Accurate and Efficient Refactoring Detection in Commit History

Nikolaos Tsantalis, Matin Mansouri,
Laleh M. Eshkevari, Davood Mazinanian
Concordia University, Montreal, Quebec, Canada

Danny Dig
Oregon State University
Corvallis, Oregon, USA

ABSTRACT

Refactoring detection algorithms have been crucial to a variety of applications: (i) empirical studies about the evolution of code, tests, and faults, (ii) tools for library API migration, (iii) improving the comprehension of changes and code reviews, etc. However, recent research has questioned the accuracy of the state-of-the-art refactoring detection tools, which poses threats to the reliability of their application. Moreover, previous refactoring detection tools are very sensitive to user-provided similarity thresholds, which further reduces their practical accuracy. In addition, their requirement to build the project versions/revisions under analysis makes them inapplicable in many real-world scenarios.

To reinvigorate a previously fruitful line of research that has stifled, we designed, implemented, and evaluated RMiner, a technique that overcomes the above limitations. At the heart of RMiner is an AST-based statement matching algorithm that determines refactoring candidates without requiring user-defined thresholds. To empirically evaluate RMiner, we created the most comprehensive oracle to date that uses triangulation to create a dataset with considerably reduced bias, representing 3,188 refactorings from 185 open-source projects. Using this oracle, we found that RMiner has a precision of 98% and recall of 87%, which is a significant improvement over the previous state-of-the-art.

CCS CONCEPTS

• **Software and its engineering → Software evolution**;

KEYWORDS

Refactoring, Commit, Git, Abstract Syntax Tree, Oracle, Accuracy

1 INTRODUCTION

Refactoring [28, 37, 60] is a key practice in agile development processes, and is well supported by refactoring tools that are standard with all major IDEs. Refactoring research is approaching now 30 years. A very active line of research focused on refactoring detection algorithms [4, 15, 18, 27, 31, 34–36, 46, 56, 62, 65, 74, 78] that compute a (likely) set of refactorings that developers applied on the source code. Other researchers used refactoring detection to empirically study [4, 6, 7, 45, 61, 63, 73] software evolution, and to support other software engineering tasks, such as library adaptation [5, 18, 40, 79], software merging [21], code completion [27, 31], and code review [1, 33, 34].

However, the accuracy of refactoring detection tools has been recently questioned. An independent study [67] has shown that Ref-Finder [46, 62] (one of the most widely used refactoring detection tools) had an overall precision of 35% and an overall recall of 24%, while a more recent study [39, 42] has shown that Ref-Finder had an overall average precision of 27%. Missing refactorings (false negatives) is a serious threat to the generalizability of empirical studies, or can cause other dependent tools to carry incomplete operations. Detecting incorrect refactorings (false positives) is even more severe as it makes the conclusions of the empirical study wrong, or causes other dependent tools to apply the wrong operations.

Moreover, the majority of the refactoring detection tools use similarity thresholds, and provide a set of default threshold values that are empirically determined through experimentation on a (rather small) number of projects (e.g., one project for UMLDiff [77], three for Ref-Finder [62] and RefactoringCrawler [18], and ten for RefDiff [65]). The derived threshold values are possibly overfitted to the characteristics of the examined projects, and thus cannot be general enough to take into account all possible ways developers apply refactorings in projects from different domains. As a result, these threshold values require a calibration to align with the particular refactoring practices applied in a project, which is tedious. Moreover, finding a *universal* threshold value might be infeasible. Several researchers proposed methods for deriving threshold values in metric-based detection techniques [2, 22, 25, 26, 59]. However, the precision and recall can vary significantly even for the same software system when using different threshold values [17], and software systems relying on different architectural styles and frameworks require different threshold values [3]. Therefore, research has shown that it is very difficult to derive *universal* threshold values that can work well for all projects, regardless of their architectural style, application domain, and development practices.

Furthermore, most refactoring detection tools take as input two *fully built versions* of a software system that contain binding information for all named code entities, linked across all library dependencies. However, a recent study [71] has shown that only 38% of the change history of software systems can be successfully compiled. This is a serious limitation for performing longitudinal refactoring detection in the commit history of projects, posing a threat to the external validity of empirical studies, since only a small number of project versions can be effectively used for extracting refactoring datasets.

Thus, we designed, implemented, and evaluated RMiner, a novel technique that overcomes the above limitations. RMiner takes as input two revisions (i.e., a commit and its parent from the commit history in git-based version control repositories) of a Java project, and returns a list of refactoring operations applied between these

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180206

two revisions. RMiner provides highly *accurate*, *efficient*, and *scalable* refactoring detection in the commit history of a project without requiring to build each individual commit.

At the heart of RMiner is an AST-based statement matching algorithm that does not require user-specified thresholds, yet it is immune to the noise introduced by the statement restructuring during refactoring operations. It relies on our two novel techniques: *abstraction* deals with changes in statements' AST type due to refactoring, and *argumentization* deals with changes in sub-expressions within statements due to parameterization. Using the matched AST statements, we designed powerful detection rules for 15 representative refactoring types.

To empirically evaluate RMiner, we first needed to create a reliable, comprehensive, and representative oracle. This is a daunting task that mounts tremendous challenges on its own. First is the danger of creating an *incomplete oracle*. Researchers previously created such oracles by inspecting release notes [18, 20] or commit messages [74]. However, only a small percentage (21%) of the release notes include refactoring operations (and only for a subset of refactorings that affect the backward compatibility of public APIs) [54]. Moreover, developers do not reliably indicate the presence of refactoring operations in commit log messages [55].

Second is the danger of creating a *biased oracle*. For example, researchers [62] created an oracle based on the findings of a single tool (i.e., Ref-Finder) configured with a more relaxed similarity threshold value in order to detect more refactoring instances (followed by removing false positives through manual inspection), and then evaluated the precision and recall of the same tool configured with a more strict similarity threshold value. However, this might still miss a large number of true instances due to an algorithm design flaw, implementation error, or inappropriate threshold value, leading to precision and recall that are significantly different than those reported by independent researchers (i.e., 35% precision and recall of 24% [67], and an overall average precision of 27% [39, 42] vs. the authors' [62] reported precision and recall of 74% and 96%).

Third is the danger of creating an *artificial oracle*. For example, researchers [65] created an oracle by asking students to apply refactorings in open-source projects. These *seeded* refactorings [11], can be used to reliably compute the recall, since all applied refactorings are known a-priori. However, seeded refactorings are not representative of real refactorings for two reasons: (i) they are *artificial*, i.e., they do not carry higher-level intents (e.g., facilitate a maintenance task, eliminate a code smell, improve code understandability), and (ii) they are *isolated* and do not overlap with typical maintenance activities (e.g., other edits in the same commit to fix bugs, add new features). A significant percentage (46% [57]) of refactored program entities are also edited or further refactored in the same commit, a practice commonly referred as *floss refactoring* [55–57, 64]. Not accounting for this real code evolution, significantly and artificially increases the signal-to-noise ratio, thus making the detection less challenging than in real-world scenarios.

To avoid the above problems with refactoring oracles, we rely on state-of-the-art procedures [24, 30, 48] that use triangulation between multiple sources (human experts and tools) to determine the ground truth. We started from an award-winning, publicly available dataset of refactoring instances [64], originating from 538 commits from 185 open-source GitHub projects. Moreover, the refactoring instances from 222 of these commits were confirmed by the developers who actually performed the refactorings (i.e., the commit authors) through surveys, and further re-validated by us manually to ensure correctness. To ensure the completeness of the dataset, we executed two tools that analyze repository commits without requiring to build the project, namely our RMiner and the previous state-of-the-art RefDiff [65], on all 538 commits of the dataset. These tools use complementary detection methods, thus are likely to detect a more comprehensive set of refactoring instances. Then we manually validated 4,108 unique refactoring instances detected by the two tools, out of which 3,188 were true positives. The validation process took 9 person-months to be completed, and involved up to three refactoring experts per instance to negotiate agreement. With this oracle we evaluate the precision and recall of RMiner and the previous state-of-the-art tool, RefDiff.

Based on these results, we launch a community call to action related to refactoring detection and refactoring oracles. We offer several actionable implications and results for researchers, tool builders, and developers. First, our oracle of 3,188 true refactorings from 538 commits across 185 projects provides an invaluable resource for validating novel refactoring tools and for comparing existing approaches. Educators can use our dataset when teaching software engineering to show examples of refactorings in their real-life contexts. Using RMiner, researchers can replicate existing empirical studies and refute or confirm previously-held beliefs. Moreover, researchers can use RMiner to reduce the noise [12, 13] created by refactorings, such as file/directory renaming, and significantly improve the accuracy of other tools. For example, tools that identify bug-introducing changes (e.g., the widely used SZZ [47, 66, 76]) can utilize RMiner to avoid flagging changes that do not alter the program behavior (i.e., refactorings) as bug-introducing. Tools that trace requirements to code [51, 52] could use RMiner to recover traceability links that are broken due to applied refactorings.

Practical aspects, such as RMiner's speed and consumption of raw code changes from commits, enable novel applications not possible before, such as *online* refactoring detection on partial input, when a developer inspects a code diff to review a change, or tries to understand code evolution selectively (e.g., using the "blame" feature on a program element of interest). As *floss refactoring* [55–57, 64] is prevalent, the trace of code changes left behind by refactorings can mask the changes actually intended by developers [16, 44]. Moreover, refactorings distract developers during code reviews [33], when the changes are inspected with text diff tools (commonly used in IDEs, repository hosting services, and code review tools). Integrating RMiner with the diff and code review tools can raise the level of abstraction for code changes originating from refactorings, thus helping developers better understand the code evolution.

