2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00132

Automated Software Entity Matching Between Successive Versions

Bo Liu
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
liubo@bit.edu.cn

Hui Liu*
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
liuhui08@bit.edu.cn

Nan Niu
Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, USA
nan.niu@uc.edu

Yuxia Zhang
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
yuxiazh@bit.edu.cn

Guangjie Li
National Innovation Institute of Defense Technology, Beijing, China
liguangjie_er@126.com

Yanjie Jiang
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
yanjiejiang@bit.edu.cn

***Abstract*—Version control systems are widely used to manage the evolution of software applications. However, such version control systems take source code as lines of plain text, and thus they cannot present the evolution of software entities embedded in the source code. To this end, a few approaches have been proposed to match software entities before and after a given commit, known as software entity matching algorithms. However, the accuracy of such algorithms requires further improvement. In this paper, we propose an automated iterative algorithm (called ReMapper) to match software entities between two successive versions. The key insight of ReMapper is that the qualified name, the implementation, and the references of a software entity together can distinguish it from others. It matches software entities iteratively because the mapping depends on the reference-based similarity whereas the reference-based similarity depends on the mapping of entities as well. We evaluated ReMapper on a benchmark consisting of 215 commits from 21 real-world projects. Our evaluation results suggest that ReMapper substantially outperformed the state of the art, reducing the number of mistakes (false positives plus false negatives) substantially by 85.8%. We also evaluated to what extent it may improve the automated refactoring discovery (mining) that relies heavily on automated entity matching. Our evaluation results suggest that it substantially improved the state of the art in refactoring discovery, improving recall by 6.9% and reducing the number of false positives by 72.6%.**

***Index Terms*—Entity Matching, Software Evolution, Software Refactoring, Entity Tracking**

I. Introduction

Software entity matching is the cornerstone of computer-aided comprehension of software evolution [1]. Although version control systems are widely used to manage the evolution of software applications, such version control systems usually take source code as lines of plain text, and thus they cannot comprehend the evolution of software entities embedded in the source code. To this end, a few approaches have been proposed to match software entities before and after a given commit, known as software entity matching algorithms [2]–[4]. Notably, matching software entities not only facilitates developers (maintainers) to comprehend the evolution of software applications, but it also serves as the basis for other automated software engineering tasks, e.g., refactoring discovery [5]–[7] and entity tracking [8]–[10]. Consequently, any mistakes (false positives and false negatives) made by software entity matching algorithms may significantly affect the performance of the subsequent automated software engineering tasks that rely heavily on them.

Entity matching originated from "origin analysis" that was opened up by Godfrey et al. [11]–[13]. Existing entity matching algorithms could be divided into two categories. The first category focuses on the evolution of system designs, and matches design entities without considering their low-level implementation (e.g., method bodies). For example, the well-known `UMLDiff` proposed by Xing and Stroulia [3]. It recovers UML class diagrams from the source code, and maps nodes (entities) between two class diagrams that represent the source code before and after a revision (commit). Such high-level design-based matching is lightweight and thus highly efficient. The second category fully considers the implementation of software entities. For example, `RefactoringMiner` proposed by Tsantalis et al. [7], [14]. It represents the implementation of software entities as abstract syntax trees (ASTs), and computes the similarity between two entities according to the name-based similarity and the statement-based similarity. With such similarities, it maps entities between two successive versions, and leverages a sequence of heuristics to discover software refactorings (e.g., rename class and extract method refactorings) based on the mapping. Although evaluation results suggest that existing entity matching algorithms are often accurate [14], the number of mismatched entities remains non-negligible (see Section III for details).

To this end, in this paper, we propose a novel matching algorithm, called `ReMapper`, to improve the accuracy in

* Corresponding author

mapping entities between two successive versions. The key insight of ReMapper is that the qualified name, the implementation, and the references of a software entity together can distinguish it from others. Consequently, it computes the name-based similarity, the implementation (AST) based similarity, and the reference-based similarity, respectively. Note that the references of an entity (i.e., entities accessing it) represent the external attributes of the entity, and thus they often survive various inner changes to the entity, e.g., changes to its implementation and changes to the entity name. To the best of our knowledge, ReMapper is the first entity matching algorithm that takes full advantage of all such three categories of similarities. Although design-based matching algorithms like UMLDiff leverage the references of entities, the references are often simply taken as connections between entities, (i.e., edges in class diagrams), and they are not distinguished from other connections (e.g., composition). Besides, design-based matching algorithms do not exploit the implementation-based similarity as ReMapper does. Although more recent matching algorithms like RefactoringMiner exploit the implementation-based similarity, they do not take advantage of the reference-based similarity. Notably, ReMapper is iterative because the mapping depends on the reference-based similarity whereas the reference-based similarity depends on the mapping of entities as well. The iterative matching differs from existing ones (e.g., the iterative matching in UMLDiff) in that it not only matches unmatched entities, but also re-matches entities that have already been temporarily matched in the previous iterations. The iteration terminates until the mapping becomes stable, i.e., further iterations would not change the mapping anymore.

We evaluated ReMapper on a benchmark consisting of 215 commits from 21 real-world projects. The evaluation results suggest that ReMapper substantially outperformed the state of the art in entity matching, reducing the number of mistakes (false positives plus false negatives) substantially by 85.8%. Since entity matching algorithms serve as the cornerstone of automated discovery of software refactorings [6], [7], we also evaluated to what extent ReMapper may improve the automated discovery of software refactorings. We replaced the entity matching algorithm in RefactoringMiner (that represents the state of the art in automated discovery of refactorings) but kept the other parts (i.e., the heuristics-based rules for detecting refactorings) intact. Our evaluation results suggest that ReMapper substantially improved the state of the art in automated discovery of software refactorings. It improved the recall in refactoring discovery by 6.9% and reduced the number of false positives by 72.6%.

In this paper, we make the following contributions:

- We propose a novel entity matching algorithm (called ReMapper) that takes full advantage of name-based similarity, implementation-based similarity, and reference-based similarity.
- We construct a new dataset for entity matching (called *em-Dataset*), validated by multiple developers [15]. Our evaluation on the dataset validates that ReMapper can substantially reduce the number of false positives and false negatives.
- We boost the automated discovery of refactorings by the proposed entity matching algorithm, resulting in a new refactoring miner (called RefactoringExtractor).

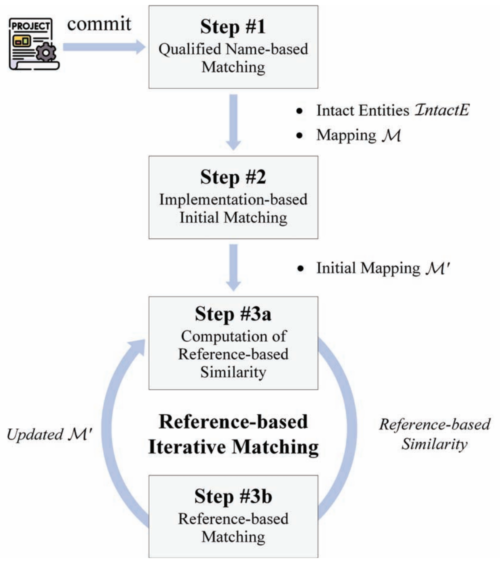

PROJECT
commit
Step #1
Qualified Name-based Matching
Intact Entities $\mathcal{I}ntactE$
Mapping $\mathcal{M}$
Step #2
Implementation-based Initial Matching
Initial Mapping $\mathcal{M}'$
Step #3a
Computation of Reference-based Similarity
Updated $\mathcal{M}'$
Reference-based Iterative Matching
Reference-based Similarity
Step #3b
Reference-based Matching

