Mining Revision Histories to Detect Cross-Language Clones without Intermediates

Xiao Cheng, Zhiming Peng, Lingxiaojan, Hao Zhong, Haibo Yu, Jianjun Zhao

1 Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
2 School of Information Systems, Singapore Management University, Singapore
3 School of Software, Shanghai Jiao Tong University, China
4 Department of Advanced Information Technology, Kyushu University, Japan

{x.cheng, zhonghao, haibo.yu}@sjtu.edu.cn, {zmpeng, lxjiang}@smu.edu.sg, zhao@ait.kyushu-u.ac.jp

ABSTRACT

To attract more users on different platforms, many projects release their versions in multiple programming languages (e.g., Java and C#). They typically have many code snippets that implement similar functionalities, i.e., cross-language clones. Programmers often need to track and modify cross-language clones consistently to maintain similar functionalities across different language implementations. In literature, researchers have proposed approaches to detect cross-language clones, mostly for languages that share a common intermediate language (such as the .NET language family) so that techniques for detecting single-language clones can be applied. As a result, those approaches cannot detect cross-language clones for many projects that are not implemented in a .NET language. To overcome the limitation, in this paper, we propose a novel approach, CLCMiner, that detects cross-language clones automatically without the need of an intermediate language. Our approach mines such clones from revision histories, which reflect how programmers maintain cross-language clones in practice. We have implemented a prototype tool for our approach and conducted an evaluation on five open source projects that have versions in Java and C#. The results show that CLCMiner achieves high accuracy and point to promising future work.

CCS Concepts

• Software and its engineering → Software libraries and repositories; Software maintenance tools;

Keywords

cross-language clone; diff; revision history

1. INTRODUCTION

Due to various considerations, many projects are implemented in different programming languages. For example, ANTLR [1] releases its versions in Java, C#, JavaScript and Python. As another example, Lucene [2] release its versions in Java and C#. When maintaining such projects, if a code snippet is modified, programmers often copy their modifications to proper locations in other language versions, and conduct further editions, according to the syntactic and semantic requirements of the target programming language. As a result, released versions can have similar code snippet in different programming languages. In literature, Kraft et al. [15] call such code snippets as cross-language code clones. Cross-language clones can be inevitable and beneficial for a project [13], even though sometimes code clones may be harmful and could be removed [6]. It also becomes necessary for programmers to locate and maintain cross-language clones. For example, after a developer D1 develops a cross-language project, another developer D2, who is not familiar with the source code, joins the project. If D2 modifies a code snippet in a programming language, all the clone instances of the code snippet in another language may require similar modifications. In particular, when a bug is reported in a programming language, D2 often needs to check versions in other languages. It can be tedious for D2 to locate the clones manually. An automated cross-language clone detection tool can be useful for D2 and reduce overlooks.

Researchers [9, 12, 10, 11] have proposed various detection approaches for code clones in one programming language. Recently, researchers [15, 3] start to detect cross-language code clones for the .NET language family. However, their approaches are limited to the languages that share a common intermediate language, while many projects are implemented in other programming languages that cannot be addressed by existing approaches. Without a common intermediate language, it becomes more challenging to detect cross-language clones. In this paper, we need to overcome the following challenges to detect such clones:

Challenge 1. Existing approaches [15, 3] can detect cross-language clones for the .NET language family, which is built on the Microsoft intermediate language. These approaches assume that different programming languages share a common intermediate language. As a result, it is feasible to reduce source code to the intermediate language and to detect clones based on such intermediates. However, most languages do not have such a common intermediate language, which makes the task challenging.

Challenge 2. Different programming languages have different grammars and APIs. As a result, even if code snippets in different programming languages implements the same functionality, their structures (even their lines of code) can be different. It becomes more challenge to determine cross-language clones than clones in a single language.

In this paper, we propose a new approach, CLCMiner, which detects cross-language clones without intermediate
languages. Our approach is based on comparing revision histories that are recorded in repository logs. Here, Diff is a change-log tool that is widely used in Version Control Systems (VCS) such as Git, SVN, and Mercurial. In this paper, we also call its generated delta as a diff. Each diff describes changes of a code fragment in the source code.