This paper makes the following contributions:

(1) We present the *first* refactoring detection algorithm that does not require any code similarity thresholds to operate.
(2) We implement our detection algorithm into a tool, RMiner (short for RefactoringMiner [69]), which operates on version control commits, and provides an API for external use.
(3) We create the most accurate, complete, and representative oracle of refactoring operations to date, comprising 3,188 refactorings

found in 538 commits from 185 open-source projects, which we validate with multiple tools and experts [70].

(4) We evaluate RMiner and find that it achieves 98% precision and 87% recall, and takes on median 58 ms to analyze a commit, a significant improvement over the previous state-of-the-art.

(5) We release a tool infrastructure to compare the accuracy of multiple refactoring detection tools, either using a user-defined fixed oracle, or a dynamically generated oracle based on tool agreement [53].

2 APPROACH

RMiner takes as input two revisions (i.e., a commit and its parent in the directed acyclic graph that models the commit history in git-based version control repositories) of a Java project, and returns a list of refactoring operations applied between these two revisions. It supports the detection of 15 refactoring types for 4 different kinds of code elements, as shown in Table 1. This is a representative set of refactoring types, because it covers all structural code elements (i.e., packages, types, methods, and fields), and also covers control-flows of program statements (e.g., Extract and Inline Method).

Table 1: Refactorings detected by RMiner

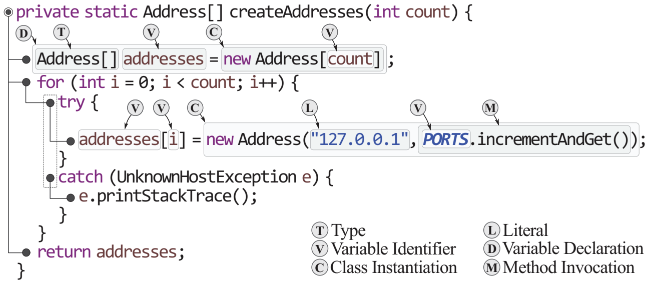
Code element	Refactorings
package	Change Package (move, rename, split)
type	Move Class, Rename Class Extract Superclass/Interface
method	Extract Method, Inline Method Pull Up Method, Push Down Method Rename Method, Move Method Extract and Move Method
field	Pull Up Field, Push Down Field Move Field

Unlike other existing refactoring detection approaches, such as Ref-Finder [46], RefactoringCrawler [18], and JDEvAn [80], which analyze all files in two snapshots/versions of a Java project, RMiner analyzes only the added, deleted, and changed files between the two revisions. This makes RMiner not only more efficient, because it has less code elements to analyze and compare, but also more accurate, because the number of code element combinations to be compared is significantly less, thus reducing the probability of incorrect code element matches.

2.1 Notation

We adopt and extend the notation defined by Biegel et al. [8] for representing the information that we extract from each revision using the Eclipse JDT Abstract Syntax Tree (AST) Parser. Notice that we configure the parser to create the ASTs of the added, deleted, and changed Java compilation units in each revision *without* resolving binding information from the compiler, and thus there is no need to build the source code. Consequently, all *referenced types* (e.g., parameters, variable/field declarations, extended superclass, implemented interfaces) are stored as they appear in the AST, as we are not able to obtain their fully qualified names. For each revision r, we extract the following information:

- TD_r: The set of type declarations (i.e., classes, interfaces, enums) affected in r. For a child commit, this set includes the type declarations inside changed and added Java files, while for a parent commit, this set includes the type declarations inside changed and removed Java files. Each element td of the set is a tuple of the form (p, n, F, M), where p is the parent of td, n is the name of td, F is the set of fields declared inside td, and M is the set of methods declared inside td. For a top-level type declaration p corresponds to the package of the compilation unit td belongs to, while for a nested/inner type declaration p corresponds to the package of the compilation unit td belongs to concatenated with the name of the type declaration td is nested under.
- F_r: The set of fields inside the type declarations of TD_r. It contains tuples of the form (c, t, n), where c is the fully qualified name of the type declaration the field belongs to (constructed by concatenating the package name p with the type declaration name n), t is the type of the field, and n is the name of the field.
- M_r: The set of methods inside the type declarations of TD_r. It contains tuples of the form (c, t, n, P, b), where c is the fully qualified name of the type declaration the method belongs to, t is the return type of the method, n is the name of the method, P is the ordered parameter list of the method, and b is the body of the method (could be `null` if the method is abstract or native).
- D_r: The set of all directories in r as returned by command `git ls-tree`. Each directory is represented by its path p.



```
private static Address[] createAddresses(int count) {
  Address[] addresses = new Address[count];
  for (int i = 0; i < count; i++) {
    try {
      addresses[i] = new Address("127.0.0.1", PORTS.incrementAndGet());
    }
    catch (UnknownHostException e) {
      e.printStackTrace();
    }
  }
  return addresses;
}
```

Figure 1: Representation of a method body as a tree.

The body of a method is represented as a tree capturing the nesting structure of the code, where each node corresponds to a statement, similar to the representation used by Fluri et al. [23]. For a composite statement (i.e., a statement that contains other statements within its body, such as `for`, `while`, `do-while`, `if`, `switch`, `try`, `catch`, `synchronized block`, `label`), the node contains the statement's type and the expression(s) appearing within parenthesis before the statement's body. For a leaf statement (i.e., a statement without a body), the node contains the statement itself. In order to avoid storing AST information into memory, for each statement/expression we keep its string representation in a pretty-printed format where all redundant whitespace and multi-line characters are removed. In addition, we use an AST Visitor to extract all variable identifiers, method invocations, class instantiations, variable declarations, types, literals, and operators appearing within each statement/expression and store them in a pretty-printed format within the corresponding statement node. Figure 1 shows the tree-like representation of the body of method `createAddresses`, along with the information extracted by the AST Visitor for two of its statements.

2.2 Statement matching

The matching of the statements between two code fragments is a core function that we use throughout the refactoring detection rules described in this paper. Our statement matching algorithm has been inspired by Fluri et al. [23], in the sense that we also match the statements in a bottom-up fashion, starting from the matching of leaf statements and then proceeding to composite statements. However, in our solution, outlined in Algorithm 1, we do not use any similarity measure to match the statements, and thus we do not require the definition of similarity thresholds.

To reduce the chances of erroneous matches, we follow a conservative approach, in which we match the statements in rounds, where each subsequent round has a less strict *match condition* than the previous round. Thus, the statements matched in earlier rounds are "safer" matches, and are excluded from being matched in the next rounds. In this way, the next round, which has a more relaxed match condition, has fewer statement combinations to check.

We match leaf statements in three rounds (lines 2-8). In the first round, we match the statements with identical string representation and nesting depth. In the second round, we match the statements with identical string representation regardless of their nesting depth. In the last round, we match the statements that become identical after replacing the AST nodes being different between the two statements. We match composite statements in three rounds as well (lines 9-16), using exactly the same match conditions as those used for leaf statements combined with an additional condition that requires at least one pair of their children to be matched (line 10), assuming that both composite statements have children.

In all rounds, we apply two pre-processing techniques on the input statements (line 5 in function `matchNodes`), namely *abstraction* and *argumentization* to deal with specific changes taking place in the code when applying Extract, Inline, and Move Method refactorings.

Abstraction: Some refactoring operations, such as Extract and Inline Method, often introduce or eliminate `return` statements when a method is extracted or inlined, respectively. For example, when an expression is extracted from a given method, it appears as a `return` statement in the extracted method. To facilitate the matching of statements having a different AST node type, we *abstract* the statements that wrap expressions. When both statements being compared follow one of the following formats:

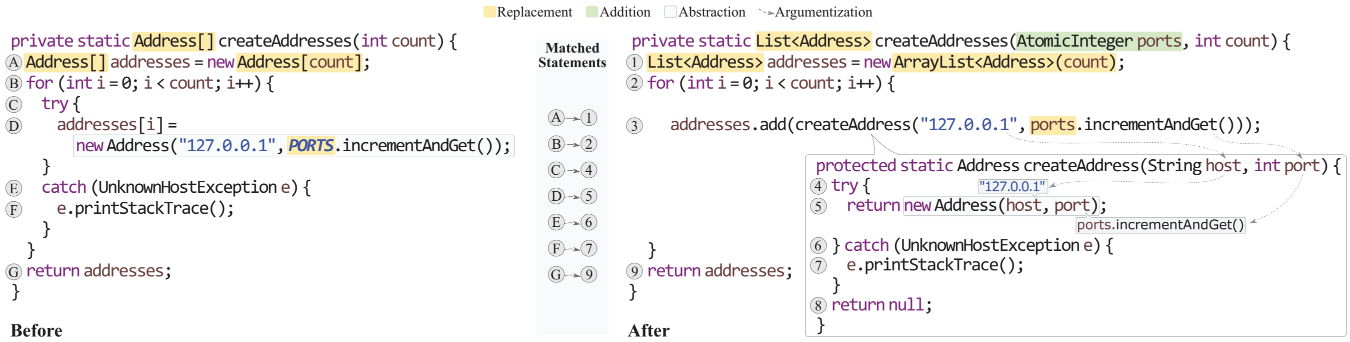
- `return expression;` i.e., returned expression
- `Type var = expression;` i.e., initializer of a variable declaration
- `var = expression;` i.e., right hand side of an assignment
- `call(expression);` i.e., single argument of a method invocation
- `if(expression)` i.e., condition of a composite statement

then they are abstracted to `expression` before their comparison. Figure 2 shows an example of abstraction, where the assignment statement Ⓓ from the code before refactoring, and the return statement ⑤ from the code after refactoring, are abstracted to expressions `new Address("127.0.0.1", PORTS.incrementAndGet())` and `new Address(host, port)`, respectively.