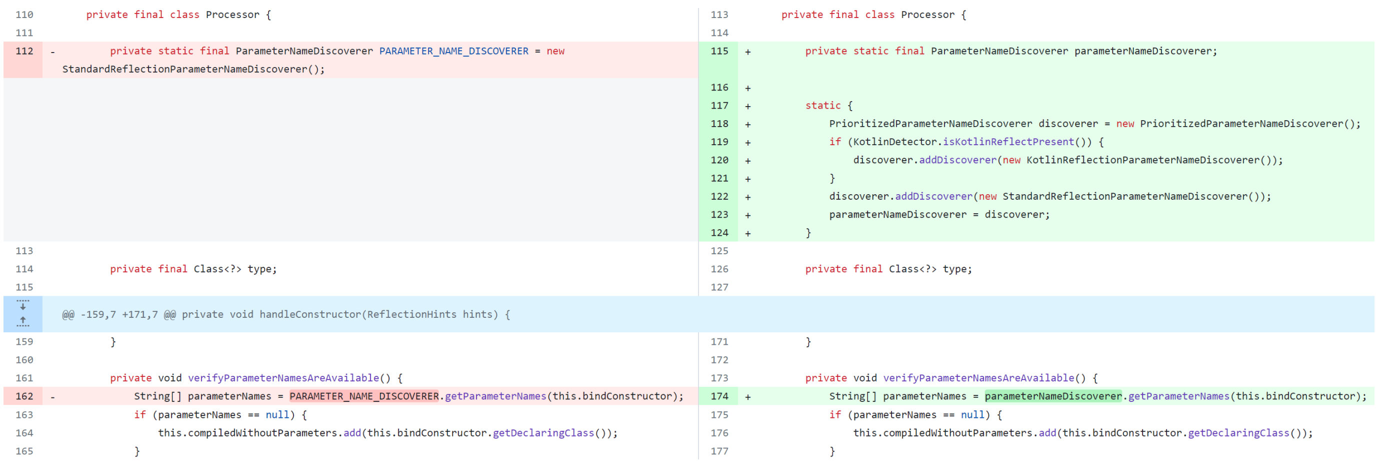
Fig. 1. Overview of ReMapper

II. Approach

A. Overview

Fig. 1 presents an overview of ReMapper. It takes as input a Java project repository and a commit ID, and returns a mapping between software entities before and after the commit. Given the repository and commit ID, ReMapper works as follows:

- First, it retrieves a list of files involved in the commit, i.e., files that have been removed, modified, or newly added by the commit. From the retrieved files, it retrieves all to-be-matched Java entities, and identifies intact entities (noted as $\mathcal{I}ntactE$) that have not been changed by the given commit. It also leverages a qualified name-based matching algorithm to match other entities, and the resulting mapping is noted as $\mathcal{M}$.
- Second, it leverages an implementation-based matching algorithm to initially match entities that have not yet been matched so far, and the resulting initial mapping is noted as $\mathcal{M}'$.
- Third, it leverages a reference-based iterative matching algorithm to match (or re-match) entities (excepted for intact entities in $\mathcal{I}ntactE$ and matched entities in $\mathcal{M}$). The iteration is composed of two steps. In the first step, it computes (or re-computes) the reference-based similarity between entities based on the initial mapping $\mathcal{M}'$. In the second step, it matches (or re-matches) entities according to the reference-based similarity, and updates mapping $\mathcal{M}'$. The iteration terminates when $\mathcal{M}'$ becomes stable.



```
    private final class Processor {

-       private static final ParameterNameDiscoverer PARAMETER_NAME_DISCOVERER = new
 StandardReflectionParameterNameDiscoverer();

        private final Class<?> type;

@@ -159,7 +171,7 @@ private void handleConstructor(ReflectionHints hints) {
        }

        private void verifyParameterNamesAreAvailable() {
-           String[] parameterNames = PARAMETER_NAME_DISCOVERER.getParameterNames(this.bindConstructor);
            if (parameterNames == null) {
                this.compiledWithoutParameters.add(this.bindConstructor.getDeclaringClass());
            }
```

```
    private final class Processor {

+       private static final ParameterNameDiscoverer parameterNameDiscoverer;
+
+       static {
+           PrioritizedParameterNameDiscoverer discoverer = new PrioritizedParameterNameDiscoverer();
+           if (KotlinDetector.isKotlinReflectPresent()) {
+               discoverer.addDiscoverer(new KotlinReflectionParameterNameDiscoverer());
+           }
+           discoverer.addDiscoverer(new StandardReflectionParameterNameDiscoverer());
+           parameterNameDiscoverer = discoverer;
+       }

        private final Class<?> type;

        }

        private void verifyParameterNamesAreAvailable() {
+           String[] parameterNames = parameterNameDiscoverer.getParameterNames(this.bindConstructor);
            if (parameterNames == null) {
                this.compiledWithoutParameters.add(this.bindConstructor.getDeclaringClass());
            }
```

Fig. 2. An Example of Code Changes from Commit b9e57c7 in spring-boot

Key steps of the proposed approach are explained in details in the following sections. Moreover, we leverage a real-world example from project spring-boot [16], depicted in Fig. 2, to illustrate how each step of the proposed approach works.

B. *Qualified Name-based Matching*

Given a Java project repository and a commit ID, we retrieve source code files that have been added, removed, or modified by the commit. Notably, entity matching algorithms usually ignore intact files and non-source code files [7], [14]. All newly added files are added to set $\mathcal{AD}$, and all removed files are added to set $\mathcal{RD}$. The modified files are recorded by a set $\mathcal{MD}$. Each item in $\mathcal{MD}$ represents a pair of documents $<d, d'>$ where d and d' represent the old version (V_n) and the new version (V_{n+1}) of the same document that was modified by the given commit.

From each tuple $<d, d'> \in \mathcal{MD}$, we retrieve all Java entities declared in d and d', respectively. All entities declared in d are added to a set (noted as $oldE$), and entities declared in d' are added to another set (noted as $newE$). Notably, the proposed approach only matches methods, fields, classes, interfaces, enums, @interface (annotation type declarations), initializers, enum constants, and annotation members. It does not match Java entities within methods (e.g., parameters, variables, and statements), and thus such entities are ignored. Notably, the number of entity types handled by the proposed approach is significantly greater than that handled by existing approaches. For example, the well-known software entity matching algorithm [4] matches method-level software entities only.

We identify intact entities in d by matching entities in $oldE$ against entities in $newE$. Entity $oe \in oldE$ matches entity $ne \in newE$ if and only if:

- oe and ne are of the same entity type;
- oe and ne share identical fully qualified name; and
- oe and ne share identical implementation.

The implementation of an entity depends on its entity type. For example, the implementation of a method includes the signature and its method body whereas the implementation of a field includes its complete declaration (including its initialization). We retrieve the implementation of an entity by using `ASTNode.toString()` [17] on it. The comparison of entities' implementations is a pure text-based comparison and thus any changes in the implementation will prevent the matching of entities in this step. If oe and ne match, we add a tuple $<oe, ne>$ to set $\mathcal{I}ntactE$, and remove oe from $oldE$ and ne from $newE$. In Fig. 2, field `type` and method `handleConstructor` are added to $\mathcal{I}ntactE$ because their fully qualified names and implementations keep intact (without any changes) in the given commit.

For the remaining entities in $oldE$ and $newE$ that we fail to match in the preceding step, we try to match them again, ignoring their implementation. That is, entity $oe \in oldE$ matches entity $ne \in newE$ if and only if:

- oe and ne are of the same entity type;
- oe and ne share identical fully qualified name;
- oe and ne share the same signature if they are methods; and
- oe and ne share the same data type if they are fields or annotation members.

This matching step may help identify entities whose implementation has been modified by the given commit but their identity (including fully qualified names, method signatures, and data types) is kept intact. If oe and ne match, we add a tuple $<oe, ne>$ to set $\mathcal{M}$, and remove oe from $oldE$ and ne from $newE$. In Fig. 2, class `Processor` and method `verifyParameterNamesAreAvailable` are added to $\mathcal{M}$ because their identities (without considering their internal implementation) keep intact in the given commit.

C. *Implementation-based Initial Matching*

Entities to be matched are classified into two categories. The first category is composed of Java entities that have the potential to contain other to-be-matched entities. For example, classes have the potential to contain methods and fields that should be matched as well. The other category is composed of Java entities that cannot contain other to-be-matched entities.

```
Algorithm 1: Initial Matching of Atomic Entities
Input  : $2bmE_n$ and $2bmE_{n+1}$
Output: $\mathcal{M}'$
1  $\mathcal{L} \leftarrow \varnothing$
2  foreach $oe \in 2bmE_n.atomicEntities$ do
3      foreach $ne \in 2bmE_{n+1}.atomicEntities$ do
4          if $oe.type == ne.type$ then
5              $implSim(oe, ne) =$ compImplSim($oe, ne$)
6              if $implSim(oe, ne) \geq 0.5$ then
7                  $\mathcal{L}$.add(<$oe, ne, implSim(oe, ne)$>)
8              end
9          end
10     end
11 end
12 sortBySim($\mathcal{L}$)  // in descending order
13 return similarityBasedMatching($\mathcal{L}$)
14 Function similarityBasedMatching($\mathcal{L}$)
15     $\mathcal{M}' \leftarrow \varnothing$, $\mathcal{O} \leftarrow \varnothing$, $\mathcal{N} \leftarrow \varnothing$
16     for $int\ i = 0;\ i < \mathcal{L}.length;\ i{+}{+}$ do
17         if $\mathcal{L}[i].oe \notin \mathcal{O}$ and $\mathcal{L}[i].ne \notin \mathcal{N}$ then
18             $\mathcal{M}'$.add(<$\mathcal{L}[i].oe, \mathcal{L}[i].ne$>)
19             $\mathcal{O}$.add($\mathcal{L}[i].oe$)
20             $\mathcal{N}$.add($\mathcal{L}[i].ne$)
21         end
22     end
23     return $\mathcal{M}'$
24 end
```

For example, methods belong to this category because the proposed approach does not match Java entities within a method (like parameters, variables, and statements). For the sake of simplicity, we call the two categories *compound entities* and *atomic entities*, respectively.