The rationale for our approach is that, in multi-language projects, versions in different languages can have similar diffs since different versions should have similar functionalities and developers may change all versions in similar ways (i.e., diffs) to perform similar tasks. Based on this insight, our approach detects cross-language clones through comparing the similarity among pieces of diffs in different programming languages and aligning each diff with the most similar one, which is called diff matching. Meanwhile, as a diff contains both its changed lines of code and surrounding code lines, it becomes easier to determine the granularity of cross-language clones based on diffs.

This paper makes the following contributions:

- To the best of our knowledge, we proposed the first approach that detects cross-language clones for programming languages that do not have an intermediate language. Our approach is based on comparing change histories, and thus reduces cross-language clone detection into a diff matching problem.
- We conducted an evaluation on five open source projects that release versions in Java and C#. Our results show that our approach achieves an average precision of 87% and recall of 93%.

2. RUNNING EXAMPLE

Figure 1 shows an example of two matched diffs in Java and C# code fragments. We use the example to illustrate the problem and how our approach works. The diff on the left records two lines of changes in an if-block in Java class MachineProbe while the one on the right records four lines of changes in a block in C# class MachineProbe.

The matched diff pair indicates a cross-language clone, which has similar functionality. Both of the code fragments intend to set the fields (i.e., line and charPositionInLine) of the object token. The Java code achieves this through method invocations (i.e., setLine() and setCharPositionInLine()), while the C# code achieves this through assigning them directly. In addition, the Java jumps out of the if-block through a break statement, while the C# code uses a goto statement. Our approach extracts all the diffs from the project (in both Java and C#) and matches each diff in Java code to a diff in C# code according to the class name (e.g., MachineProbe) and the text similarity (e.g., the identifier names and the words). Thus, our approach is able to detect the cross-language clone in Figure 1. The detailed algorithm to match the diffs will be presented in Section 3.

3. APPROACH AND IMPLEMENTATION

3.1 Overview

The same functionality implemented in different languages may diverge in the syntax, but the functionality in one language (e.g., Java) can be used as a reference for implementation in another language (e.g., C#). As a result, similar variable or method names can be used in such cases. To detect cross-language clones, CLCMiner adapts natural language processing (NLP) techniques to calculate the similarity among pieces of diffs in different programming languages and selects the most similar one for each diff as a pair of matched diffs. Each pair of matched diffs refers to a pair of potential clones. Based on the most similar one, we expect that other similar ones can be further detected by single-language clone detection tools. Therefore, CLCMiner so far does not report the second most similar or other similar ones for each diff. Finally, CLCMiner ranks the matched pairs of diffs according to their diff similarity and reports top ones as potential cross-language clones.

Figure 2 shows an overview of CLCMiner. Each blue rectangle represents a processing step, and each red rounded rectangle represents an entity. The input of CLCMiner is git logs, and its output is a ranked list of detected potential cross-language clones. The approach has four main steps:

1. Log Parsing. This step extracts diffs and their attributes from revision logs.
2. Normalizing. This step normalizes diffs and prepares for the comparison in the next step.
3. Diff Matching. This step matches diffs in different languages by comparing their similarity values. For each diff, its matched one is the most similar one.
4. Ranking & Reporting. This step ranks matched diffs according to their similarity and reports cross-language clones.
3.2 Log Parsing

In a Version Control System (VCS), repository logs record code evolution histories. For example, the structure of git logs is organized as follows: a git log consists of several commits; each commit is related to one or more files; each file is related to one or more diffs; each diff records one or more change hunks that occur in a code fragment [5].

Log parsing is a preparation step that extracts useful information from repository logs. CLCMiner parses a log into a list of diffs, and attaches each diff with a set of attributes, including Commit Date (CD), Commit Author (CA), Commit ID (CID), File Name (FN), and Commit Message (CM).

For example, Table 1 lists the attributes of the diffs in Figure 1. Some attributes (e.g., FN) are useful for matching diffs, and others (e.g., CID) help to uniquely locate the code.

3.3 Normalizing

Normalizing is to remove uninteresting contents from the diffs and transform the rest contents into normalized comparison units. CLCMiner uses the token streams of the diffs as the comparison unit, and normalizes them as follows:

1. **Removing Comments.** To relieve the impact of comments in natural language, CLCMiner removes the comments from the diff firstly.

2. **Lexing.** CLCMiner employs a lexer to lex the code in the diff without comments into a token stream.

3. **Removing Punctuations.** Punctuations and numbers are removed from the token stream, as they often do not indicate significant semantics.