Argumentization: Some refactoring operations may replace expressions with parameters, and vice versa. For example, when duplicated code is extracted into a common method, all expressions being different among the duplicated code fragments are *parameterized* (i.e., they are replaced with parameters in the extracted method). The duplicated code fragments are replaced with calls to the extracted method, where each expression being different is passed as an argument. In many cases, the arguments may differ substantially from the corresponding parameter names, leading to a low textual similarity of the code before and after refactoring. Argumentization is the process of replacing parameter names with the corresponding arguments in the code after refactoring. Figure 2 shows an example of argumentization, where parameter names `host` and `port` are replaced with arguments `"127.0.0.1"` and `ports.incrementAndGet()`, respectively, in statement ⑤.

Algorithm 1: Statement matching

```
   Input  : Trees $T_1$ and $T_2$
   Output : Set $M$ of matched node pairs, Sets of unmatched
            nodes $U_{T_1}$, $U_{T_2}$ from $T_1$ and $T_2$, respectively
1  $M \leftarrow \emptyset$, $U_{T_1} \leftarrow \emptyset$, $U_{T_2} \leftarrow \emptyset$
2  $L_1 \leftarrow T_1$.leafNodes, $L_2 \leftarrow T_2$.leafNodes
3  condition1 $(n_1, n_2) \rightarrow n_1$.text = $n_2$.text $\wedge$ $n_1$.depth = $n_2$.depth
4  condition2 $(n_1, n_2) \rightarrow n_1$.text = $n_2$.text
5  condition3 $(n_1, n_2) \rightarrow$ replacements ($n_1$.text, $n_2$.text)
6  $L'_1$ , $L'_2$ = matchNodes ($L_1$, $L_2$, condition1) // round #1
7  $L''_1$, $L''_2$ = matchNodes ($L'_1$, $L'_2$, condition2) // round #2
8  matchNodes ($L''_1$, $L''_2$, condition3) // round #3
9  $C_1 \leftarrow T_1$.compositeNodes, $C_2 \leftarrow T_2$.compositeNodes
10 condition4 $(n_1, n_2) \rightarrow$
     $\exists\,(k_1, k_2) \in M \mid k_1 \in n_1$.children $\wedge$ $k_2 \in n_2$.children
11 condition1 $(n_1, n_2)$ = condition1 $\wedge$ condition4
12 condition2 $(n_1, n_2)$ = condition2 $\wedge$ condition4
13 condition3 $(n_1, n_2)$ = condition3 $\wedge$ condition4
14 $C'_1$ , $C'_2$ = matchNodes ($C_1$, $C_2$, condition1) // round #1
15 $C''_1$, $C''_2$ = matchNodes ($C'_1$, $C'_2$, condition2) // round #2
16 matchNodes ($C''_1$, $C''_2$, condition3) // round #3
17 $U_{T_1} \leftarrow T_1$.nodes $\setminus M_{T_1}$, $U_{T_2} \leftarrow T_2$.nodes $\setminus M_{T_2}$
1  Function matchNodes($N_1$, $N_2$, matchCondition)
2      foreach $n_1 \in N_1$ do
3          $P \leftarrow \emptyset$
4          foreach $n_2 \in N_2$ do
5              $pn_1, pn_2 \leftarrow$ preprocessNodes($n_1$,$n_2$)
6              if matchCondition($pn_1$,$pn_2$) then
7                  $P \leftarrow P \cup (n_1, n_2)$
8              end
9          end
10         if $|P| > 0$ then
11             bestMatch $\leftarrow$ findBestMatch($P$)
12             $M \leftarrow M \cup$ bestMatch
13             $N_1 \leftarrow N_1 \setminus$ bestMatch.$n_1$
14             $N_2 \leftarrow N_2 \setminus$ bestMatch.$n_2$
15         end
16     end
17     return $N_1$, $N_2$
18 end
```



```
private static Address[] createAddresses(int count) {
Ⓐ Address[] addresses = new Address[count];
Ⓑ for (int i = 0; i < count; i++) {
Ⓒ   try {
Ⓓ     addresses[i] =
          new Address("127.0.0.1", PORTS.incrementAndGet());
    }
Ⓔ   catch (UnknownHostException e) {
Ⓕ     e.printStackTrace();
    }
  }
Ⓖ return addresses;
}
```

```
private static List<Address> createAddresses(AtomicInteger ports, int count) {
① List<Address> addresses = new ArrayList<Address>(count);
② for (int i = 0; i < count; i++) {

③   addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));

  }
⑨ return addresses;
}
```

```
protected static Address createAddress(String host, int port) {
④ try {
⑤   return new Address(host, port);
⑥ } catch (UnknownHostException e) {
⑦   e.printStackTrace();
  }
⑧ return null;
}
```

Figure 2: Statement matching for an Extract Method refactoring in project hazelcast [38].

The same process is applied to the statements of inlined and moved methods. In particular, when an instance method is moved to a target class, we might have a parameter (or a source class field access) of target type that is removed from the original method, or a parameter of source type that is added to the original method. In the case of removal, the removed parameter (or field access) might be replaced with `this` reference in the moved method, while in the case of addition, `this` reference might be replaced with the added parameter in the moved method.

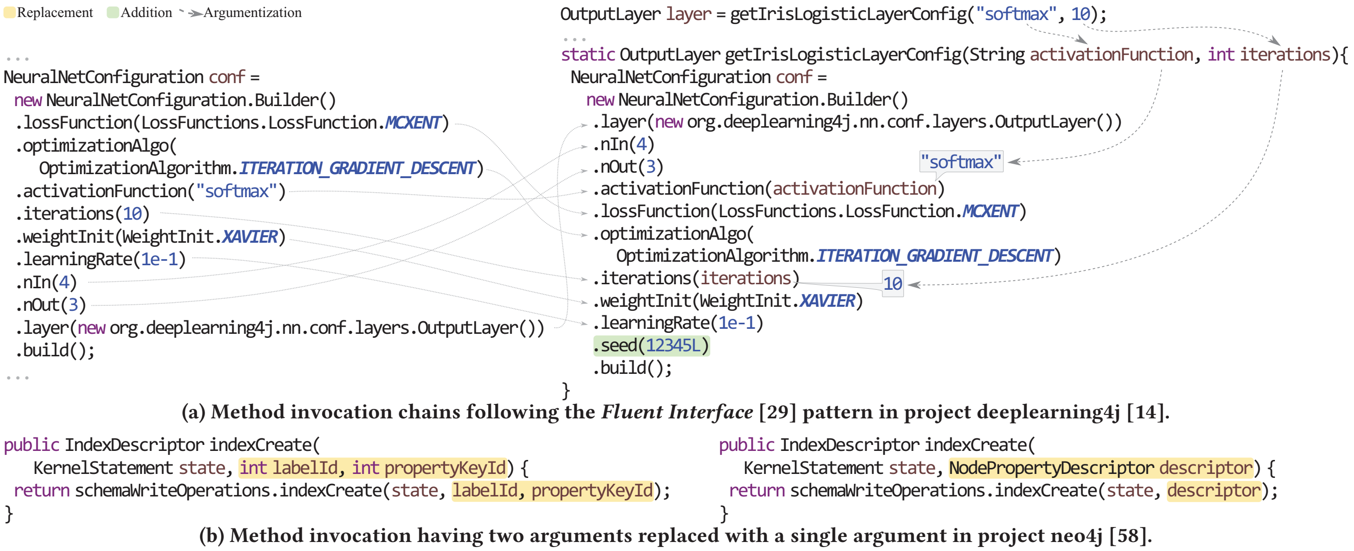
By applying the techniques of abstraction and argumentization the original statements Ⓓ and ⑤ in Figure 2 are transformed to `new Address("127.0.0.1", PORTS.incrementAndGet())` and `new Address("127.0.0.1", ports.incrementAndGet())`, respectively, and thus can be identically matched by replacing static field `PORTS` with parameter `ports`. On the other hand, string similarity measures would require a very low threshold to match these statements. For instance, the Levenshtein distance [50] (commonly used for computing string similarity) between the original statements Ⓓ and ⑤ is 44 edit operations, which can be normalized to a similarity of $1 - 44/65 = 0.32$, where 65 is the length of the longest string corresponding to statement Ⓓ. The bigram similarity [49] (used by ChangeDistiller [23]) between statements Ⓓ and ⑤ is equal to 0.3. It is clear that the string similarity measures used by the majority of the refactoring detection tools are susceptible to code changes applied by some refactoring operations, such as parameterization, especially when the arguments differ substantially from the parameter names. Therefore, our pre-processing techniques facilitate the matching of statements with low textual similarity.

