.B.D., Chu, D.H., Lo, D., Le Goues, C., Visser, W.: S3: syntax-and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, pp. 593–604. ACM, Paderborn, Germany (2017)
42. Le, X.B.D., Le, Q.L., Lo, D., Le Goues, C.: Enhancing automated program repair with deductive verification. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), pp. 428–432. IEEE, Raleigh, NC, USA (2016)
43. Le, X.D., Lo, D., Le Goues, C.: History driven program repair. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, vol. 1, pp. 213–224. IEEE (2016)
44. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. TSE **38**(1), 54–72 (2012)
45. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Trans. Software Eng. **38**(1), 54–72 (2012)
46. Lee, J., Kim, D., Bissyandé, T.F., Jung, W., Le Traon, Y.: Bench4bl: reproducibility study on the performance of ir-based bug localization. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 61–72. ACM (2018)
47. Lin, W., Chen, Z., Ma, W., Chen, L., Xu, L., Xu, B.: An empirical study on the characteristics of python fine-grained source code change types. In: Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on, pp. 188–199. IEEE (2016)
48. Liu, K., Kim, D., Bissyandé, T.F., Yoo, S., Le Traon, Y.: Mining fix patterns for findbugs violations. IEEE Transactions on Software Engineering (2018)

49. Liu, K., Kim, D., Koyuncu, A., Li, L., Bissyandé, T.F., Le Traon, Y.: A closer look at real-world patches. In: 2018 IEEE International Conference on Software Maintenance and Evolution, pp. 275–286. IEEE (2018)
50. Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J., Le Traon, Y.: You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 102–113. IEEE (2019)
51. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In: Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, pp. 456–467. IEEE (2019)
52. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: TBar: revisiting template-based automated program repair. In: Proceedings of the 28th International Symposium on Software Testing and Analysis (2019)
53. Liu, K., Koyuncu, A., Kim, K., Kim, D., Bissyandé, T.F.: LSRepair: Live search of fix ingredients for automated program repair. In: Proceedings of the 25th Asia-Pacific Software Engineering Conference, pp. 658–662 (2018)
54. Liu, X., Zhong, H.: Mining stackoverflow for program repair. In: Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, pp. 118–129. IEEE (2018)
55. Livshits, B., Zimmermann, T.: DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 296–305. ACM, New York, NY, USA (2005). DOI 10.1145/1081706.1081754
56. Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, pp. 727–739. ACM, Paderborn, Germany (2017)
57. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 166–178. ACM, Bergamo, Italy (2015)
58. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 298–312. ACM, St. Petersburg, FL, USA (2016)
59. Martinez, M., Duchien, L., Monperrus, M.: Automatically extracting instances of code change patterns with ast analysis. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, pp. 388–391. IEEE (2013)
60. Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M.: Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empirical Software Engineering **22**(4), 1936–1964 (2017)
61. Martinez, M., Monperrus, M.: Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering **20**(1), 176–205 (2015)
62. Martinez, M., Monperrus, M.: Astor: A program repair library for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 441–444. ACM, Saarbrücken, Germany (2016)
63. Martinez, M., Monperrus, M.: Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: Proceedings of the 10th SSBSE, pp. 65–86. Springer (2018)
64. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: Looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 448–458. IEEE, Florence, Italy (2015)
65. Meng, N., Kim, M., McKinley, K.S.: Systematic editing: generating program transformations from an example. ACM SIGPLAN Notices **46**(6), 329–342 (2011)
66. Meng, N., Kim, M., McKinley, K.S.: Lase: locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 502–511. IEEE Press (2013)

67. Molderez, T., Stevens, R., De Roover, C.: Mining change histories for unknown systematic edits. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 248–256. IEEE Press (2017)
68. Monperrus, M.: Automatic software repair: a bibliography. ACM Computing Surveys (CSUR) **51**(1), 17 (2018)
69. Myers, E.W.: Ano (nd) difference algorithm and its variations. Algorithmica **1**(1-4), 251–266 (1986)
70. Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Software Engineering Notes **30**(4), 1–5 (2005)
71. Nguyen, H.A., Nguyen, A.T., Nguyen, T.N.: Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pp. 138–147. IEEE (2013)
72. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: Proceedings of the 35th ICSE, pp. 772–781. IEEE (2013)
73. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J., Nguyen, T.N.: Recurring bug fixes in object-oriented programs. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, vol. 1, pp. 315–324. IEEE (2010)
74. Osman, H., Lungu, M., Nierstrasz, O.: Mining frequent bug-fix code changes. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on, pp. 343–347. IEEE (2014)
75. Oumarou, H., Anquetil, N., Etien, A., Ducasse, S., Taiwe, K.D.: Identifying the exact fixing actions of static rule violation. In: Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, pp. 371–379. IEEE (2015)
76. Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and Automating Collateral Evolutions in Linux Device Drivers. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, Eurosys '08, pp. 247–260. ACM, New York, NY, USA (2008). DOI 10.1145/1352592.1352618
77. Pan, K., Kim, S., Whitehead, E.J.: Toward an understanding of bug fix patterns. Empirical Software Engineering **14**(3), 286–315 (2009)
78. Park, J., Kim, M., Ray, B., Bae, D.H.: An empirical study of supplementary bug fixes. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pp. 40–49. IEEE Press (2012)
79. Pawlik, M., Augsten, N.: Rted: a robust algorithm for the tree edit distance. Proceedings of the VLDB Endowment **5**(4), 334–345 (2011)
80. Rolim, R., Soares, G., Gheyi, R., D'Antoni, L.: Learning quick fixes from code repositories. arXiv preprint arXiv:1803.03806 (2018)
81. Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R.: Elixir: Effective object-oriented program repair. In: Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on, pp. 648–659. IEEE (2017)
82. Skiena, S.S.: The stony brook algorithm repository. URL http://www. cs. sunysb. edu/algorith/implement/nauty/implement. shtml (1997)
83. Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., Maia, M.A.: Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In: Proceedings of SANER (2018)
84. Tan, S.H., Roychoudhury, A.: relifix: Automated repair of software regressions. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 471–482. IEEE Press (2015)
85. Tao, Y., Kim, S.: Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp. 180–190. IEEE (2015)
86. Thomas, S.W., Nagappan, M., Blostein, D., Hassan, A.E.: The impact of classifier configuration and classifier combination on bug localization. TSE **39**(10), 1427–1443 (2013)
87. Tian, Y., Lawall, J., Lo, D.: Identifying linux bug fixing patches. In: Proceedings of the 34th International Conference on Software Engineering, pp. 386–396. IEEE Press (2012)
88. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, May 16-24,, pp. 364–374. IEEE, Vancouver, Canada (2009)

89. Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on, pp. 231–240. IEEE (2006)
90. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.C.: Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering, pp. 1–11. ACM (2018)
91. Wen, M., Wu, R., Cheung, S.C.: Locus: Locating bugs from software changes. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 262–273. IEEE (2016)
92. Winkler, W.E.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. (1990)
93. Xin, Q., Reiss, S.P.: Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 660–670. IEEE (2017)
94. Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering, pp. 416–426. IEEE, Buenos Aires, Argentina (2017)
95. Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M.: Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering **43**(1), 34–55 (2017)
96. Ying, A.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C.: Predicting source code changes by mining change history. IEEE transactions on Software Engineering **30**(9), 574–586 (2004)
97. Yue, R., Meng, N., Wang, Q.: A characterization study of repeated bug fixes. In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, pp. 422–432. IEEE (2017)

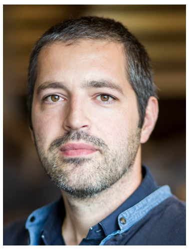
Anil Koyuncu is a PhD student at the University of Luxembourg. He received a Master's degree from Politecnico di Milano, Italy. His research interest includes automatic patch repair, fault localization.

Kui Liu received the master degree in computer engineering from Southwest University, China, in 2013. He is working toward the PhD degree in software engineering at the University of Luxembourg from 2016. His current research focuses on automated program repair.

Tegawendé F. Bissyandé is research scientist with the Interdisciplinary Center for Security, Reliability and Trust at the University of Luxembourg. He holds a PhD in computer from the Université de Bordeaux in 2013, and an engineering degree (MSc) from ENSEIRB. His research interests are in debugging, including bug localization and program repair, as well as code search, including code clone detection and code classification. He has published research results

in all major venues in Software engineering (ICSE, ESEC/FSE, ASE, ISSTA, EMSE, TSE). His research is supported by FNR (Luxembourg National Research Fund). Dr. Bissyandé is the PI of the CORE RECOMMEND project on program repair, under which the current work has been performed.

Dongsun Kim is a Software Engineer at Furiosa.ai. He was formerly a research associate at the University of Luxembourg and a post-doctoral fellow at the Hong Kong University of Science and Technology. His research interest includes testing AI systems, automatic patch generation, fault localization, static analysis, and search-based software engineering. In particular, automated debugging is his current focus. His recent work has been recognized by several awards such as a featured article of the IEEE Transactions on Software Engineering (TSE) and ACM SIGSOFT Distinguished Paper of the International Conference on Software Engineering (ICSE). He is leading the FIXPATTERN project funded by FNR (Luxembourg National Research Fund) CORE programme.



Jacques Klein is senior research scientist at the University of Luxembourg, and at the Interdisciplinary Centre for Security, Reliability and Trust (SnT). He received his Ph.D. degree in Computer Science from the University of Rennes, France in 2006. His main areas of expertise are threefold: (1) Mobile Security (malware detection, prevention and dissection, static analysis for security, vulnerability detection, etc.); (2) Software Reliability (software testing, semi-automated and fully-automated program repair, etc.); (3) Data Analytics (multi-objective reasoning and optimization, model-driven data analytics, time series pattern recognition, text mining, etc.). In addition to academic achievements, Dr. Klein has also standing experience and expertise on successfully running industrial projects with several industrial partners in various domains by applying data analytics, software engineering, information retrieval, etc., to their research problems.

Martin Monperrus is Professor of Software Technology at KTH Royal Institute of Technology. He was previously associate professor at the University of Lille and adjunct researcher at Inria. He received a Ph.D. from the University of Rennes, and a Master's degree from the Compiègne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, program hardening and chaos engineering.

Yves Le Traon is professor at University of Luxembourg, in the domain of software engineering, testing, security and model-driven engineering. He received his engineering degree and his PhD in Computer Science at the "Institut National Polytechnique" in Grenoble, France, in 1997. From 1998 to 2004, he was an associate professor at the University of Rennes, in Brittany, France. From 2004 to 2006, he was an expert in Model-Driven Architecture and Validation at "France Te le com R&D". In 2006, he became professor at Telecom Bretagne (Ecole Nationale des Tlcommunications de Bretagne). He is currently the head of the CSC Research Unit (e.g. Department of Computer Science) at University of Luxembourg. He is a member of the Interdisciplinary Centre for Security, Reliability and Trust (SnT), where he leads the research group SERVAL (SEcurity Reasoning and VALidation). His research interests include software testing, model-driven engineering, model based testing, evolutionary algorithms, software security, security policies and Android security. The current key-topics he explores are related to Internet of things (IoT), Big Data (stress testing, multi-objective optimization and data protection), and mobile security and reliability. He is author of more than 140 publications in international peer-reviewed conferences and journals.