2-2017

CLCMiner: Detecting cross-language clones without intermediates

Xiao CHENG
Singapore Management University, xcheng@smu.edu.sg

Zhiming PENG

Lingxiao JIANG
Singapore Management University, lxjiang@smu.edu.sg

Hao ZHONG

Haibo YU

See next page for additional authors

DOI: https://doi.org/10.1587/transinf.2016EDP7334

Published in IEICE Transactions on Information and Systems, 2017 Feb, Vol. E100-D, No. 2, pp. 273-284.
http://doi.org/10.1587/transinf.2016EDP7334

PAPER

CLCMiner: Detecting Cross-Language Clones without Intermediates*

Xiao CHENG[†a)], **Zhiming PENG**[††], **Lingxiao JIANG**[††b)], **Hao ZHONG**[†], **Haibo YU**[†††], ***Nonmembers***, ***and*** **Jianjun ZHAO**[††††], ***Member***

SUMMARY The proliferation of diverse kinds of programming languages and platforms makes it a common need to have the same functionality implemented in different languages for different platforms, such as Java for Android applications and C# for Windows phone applications. Although versions of code written in different languages appear syntactically quite different from each other, they are intended to implement the same software and typically contain many code snippets that implement similar functionalities, which we call *cross-language clones*. When the version of code in one language evolves according to changing functionality requirements and/or bug fixes, its cross-language clones may also need be changed to maintain consistent implementations for the same functionality. Thus, it is needed to have automated ways to locate and track cross-language clones within the evolving software. In the literature, approaches for detecting cross-language clones are only for languages that share a common intermediate language (such as the .NET language family) because they are built on techniques for detecting single-language clones. To extend the capability of cross-language clone detection to more diverse kinds of languages, we propose a novel automated approach, *CLCMiner*, without the need of an intermediate language. It mines such clones from revision histories, based on our assumption that revisions to different versions of code implemented in different languages may naturally reflect how programmers change cross-language clones in practice, and that similarities among the revisions (referred to as *clones in diffs* or *diff clones*) may indicate actual similar code. We have implemented a prototype and applied it to ten open source projects implementations in both Java and C#. The reported clones that occur in revision histories are of high precisions (89% on average) and recalls (95% on average). Compared with token-based code clone detection tools that can treat code as plain texts, our tool can detect significantly more cross-language clones. All the evaluation results demonstrate the feasibility of revision-history based techniques for detecting cross-language clones without intermediates and point to promising future work.

key words: *cross-language clone, code clone, revision, diff, similarity*

1. Introduction

With diverse programming languages available on different platforms catering to varieties of users, it is a common need for the same functionality and even a whole software project to be implemented in different programming languages. A sample case is the availability of a mobile application that is implemented in three languages, Java for Android, C# for Windows phones and Objective-C for iPhones. Such re-implementations of a software project in different languages are also common for desktop applications. For example, Antlr [1], a parser generator, has versions implemented in Java, C#, JavaScript and Python. For another example, Lucene [2], a text search engine, has implementations in Java and C#. When implementing or changing a functionality in one version of such projects, it is naturally needed to change another version in a different language to maintain consistencies. It is also reasonable, in order to save coding efforts, for programmers to copy their modifications for one version into another version and adapt the copies to fit the different language. Even when the syntactic structures of two languages differ a lot, there should still be pieces of code in different languages having similar semantics for a similar functionality. As a result, projects in different languages can have similar code snippets in different programming languages too. In the literature [3], such code snippets are referred to as *cross-language code clones*.

Code clones may be considered harmful and removable [4] or useful and should be kept [5] depending on their usages. For cross-language clones, we take the view that they are often inevitable, and cannot be removed, based on our experiences and diverse landscape of available programming languages and platforms. Instead of removing clones, we thus need automated techniques to help programmers locate and maintain cross-language clones to save costs and improve developer productivity. For example, a developer D1 develops a cross-language project at the beginning, and later another developer D2, who is not so familiar with the source code takes over the project. When D2 modifies a code snippet in one programming language, all relevant code snippets in other languages may require similar modifications to maintain consistencies. In particular, when a bug is found and fixed in one programming language, D2 needs to check versions in other programming languages to fix similar bugs too. It would be rather tedious and error-prone for D2 to locate such cross-language clones manually, especially when their programming languages are quite different.