Function `matchNodes`, finds all possible matching nodes in tree T_2 for a given node in tree T_1 and stores the matching node pairs into set P. Function `findBestMatch`(P) (line 11), sorts the node pairs in P and selects the top-sorted one. Leaf node pairs are sorted based on 3 criteria. First, based on the string edit distance [50] of the nodes in ascending order (i.e., more textually similar node pairs rank higher). Second, based on the absolute difference of the nodes' depth in ascending order (i.e., node pairs with more similar depth rank higher). Third, based on the absolute difference of the nodes' index in their parent's list of children in ascending order (i.e., node pairs with more similar position in their parent's list of children rank higher). Composite node pairs are sorted with an additional criterion, which is applied right after the first criterion: based on the ratio of the nodes' matched children in descending order (i.e., node pairs with more matched children rank higher).

Algorithm 2: Syntax-aware replacements of AST nodes

```
Input   :Statements $s_1$ and $s_2$
Output: True if statements can be identically matched after
        syntax-aware replacements, otherwise false
Function replacements($s_1$, $s_2$)
    $N_{s_1} \leftarrow \emptyset$, $N_{s_2} \leftarrow \emptyset$, $R \leftarrow \emptyset$
    foreach t ∈ nodeTypes do
        $\text{common}_t \leftarrow s_1.\text{nodes}_t \cap s_2.\text{nodes}_t$
        $N_{s_1} \leftarrow N_{s_1} \cup \{ s_1.\text{nodes}_t \setminus \text{common}_t \}$
        $N_{s_2} \leftarrow N_{s_2} \cup \{ s_2.\text{nodes}_t \setminus \text{common}_t \}$
    end
    $d$ = distance($s_1$, $s_2$)
    foreach $n_{s_1} \in N_{s_1}$ do
        $C \leftarrow \emptyset$
        foreach $n_{s_2} \in N_{s_2}$ do
            if compatibleForReplacement($n_{s_1}$, $n_{s_2}$) then
                $d'$ = distance($s_1$.replace($n_{s_1}$, $n_{s_2}$), $s_2$)
                if $d' < d$ then
                    $C \leftarrow C \cup (n_{s_1}, n_{s_2})$
                end
            end
        end
        if $|C| > 0$ then
            best ⟵ smallestDistance($C$)
            $d$ = best.distance
            $r$ = best.replacement
            $R \leftarrow R \cup r$
            $s_1$ = $s_1$.replace($r.n_{s_1}$, $r.n_{s_2}$)
        end
    end
    if $s_1 = s_2$ then
        return true
    else
        return false
end
```

Function `replacements` (Algorithm 2), takes as input two statements and performs replacements of AST nodes until the statements



```
...
NeuralNetConfiguration conf =
 new NeuralNetConfiguration.Builder()
 .lossFunction(LossFunctions.LossFunction.MCXENT)
 .optimizationAlgo(
    OptimizationAlgorithm.ITERATION_GRADIENT_DESCENT)
 .activationFunction("softmax")
 .iterations(10)
 .weightInit(WeightInit.XAVIER)
 .learningRate(1e-1)
 .nIn(4)
 .nOut(3)
 .layer(new org.deeplearning4j.nn.conf.layers.OutputLayer())
 .build();
...
```

```
OutputLayer layer = getIrisLogisticLayerConfig("softmax", 10);
...
static OutputLayer getIrisLogisticLayerConfig(String activationFunction, int iterations){
 NeuralNetConfiguration conf =
   new NeuralNetConfiguration.Builder()
   .layer(new org.deeplearning4j.nn.conf.layers.OutputLayer())
   .nIn(4)
   .nOut(3)
   .activationFunction(activationFunction)
   .lossFunction(LossFunctions.LossFunction.MCXENT)
   .optimizationAlgo(
      OptimizationAlgorithm.ITERATION_GRADIENT_DESCENT)
   .iterations(iterations)
   .weightInit(WeightInit.XAVIER)
   .learningRate(1e-1)
   .seed(12345L)
   .build();
}
```

(a) Method invocation chains following the *Fluent Interface* [29] pattern in project deeplearning4j [14].

```
public IndexDescriptor indexCreate(
    KernelStatement state, int labelId, int propertyKeyId) {
 return schemaWriteOperations.indexCreate(state, labelId, propertyKeyId);
}
```

```
public IndexDescriptor indexCreate(
    KernelStatement state, NodePropertyDescriptor descriptor) {
 return schemaWriteOperations.indexCreate(state, descriptor);
}
```

(b) Method invocation having two arguments replaced with a single argument in project neo4j [58].

Figure 3: Replacement of method invocations.

become textually identical. This approach has two main advantages over existing methods relying on textual similarity. First, there is no need to define a similarity threshold. There is empirical evidence that developers interleave refactoring with other types of programming activity (e.g., bug fixes, feature additions, or other refactoring operations) [55, 56, 64]. In many cases, the changes caused by these different activities may overlap [57]. Some of these changes may even change substantially the original code being part of a refactoring operation. For example, a code fragment is originally extracted, and then some temporary variables are inlined in the extracted method. The longer the right-hand-side expressions assigned to the temporary variables, the more textually different the original statements will be after refactoring. Therefore, it is impossible to define a *universal* similarity threshold value that can cover any possible scenario of overlapping changes. On the other hand, our approach does not pose any restriction on the replacements of AST nodes, as long as these replacements are syntactically valid. Second, the replacements found within two matched statements can help to infer other edit operations taking place on the refactored code (a phenomenon called *refactoring masking* [68]), such as renaming of variables, generalization of types, and merging of parameters. On the other hand, similarity-based approaches lose this kind of valuable information.

Initially, our algorithm computes the intersection between the sets of variable identifiers, method invocations, class instantiations, types, literals, and operators extracted from each statement, respectively, in order to exclude from replacements the AST nodes being common in both statements, and include only those being different between the statements (lines 3-7). AST nodes that cover the entire statement (e.g., a method invocation followed by `;`) are also excluded from replacements in order to avoid having an excessive number of matching statements. All attempted replacements are *syntax-aware*, in the sense that only *compatible* AST nodes are allowed to be replaced (line 12), i.e., types can be replaced only by types, operators can be replaced only by operators, while all remaining expression types can be replaced by any of the remaining expression types (e.g., a variable can be replaced by a method invocation). Out of all possible replacements for a given node from the first statement that decrease the original edit distance of the input statements, we select the replacement corresponding to the smallest edit distance (line 20).

In the special case when two method invocations are considered for replacement, function compatibleForReplacement(n_{s_1}, n_{s_2}) examines the expressions used for invoking the methods. If these expressions are chains of method invocations, as the case shown in Figure 3a (commonly known as the *Fluent Interface* [29] pattern in API design), then we extract the individual method invocations being part of each chain and compute their intersection ignoring any differences in the order of the invocations inside each chain. If the number of common invocations is larger than the uncommon ones, then we consider the original method invocations as compatible for replacement. In the example of Figure 3a, there are 9 common invocations (two of them are identically matched after applying the argumentization technique), and only 1 uncommon. Notice that string similarity measures produce very low similarity value for this case. For instance, the normalized Levenshtein similarity between the two statements is 0.47, while the bigram similarity is 0.46.

Handling of changes not supported by Algorithm 2: As explained before, AST nodes covering the entire statement, such as the method invocations shown in Figure 3b, are excluded from replacements to avoid having an excessive number of matching statements. However, there might be changes in their list of arguments that cannot be handled by Algorithm 2, such as the insertion or deletion of an argument, and the replacement of multiple arguments with a single one and vice versa. This is because we designed the algorithm to perform only one-to-one AST node replacements and does not support one-to-many, many-to-one, one-to-zero (i.e., deletion), zero-to-one (i.e., insertion) replacements, as this would increase substantially its computational cost. To overcome this limitation, we allow the replacements of textually different method invocations covering the entire statement, as long as they have an identical invocation expression, an identical method name, and a non-empty intersection of arguments (e.g., argument `state` in the example of Figure 3b).