All to-be-matched entities (except for intact entities in $\mathcal{I}ntactE$ and matched entities in $\mathcal{M}$) in V_n and V_{n+1} are noted as $2bmE_n$ and $2bmE_{n+1}$, respectively. We initially match atomic entities by Algorithm 1. For each atomic entity $oe \in 2bmE_n$, we compare it against each atomic entity $ne \in 2bmE_{n+1}$. If they are of different entity types, they are deemed unmatched. For example, we cannot map a field in the old version to a method in the new version. If they are of the same entity type, on Line 5 in Algorithm 1, we retrieve their corresponding ASTs and compute the implementation-based similarity (i.e., $implSim(oe, ne)$) between them. We add a triple $<oe, ne, implSim(oe, ne)>$ to list $\mathcal{L}$ (Line 7) if and only if:

- oe and ne are of the same entity type; and
- $implSim(oe, ne) \geq 0.5$ (i.e., more than half of their AST nodes are common).

The triple suggests that oe and ne have the potential to be matched. We compute the implementation-based similarity of two atomic entities with *dice coefficient* [18] that is widely used to measure the similarity between two ASTs [19]–[21]. The implementation-based similarity is computed as follows:

$$\begin{aligned} implSim(e_1, e_2) &= dice(t_1, t_2) \\ &= 2 \times \frac{|nodes(t_1) \cap nodes(t_2)|}{|nodes(t_1)| + |nodes(t_2)|}, \end{aligned} \quad (1)$$

where t_1 and t_2 are the ASTs associated with entities e_1 and e_2, respectively. $nodes(t_1)$ and $nodes(t_2)$ denote the AST nodes in t_1 and t_2, respectively. $nodes(t_1) \cap nodes(t_2)$ denotes the common nodes between the two sets. Nodes $nd_i \in nodes(t_1)$ and $nd_j \in nodes(t_2)$ represent a common node if they share the same node type (i.e., $nd_i.getNodeType() == nd_j.getNodeType()$) and equivalent value (i.e., $nd_i.toString() == nd_j.toString()$).

We sort $\mathcal{L}$ by implementation-based similarity in descending order (Line 12). Function similarityBasedMatching (Lines 14-24) matches atomic entities to maximize the similarity between matched entities whereas each entity matches no more than one entity.

The initial matching of compound entities is exactly the same as the matching of atomic entities except that the similarity of compound entities is computed as follows:

$$implSim(e_1, e_2) = 2 \times \frac{|subE(e_1) \cap subE(e_2)|}{|subE(e_1)| + |subE(e_2)|}, \quad (2)$$

where $subE(e)$ denotes all entities contained in e. For example, if e is a class, $subE(e)$ is composed of all methods, fields, and inner types (e.g., inner classes and inner interfaces) declared within e. $subE(e_1) \cap subE(e_2)$ denotes the common entities within $subE(e_1)$ and $subE(e_2)$. The two entities $e_1 \in subE(e_1)$ and $e_2 \in subE(e_2)$ represent a single common entity if and only if e_1 and e_2 have been matched (or temporarily matched), i.e., $<e_1, e_2> \in (\mathcal{I}ntactE \cup \mathcal{M} \cup \mathcal{M}')$. Because the similarity of two compound entities as defined in Equation 2 depends on the matching of their sub-entities, the initial matching should take a bottom-up strategy.

In the example depicted in Fig. 2, two atomic entities (field PARAMETER_NAME_DISCOVERER on Line 112 in V_n and field parameterNameDiscoverer on Line 115 in V_{n+1}) are initially matched because they share more than half of their AST nodes, and they are added to $\mathcal{M}'$.

D. Reference-based Iterative Matching

As suggested by the overview in Fig. 1, reference-based iterative matching is a loop containing two iterative steps, i.e., computation of reference-based similarity and reference-based matching. We explain the loop (iteration) first, and then the two key steps within the loop.

1) Iterative Matching: The proposed approach leverages Algorithm 2 to conduct the reference-based iterative matching. In each iteration, the approach first (Lines 2-12) computes/updates the reference-based similarity based on the temporary mapping (i.e., $\mathcal{M}'$) generated by the previous iteration. With the updated similarity, the approach (Lines 13-19) rematches temporarily matched entities (as well as unmatched ones), and updates $\mathcal{M}'$ if it is necessary. The algorithm terminates when the mapping $\mathcal{M}'$ survives the latest iteration, i.e., the iteration does not bring any changes to $\mathcal{M}'$.

Algorithm 2: Reference-based Iterative Matching

```
Input  : $2bmE_n$, $2bmE_{n+1}$, $\mathcal{I}ntactE$, $\mathcal{M}$, and $\mathcal{M}'$
Output: $\mathcal{M}'$
1  while true do
2      $\mathcal{L} \leftarrow \varnothing$
3      foreach $e_1 \in 2bmE_n$ do
4          foreach $e_2 \in 2bmE_{n+1}$ do
5              if type-compatible($e_1$, $e_2$) then
6                  $sim(e_1, e_2) =$
                     compSim($e_1, e_2, \mathcal{I}ntactE, \mathcal{M}, \mathcal{M}'$)
7                  if $sim(e_1, e_2) \geq 0.5$ then
8                      $\mathcal{L}$.add(<$e_1, e_2, sim(e_1, e_2)$>)
9                  end
10             end
11         end
12     end
13     sortBySime($\mathcal{L}$)  // in descending order
14     $\mathcal{S} \leftarrow$ similarityBasedMatching($\mathcal{L}$)
15     if $\mathcal{M}' == \mathcal{S}$ then  // no change on mapping
16         return $\mathcal{M}'$  // terminate loop
17     else
18         $\mathcal{M}' \leftarrow \mathcal{S}$
19     end
20 end
```

2) Computation of Reference-based Similarity: For each pair of entities $e_1 \in 2bmE_n$ and $e_2 \in 2bmE_{n+1}$, we compute their reference similarity as follows:

$$refSim(e_1, e_2) = \frac{2 \times |callers(e_1) \cap callers(e_2)|}{|callers(e_1)| + |callers(e_2)|}, \quad (3)$$

where $callers(e)$ denotes Java entities that access e. $callers(e_1) \cap callers(e_2)$ denotes entities accessing both e_1 and e_2. Notably, e_i and e_j represent an entity accessing both e_1 and e_2 if and only if $e_i \in callers(e_1)$, $e_j \in callers(e_2)$, and e_i has been matched (or initially matched) to e_j (i.e., $<e_i, e_j> \in (\mathcal{I}ntactE \cup \mathcal{M} \cup \mathcal{M}')$).

The reference-based similarity between e_i and e_j is the average of their reference similarity and implementation-based similarity:

$$sim(e_1, e_2) = \frac{refSim(e_1, e_2) + implSim(e_1, e_2)}{2} \quad (4)$$

3) Reference-based Matching: To match entities according to the reference-based similarity defined in Equation 4, Algorithm 2 (Lines 3-12) computes the reference-based similarity between each type-compatible entity pair $<e_1, e_2>$ (Line 5). If two entities are of the same entity type, they are surely type compatible. Following Tsantalis et al. [7], [14], we also take classes, interfaces, and enums as type-compatible to each other because developers often switch among such types, e.g., changing an interface into a class. If we do not try to match interfaces against classes, we can never identify such refactorings. The same is true for enums.

If the reference-based similarity between e_1 and e_2 is greater than or equal to a preset threshold (empirically set to 0.5), we add the item $<e_1, e_2, sim(e_1, e_2)>$ to list $\mathcal{L}$ (Line 8), suggesting that they have the potential to match each other. We sort items in $\mathcal{L}$ by their reference-based similarity in descending order (Line 13). In case two items have equivalent reference-based similarity, we sort them according to the name-based similarity in descending order. The name-based similarity between e_1 and e_2 is the *n-gram* similarity [22] of their fully qualified names:

$$nameSim(e_1, e_2) = \frac{2 \times |2gs(qn(e_1)) \cap 2gs(qn(e_2))|}{|2gs(qn(e_1))| + |2gs(qn(e_2))|}, \quad (5)$$

where $qn(e)$ is the fully qualified name of e, and $2gs(qn(e))$ denotes the set of 2-grams within $qn(e)$.