Many approaches have been proposed for detecting clones within one programming language [6–9]. A number of researchers [3, 10] have started to detect cross-language code clones too. However, their approaches are limited to

[†]The authors are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China.
[††]The authors are with the School of Information Systems, Singapore Management University, Singapore.
[†††]The author is with the School of Software, Shanghai Jiao Tong University, China.
[††††]The author is with the Department of Advanced Information Technology, Kyushu University, Japan.
*This paper is extended from the New Ideas Paper at the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).
a) E-mail: x.cheng@sjtu.edu.cn
b) E-mail: lxjiang@smu.edu.sg

```
@@ -129,11 +129,11 @@ public class MachineProbe {
if (!t.isEpsilon() && !t.label.getSet().and(label).isNil() &&
    next.contains(t.target)) {
  if (p.associatedASTNode != null) {
-     antlr.Token oldtoken = p.associatedASTNode.token;
+     Token oldtoken = p.associatedASTNode.token;
    CommonToken token = new CommonToken(oldtoken.getType(), oldtoken.getText());
    token.setLine(oldtoken.getLine());
-     token.setColumn(oldtoken.getColumn());
+     token.setCharPositionInLine(oldtoken.getCharPositionInLine());
    tokens.add(token);
    break nfaConfigLoop; // found path, move to next
                         // NFAState set
……
```

(a) MachineProbe.java

```
@@ -143,11 +148,11 @@ namespace Antlr3.Analysis
{
  IToken oldtoken = p.associatedASTNode.Token;
  CommonToken token = new CommonToken(oldtoken.Type, oldtoken.Text);
-   token.Line = (oldtoken.Line);
-   token.CharPositionInLine = (oldtoken.CharPositionInLine);
+   token.Line = oldtoken.Line;
+   token.CharPositionInLine = oldtoken.CharPositionInLine;
  tokens.Add(token);
-   goto endNfaConfigLoop; // found path, move to next
-                          // NFAState set
+   // found path, move to next NFAState set
+   goto endNfaConfigLoop;
}
……
```

(b) MachineProbe.cs

Fig. 1 A Pair of Matched Diffs

detect clones in the .NET language family that share a common intermediate language. In practice, many projects are implemented in other programming languages that may not be addressed by the existing approaches. Without a common intermediate language, we need to overcome the following challenges to detect cross-language clones:

Challenge 1. For different languages that do not share a common intermediate language, it is no longer feasible to reduce source code to an intermediate language and detect similar code based on the intermediates. We need to find *a new way to represent code* so that the similarity among code snippets can be measured.

Challenge 2. For different programming languages that have different grammars and APIs, it is much less likely to measure code similarity through syntactical structures, since even when code snippets in different programming languages implement the same functionality, their syntactical structures can be quite different. We need to find *a code similarity measure* that can be applied to code in different languages.

In short, we need design a language-agnostic way to represent code and measure code similarity for detecting clones across languages. In this paper, we propose a new approach, named *CLCMiner*, that can detect cross-language clones without intermediate languages. *CLCMiner* works by comparing revision histories recorded as *diffs* in software repositories. Here, *diff* refers the change-log tool widely used in Version Control Systems (VCS) such as Git and SVN to identify the differences between files; a *diff* also refers to the differences produced by the *diff* tool.