Table 2: Refactoring detection rules

Refactoring type	Rule
Change Method Signature m_a to m_b	$\exists (M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid m_a \in M^- \wedge m_b \in M^+ \wedge m_a.c = m_b.c \wedge$ $\boxed{m_a.n \neq m_b.n \Rightarrow \textsc{Rename Method}}$ ① $(U_{T_1} = \emptyset \wedge U_{T_2} = \emptyset \wedge \texttt{allExactMatches}(M)) \vee$ ② $(\lvert M\rvert > \lvert U_{T_1}\rvert \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge \texttt{locationHeuristic}(m_a, m_b) \wedge \texttt{compatibleSignatures}(m_a, m_b)) \vee$ ③ $(\lvert M\rvert > \lvert U_{T_2}\rvert \wedge \texttt{locationHeuristic}(m_a, m_b) \wedge \exists\, \texttt{extract}(m_a, m_x)) \vee$ ④ $(\lvert M\rvert > \lvert U_{T_1}\rvert \wedge \texttt{locationHeuristic}(m_a, m_b) \wedge \exists\, \texttt{inline}(m_x, m_b))$
Extract Method m_b from m_a	$\exists (M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^+ \wedge m_a.c = m_b.c \wedge \neg\texttt{calls}(m_a, m_b) \wedge \texttt{calls}(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_2}\rvert$
Inline Method m_b to $m_{a'}$	$\exists (M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_b.b, m_{a'}.b) \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^- \wedge m_{a'}.c = m_b.c \wedge \texttt{calls}(m_a, m_b) \wedge \neg\texttt{calls}(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_1}\rvert$
Change Class Signature td_a to td_b	$\exists (td_a, td_b) \mid td_a \in TD^- \wedge td_b \in TD^+ \wedge (td_a.M \supseteq td_b.M \vee td_a.M \subseteq td_b.M) \wedge (td_a.F \supseteq td_b.F \vee td_a.F \subseteq td_b.F)$ $\boxed{td_a.p \neq td_b.p \Rightarrow \textsc{Move Class}}$ $\boxed{td_a.n \neq td_b.n \Rightarrow \textsc{Rename Class}}$
Move Method m_a to m_b	$\exists (M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid m_a \in M^- \wedge m_b \in M^+ \wedge m_a.c \neq m_b.c \wedge \lvert M\rvert > \lvert U_{T_1}\rvert \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge$ $(td_a, td_{a'}) \in TD^= \wedge m_a \in td_a \wedge (td_b, td_{b'}) \in TD^= \wedge m_b \in td_{b'} \wedge (\texttt{importsType}(td_{a'}, m_b.c) \vee \texttt{importsType}(td_b, m_a.c))$ $\boxed{\texttt{subType}(m_a.c, m_b.c) \Rightarrow \textsc{Pull Up Method}}$ $\boxed{\texttt{subType}(m_b.c, m_a.c) \Rightarrow \textsc{Push Down Method}}$
Move Field f_a to f_b	$\exists (f_a, f_b) \mid f_a \in F^- \wedge f_b \in F^+ \wedge f_a.c \neq f_b.c \wedge f_a.t = f_b.t \wedge f_a.n = f_b.n \wedge$ $(td_a, td_{a'}) \in TD^= \wedge f_a \in td_a \wedge (td_b, td_{b'}) \in TD^= \wedge f_b \in td_{b'} \wedge (\texttt{importsType}(td_{a'}, f_b.c) \vee \texttt{importsType}(td_b, f_a.c))$ $\boxed{\texttt{subType}(f_a.c, f_b.c) \Rightarrow \textsc{Pull Up Field}}$ $\boxed{\texttt{subType}(f_b.c, f_a.c) \Rightarrow \textsc{Push Down Field}}$
Extract m_b from m_a & Move to $m_b.c$	$\exists (M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^+ \wedge m_a.c \neq m_b.c \wedge$ $\neg\texttt{calls}(m_a, m_b) \wedge \texttt{calls}(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge (td_a, td_{a'}) \in TD^= \wedge m_a \in td_a \wedge \texttt{importsType}(td_{a'}, m_b.c)$
Extract Supertype td_b from td_a	$\exists (td_a, td_b) \mid (td_a, td_{a'}) \in TD^= \wedge td_b \in TD^+ \wedge \texttt{subType}(\texttt{type}(td_{a'}), \texttt{type}(td_b))$ $\boxed{\exists\, \texttt{pullUp}(m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \vee \exists\, \texttt{pullUp}(f_a, f_b) \mid f_a \in td_a \wedge f_b \in td_b \Rightarrow \textsc{Extract Superclass}}$ $\boxed{\exists (m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \wedge \texttt{identicalSignatures}(m_a, m_b) \wedge m_b.b = \texttt{null} \Rightarrow \textsc{Extract Interface}}$
Change Package p_a to p_b	$\exists (p_a, p_b) \mid \texttt{path}(p_a) \in D^- \wedge \texttt{path}(p_b) \in D^+ \wedge \exists\, \texttt{MoveClass}(td_a, td_b) \mid td_a.p = p_a \wedge td_b.p = p_b$

$\texttt{matching}(T_1, T_2)$ returns a set of matched statement pairs (M) between the trees T_1 and T_2 representing method bodies, and two sets of unmatched statements from T_1 (U_{T_1}) and T_2 (U_{T_2}), respectively
$\texttt{indexOf}(m, td)$ returns the position of m inside type declaration td $\texttt{typeDecl}(c)$ returns the type declaration of type c $\texttt{type}(td)$ returns the qualified name of type declaration td
$\texttt{locationHeuristic}(m_a, m_b) = \lvert\texttt{indexOf}(m_a, \texttt{typeDecl}(m_a.c)) - \texttt{indexOf}(m_b, \texttt{typeDecl}(m_b.c))\rvert \leq \lvert M_c^- - M_c^+\rvert$ $\texttt{importsType}(td, t)$ returns true if type declaration td depends on type t
$\texttt{compatibleSignatures}(m_a, m_b) = m_a.P \supseteq m_b.P \vee m_a.P \subseteq m_b.P \vee \lvert m_a.P \cap m_b.P\rvert \geq \lvert(m_a.P \cup m_b.P) \setminus (m_a.P \cap m_b.P)\rvert$ $\texttt{calls}(m_a, m_b)$ returns true if method m_a calls m_b
$\texttt{subType}(c_a, c_b)$ returns true if c_a is a direct or indirect subclass of c_b or implements interface c_b $\texttt{path}(p)$ returns the directory path for package p

2.3 Refactoring detection

The detection of refactorings takes place in two phases. The first phase is less computationally expensive, since the code elements are matched only based on their signatures. Our assumption is that two code elements having an identical signature in two revisions correspond to the same code entity, regardless of the changes that might have occurred within their bodies. The second phase is more computationally expensive, since the remaining code elements are matched based on the statements they have in common within their bodies. In a nutshell, in the first phase, our algorithm matches code elements in a top-down fashion, starting from classes and continuing to methods and fields. Two code elements are matched only if they have an identical *signature*. Assuming a and b are two revisions of a project:

- Two type declarations td_a and td_b have an identical signature, if $td_a.p = td_b.p \wedge td_a.n = td_b.n$
- Two fields f_a and f_b have an identical signature, if $f_a.c = f_b.c \wedge f_a.t = f_b.t \wedge f_a.n = f_b.n$
- Two methods m_a and m_b have an identical signature, if $m_a.c = m_b.c \wedge m_a.t = m_b.t \wedge m_a.n = m_b.n \wedge m_a.P = m_b.P$
- Two directories d_a and d_b are identical, if $d_a.p = d_b.p$

After the end of the first phase, we consider the unmatched code elements from revision a as *potentially deleted*, and store them in sets TD^-, F^-, M^-, and D^-, respectively. We consider the unmatched code elements from revision b as *potentially added*, and store them in sets TD^+, F^+, M^+, and D^+, respectively. Finally, we store the pairs of matched code elements between revisions a and b in sets $TD^=$, $F^=$, $M^=$, and $D^=$, respectively.

In the second phase, our algorithm matches the remaining code elements (i.e., the *potentially deleted* code elements with the *potentially added* ones) in a bottom-up fashion, starting from methods and continuing to classes, to find code elements with signature changes or code elements involved in refactoring operations.

Examination order of refactoring types: We detect the refactoring types in the order they appear in Table 2 by applying the rules shown in the second column of the table. The order of examination is very important for the accuracy of our approach. We order the refactoring types according to their *locality of change* [19], starting from local refactoring types (i.e., within a single method/class) and proceeding with global ones (i.e., among different classes or packages). The intuition behind this order comes from empirical evidence showing that small and local refactorings are more frequent than big and global ones [9, 56], and thus there is a higher probability that the potentially added/deleted code elements resulted from local rather than global refactorings. Whenever a refactoring type is processed, we remove the matched code elements from the sets of potentially deleted/added code elements, and add them to the corresponding sets of matched code elements. This affects the code elements examined in the refactoring types that follow, thus reducing the noise level and improving accuracy.

Best match selection: For the refactoring types involving statement matching in their detection rule, when a code element (i.e., method) has multiple matches, we always select the best match. The reason is that the same piece of code cannot be part of multiple refactoring operations. For example, a method cannot be renamed to multiple methods. Our algorithm sorts the matching method pairs based on 4 criteria, which serve as proxies for method similarity at statement level. First, based on the total number of matched statements in descending order (i.e., method pairs with more matched statements rank higher). Second, based on the total number of exactly matched statements in descending order (i.e., method pairs with more identical statements rank higher). Third, based on the total edit distance [50] between the matched statements in ascending order (i.e., method pairs with more textually similar statements rank higher). Fourth, based on the edit distance between the method names in ascending order (i.e., method pairs with more textually similar names rank higher).

As Table 2 shows, the refactoring types examined first have more elaborate and strict rules. This is crucial to avoid early erroneous matches that would negatively affect the accuracy of the detected instances for the refactoring types that follow. For example, the *location heuristic* applied in sub-rules ②, ③, and ④ of the Change Method Signature refactoring type, ensures that the positional difference of two matched methods is less or equal to the absolute difference in the number of methods added to and deleted from a given type declaration. The intuition behind this heuristic is that developers do not tend to change the position of an already existing method inside its type declaration when changing its signature. Assuming that only method renames take place in a type declaration, the number of potentially added and deleted methods will be equal, and thus the location heuristic will be satisfied only for the method pairs having the same position before and after refactoring. This heuristic is particularly effective in cases of extensive method signature changes in test classes (e.g., see the case of extensive unit test renames in project cassandra [10]), where developers tend to copy-and-modify older unit tests to create new ones [72], and thus several methods share very similar statements with each other. Sub-rules ③ and ④ take into account the case where a method with a signature change has a significant portion of its body extracted or inlined, respectively. For instance, in the case shown in Figure 2, the result of statement matching between methods `createAddresses` before and after refactoring is $M = \{(A, 1), (B, 2), (G, 9)\}$, i.e., $|M| = 3$, while $U_{T_1} = \{C, D, E, F\}$, i.e., $|U_{T_1}| = 4$, and $U_{T_2} = \{3\}$, i.e., $|U_{T_2}| = 1$, and thus sub-rule ② fails to match the methods. On the other hand, sub-rule ③ matches successfully the methods, because $|M| > |U_{T_2}|$ and there exists at least one method extracted from the original `createAddresses`.