4. **Post Processing.** Camel case tokens are split by the uppercase letters and tokens with underscores are split by the underscores. After that, all tokens are transformed to lowercases. This step paves difference between programming styles.

In Table 1, Column “TS” lists the two normalized token streams of the diffs in the running example.

3.4 Diff Matching

Diff matching is the process to align a diff in a language (e.g., Java) to the diff in the other language (e.g., C#), according to their similarity. Bag of Words (BOW) [8] represents a piece of text as a bag (multiset) of its words, disregarding grammar and the ordering of words. CLCMiner adopts BOW to build a characteristic vector, each dimension of which represents the number of times that a word appears in the token stream of a diff, to calculate the similarity between two diffs.

Table 2 shows the characteristic vectors for the token streams in Table 1. Column “Token” lists the words appearing in the token streams. Columns “Java” and “C#” list the numbers of times that each word occurs in the diff of MachineProbe.java and MachineProbe.cs respectively. Column “Difference” lists the absolute value of the difference between the numbers of occurrence. For example, token “break” appears in the diff of Java code once but does not appear in the C# code, and the difference is 1 (|1 − 0|).

We use the distance between two vectors to measure the similarity of two diffs. For two vectors, $V_i(v_1, v_2, \ldots, v_n)$ and $V_j(v'_1, v'_2, \ldots, v'_n)$, their distance is defined as:

$$\text{Distance}(V_i, V_j) = \sum_{k=1}^{n} \frac{|v_k - v'_k|}{\sqrt{\sum_{k=1}^{n} (v_k + v'_k)}}$$

In the running example, the distance between two pieces of diffs is $61/(80 + 59) = 0.4388$. The smaller the distance is, the more similar two pieces of diffs appear.

Algorithm [8] shows the details for matching diffs. It takes as input two lists of diffs, each of which represents changes of the code fragments in a programming language. The output is a list of matched diff pairs, each of which is from different input lists. CLCMiner compares the sizes of the two diff lists and sets the small one and the large one as source and target respectively (Lines 1–2). The diffs, whose file names are the same, are called neighbors of each other. For each diff in source (d.), CLCMiner searches target for its nearest neighbors by comparing the distances from $d_s$ to all of its neighbors in target (Lines 3–18). The shortest distance indicates the nearest one. As long as there exists a neighbor in target for $d_s$, $d_s$ can be matched; otherwise, it cannot.

CLCMiner only matches a diff to its nearest neighbor to report clone pairs, instead of reporting all its top-k nearest neighbors to form clone groups. This takes into consideration that, with the nearest neighbor, the other top-k nearest neighbors and even clones in files with different names can be detected by a single-language clone detector to build more comprehensive clone groups. Section [5] discusses more about this setting for future work.

3.5 Ranking and Reporting

Each pair of matched diffs is called clone candidates. We rank all such pairs according to their distances. The pairs whose diff distances are lower than 0.5 are to be reported as code clones because it is empirically determined (cf. Section 4) that such short distance pairs of diffs are highly likely to be cross-language clones.

4. EVALUATION

We aim to answer the following research questions:
Algorithm 1: Diff Matching

Input: List $d_{List_i}$, $d_{List_o}$
Output: List $d_{Pair}$
1. $source = \min_{List_i}(d_{List_i}, d_{List_o})$
2. $target = \min_{List_o}(d_{List_i}, d_{List_o})$
3. foreach $d_i \in source$ do
   4. $distance \leftarrow 1$
   5. foreach $d_t \in target$ do
      6. if $d_i.fileName().equals(d_t.fileName())$ then
         7. if Distance($d_i, d_t) == distance then
            8. pairs.add($d_i, d_t$);
         end
      9. if Distance($d_i, d_t) < distance then
         10. pairs.add($d_i, d_t$);
         11. distance $\leftarrow$ Distance($d_i, d_t$);
      end
   end
end
16. $d_{Pair}.addAll(pairs)$;
17. return $d_{Pair}$;

- **RQ 1.** What is the clone ratio distribution with respect to the diff distances?
- **RQ 2.** What is the accuracy of CLCMiner?
- **RQ 3.** What is the impact of the other attributes on cross-language clones?

### 4.1 Setup

In our evaluation, we use five open source projects implemented in both Java and C#, i.e., ANTLR3, FpML, Log4j (Log4Net), Spring, and Lucene. Table 3 shows the projects and lists LOCs, log sizes, numbers of commits and diffs.