The key assumption for our approach is that in multi-language projects, versions in different languages can have similar *diffs* since developers may need to change all versions in similar ways to implement the same functionalities in the versions. Also, when *diffs* are relatively small, the syntactic differences among *diffs* may not be that significant; instead, lexical appearances, such as identifier names, may give more hints whether two code snippets are implementing similar functionalities. Based on these intuitions, our approach detects cross-language clones by comparing the similarity among pieces of *diffs* in different languages and aligning each *diff* with the most similar one. We call this process *diff matching*. As a *diff* contains both the changed lines of code and their surrounding code, matched *diffs* make it easy to determine whether the involved lines are cross-language clones.

This paper makes the following contributions:

- To the best of our knowledge, we propose the first approach that detects cross-language clones for programming languages that do not have an intermediate language. Our approach is based on comparing code change histories, and thus reduces cross-language clone detection into a *diff matching* problem.
- We conduct an evaluation on 10 open source projects that have versions implemented in both Java and C#. Our results show that our approach achieves a high precision and recall. For the 10 projects, the average precision is 89.1% and the average recall is 95.0%
- To improve our previous work [11], this paper introduces a *sliding window* into our *diff* matching algorithm and has increased the precisions and recalls greatly.
- We further demonstrate the capability of *CLCMiner* for detecting significantly more cross-language clones by comparing it with token-based clone detection tools, *CCFinder* [7] and *ConQAT* [9], because those tools can treat code in different languages as plain texts and try to detect clones in plain texts.

Different from other clone detection techniques that aim to find *all clones in a set of code*, *CLCMiner* is designed to detect only clones *in code that has ever been changed in the revision history* (which are referred to as *diff clones* in this paper) because *CLCMiner* technically detects clones based on the *diff*s. Such clones may naturally overlap with each other or disappear along the project history in its latest version (Sections 4 and 5 will provide more statistics about *diff* clones). Despite the difference, detected clones in code change history can still help maintainers to understand the correspondence between code in different programming languages and facilitate many software engineering tasks that involve changing code clones, such as tracking and studying clone genealogies, refactoring code, detecting potential bugs from consistent and inconsistent clone changes [12–15].

The rest of this paper is organized as follows: Section 2 gives an example to illustrate the whole picture of our new idea. Section 3 presents the detail of our approach. Section 4 evaluates our approach. Section 5 discusses related issues. Section 6 presents related work. Section 7 concludes.

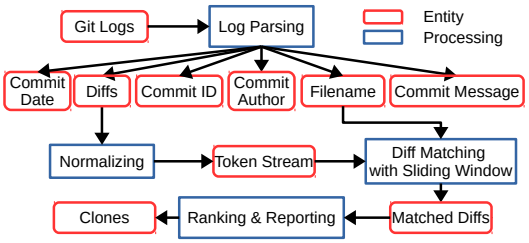


Fig. 2 Approach Overview

2. An Example

An example of two matched *diffs* in Java and C# code snippets is shown in Figure 1. The matched *diff* pair indicates a cross-language clone which has a similar functionality. The *diff* on the left-hand side 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`. This example is used to illustrate the problem and how our approach works. It is also used in the later part of the paper to explain the algorithm details.

In the example, both of the code snippets in the *diffs* intend to set the fields (*i.e.*, `line` and `charPositionInLine`) of the object pointed to by the reference `token`. The Java code achieves this through invoking the object's methods (*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 is able to detect cross-language clones from such *diffs* in Figure 1, since it reduces cross-language clone detection to a *diff* matching problem. It 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 filename without extension (*e.g.,* `MachineProbe`) and the vocabulary similarity (*e.g.,* the tokens of the identifier names and the words). The detailed algorithm to match the *diffs* will be presented in Section 3.

3. Approach and Implementation

The similar functionality implemented in different programming languages may diverge in the syntax, but the code snippet 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* adopts *Natural Language Precessing* (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. Finally, *CLCMiner* ranks the matched pairs of *diffs* according to their *diff* similarity and reports top ones as cross-language clones.