3 EVALUATION

We empirically evaluate the usefulness of RMiner by answering the following research questions:

RQ1: What is the accuracy of RMiner and how does it compare to the previous state-of-the-art?

RQ2: What is the execution time of RMiner and how does it compare to the previous state-of-the-art?

We answer the first research question by computing standard metrics from information retrieval (i.e., precision and recall). As these metrics require having a reliable oracle, we use complementary methods to create the most accurate oracle to date. Moreover, we compare the accuracy and running time of RMiner against that of the previous state-of-the-art tool, RefDiff, as Silva and Valente [65] established that RefDiff significantly outperforms other widely used refactoring detection tools, such as Ref-Finder and RefactoringCrawler.

3.1 Oracle construction

Having a correct, complete and representative oracle of refactorings is fundamental for computing precision and recall in a reliable manner. Therefore, we used a publicly available dataset of refactoring instances [64], comprising 538 commits from 185 open-source GitHub-hosted projects monitored over a period of two months (between June 8th and August 7th, 2015). The authors of [64] manually validated all refactoring instances in the dataset. Moreover, the instances found in 222 of these commits were confirmed by the developers who actually performed the refactorings (i.e., the commit authors) through surveys. We re-validated all instances to ensure their correctness. Fourteen cases actually corresponded to multiple instances summarized as a single refactoring operation (e.g., a refactoring reported as "method `foo` extracted from `bar` and x other methods" corresponds to $x + 1$ separate Extract Method instances). We broke down these cases to separate instances by manually finding the summarized code elements. This dataset can be considered correct, since all instances went through rigorous manual validation by multiple authors and in several cases were confirmed by the developers who actually performed them. It is one of the most representative datasets to date, since all instances are real refactorings found in 185 different Java projects, they are motivated by a variety of reasons [64], and take place along with other changes/refactorings in the same commit. However, the completeness of the dataset is not guaranteed, since there is no reported recall for the refactoring detection tool used in [64]. To ensure the completeness of the dataset, we executed two tools that analyze repository commits without requiring to build the project, namely RMiner and RefDiff [65], on all 538 commits of the dataset. These tools use complementary detection methods (i.e., a more conservative threshold-free approach based on statement matching vs. a more relaxed threshold-based approach based on token similarity), thus are likely to detect a more comprehensive set of refactoring instances. For the validation process, we created a web application, which listed all detected refactorings along with links to the corresponding GitHub commits. Through this web application, the validators were able to inspect the change diff provided by GitHub, and enter their validation and comments. In total, we manually validated 4,108 unique refactoring instances detected by the two tools, out of which 3,188 were true positives and 920 were false positives. The validation process was labor-intensive and involved 3 validators for a period of 3 months (i.e., 9 person-months). To give a sense of the manual inspection difficulty, on average, a commit contained 7.89 refactoring instances (median = 2), 14.25 changed files (median = 5), and 1,047 changed lines of code (median = 211).

Table 3: Precision and recall per refactoring type

Refactoring Type	RMiner		RefDiff	
	Precision	Recall	Precision	Recall
Inline Method	98.96	86.36	84.35	88.18
Extract Method	98.63	84.72	93.03	90.95
Move Field	88.42	95.45	30.19	45.45
Move Class	100	96.24	99.90	93.53
Extract Interface	100	100	76.92	55.56
Push Down Method	100	100	95.00	61.29
Push Down Field	100	86.21	100	100
Change Package	85.00	100	N/A	N/A
Pull Up Method	100	90.48	80.60	85.71
Pull Up Field	100	96.30	64.00	59.26
Move Method	95.17	76.36	32.25	92.25
Rename Method	97.78	83.28	85.54	89.59
Extract Superclass	95.08	100	100	18.97
Rename Class	98.33	71.08	89.71	73.49
Extract & Move Method	95.92	41.23	73.02	80.70
Overall	97.96	87.20	75.71	85.76

3.2 Precision and Recall

Since RefDiff does not support the detection of Change Package refactoring, we did not consider the instances detected by RMiner as false negatives for RefDiff. Moreover, to ensure a fair comparison for RMiner, we considered all classes within a changed package as being moved from the original package to the new one, and added the corresponding Move Class instances to those originally detected by RMiner.

Table 3 shows the precision and recall of RMiner and RefDiff. Notice that RMiner has better precision than RefDiff in all refactoring types, except for Extract Superclass (95% vs. 100%), where RefDiff seems to be extremely conservative due to its low recall. This is a result of RMiner following a conservative threshold-free approach for detecting refactorings, which results in high precision. The lowest precision for RMiner is observed for Change Package and Move Field (85% and 88.4%, respectively), while it has over 95% precision in all other refactoring types. The Achilles' heel of RefDiff, in terms of precision, is the detection of Move Method and Move Field refactorings (32% and 30%, respectively). We found two recurring scenarios causing such false positives for RefDiff. In the first scenario, RefDiff misses the detection of a class move to another package, and consequently reports the methods and fields of that class as being moved from the original class, which is assumed to be deleted, to another class, which is assumed to be newly added. In the second scenario, a subclass extending/implementing a given superclass/interface is deleted, and a new subclass is added, which overrides the superclass/interface methods in a similar way. RefDiff reports these methods as being moved from the deleted to the added subclass. We believe both scenarios occur because RefDiff does not examine if there is an import dependency between the source and target class of a candidate Move Method/Field refactoring, but relies only on code similarity.

Since RMiner achieves very high precision, does RefDiff have better recall due to its less conservative threshold-based approach? We found this is true only for 7 refactoring types, while for the other 7 types RMiner has better recall. In particular, RMiner has an increased recall of 37% to 81% for inheritance-related refactorings (i.e., Pull Up Field, Push Down Method, Extract Superclass/Interface), and 50% for Move Field refactoring, while the increase in recall for Move Class and Pull Up Method refactorings is smaller, 3% and 5%, respectively. In contrast, RefDiff has only a slightly increased recall of 2% to 6% for local refactorings, such as Extract/Inline/Rename Method and Rename Class, while the increase in recall for inter-class refactorings, such as Push Down Field, Move Method, and Extract and Move Method, is larger, 14%, 16% and 39%, respectively.

An inherent advantage of RefDiff, helping it to achieve higher recall in refactoring types involving code similarity, is that it ignores the structure of the code by treating code fragments as bags of tokens. Therefore, any change in the structure of the code (e.g., merging/splitting of conditionals, as in the case found in project jetty [41]) before or after the actual refactoring will not affect its detection ability, as long as the tokens remain the same. A disadvantage of RefDiff is its inability to deal with changes in the tokens caused by the refactoring itself (e.g., parameterization of expressions in Extract Method refactoring), or another overlapping refactoring (e.g., local variable renames inside the body of a refactored method). On the other hand, RMiner deals robustly with this kind of changes by applying statement pre-processing techniques, such as argumentization, and allowing syntax-aware replacement of AST nodes within matched statements. Further research on hybrid methods that combine the advantages of RMiner and RefDiff seems to have great potential.

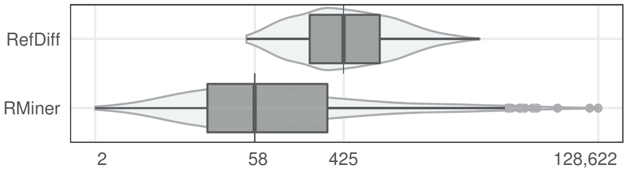


Figure 4: Execution time per commit (ms).

3.3 Execution Time

Figure 4 shows the distribution of the execution time of RMiner and RefDiff for each examined commit, collected by executing separately each tool on the same machine with the following specifications: Intel Core i7-2620M CPU @ 2.70GHz, 16 GB DDR3 memory, 1 TB SSD, Windows 10 OS, and Java 1.8.0 x64. For each tool, we recorded the time taken for parsing the source code of the examined and its parent commit, and the time taken to detect refactorings using the `System.nanoTime` Java method. On median, RMiner is 7 times faster than RefDiff (58 vs. 425 ms). We also applied the Wilcoxon signed rank test on the paired samples of the time execution for each commit, which rejected the null hypothesis "RefDiff execution time is smaller than that of RMiner" with a p-value $<$ 2.2e-16, and thus we can statistically conclude that RMiner is faster on our commit sample. We should note that RMiner has 70 outlier commits that were processed in over one second, representing 13% of the examined commits. Among these commits 39 took between 1-5 sec, 14 between 5-10 sec, 11 between 10-30 sec, and 6 took over 30 sec with the most time consuming commit taking 128 sec.