After sorting $\mathcal{L}$, the proposed approach rematches entities by using function `similarityBasedMatching` on Line 14. Notably, this function has been defined in Algorithm 1.

In the example depicted in Fig. 2, after one iteration of matching, field `PARAMETER_NAME_DISCOVERER` on Line 112 in V_n and field `parameterNameDiscoverer` on Line 115 in V_{n+1} are matched and added to the temporary mapping ($\mathcal{S}$) because they are accessed by the same method `verifyParameterNamesAreAvailable`. Then, the algorithm terminates and returns all matching pairs because $\mathcal{S}$ equals the previous mapping ($\mathcal{M}'$).

III. Evaluation

The evaluation investigates the following research questions:

RQ1. Can `ReMapper` improve the state of the art in matching software entities between successive versions?

RQ2. How does the patch size influence the performance of `ReMapper`?

RQ3. Is `ReMapper` scalable?

To answer RQ1, we compared `ReMapper` against `RefactoringMiner` [14]. Although it was originally designed for refactoring discovery, its novel entity matching algorithm serves as its cornerstone (and its major contribution). We employed its entity matching algorithm only as the baseline. Pure entity matching algorithms were not employed for comparison because they were rather old (published more than fifteen years ago), and more recent advances in this line are often closely coupled to specific tasks, like discovery of refactorings [14], [23] and extraction of fine-grained code changes [21]. `RefactoringMiner` [14] is the latest advance in this line. For the sake of integrity, however, we also compared `ReMapper` against the pure entity matching algorithm `JDiff` [24], and published the results online [25].

RQ2 concerns whether and to what extent the performance of `ReMapper` may be influenced by the patch size (i.e., in a commit, the number of deleted or modified lines plus the number of added lines). It is likely that the larger the change (patch) is, the more challenging the entity matching is. By answering RQ2, we intend to examine the hypothesis. RQ3 concerns the scalability of the proposed approach, i.e., whether it works efficiently on large projects with large-scale patches.

A. *Setup*

1) Subjects: We reused 20 popular Java projects that were chosen by Grund et al. [9] as the subject projects. All of the projects contain rich evolution histories and the number of commits per project varies from 2,506 to 397,649 with a median of 14,748. They also cover a range of domains, including code analysis, unit testing, search engine, distribution framework, and web server. The diversity may help reduce the potential bias in the evaluation.

To further explore the performance of our approach in industrial applications, we selected a closed-source Java project. This is a software refactoring engine developed by one of the world's leading IT companies, Huawei. It is designed to automatically detect code smells in software applications and to provide tailored solutions for each identified issue. This project contains 298 commits submitted during 2021 and 2023. Therefore, in total we selected 21 subject projects, including 20 open-source projects and one closed-source project.

2) Process: We applied the evaluated approaches independently to the selected subject projects, starting from the latest commit. On each commit, if the evaluated approaches generated identical results (i.e., for all of the changed entities, the matching relationship created by `ReMapper` and the selected baseline approach was exactly the same) or neither reported any matching relationship (e.g., the commit only modified documents not associated with source code files), we simply dropped the commit and turned to the next one. In contrast, if the evaluated approaches generated inconsistent results (i.e., non-identical results), we collected the commit as a conflict-generating commit, and requested three experienced developers to manually inspect all the matching relations in each conflict-generating commit. Therefore, we could count the number of true positives, the number of false positives, and the number of false negatives for each evaluated approach, thereby computing its precision and recall. All of the participants were required to have Java background. They had a median of 8.5 years of programming experience, 3.5 years working as professional software developers, and 6 years experience with version control systems. In case of inconsistent manual inspection (112 matching relations in total), the participants were requested to discuss together and reach an agreement. Notably, the three developers achieved a high consistency with a Fleiss' kappa coefficient [26] of 0.85. To control the cost of manual validation, we only selected the most recent 10 conflict-generating commits per open-source project.

For the closed-source project, we requested the original developers of the source code instead of the third-party participants to inspect the matching relations. The developers knew the source code and its evolution well, and thus their inspection could be more accurate and much more cost-effective. To this end, we requested the original developers to validate all (15 in total) conflict-generating commits of the project.

Based on the manual inspection, we computed the performance metrics for the evaluated approaches, i.e., the number of false positives, the number of false negatives, precision, and recall. We also manually analyzed some incorrect matching pairs to gain insights into the reasons of the mistakes.

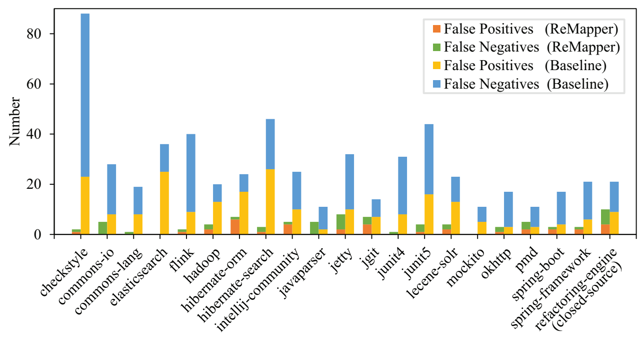


Fig. 3. Number of Mistakes per Project

B. *Improving the State of the Art*

To measure the performance of our approach in entity matching, we focused on the number of false positives (#FP) and the number of false negatives (#FN). A false positive is a pair of entities $<e_1, e_2>$ that is reported as *matched* whereas manual checking marked them as *mismatched*. In contrast, a false negative is a pair of entities that is manually marked as *matched* but the matching algorithm does not report it as a matched entity pair. #MST = #FP + #FN (i.e., the number of false positives plus the number of false negatives) represents how frequently the evaluated approaches make mistakes. The key concern of RQ1 is to what extent `ReMapper` can reduce the frequency of mistakes (i.e., #MST). Besides #MST, #FP, and #FN, we also computed the precision and recall where precision measures how often the reported pairs were manually confirmed, and recall measures how often matched pairs were retrieved by the evaluated algorithms.

We ignored entities whose scope is confined within a single method, e.g., parameters, variables, and statements, because they are out of the scope of `ReMapper`. The selected baseline approach ignores "annotation type" and "annotation member". To favor the baseline, we ignored all entities of these two categories while accounting for false positives and false negatives for the baseline approach (i.e., assuming that it makes no mistakes at all in matching such entities).

Our evaluation results are presented in Fig. 3. The horizontal axis presents the involved projects where the last one ("refactoring-engine") is the closed-source project whereas others are open-source projects. The vertical axis presents the number of false positives and false negatives as well their sum (i.e., #FP, #FN, and #MST) on each subject project.

From Fig. 3, we observe that compared against the state of the art, `ReMapper` substantially reduced the frequency of mistakes: The total number of mistakes (i.e., #MST) was reduced from 579 to 82, with a substantial reduction of 85.8%=(579-82)/579. On average, the number of false positives per project was reduced by 84.1%=(10.7-1.7)/10.7 and the number of false negatives per project was reduced