Figure 2 shows the overview of *CLCMiner*. Each rectangle in blue represents a processing step, and each rounded rectangle in red represents an entity of each step. The input of *CLCMiner* is git logs, and its output is a ranked list of detected potential cross-language clones. The approach includes 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 distance. For each *diff*, its matched one is the nearest one.
4. **Ranking & Reporting.** This step ranks matched *diffs* based on similarity and reports cross-language clones.

Table 1 Attributes of Example *Diffs*

FN	`MachineProbe.java`	`MachineProbe.cs`
CID	7288ec. . . 36ed66	e589c6. . . 3b1d56
CA	Sharwell	Sharwell
CD	Mon Mar 28 15:33:44 2011 -0800	Tue May 3 20:16:15 2011 -0800
CM	Convert all Tool grammars to ANTLR v3. The only remaining dependency on v2 is the StringTemplate 3.2's use of the v2 runtime	(C# 3) Code cleanup
TS	if t is epsilon t label get set and label is nil next contains t target if p associated ast node null antlr token oldtoken p associated ast node token token oldtoken p associated ast node token common token token new common token oldtoken get type oldtoken get text token set line oldtoken get line token set column oldtoken get column token set char position in line oldtoken get char position in line tokens add token break nfa config loop	i token oldtoken p associated ast node token common token token new common token oldtoken type oldtoken text token line oldtoken line token char position in line oldtoken char position in line token line oldtoken line token char position in line oldtoken char position in line tokens add token goto end nfa config loop goto end nfa config loop

3.1 Log Parsing

In a Version Control System (VCS), repository logs record the evolution history information. 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 [16].

Log parsing is a preparation process to extract useful information from repository logs. *CLCMiner* parses a repository 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), *filename* (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.

3.2 Normalizing

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

1. **Removing Comments.** To relieve the impact of the comments in natural language, CLCMiner first removes the comments from the code snippets in the *diffs*.
2. **Lexing.** CLCMiner lexes the code snippets in the *diffs* 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.** `CamelCaseTokens` and `tokens_with_underscores` are split by uppercase letters and the underscore to reduce differences between programming styles. At last, all letters are transformed to lowercase.

Row "TS" in Table 1 shows the token streams of the *diffs* in Figure 1. Such token streams will be compared with each other to detect clones.

3.3 Diff Matching

Diff matching is a process to align a *diff* in a language (*e.g.*, Java) to a *diff* in the other language (*e.g.*, C#), according to their similarity. In our approach, we define a distance between *diffs* to measure their similarity.

3.3.1 Distance between Token Streams

Since the distance between *diffs* is based on the distance between their token streams, we first define the distance between token streams.

CLCMiner adopts *Bag of Words* (BOW) [17], a text representation technique widely used in NLP, which represents a piece of text as a bag of its words disregarding grammar and the ordering of words, to represent token streams as *characteristic vectors*. Each dimension of the characteristic vector denotes the number of a specific token in the token stream. *CLCMiner* defines the distance between two token streams through comparing the characteristic vectors. For two vectors, $V_i(v_{i1}, v_{i2}, \ldots, v_{in})$ and $V_j(v_{j1}, v_{j2}, \ldots, v_{jn})$, their distance is defined as follows:

$$D_{ts}(V_i, V_j) = \frac{\sum_{k=1}^{n} \left| v_{ik} - v_{jk} \right|}{\sum_{k=1}^{n} (v_{ik} + v_{jk})} \tag{1}$$

3.3.2 Distance between *Diffs*

A straightforward way to calculate the distance between two *diff*s is to measure the distance between the two token streams of the *diffs* as we did previously [11]. However, we found in quite a few cases the length of two token streams are greatly different. As a result, the distance between the two token streams is so large that the two *diffs* are excluded from the clone reports. However, for these cases, the token stream of the shorter *diff* is in fact *similar to a subsequence* of the token stream of the longer one. It may help to detect more clones if we can measure the distance between subsequences of token streams. Thus, we improve our previous distance measurement by utilizing *sliding windows* to select a number of subsequences of token streams from the longer one to compare with the token stream for the shorter one.