3.4 Limitations

Missing context: As explained in Section 2, RMiner analyzes only the added, deleted, and changed files between two revisions. However, the missing context (i.e., the unchanged files) can make RMiner to report an incorrect refactoring type for certain operations. For example, if a method or field is pulled multiple levels up to the inheritance hierarchy and some classes between the source and destination are unchanged, then RMiner will report it as a move, because it cannot detect the inheritance relationship between the source and destination classes due to the missing context. In our oracle, this scenario occurred only once in project cascading [75], where 4 methods were pulled three levels up (`TezNodeStats` → `BaseHadoopNodeStats` → `FlowNodeStats` → `CascadingStats`), but class `FlowNodeStats` remained unchanged in the commit.

Nested refactorings: RMiner is currently unable to detect nested refactoring operations, e.g., Extract Method applied within an extracted method. A notable exception is the detection of Extract and Move Method, which is a sequence of two nested refactoring operations. A possible solution is to include recursively the statements of called methods when performing the statement matching process. In this way, it will be possible to reconstruct the sequence of nested refactoring operations, regardless of the nesting depth.

Unsupported refactorings: In this paper, we present and evaluate the detection rules for 15 refactoring types, while Fowler's catalog [28] includes 72 different types. RMiner already supports the detection of refactorings related to method signature changes (add/remove parameter, change return/parameter type, hide/unhide method), but we didn't validate the detected instances in our dataset due to their large number and time constraints. Moreover, the refactoring types taking place within method bodies, such as rename variable/parameter/field, extract/inline variable, and replace magic number with constant, can be inferred by utilizing the AST node replacements collected through the statement matching process. With these additions, RMiner will be able to support the majority of the most popular refactoring types applied by developers [55, 56].

Oracle bias: Although we did our best effort to reduce bias in the construction of our oracle by incorporating the input of two tools and manual validations by multiple authors, we cannot claim the oracle is unbiased. We tried to incorporate the input of snapshot-based tools, such as Ref-Finder, but the vast majority of commits failed to build due to broken dependencies. Moreover, whenever the inspection of a case was challenging, multiple authors performed an independent validation followed by a thorough discussion. Overall, 3,333 cases were inspected by one validator (out of which 1,411 cases were already assessed as true positives by the authors of [64]), 652 by two validators, and 123 by three validators.

4 RELATED WORK

Weißgerber and Diehl [74] developed the first technique for the detection of local-scope and class-level refactorings in the commit history of CVS repositories. Their approach uses a clone detection tool (CCFinder [43]) to compare the bodies of the code elements that are candidates for refactorings. They manually inspected the commit log messages of two open-source projects to find documented refactorings and compute the recall, and used random sampling to estimate the precision of their approach. Dig et al. [18] developed a tool, RefactoringCrawler, which first performs a fast syntactic analysis (based on techniques from Information Retrieval) to find refactoring candidates, and then a precise semantic analysis (based on similarity of call graphs) to find the actual refactorings. To compute the recall, the authors manually discovered the applied refactorings in three projects by inspecting their release notes, while they inspected the source code to compute precision. Xing and Stroulia [78] developed a tool, JDEvAn [80], which detects and classifies refactorings based on the design-level changes reported by UMLDiff [77]. They evaluated the recall of JDEvAn on two software systems, and found that all documented refactorings were recovered. Prete et al. [46, 62] developed a tool, Ref-Finder, which detects the largest number of refactoring types (63 of 72) from Fowler's catalog [28]. Ref-Finder encodes each program version using logic predicates that describe code elements and their containment relationships, as well as structural dependencies, and encodes refactorings as logic rules. Prete et al. created a set of correct refactorings by running Ref-Finder with a low similarity threshold (σ=0.65) and manually verified them. Then, they computed recall by comparing this set with the results found using a higher threshold (σ=0.85) and computed precision by inspecting a sampled data set. Silva and Valente [65] developed a tool, RefDiff, which takes as input two revisions of a git repository and employs heuristics based on static analysis and code similarity to detect 13 refactoring types. RefDiff represents a source code fragment as a bag of tokens, and computes the similarity of code elements using a variation of the TF-IDF weighting scheme. To determine the similarity threshold values the authors applied a calibration process on a randomly selected set of ten commits from ten different open-source projects, for which the applied refactorings are known and have been confirmed by the project developers themselves [64]. They evaluated the accuracy of their tool using an oracle of seeded refactorings applied by graduate students in 20 open-source projects.

Unlike these previous tools, RMiner neither requires similarity thresholds (that are tedious to calibrate, and might not be generalizable), nor does it require operating on fully built snapshots of software systems, thus it is applicable in many more contexts. Moreover, whereas previous tools have been evaluated against 2-3 projects with a small number of refactoring instances (a notable exception is RefDiff, which was evaluated on 20 projects with 448 seeded refactorings), our oracle is orders of magnitude larger comprising 185 projects and 3,188 true refactoring instances. We use triangulation between multiple sources to create one of the most reliable, comprehensive, and representative oracles to date.

A totally different approach to detect refactorings in real-time is to continuously monitor code changes inside the IDE. BeneFactor [31] and WitchDoctor [27] detect manual refactorings in progress and offer support for completing the remaining changes, whereas CodingTracker [56], GhostFactor [32] and ReviewFactor [34] infer fully completed refactorings. While these tools highlight novel usages of fine-grained code changes inside the IDE, RMiner focuses on changes from commits, thus it can be more broadly applied as it is not dependent on an IDE or text editor.

5 CONCLUSIONS

In this work, we presented the first refactoring detection algorithm that does not rely on code similarity thresholds. We utilize novel techniques, such as *abstraction* and *argumentization* to deal with changes taking place on code statements during refactoring. In addition, we apply syntax-aware replacement of AST nodes when matching two statements to deal with overlapping refactorings (e.g., variable renames), or changes caused by other maintenance activities (e.g., bug fixing). Our evaluation, using one of the most accurate, complete, and representative refactoring oracles to date, showed that our approach achieves very high precision (98%) with a recall that is competitive to the previous state-of-the-art (87%), and has very small computation cost (on median, it takes 58 ms to process a commit). Moreover, RMiner's ability to operate on commits opens new avenues: (1) empirical researchers can create refactoring datasets with high precision from the entire commit history of projects, and study various software evolution phenomena at a fine-grained level, (2) bug-inducing analysis techniques can improve their accuracy utilizing commit-level refactoring information, (3) refactoring operations can be automatically documented at commit-time to provide a more detailed description of the applied changes in the commit message, (4) commit diff visualization can be overlaid with refactoring information to assist code review and evolution comprehension.

REFERENCES

[1] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE ’14)*. ACM, New York, NY, USA, 751–754. https://doi.org/10.1145/2635868.2661674

[2] Tiago L. Alves, Christiaan Ypma, and Joost Visser. 2010. Deriving Metric Thresholds from Benchmark Data. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM ’10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609747

[3] Maurício Aniche, Christoph Treude, Andy Zaidman, Arie van Deursen, and Marco Aurélio Gerosa. 2016. SATT: Tailoring Code Metric Thresholds for Different Software Architectures. In *Proceedings of the IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM ’16)*. 41–50. https://doi.org/10.1109/SCAM.2016.19

[4] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. 2004. An Automatic Approach to identify Class Evolution Discontinuities. In *7th International Workshop on Principles of Software Evolution*. 31–40. https://doi.org/10.1109/IWPSE.2004.1334766

[5] Ittai Balaban, Frank Tip, and Robert M. Fuhrer. 2005. Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 265–279. https://doi.org/10.1145/1094811.1094832

[6] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM ’12)*. 104–113. https://doi.org/10.1109/SCAM.2012.20

[7] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An Experimental Investigation on the Innate Relationship Between Quality and Refactoring. *Journal of Systems and Software* 107 (Sept 2015), 1–14. https://doi.org/10.1016/j.jss.2015.05.024

[8] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. 2011. Comparison of Similarity Metrics for Refactoring Detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR ’11)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/1985441.1985452

[9] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. 2005. Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice* 17, 5 (Sept. 2005), 309–332. https://doi.org/10.1002/smr.v17:5

[10] Apache Cassandra. 2018. Mirror of Apache Cassandra. (2018). https://github.com/apache/cassandra/commit/446e2537895c15b404a74107069a12f3fc404b15#diff-8d5005607847694afae01a22fa8fdbce

[11] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. 2014. On the Impact of Refactoring Operations on Code Quality Metrics. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME ’14)*. IEEE Computer Society, Washington, DC, USA, 456–460. https://doi.org/10.1109/ICSME.2014.73

[12] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering* 43, 7 (July 2017), 641–657. https://doi.org/10.1109/TSE.2016.2616306

[13] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process* 26, 1 (2014), 107–139. https://doi.org/10.1002/smr.1619

[14] Eclipse Deeplearning4J. 2018. Deep Learning for Java, Scala & Clojure on Hadoop & Spark. (2018). https://github.com/deeplearning4j/deeplearning4j/commit/91cdfa1ffd937a4cb01cdc0052874ef7831955e2#diff-367fe3c8ca7846530b2d0562b3b83324R61

[15] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2000. Finding Refactorings via Change Metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA ’00)*. ACM, New York, NY, USA, 166–177. https://doi.org/10.1145/353171.353183

[16] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER ’15)*. 341–350. https://doi.org/10.1109/SANER.2015.7081844

[17] Daniel Dig. 2007. *Automated Upgrading of Component-based Applications*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL, USA.