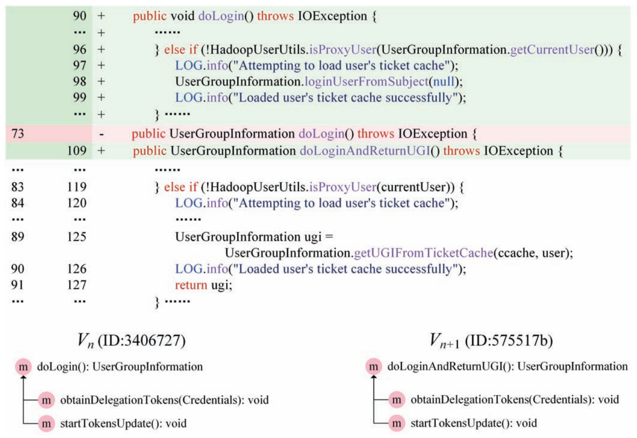
TABLE I
PERFORMANCE PER ENTITY TYPE

Entity Type	Approaches	#MST	#FP	#FN	Precision	Recall
Class	ReMapper	0	0	0	100%	100%
	Baseline	51	10	41	99.33%	97.31%
	Δ Improvement	51	10	41	0.67%	2.69%
Interface	ReMapper	0	0	0	100%	100%
	Baseline	1	0	1	100%	99.01%
	Δ Improvement	1	0	1	0%	0.99%
Enum	ReMapper	0	0	0	100%	100%
	Baseline	0	0	0	100%	100%
	Δ Improvement	0	0	0	0%	0%
Annotation Type	ReMapper	0	0	0	100%	100%
	Baseline			N/A		
	Δ Improvement					
Initializer	ReMapper	1	0	1	100%	88.89%
	Baseline	8	6	2	53.85%	77.78%
	Δ Improvement	7	6	1	46.15%	11.11%
Field	ReMapper	16	8	8	98.3%	98.3%
	Baseline	95	31	64	92.91%	86.38%
	Δ Improvement	79	23	56	5.39%	11.92%
Method	ReMapper	64	27	37	99.21%	98.92%
	Baseline	421	177	244	94.74%	92.89%
	Δ Improvement	357	150	207	4.47%	6.03%
Enum Constant	ReMapper	1	0	1	100%	98.31%
	Baseline	3	1	2	98.28%	96.61%
	Δ Improvement	2	1	1	1.72%	1.7%
Annotation Member	ReMapper	0	0	0	100%	100%
	Baseline			N/A		
	Δ Improvement					
Total	ReMapper	82	35	47	99.38%	99.16%
	Baseline	579	225	354	95.9%	93.7%
	Δ Improvement	497	190	307	3.48%	5.46%

by 87%=(16.9-2.2)/16.9. We performed a significance test to validate whether there is a statistically significant difference between the total number of mistakes caused by the two approaches. Our evaluation results (p-value [27]=2.57E-7 and effect size of Cohen's d [28]=1.96) confirmed that the reduction in #MST was statistically significant. Similar significance tests confirmed that the improvement in precision (p-value=1.87E-6 and Cohen's d=1.76) and recall (p-value=1.3E-5 and Cohen's d=1.57) was statistically significant.

We also observe from Fig. 3 that the frequency of mistakes (#MST) was reduced on each of the involved 21 projects (both open-source projects and closed-source projects), which may suggest that `ReMapper` can stably improve the state of the art regardless of the involved projects. As mentioned in Section III-A1, the 21 subject projects were from different domains. Consequently, the consistent performance on all such diverse projects may suggest that `ReMapper` works well.

We further investigated their performance on matching different categories of software entities, e.g., "classes", "interfaces" and "methods". The evaluation results are presented in Table I. From this table, we observe that `ReMapper` outperforms the baseline on all of the involved entity types except for "enum" where neither of them made any mistakes.



```
        90  +   public void doLogin() throws IOException {
        ··· +       ······
        96  +       } else if (!HadoopUserUtils.isProxyUser(UserGroupInformation.getCurrentUser())) {
        97  +           LOG.info("Attempting to load user's ticket cache");
        98  +           UserGroupInformation.loginUserFromSubject(null);
        99  +           LOG.info("Loaded user's ticket cache successfully");
        ··· +       } ······
73          -   public UserGroupInformation doLogin() throws IOException {
        109 +   public UserGroupInformation doLoginAndReturnUGI() throws IOException {
···     ···         ······
83      119         } else if (!HadoopUserUtils.isProxyUser(currentUser)) {
84      120             LOG.info("Attempting to load user's ticket cache");
···     ···             ······
89      125             UserGroupInformation ugi =
                                UserGroupInformation.getUGIFromTicketCache(ccache, user);
90      126             LOG.info("Loaded user's ticket cache successfully");
91      127             return ugi;
···     ···         } ······
```

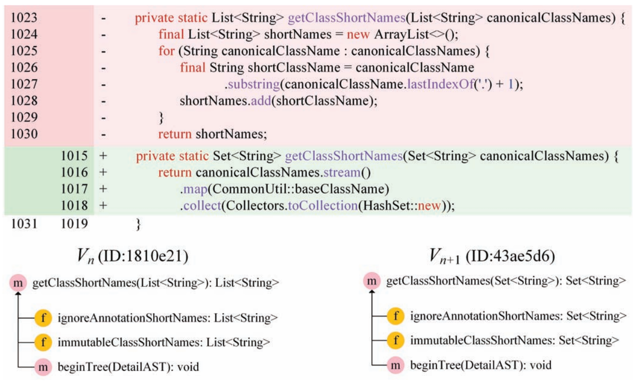
V_n (ID:3406727)
V_{n+1} (ID:575517b)
m doLogin(): UserGroupInformation
m obtainDelegationTokens(Credentials): void
m startTokensUpdate(): void
m doLoginAndReturnUGI(): UserGroupInformation
m obtainDelegationTokens(Credentials): void
m startTokensUpdate(): void

Fig. 4. False Positive Avoided by ReMapper

We also notice that `ReMapper` resulted in high precision and recall on all of the entity types. The minimal precision (on "field") was 98.3%, and the minimal recall (on "initializer") was 88.89%. It may suggest that `ReMapper` worked well on all entity types. The evaluated approaches reported the highest numbers of mistakes in matching entities of "method", followed by "field". It is reasonable because the numbers of involved methods and fields are significantly greater than those of classes, interfaces, enums, and annotation types.

We leverage a real-world example from project flink [29] in Fig. 4 to illustrate how `ReMapper` avoided some false positives. The method `doLogin` on Line 73 in V_n was renamed as `doLoginAndReturnUGI` on Line 109 in V_{n+1}, and a new method named `doLogin` on Line 90 in V_{n+1} was added. However, the baseline approach mapped method `doLogin` in V_n by mistake to method `doLogin` in V_{n+1} because their method signatures were identical except for the return types. In contrast, our approach correctly mapped method `doLogin` in V_n to method `doLoginAndReturnUGI` in V_{n+1} because 1) their method bodies were exactly the same and 2) they were called by the same software entities, i.e., *startTokensUpdate* and *obtainDelegeationTokens*. It did not map `doLogin` in V_n to `doLogin` in V_{n+1} although their signatures were highly similar because they were significantly different concerning their references (callers) and method bodies.

We leverage another real-world example from project checkstyle [30] in Fig. 5 to illustrate how `ReMapper` successfully retrieved the pairs (i.e., false negatives) missed by the baseline approach. The baseline approach failed to match method `getClassShortNames` on Line 1023 in V_n with method `getClassShortNames` on Line 1015 in V_{n+1}. The change between these two methods involves a complex refactoring (*replace loop with pipeline* [31], [32]). The refactoring made the two method bodies dissimilar, which causes the failure of the baseline approach that heavily depends on the statement-level similarity of method bodies. Our approach succeeded because it leverages the reference-based similarity: The two methods are called by exactly the same Java entities.



```diff
-   private static List<String> getClassShortNames(List<String> canonicalClassNames) {
-       final List<String> shortNames = new ArrayList<>();
-       for (String canonicalClassName : canonicalClassNames) {
-           final String shortClassName = canonicalClassName
-                   .substring(canonicalClassName.lastIndexOf('.') + 1);
-           shortNames.add(shortClassName);
-       }
-       return shortNames;
+   private static Set<String> getClassShortNames(Set<String> canonicalClassNames) {
+       return canonicalClassNames.stream()
+           .map(CommonUtil::baseClassName)
+           .collect(Collectors.toCollection(HashSet::new));
    }
```

V_n (ID:1810e21)
m getClassShortNames(List<String>): List<String>
f ignoreAnnotationShortNames: List<String>
f immutableClassShortNames: List<String>
m beginTree(DetailAST): void
V_{n+1} (ID:43ae5d6)
m getClassShortNames(Set<String>): Set<String>
f ignoreAnnotationShortNames: Set<String>
f immutableClassShortNames: Set<String>
m beginTree(DetailAST): void

Fig. 5. True Positive Missed by Baseline

TABLE II
IMPACT OF PATCH SIZE

Patch Size	ReMapper			Baseline		
	#MST	Precision	Recall	#MST	Precsion	Recall
Q1 [4, 38, 72]	0.11	99.51%	99.16%	1.84	90.14%	81.59%
Q2 [73, 118, 156]	0.32	98.01%	98.53%	2.02	94.86%	92.76%
Q3 [157, 244, 364]	0.37	98.75%	98.86%	3	95.61%	91.13%
Q4 [376, 833, 7154]	0.74	99.03%	98.7%	3.94	94.87%	93.58%

C. Impact of Patch Size

We investigated the impact of patch size on the performance of the evaluated approaches, and the evaluation results are presented in Table II. Notably, to reduce the randomness, we partitioned the involved commits into four equally sized groups according to their patch size: Q1, Q2, Q3, and Q4 where Q1 was composed of the smallest 25% commits and Q4 was composed of the largest 25% commits. The first column of Table II specifies the data groups along with the minimal size, average size, and maximal size of the patches within each data group. Columns 2 to 4 present the performance of ReMapper on the given data group. Columns 5 to 7 present the performance of the baseline approach on the given data group. Notably, the performance metrics are the averages. For example, #MST is the average number of mistakes per commit in the given group.