CLCMiner sets the size of *sliding window* the same as the length of the shorter token stream and moves the *sliding window* along the longer token stream. Considering the accuracy, *CLCMiner* slides the *sliding window* one token per step. At each stop, a characteristic vector for the token stream in the *sliding window* is built. *CLCMiner* calculates the distance between it and the characteristic vector of the shorter token stream. Among all such calculated distances, the shortest distance is used as the distance between the two *diffs*. Formally, supposing *diff* d_1 and *diff* d_2 have m and n tokens in their own token streams respectively. If $m > n$, the length of *sliding window* will be set as n and the *sliding window* will be moved along the token stream of d_1. Then, $m - n + 1$ characteristic vectors $(V_{11}, V_{12}, \ldots, V_{1(m-n+1)})$ of *diff* d_1 will be built. If $m < n$, the length of *sliding window* will be set as m and the *sliding window* will be moved along the token stream of d_2. Then, $n - m + 1$ characteristic vectors $(V_{21}, V_{22}, \ldots, V_{2(n-m+1)})$ will be built. If $m = n$, there will be no *sliding window*. In brief, the distance between *diffs* d_1 and d_2 is defined as follows:

$$D_d(d_1, d_2) = \begin{cases} \min_{1 \le i \le m-n+1} D_{ts}(V_{1i}, V_2) & m > n \\ D_{ts}(V_1, V_2) & m = n \\ \min_{1 \le i \le n-m+1} D_{ts}(V_1, V_{2i}) & m < n \end{cases} \tag{2}$$

For the example in Section 2, the lengths of Java and C# token streams (in Table 1) are 80 and 59 respectively. Therefore, the length of *sliding window* is 59 as the C# one. *CLCMiner* moves the *sliding window* along the Java token stream, builds characteristic vector for each token stream in the *sliding window*. When the *sliding window* stop at the end of the Java token stream (the *sliding windows* contains the underlined tokens in Table 1), it has the shortest distance with the C# one. Table 2 shows the characteristic vectors, which have the shortest distances. Column "Token" lists the words appearing in the token streams. Columns "V_i" and "V_j" list the vector of token stream in the *sliding window* in `MachineProbe.java` and the vector of token stream in `MachineProbe.cs` respectively. Column "$\left|V_i - V_j\right|$" lists the absolute value of the difference between the values of the corresponding dimension of the vectors. For example, token "break" appears in the *sliding window* of the Java token stream but does not appear in the C# one, and the difference is 1 ($|1 - 0|$). In this way, the distance between two *diffs* is 0.305 ($36/(59 + 59)$).

Table 2 Characteristic Vectors

Token	V_i	V_j	$\left\|V_i - V_j\right\|$
add	1	1	0
antlr	1	0	1
associated	2	1	1
ast	2	1	1
break	1	0	1
...	...	...	...
token	11	10	1
tokens	1	1	0
type	1	1	0
Total	59	59	36

3.3.3 Matching Algorithm

Algorithm 1 shows the details for matching *diffs*. It takes as input two lists of *diffs*. 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 smaller one and the larger one as *source* and *target* respectively (Lines 1–2). The *diffs*, whose file names are the same (filename extensions are ignored), are called *neigh-*

Algorithm 1: Diff Matching

```
Input: List dList_j dList_cs
Output: List dPair
1  source = minList(dList_j, dList_cs);
2  target = maxList(dList_j, dList_cs);
3  foreach d_s ∈ source do
4      distance ← 1;
5      foreach d_t ∈ target do
6          if d_t.fileName().equals(d_s.fileName()) then
7              if D_d(d_s, d_t) == distance then
8                  pairs.add(d_s, d_t);
9              end
10             if D_d(d_s, d_t) < distance then
11                 pairs.clean();
12                 pairs.add(d_s, d_t);
13                 distance ← D_d(d_s, d_t);
14             end
15         end
16     end
17     dPair.addAll(pairs);
18 end
19 return dPair;
```

bors of each other. For each *diff* in *source* (d_s), *CLCMiner* searches in *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.