[18] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP ’06)*. Springer-Verlag, Berlin, Heidelberg, 404–428. https://doi.org/10.1007/11785477_24

[19] Danny Dig, William G. Griswold, Emerson Murphy-Hill, and Max Schäfer. 2014. The Future of Refactoring (Dagstuhl Seminar 14211). *Dagstuhl Reports* 4, 5 (2014), 40–67. https://doi.org/10.4230/DagRep.4.5.40

[20] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (March 2006), 83–107. https://doi.org/10.1002/smr.v18:2

[21] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. 2008. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering* 34, 3 (May 2008), 321–335. https://doi.org/10.1109/TSE.2008.29

[22] Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, and Heitor C. Almeida. 2012. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software* 85, 2 (2012), 244–257. https://doi.org/10.1016/j.jss.2011.05.044 Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.

[23] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (Nov. 2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[24] Francesca Arcelli Fontana, Andrea Caracciolo, and Marco Zanoni. 2012. DPB: A Benchmark for Design Pattern Detection Tools. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR ’12)*. 235–244. https://doi.org/10.1109/CSMR.2012.32

[25] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics (WETSoM ’15)*. IEEE Press, Piscataway, NJ, USA, 44–53. http://dl.acm.org/citation.cfm?id=2821491.2821501

[26] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and Experimenting Machine Learning Techniques for Code Smell Detection. *Empirical Software Engineering* 21, 3 (June 2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

[27] Stephen R. Foster, William G. Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering (ICSE ’12)*. 222–232. https://doi.org/10.1109/ICSE.2012.6227191

[28] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[29] Martin Fowler. 2005. Fluent Interface. (2005). https://martinfowler.com/bliki/FluentInterface.html

[30] Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. 2008. Towards a Benchmark for Evaluating Design Pattern Miner Tools. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR ’08)*. 143–152. https://doi.org/10.1109/CSMR.2008.4493309

[31] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling Manual and Automatic Refactoring. In *Proceedings of the 34th International Conference on Software Engineering (ICSE ’12)*. IEEE Press, Piscataway, NJ, USA, 211–221. http://dl.acm.org/citation.cfm?id=2337223.2337249

[32] Xi Ge and Emerson Murphy-Hill. 2014. Manual Refactoring Changes with Automated Refactoring Validation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE ’14)*. ACM, New York, NY, USA, 1095–1105. https://doi.org/10.1145/2568225.2568280

[33] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. 2014. Towards Refactoring-aware Code Review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE ’14)*. ACM, New York, NY, USA, 99–102. https://doi.org/10.1145/2593702.2593706

[34] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. 2017. Refactoring-Aware Code Review. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC ’17)*. 71–79. https://doi.org/10.1109/VLHCC.2017.8103453

[35] Michael W. Godfrey and Lijie Zou. 2005. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering* 31, 2 (2005), 166–181. https://doi.org/10.1109/TSE.2005.28

[36] Carsten Görg and Peter Weissgerber. 2005. Detecting and visualizing refactorings from software archives. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC ’05)*. 205–214. https://doi.org/10.1109/WPC.2005.18

[37] William G. Griswold. 1992. *Program Restructuring As an Aid to Software Maintenance*. Ph.D. Dissertation. Seattle, WA, USA.

[38] Hazelcast. 2018. Open Source In-Memory Data Grid. (2018). https://github.com/hazelcast/hazelcast/commit/76d7f5e3fe4eb41b383c1d884bc1217b9fa7192e#diff-17f53e9abe4ccd40013a293698fa234dL143

[39] Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. 2017. Empirical Evaluation of Software Maintainability Based on a Manually Validated Refactoring Dataset. *Information and Software Technology* (Nov. 2017). https://doi.org/10.1016/j.infsof.2017.11.012 Accepted, to appear.

[40] Johannes Henkel and Amer Diwan. 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In *27th International Conference on Software Engineering*. 274–283. https://doi.org/10.1145/1062455.1062512

[41] Eclipse Jetty. 2018. Web Container & Clients. (2018). https://github.com/eclipse/jetty.project/commit/1f3be625e62f44d929c01f6574678eea05754474#diff-ff02a462f6cc50644669e515c691229dR580

[42] István Kádár, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '16)*. ACM, New York, NY, USA, Article 10, 4 pages. https://doi.org/10.1145/2972958.2972962

[43] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering* 28, 7 (July 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[44] David Kawrykow and Martin P. Robillard. 2011. Non-essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 351–360. https://doi.org/10.1145/1985793.1985842

[45] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of API-level Refactorings During Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/1985793.1985815

[46] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. RefFinder: A Refactoring Reconstruction Tool Based on Logic Query Templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 371–372. https://doi.org/10.1145/1882291.1882353

[47] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*. IEEE Computer Society, Washington, DC, USA, 81–90. https://doi.org/10.1109/ASE.2006.23

[48] Günter Kniesel and Alexander Binun. 2009. Standing on the shoulders of giants - A data fusion approach to design pattern detection. In *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*. 208–217. https://doi.org/10.1109/ICPC.2009.5090044

[49] Grzegorz Kondrak. 2005. N-gram Similarity and Distance. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE'05)*. Springer-Verlag, Berlin, Heidelberg, 115–126. https://doi.org/10.1007/11575832_13

[50] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10, 8 (1966), 707–710.

[51] Anas Mahmoud and Nan Niu. 2013. Supporting requirements traceability through refactoring. In *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE '13)*. 32–41. https://doi.org/10.1109/RE.2013.6636703

[52] Anas Mahmoud and Nan Niu. 2014. Supporting Requirements to Code Traceability Through Refactoring. *Requirements Engineering* 19, 3 (Sept 2014), 309–329. https://doi.org/10.1007/s00766-013-0197-0

[53] Matin Mansouri. 2018. Refactoring Benchmark. (2018). https://github.com/MatinMan/RefactoringBenchmark

[54] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2017. ARENA: An Approach for the Automated Generation of Release Notes. *IEEE Transactions on Software Engineering* 43, 2 (Feb 2017), 106–127. https://doi.org/10.1109/TSE.2016.2591536

[55] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[56] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[57] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. 2012. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 79–103. https://doi.org/10.1007/978-3-642-31057-7_5

[58] Neo4j. 2018. Graphs for Everyone. (2018). https://github.com/neo4j/neo4j/commit/f6f87f7d5c5d3987db45db7845d221d7abc33146#diff-0694c9de7c6c3b2738144757b771b751L441

[59] Paloma Oliveira, Marco Tulio Valente, and Fernando Paim Lima. 2014. Extracting relative thresholds for source code metrics. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE '14)*. 254–263. https://doi.org/10.1109/CSMR-WCRE.2014.6747177

[60] William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph.D. Dissertation. Champaign, IL, USA.

[61] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship Between Changes and Refactoring. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 176–185. https://doi.org/10.1109/ICPC.2017.38

[62] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*. 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[63] Napol Rachatasumrit and Miryung Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM '12)*. 357–366. https://doi.org/10.1109/ICSM.2012.6405293

[64] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[65] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 269–279. https://doi.org/10.1109/MSR.2017.14

[66] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083147

[67] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. 2013. Comparing Approaches to Analyze Refactoring Activity on Software Repositories. *Journal of Systems and Software* 86, 4 (Apr 2013), 1006–1022. https://doi.org/10.1016/j.jss.2012.10.040

[68] Quinten David Soetens, Javier Pérez, Serge Demeyer, and Andy Zaidman. 2015. Circumventing Refactoring Masking Using Fine-grained Change Recording. In *Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015)*. ACM, New York, NY, USA, 9–18. https://doi.org/10.1145/2804360.2804362

[69] Nikolaos Tsantalis. 2018. RefactoringMiner. (2018). https://github.com/tsantalis/RefactoringMiner

[70] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, and Davood Mazinanian. 2018. Refactoring Oracle. (2018). http://refactoring.encs.concordia.ca/oracle/

[71] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017). https://doi.org/10.1002/smr.1838

[72] Arie van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2001)*. 92–95.

[73] Peter Weissgerber and Stephan Diehl. 2006. Are Refactorings Less Error-prone Than Other Changes?. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 112–118. https://doi.org/10.1145/1137983.1138011

[74] Peter Weissgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*. 231–240. https://doi.org/10.1109/ASE.2006.41

[75] Chris K. Wensel. 2018. Cascading. (2018). https://github.com/cwensel/cascading/commit/f9d3171f5020da5c359cdda28ef05172e858c464

[76] Chadd Williams and Jaime Spacco. 2008. SZZ Revisited: Verifying when Changes Induce Fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems (DEFECTS '08)*. ACM, New York, NY, USA, 32–36. https://doi.org/10.1145/1390817.1390826

[77] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1101908.1101919

[78] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring Detection Based on UMLDiff Change-Facts Queries. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. IEEE Computer Society, Washington, DC, USA, 263–274. https://doi.org/10.1109/WCRE.2006.48

[79] Zhenchang Xing and Eleni Stroulia. 2007. API-Evolution Support with DiffCatchUp. *IEEE Transactions on Software Engineering* 33, 12 (Dec 2007), 818–836. https://doi.org/10.1109/TSE.2007.70747

[80] Zhenchang Xing and Eleni Stroulia. 2008. The JDEvAn Tool Suite in Support of Object-oriented Evolutionary Development. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 951–952. https://doi.org/10.1145/1370175.1370203