We apply our approach to each project to obtain the ranked list of cross-language clone pairs as the report. Column “#Matched Diff Pairs” in Table 3 lists the number of matched diff pairs according to the file name and diff similarity. Due to the large number of clone candidates and limited manpower, we randomly sampled, in a uniform way, a small percentage of the clone candidates in the reported ranked lists and manually labelled whether they were actual clones. As listed in Table 3 for ANTLR3, FpML, Log4j (Log4Net), and Spring, we sampled over 6% of the reported clone candidates in each project; for Lucene, we sampled about 2%. Two co-authors manually labelled whether they were actual clones separately based on the clone definition of Bellon [4] and the functionality equivalence. If there exists a difference between the labels given by the two co-authors, it will be labelled and decided by a third co-author. We calculated the clone ratio and its distribution w.r.t. the distances, where the clone ratio is defined as $CR = \frac{\#\text{clones}}{\#\text{candidates}} \times 100\%$.

### 4.2 Result

#### 4.2.1 RQs 1 & 2. Distribution and Accuracy

Figure 2 shows the clone ratio distribution and the accumulated clone ratio, w.r.t. the diff distances. The clone ratio distribution in Figure 2(a) indicates: 1) almost all the candidates whose diff distances are lower than 0.3 are clones; 2) almost none of the candidates whose diff distances are larger than 0.7 is clone; 3) with the distance increasing from 0.3 to 0.5, the clone ratio decreases gradually; 4) with the distance increasing from 0.5 to 0.7, the clone ratio decreases greatly.

The accumulated clone ratio in Figure 2(b) also decreases with the increasing of the diff distance. When the diff distance is lower than 0.5, the clone ratio decreases slowly and when the diff distance is larger than 0.5, the clone ratio decreases greatly.

Based on the above observation, it is reasonable to set 0.5 as the proper threshold distance. If the diff distance is lower than 0.5, its related clone candidate is considered as a clone; if the diff distance is larger than 0.5, its related clone candidate is not considered as a clone. In other words, we only report as clones the pairs of code fragments in the ranked list whose diff distance is lower than 0.5.

We use precision and recall to evaluate the accuracy of CLCMiner. In this way, for ANTLR3, FpML, Log4j (Log4Net), Spring and Lucene, w.r.t. the manually labelled clone samples, the report precisions are 86%, 90%, 71%, 68% and 90% respectively and the average precision is about 87%. For the clone candidates in the five projects whose diff distance is between 0.5 and 1, the clone ratios are 3%, 8%, 2%, 5%, and 2% respectively. Since it is impossible to know how many actual cross-language clones in the projects, we calculate the recall based on the number of the missed clones whose distance is larger than 0.5. In this way, the recalls of the five projects are 90%, 97%, 71%, 69% and 98% respectively and the average recall is about 93%.

#### 4.2.2 RQ3. Impact of More Attributes of Diffs

For matching diffs, BOW is not the only choice. We identify the following attributes that may be used to improve the effectiveness of matching cross-language clones in future.

**Author.** As a developer may have a programming style that may persist even across different languages, we hypothesize that a pair of diffs from different language versions of a project may be more likely to be clones if they are authored by the same developer. To investigate this hypothesis, we look into the labels for the clone reports sampled in the way mentioned in Section 4.1. Among these five projects, all sampled pairs of diffs in Spring and Log4j (Log4Net) were committed by different persons; about only 0.5% of the diff pairs in Lucene were committed by the same developer, and about only 1% in FpML were committed by the same developer. ANTLR3 has a more pronounced difference: about 74% of diff pairs were made by different authors. So for each pair of diffs in the sampled reports, we have a variable indicating whether it is clones and another variable indicating whether it is made by the same author. A simple t-test showed that diff pairs made by the same author are statistically more likely to be clones than those made by different authors, but the correlation between the two variables is very weak (Pearson’s correlation coefficient is about 0.08).