For projects that have multiple implementations in different languages, code in one language is often used as a reference for the implementation in another language and the same functionality is likely to be encapsulated in a file with the same name, especially for object-oriented languages (*e.g.* Java and C#). With this heuristic, *CLCMiner* only matches a *diff* with its neighbors having the same file name.

CLCMiner by default only matches a *diff* to its nearest *neighbor* to report a clone *pair* (or clone pairs if there are more than one nearest neighbors having the same shortest distance), instead of reporting all its top-k nearest *neighbors* to form a clone *group*. 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.

3.4 Ranking and Reporting

For each pair of matched *diffs*, the code fragments can be located via their attributes (*e.g.*, FN and CID). These pairs of code fragments are considered as potential clones, which are called *clone candidates*. *CLCMiner* ranks all such pairs according to their *diff* distances. The pairs whose *diff* distances are lower than a distance threshold are to be reported as clones. The distance threshold is empirically determined, which will be explained in detail in Section 4.1.

4. Evaluation

In order to justify the effectiveness of *CLCMiner* in detecting clones in code change history and in comparison with related work, we perform empirical evaluation to answer the following research questions:

- **RQ 1.** How accurate is *CLCMiner*? How do *sliding windows* help improve *CLCMiner* accuracy?
- **RQ 2.** How effectively does *CLCMiner* detect cross-language clones compared with other token-based clone detection tools that may detect cross-language clones by treating code as *plain texts*?
- **RQ 3.** What is the impact of the other code-related attributes on cross-language clones?

Table 3 Characteristics of subject projects

Projects		#LOC	#Rev.	Logs (MB)	#Commit	#Diffs	#Cand.	#Spl.
Antlr3	Java	49,617	576	32	572	2,954	9,502	471
	C#	97,304	648	31	648	19,372		
cordova-android		5,350	3,535	45	3,277	9,275	1,549	75
cordora-wp8		2,235	1,251	79	1,161	3,461		
DataStax	Java	66,131	3,047	32	2,917	25,974	9,997	196
	C#	56,641	1,349	56	1,330	20,410		
Factual	Java	3,668	307	2	279	1,535	577	112
	C#	4,080	178	1	178	823		
FpML	Java	17,810	329	244	329	2,919	4,602	454
	C#	16,548	183	227	183	2,278		
Log4j		30,287	3,561	46	2,644	21,032	3,620	359
Log4net		30,885	977	36	925	7,909		
Lucene	Java	867,110	41,081	821	24,988	308,421	72,321	718
	C#	434,577	2,911	883	1,320	45,165		
Spring	Java	551,475	14,094	335	11,971	171,090	7,237	356
	C#	224,807	1,752	316	1,747	20,672		
ua-parser	Java	839	34	0.1	34	99	31	31
	C#	850	50	0.8	50	129		
jeromq		23,621	504	5	504	5,242	6,605	326
netmq		21,836	1,679	30	1,705	13,515		

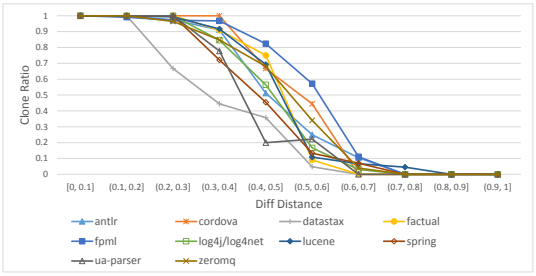
4.1 RQ1. Accuracy

With respect to the previous version of *CLCMiner* that does not use sliding windows [11], we further evaluate the accuracy of the improved *CLCMiner* for more projects with sliding windows. In the paper, *CLCMiner* detects cross-language clones from 10 open source projects implemented in both Java and C#. For each project, *CLCMiner* reports a ranked list of cross-language clone pairs.