We observe from Table II that the evaluated approaches made more mistakes with the increase in patch size. This is reasonable because the larger the patches are, the more impacted entities there are (and thus the more chance to make mistakes). From Table II, we also observe that ReMapper outperformed the baseline on all categories of the patches, i.e., Q1, Q2, Q3, and Q4. It may suggest that ReMapper can stably improve the state of the art in entity matching regardless of the patch size in the commits. However, as shown in Table II, the patch size has a substantially greater impact on the performance of the baseline approach than that on ReMapper. The average precision of ReMapper varies slightly from 99.51% (Q1) to 99.03% (Q4) whereas the recall varies slightly from 99.16% (Q1) to 98.7% (Q4), suggesting that the patch size had little influence on the performance of ReMapper.

TABLE III
EXECUTION TIME PER COMMIT

Patch Size	Median (s)	Average (s)
Q1 [4, 38, 72]	0.56	1.76
Q2 [73, 118, 156]	1.18	2.31
Q3 [157, 244, 364]	2.06	3.05
Q4 [376, 833, 7154]	2.71	4.92

To quantitatively measure the correlation between the patch size and the performance of the evaluated approaches, we compute their Pearson correlation coefficient [33] by taking each commit (patch) as an individual. The computation results suggest that the correlation coefficients between the patch size and our approach's precision and recall are 0.02 and 0.01, respectively. The correlation coefficients between the patch size and the baseline approach's precision and recall are 0.11, and 0.1, respectively. It may suggest that the correlation between the patch size and the performance of the evaluated approaches is relatively weak.

D. RQ3: Scalability

The evaluation was conducted on a machine with Intel Core i7-11700 CPU @ 2.50GHz, 16 GB DDR4 memory, 512 GB SSD, Windows 10 OS, and Java 11.0.17 x64 with a maximum of 8GB Java heap memory (i.e., -Xmx8g). We recorded the execution time of ReMapper by using `System.nanoTime()` Java method [34]. Our evaluation results are presented in Table III. The first column not only specifies the categories of the patches (the same as those in Table II), but also presents the minimal size, average size, and maximal size of the given category. For example, within the first category Q1, the minimal size, average size, and maximal size of the patches are 4, 38, and 72, respectively.

From Table III, we observe that the execution time increased with the increase in patch size. However, ReMapper kept efficient even on the largest patches: The median and average execution time for the largest 25% patches (i.e., Q4) was 2.71 seconds and 4.92 seconds, respectively, with a maximum execution time of 28.3 seconds.

IV. INFLUENCE ON AUTOMATED DISCOVERY OF SOFTWARE REFACTORINGS

Automated entity matching serves as the cornerstone of automated discovery of refactorings [6], [7]. Consequently, in this section, we investigate the following question:

RQ4. To what extent can the proposed approach improve the state of the art in automated discovery of refactorings?

A. Automated Discovery of Refactorings

Automated refactoring discovery is to mine automatically refactoring operations in the software evolution history. On one side, discovering software refactorings help researchers/ developers comprehend and analyze the evolution process of the code [35], [36]. On the other side, the resulting refactoring histories are useful for various tasks, e.g., evaluation of refactoring-related approaches/tools [37]–[40] and refactoring opportunity identification [41], [42].

A few approaches have been proposed to automate the discovery of refactorings [5]–[7], [14], [23], [43], [44]. As the first step, such approaches should match entities between two (successive) versions. Based on the matched entity pairs, they leverage a set of predefined heuristics to discover refactoring operations. For example, if method m_1 in the old version matches method m_2 in the new version, and they have different names, such approaches would report a renaming: method m_1 in the old version has been renamed with the name of m_2.

B. Baseline and Scope

`RefactoringMiner` [14] represents the state of the art in automated refactoring discovery. Consequently, in this section, we take it as the baseline approach. Notably, our approach (`ReMapper`) only matches entities across versions, but it does not propose any heuristics to detect refactoring based on the matched entity pairs. To this end, during the evaluation, we simply reused all heuristics implemented by `RefactoringMiner`, and note the resulting approach as `RefactoringExtractor` (ReExtractor for short). As a result, the only difference between `ReExtractor` and `RefactoringMiner` is the entity matching algorithms.

It should be noted that the refactoring detection heuristics implemented in `RefactoringMiner` [14], [45] was not completely separated from the implementation of the entity matching algorithm. Consequently, we cannot simply feed the mapping of `ReMapper` into its heuristic implementation. In contrast, we had to migrate the heuristics from `RefactoringMiner` to ours, which was time-consuming. To minimize the cost, the evaluation focused on the following 9 refactoring types only: *rename method*, *rename field*, *rename class*, *move class*, *extract class*, *extract method*, *inline method*, *change return type*, and *change field type*. They were selected because they are popular and frequently applied [46]–[48].

C. Process

We applied the two evaluated approaches to each of the projects selected in Section III-A1. Note that some of the projects have long evolution histories, and the automated discovery of refactorings is time-consuming. Consequently, we only leveraged the latest 1,000 commits if the number of commits in one project was greater than 1,000. If either of the evaluated approaches reported any refactorings from a given commit, we collected the commit as a refactoring-discovering commit. The refactoring-discovering commits containing “extract class” or “inline method” refactorings were first selected until the number of “extract class” and the number of “inline method” reported by the evaluated approaches both exceeded 100 because these two types of refactorings are substantially less popular than the others. Giving them priorities helps to strike a balance among different refactoring types. Other refactoring-discovering commits were selected chronologically so that 20 refactoring-discovering commits per open-source project were selected for the evaluation.

TABLE IV
Performance in Automated Discovery of Refactorings

Metrics	ReExtractor	RefactoringMiner
#TP	1696	1587
#FP	23	84
#FN	33	142
Precision	98.66%	94.97%

We requested three refactoring experts (different from the participants in Section III-A2) to manually and independently validate the refactorings discovered from the selected commits. All of the participants were required to hava Java background and were familiar with software refactoring. They had a median of 6.5 years of programming experience and 4 years experience with software refactoring. In case of inconsistency (52 refactoring operations in total), the participants were requested to discuss together and reach an agreement. The three experts achieved a high consistency with a Fleiss’ kappa coefficient [26] of 0.86. Notably, for the closed-source project, we requested the original developers of the source code instead of the third-party participants to validate all (28 in total) refactoring-discovering commits.

D. Results and Analysis

To measure the performance in refactoring discovery, we counted the number of true positives (#TP), the number of false positives (#FP), and the number of false negatives (#FN). If a refactoring operation was reported by only one of the evaluated approaches and then was manually confirmed by the participants, the refactoring operation was taken as a false negative for the other approach that failed to report it as a potential refactoring. We also computed precision = #TP / (#TP + #FP). The recall was not computed because we had not manually checked all of the commits, and thus we did not know exactly how many refactoring operations had been missed by the evaluated approaches.

The evaluation results are presented in Table IV. From Table IV, we observe that compared against the baseline approach, `ReExtractor` substantially reduced the number of false positives and false negatives: On average, the number of false positives was reduced by 72.6%=(84-23)/84 and the number of false negatives was reduced by 76.8%=(142-33)/142. We also notice that the number of true positives was substantially improved by `ReExtractor` with an improvement of 6.9%=(1696-1587)/1587. Since both approaches were evaluated on the same dataset, they should have equivalent `#Positives` (i.e., the actually conducted refactorings), and thus the relative improvement in recall equals the relative improvement (6.9%) in #TP because recall = #TP / #Positives.

We notice that `ReExtractor` achieved comparable improvement on the closed-source project. Compared against the baseline, it improved #TP and precision on this project by 6.8%=(63-59)/59 and 11.4%=(91.3%-81.94%)/81.94%, re-

TABLE V
PERFORMANCE IN DISCOVERING DIFFERENT REFACTORINGS

Refactoring Type	Approaches	#TP	#FP	#FN	Precision
Rename Method	ReExtractor	305	6	9	98.07%
	RefactoringMiner	286	18	28	94.08%
Rename Field	ReExtractor	152	2	2	98.7%
	RefactoringMiner	131	11	23	92.25%
Rename Class	ReExtractor	135	0	1	100%
	RefactoringMiner	127	1	9	99.22%
Move Class	ReExtractor	190	0	0	100%
	RefactoringMiner	187	2	3	98.94%
Extract Class	ReExtractor	93	3	3	96.88%
	RefactoringMiner	85	7	11	92.39%
Extract Method	ReExtractor	231	3	10	98.72%
	RefactoringMiner	216	13	25	94.32%
Inline Method	ReExtractor	88	3	4	96.7%
	RefactoringMiner	86	11	6	88.66%
Change Return Type	ReExtractor	261	5	3	98.12%
	RefactoringMiner	240	12	24	95.24%
Change Field Type	ReExtractor	241	2	1	99.18%
	RefactoringMiner	229	8	13	96.62%

spectively. It may suggest that `ReExtractor` can outperform the baseline on both open-source and closed-source projects.