**Commit Time.** As the functionalities in different language versions of a project are likely to remain consistent, changes in one language version may induce similar changes in another within a short period of time. Similarly, we in-
vestigate whether the commit time difference between the two diffs in a reported pair is correlated with whether the pair is a clone with t-test and Pearson’s correlation. We noticed that five projects exhibit different correlations between commit time differences and clones. In FpML and Spring, the diff pairs with shorter time differences are statistically more likely to be clones, but the correlation coefficients between these two variables are weak (-0.19 and -0.11). In Log4j (Log4Net), the effect is reversed: the diff pairs with longer time differences are statistically more likely to be clones, although the correlation is still weak (0.33). In ANTLR3 and Lucene, whether diff pairs are clones statistically has no effect on their time differences.

Commit Message. As a commit message often summarizes the changes in the commit, a diff pair may be more likely to be clones if they share similar commit messages. So we also investigate whether the distance between the commit messages of a diff pair is correlated with whether the pair is a clone. We calculate the distance between commit messages via the same technique we used for code (Section 3.4), and we check the relationship with t-test and Pearson’s correlation. We found mixed results too as many commit messages are empty or very brief and non-informative: in FpML, ANTLR3, and Spring, clone pairs have statistically longer message distance, while in Log4j (Log4Net) and Lucene, clone pairs have statistically longer distance, and the correlation coefficients are all weak.

As a summary, there is no clear deciding attribute for diff pairs to be clones, besides the code itself. It could be a combined effect of various attributes, even some contexts beyond diffs. In our future work, we plan to investigate whether the combination of more attributes, together with additional ones discussed in Section 5, can be used to improve cross-language clone detection.

5. DISCUSSION AND FUTURE WORK

Using comments in code. In diff normalization (Sec-
(e.g., File I/O, HttpHeaders) in different ways. In our future work, we will refine CLCMiner to handle such cases.

**Comparing with token-based clone detection.** Some token-based clone detection techniques \[19\], can run in plain text mode to detect some cross-language clones. For example, **CCFinder** lexes each line of source files into token sequence and utilizes suffix-tree-based substring matching algorithm to search for similar subsequences. Different from **CCFinder**, **CLCMiner** splits each camel case identifier (e.g., the variable name) and utilizes the statistical method to calculate the distance between **diff**s and search for similar **diffs**. We will compare **CLCMiner** with **CCFinder** in future work.

## 6. RELATED WORK

**Cross-language clone detection.** The number of various software systems implemented in multiple languages is increasing considerably \[14\], but cross-language clone detection is limited. Kraft et al. \[15\] conduct the first study on code clones that span over multiple languages. They implemented a tool called C2D2 based on the CodeDOM library in the Microsoft .NET framework, which uses NRefactory Library to generate the Unified CodeDOM graph for both C# and VB.NET. Al-omari et al. \[3\] present a clone detection approach for the .NET language family too, based on the Common Intermediate Language (CIL). It can detect cross-language clone pairs in C#, J#, and VB.NET. Compared with these work, our approach focuses on detecting cross-language clone detection on different platforms without common intermediate languages.

**Data mining in VCS.** There are considerable studies of data mining in Version Control Systems (VCS). Zimmermann et al. \[21\] apply data mining on version histories to recommend related syntactic changes. Girba et al. \[7\] apply concept analysis on VCS to identify groups of co-changes. McIntosh, et al. \[16\] mine source and test code for accompanying build changes. Meng et al. \[17\] mine revision histories to identify updated API interfaces. We mine VCS for a different purpose, i.e., detecting cross-language clones.

## 7. CONCLUSION

This paper proposes a novel approach, **CLCMiner**, that detects cross-language clones without common intermediate languages. Our key idea is to utilize **diff** similarity. We have implemented and evaluated its prototype on five open source projects. The results show that **CLCMiner** can detect many cross-language code clones with a high precision of 87% and recall of 93% on average (w.r.t. distance threshold 0.5).

To improve **CLCMiner** in our future work, we plan to refine the handling of false positives, detect more cross-language clones not captured in revision histories by incorporating in single-language clone detectors, and detect more clone groups across more languages (e.g., Objective-C, Swift, and Java) as described in Section 8.

## 8. ACKNOWLEDGMENTS

This work is sponsored by the 973 Program in China (No. 2015CB352203), the National Nature Science Foundation of China (No. 61572312, No. 61572313, and No. 61272102) and the grant of Science and Technology Commission of Shanghai Municipality (No. 15DZ1100305). This work is performed during Xiao Cheng’s visit to Singapore Management University (SMU) and partially supported by SMU. Also, we thank all anonymous reviewers for their feedbacks.

## 9. REFERENCES