We further investigated their performance on different refactoring types. The evaluation results are presented in Table V. We observe from Table V that `ReExtractor` outperformed the baseline approach on all of the involved refactoring types, concerning both recall (#TP) and precision. The maximal improvement in precision was up to 8.04=96.7-88.66 percentage points (for *inline method* refactorings). We also notice that on all types of refactorings, `ReExtractor` substantially reduced the number of false positives and false negatives.

V. DISCUSSION

A. Threats to Validity

A threat to internal validity is that the ground truth in the employed benchmark could be inaccurate. To construct the ground truth, we requested three developers to manually match entities involved in the selected commits (from open-source projects). However, the participants were not the original developers, and thus their manual matching could be inaccurate. To mitigate this threat, we requested three experienced developers to independently validate each matching pair, and computed the Fleiss' kappa coefficient (0.85) to validate the high level of consistency. To further mitigate the threat, we also selected a closed-source peoject for the evaluation where the original developers constructed the ground truth (both for entity matching and refactoring discovery).

A threat to external validity is the limited testing data. Note that the evaluation involved difficult and time-consuming manual construction of ground truth. Consequently, it is difficult to enlarge the testing dataset. However, evaluation results on small testing dataset may suffer from limited generality. To increase the diversity of the dataset, we selected the most recent 10 conflict-generating commits from each of the 20 subject projects. On one side, the testing data covers diverse applications, increasing the diversity of the dataset. On the other side, it limited the total number of to-be-matched entities, which reduced the cost of manual checking. We did not employ random sampling because it increased the uncertainty in the replication of the evaluation.

A threat to construct validity is that the re-implementation of the refactoring detection heuristics in `RefactoringMiner` could be inaccurate. Ideally, we should replace the entity matching algorithm in `RefactoringMiner` and keep other parts (refactoring detection heuristics) intact. However, in `RefactoringMiner`, the detection heuristics are often combined with the entity matching algorithm, and thus it is difficult to replace the matching algorithm in `RefactoringMiner`. To this end, we re-implemented the heuristics because such heuristics are simple and easy to implement, and connected the re-implementation to our matching algorithm. However, any bugs in the re-implementation may have biased the evaluation results.

B. Limitations

Currently, `ReMapper` works on Java programs only. Although the rationale of `ReMapper` is not language-specific, its implementation depends on the automated parsing of source code, and thus it is language-specific. Besides that, `ReMapper` depends on the data types of the entities, and thus it may not be easily adapted to dynamically typed languages, e.g, *Python* and *JavaScript* where the data types could not be determined via static analysis.

It is often time-consuming to retrieve all references for each of the involved entities within a given commit. That is one of the reasons why existing entity matching algorithms do not exploit references. To minimize the cost, `ReMapper` takes the following two measures. First, `ReMapper` only retrieves references for those that could not be matched by qualified names. Most of the involved entities within the changed files could be matched by qualified names, which significantly reduces the cost of retrieving references. Second, `ReMapper` only considers references whose callers and callees are within the changed files. That is, it does not search unchanged files for references because software entities accessed by such unchanged files must have kept their qualified names intact (otherwise, the qualified name-based access would be obsolete). However, such entities whose qualified names are unchanged should have been matched by qualified names (the first step in Fig. 1), and thus there is no need to retrieve their references (for reference-based matching). Notably, ignoring such unchanged files could significantly reduce the cost of retrieving references because most of the files in the repository are not influenced by the given commit.

VI. RELATED WORK

Entity matching algorithms could be divided into two categories. The first category takes two design models as input and maps one model into the other, and thus they do not exploit the detailed implementation (e.g., method bodies) of the software entities. `UMLDiff`, proposed by Xing and Stroulia [3], is the most famous algorithm of this category. Although it takes the source code as input, it actually works on the class diagrams automatically generated according to the source code. Consequently, it maps software entities by mapping one graph into another one where a graph represents a class diagram. Nodes of the graph represent software entities whereas edges represent the relations between entities. Nodes and edges are matched according to their labels' similarity (i.e., name-based similarity) and similarity in connections (i.e., edges). `SiDiff`, proposed by Kelter et al. [49] is another algorithm belonging to this category. It matches leaf entities (e.g., methods) according to their types, names, and signatures (like parameters of methods), and then takes a bottom-up tactic to match high-level entities (like classes) by computing their similarity in names, matched sub-entities, common generalization targets, and common packages. Once a high-level entity is matched, it takes a top-down tactic to match its sub-entities. Kim et al. [4] presented an approach to automatically infer changes at or above method headers. They defined a set of rules, i.e., low-level transformations (e.g., replacing return types of methods) on method headers. Two methods match each other if there is a sequence of rules that can transform the head of one method into the head of the other.

The second category takes full advantage of the involved source code. `JDiff`, proposed by Apiwattanapong et al. [2], [24], is of this category. It first matches classes and interfaces based on their fully qualified names, and then matches methods within matched pairs of classes/interfaces according to method signatures. To figure out the difference between a pair of matched methods, it represents the method bodies as enhanced control flow graphs and matches nodes in the graphs by graph isomorphism. `ChangeDistiller`, proposed by Fluri et al. [20], is widely-used for fine-grained source code change extraction. Given two ASTs representing the source code before and after the revision, `ChangeDistiller` discovers a sequence of changes that can turn one tree into another. The key of the algorithm is the similarity-based matching between two AST nodes where the similarity between nodes indicates how many percentages of their descendant nodes are common. `GumTree`, proposed by Falleri et al. [21] improves `ChangeDistiller` by removing the assumption that leaf nodes contain a lot of text. It exploits a greedy top-down search algorithm to find the greatest isomorphic subtrees between two ASTs, and exploits a bottom-up algorithm to match software entities based on common (matched) subtrees. `RefDiff` proposed by Silva et al. [6], [23] leveraged a variation of the TF-IDF weighting scheme [50], and a weighted Jaccard coefficient [51] to compute the similarity between software entities (taken as plain texts), and matched entities according to predefined similarity thresholds. `RefactoringMiner` proposed by Tsantalis et al. [7], [14] matches entities without any similarity thresholds, which significantly facilitates the usage of the algorithm. Another significant contribution of `RefactoringMiner` is that it proposed two novel preprocessing techniques (i.e., *abstraction* and *argumentization*) to facilitate statement matching. We notice that none of the matching algorithms of this category exploits the references of software entities as our approach does.

Entity matching algorithms are the cornerstone of a variety of software engineering tasks, e.g., refactoring discovery, entity tracking, and change comprehension. Here we present recent applications of the entity matching algorithms introduced in the preceding paragraphs. Xing and Stroulia [52] developed `JDEvAn` [53] to discover refactoring operations based on the structural changes reported by `UMLDiff` [3]. `RefDiff` [6], [23] has also been successfully employed to discover software refactorings. `RefactoringMiner` [7], [14] is one of the most widely used refactoring discovery tools, and the major reason for its improvement over the state of the art is that it proposed a novel entity matching algorithm as introduced in the preceding paragraph proposed. `CodeShovel`, proposed by Grund et al. [9], leverages an entity matching algorithm to track evolved methods in the commit history. `CodeTracker` proposed by Jodavi and Tsantalis [10] utilizes the state-of-the-art entity matching algorithm in `RefactoringMiner` [7], [14] to track methods in evolution history. `ClDiff`, proposed by Huang et al. [54], is a visual code difference tool, leveraging `GumTree` [21] to map source code entities across versions.

VII. Conclusions and Future Work

In this paper, we propose an automated iterative approach (called `ReMapper`) to match software entities between two successive versions. It takes full advantage of the qualified names, the implementations, and the references of software entities. It leverages an iterative matching algorithm to handle the interdependence between entity matching and the computation of reference-based similarity. We evaluate `ReMapper` on both open-source and closed-source real-world projects. Our evaluation results suggest that `ReMapper` substantially improved the state of the art in entity matching, reducing the number of mistakes by 85.8%. It also substantially improved the state of the art in automated discovery of software refactorings, improving recall by 6.9% and reducing the number of false positives by 72.6%.

In the future, we would like to adapt the proposed approach to more programming languages, which may significantly improve the usefulness of the proposed approach. We also plan to implement more refactoring detection heuristics so that we can discover more refactorings with `ReExtractor`.

VIII. Data Availability

The replication package, including the tools and the data, is publicly available [55], [56].

Acknowledgment

The authors would like to thank the anonymous reviewers from ASE 2023 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037).

References

[1] M. Kim and D. Notkin, "Program element matching for multi-version program analyses," in *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. Shanghai, China: ACM, 2006, pp. 58–64, https://doi.org/10.1145/1137983.1137999.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*. Linz, Austria: IEEE, 2004, pp. 2–13, https://doi.org/10.1109/ASE.2004.1342719.

[3] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. Long Beach, CA, USA: ACM, 2005, pp. 54–65, https://doi.org/10.1145/1101908.1101919.

[4] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. Minneapolis, MN, USA: IEEE, 2007, pp. 333–343, https://doi.org/10.1109/ICSE.2007.20.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*. Nantes, France: Springer, 2006, pp. 404–428, https://doi.org/10.1007/11785477_24.

[6] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. Buenos Aires, Argentina: IEEE, 2017, pp. 269–279, https://doi.org/10.1109/MSR.2017.14.

[7] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Gothenburg, Sweden: ACM, 2018, pp. 483–494, https://doi.org/10.1145/3180155.3180206.

[8] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking Java methods with Git mechanisms," *Journal of Systems and Software*, vol. 165, p. 110571, 2020, https://doi.org/10.1016/j.jss.2020.110571.

[9] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, "CodeShovel: Constructing method-level source code histories," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE '21)*. Madrid, Spain: IEEE, 2021, pp. 1510–1522, https://doi.org/10.1109/ICSE43902.2021.00135.

[10] M. Jodavi and N. Tsantalis, "Accurate method and variable tracking in commit history," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. Singapore, Singapore: ACM, 2022, pp. 183–195, https://doi.org/10.1145/3540250.3549079.

[11] M. W. Godfrey and Q. Tu, "Tracking structural evolution using origin analysis," in *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02)*. Orlando, FL, USA: ACM, 2002, pp. 117–119, https://doi.org/10.1145/512035.512062.

[12] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*. Paris, France: IEEE, 2002, pp. 127–136, https://doi.org/10.1109/WPC.2002.1021334.

[13] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005, https://doi.org/10.1109/TSE.2005.28.

[14] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022, https://doi.org/10.1109/TSE.2020.3007722.

[15] "The dataset of entity matching," https://github.com/lyoubo/ReMapper/tree/master/data/entity%20matching, 2023.

[16] "Spring Boot," https://github.com/spring-projects/spring-boot/commit/b9e57c7, 2023.

[17] "Eclipse JDT ASTNode.toString()," https://help.eclipse.org/latest/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html&anchor=toString(), 2023.

[18] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945, https://doi.org/10.2307/1932409.

[19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the 1988 International Conference on Software Maintenance (ICSM '98)*. Bethesda, MD, USA: IEEE, 1998, pp. 368–377, https://doi.org/10.1109/ICSM.1998.738528.

[20] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change Distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007, https://doi.org/10.1109/TSE.2007.70731.

[21] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Vasteras, Sweden: ACM, 2014, pp. 313–324, https://doi.org/10.1145/2642937.2642982.

[22] G. W. Adamson and J. Boreham, "The use of an association measure based on character structure to identify semantically related pairs of words and document titles," *Information Storage and Retrieval*, vol. 10, no. 7-8, pp. 253–260, 1974, https://doi.org/10.1016/0020-0271(74)90020-5.

[23] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020, https://doi.org/10.1109/TSE.2020.2968072.

[24] T. Apiwattanapong, A. Orso, and M. J. Harrold, "JDiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, pp. 3–36, 2007, https://doi.org/10.1007/s10515-006-0002-0.

[25] "Comparison against JDiff," https://github.com/lyoubo/ReMapper/tree/master/data/comparison%20against%20JDiff, 2023.

[26] J. L. Fleiss, "Measuring nominal scale agreement among many raters," *Psychological Bulletin*, vol. 76, no. 5, pp. 378–382, 1971, https://doi.org/10.1037/h0031619.

[27] S. McKillup, *Statistics Explained: An Introductory Guide for Life Scientists*. Cambridge, UK: Cambridge University Press, 2006.

[28] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1988.

[29] "Flink," https://github.com/apache/flink/commit/575517bbb8de36b21632e54b441b7dcbc4d061c4#diff-45c9720456d17ae7c5b5d825ef2d58a3e2d30f78bcadbceacb7a35a760288aeeL73, 2023.

[30] "Checkstyle," https://github.com/checkstyle/checkstyle/commit/43ae5d651d5b3078d9c04a0539134e811f461f5c#diff-eb159ca4c068124acd5ef9f9d88e86b478ebf0b2a079d8aec61b4c8ced305483L1023, 2023.

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

[32] ——, "The catalog of refactorings," https://refactoring.com/catalog/, 2023.

[33] A. Bravais, *Analyse Mathématique Sur Les Probabilités Des Erreurs de Situation d'un Point*. Paris, France: Impr. Royale, 1844.

[34] "System.nanoTime() Java method," https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#nanoTime(), 2023.

[35] M. Kim, D. Cai, and S. Kim, "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)*. Honolulu, HI, USA: ACM, 2011, pp. 151–160, https://doi.org/10.1145/1985793.1985815.

[36] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. Buenos Aires, Argentina: IEEE, 2017, pp. 176–185, https://doi.org/10.1109/ICPC.2017.38.

[37] P. Weißgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. Shanghai, China: ACM, 2006, pp. 112–118, https://doi.org/10.1145/1137983.1138011.

[38] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*. Riva del Garda, Italy: IEEE, 2012, pp. 104–113, https://doi.org/10.1109/SCAM.2012.20.

[39] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM '12)*. Trento, Italy: IEEE, 2012, pp. 357–366, https://doi.org/10.1109/ICSM.2012.6405293.

[40] E. L. Alves, M. Song, and M. Kim, "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)*. Hong Kong, China: ACM, 2014, pp. 751–754, https://doi.org/10.1145/2635868.2661674.

[41] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013, https://doi.org/10.1109/TSE.2013.4.

[42] H. Liu, Q. Liu, Y. Liu, and Z. Wang, "Identifying renaming opportunities by expanding conducted rename refactorings," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 887–900, 2015, https://doi.org/10.1109/TSE.2015.2427831.

[43] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*. Timisoara, Romania: IEEE, 2010, pp. 1–10, https://doi.org/10.1109/ICSM.2010.5609577.

[44] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: 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)*. Santa Fe, NM, USA: ACM, 2010, pp. 371–372, https://doi.org/10.1145/1882291.1882353.

[45] N. Tsantalis, "The implementation of RefactoringMiner (release: 2.4.0)," https://github.com/tsantalis/RefactoringMiner, 2023.

[46] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012, https://doi.org/10.1109/TSE.2011.41.

[47] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP '13)*. Montpellier, France: Springer, 2013, pp. 552–576, https://doi.org/10.1007/978-3-642-39038-8_23.

[48] A. Ketkar, O. Smirnov, N. Tsantalis, D. Dig, and T. Bryksin, "Inferring and applying type changes," in *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Pittsburgh, PA, USA: ACM, 2022, pp. 1206–1218, https://doi.org/10.1145/3510003.3510115.

[49] U. Kelter, J. Wehren, and J. Niere, "A generic difference algorithm for UML models," *Software Engineering*, pp. 105–116, 2005.

[50] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, 1986.

[51] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii, "Finding the Jaccard median," in *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*. Austin, TX, USA: SIAM, 2010, pp. 293–311, https://doi.org/10.1137/1.9781611973075.25.

[52] Z. Xing and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Benevento, Italy: IEEE, 2006, pp. 263–274, https://doi.org/10.1109/WCRE.2006.48.

[53] ——, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. Leipzig, Germany: ACM, 2008, pp. 951–952, https://doi.org/10.1145/1370175.1370203.

[54] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "ClDiff: Generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Montpellier, France: ACM, 2018, pp. 679–690, https://doi.org/10.1145/3238147.3238219.

[55] B. Liu, "Replication package," https://doi.org/10.5281/zenodo.8249717, 2023.

[56] ——, "ReMapper," https://github.com/lyoubo/ReMapper, 2023.