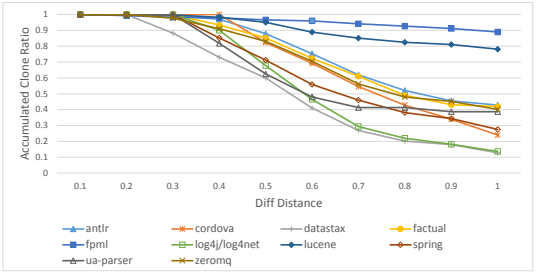
4.1.1 Setup

Table 3 shows the projects and lines of code (LOCs without comments in the latest revision), numbers of revisions, log sizes, numbers of commits and numbers of *diffs*. Column "#Cand." lists the numbers of clone candidates, which are the numbers of matched *diff* pairs. Since some *diffs* have more than one nearest neighbors, the number of matched *diff* pairs may be greater than the number of *diff*s. Due to the large number of clone candidates and limited manpower, we randomly sampled, in a uniform way, a small percentage of the candidates in the ranked lists (*cf.* Column "#Spl").

Two co-authors manually labelled whether they were actual clones separately based on the clone definition of Bellon [18] and the functionality equivalence. If there exists a difference between the labels given by them, 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 $CR = \frac{\#clones}{\#candidates} \times 100\%$.



(a) Clone Distribution



(b) Accumulated Clone Distribution

Fig. 3 Clone Ratio Distribution

4.1.2 Result

Figure 3 shows the clone ratio distribution and the accumulated clone ratio w.r.t. the *diff* distances calculated by the algorithm with *sliding window*. The clone ratio distribution in Figure 3(a) indicates:

- almost all the candidates whose *diff* distance is lower than 0.3 are clones;
- almost none of the candidates whose *diff* distance is larger than 0.7 is clone;
- with the distance increasing from 0.3 to 0.5, the clone ratio decreases gradually;
- with the distance increasing from 0.5 to 0.7, the clone ratio decreases greatly.

The accumulated clone ratio in Figure 3(b) also decreases with the increasing of the *diff* distance. Intuitively, 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.

In order to choose a threshold distance to determine cross-language clones, we plot a *Receiver Operating Characteristic* (ROC) curve for the 10 projects in Figure 4. The curve plots the *True Positive Rate* (TPR) against the *False Positive Rate* (FPR) at various threshold settings. The slope (k) of the curve reflects the relative increasing speed between TPR and FPR *w.r.t* the increase of threshold setting value: $k > 1$ means TPR increases more greatly than FPR; $k < 1$ means FPR increases more greatly than TPR.

In Figure 4, the black solid curve is the average ROC curve of the ten projects, the red dashed one is the best one (*i.e.*, Lucene) among them and the orange dotted one is the worst one (*i.e.*, ua-parser) among them. We can find that the slope of the curve decreases with the increase of TPR and

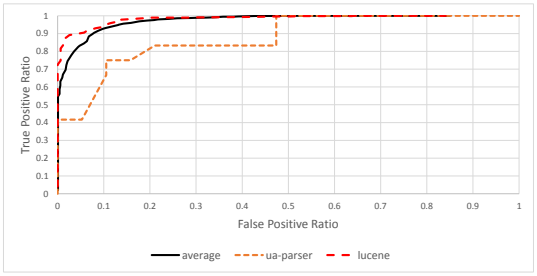


Fig. 4 ROC Curve

FPR. We can see for all of them the slope decreases *w.r.t* the increase of threshold setting value. When the slope is around 1, the corresponding threshold is the proper threshold. In our experiment, it is 0.5. 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 this experiment, we use precision and recall to measure the accuracy of *CLCMiner*. For the distance threshold 0.5, *CLCMiner* reports as clones the pairs of code fragments in the ranked list whose *diff* distance is lower than 0.5. In this way, the precision and recall are defined as follows:

$$precision = \frac{TP_{d\leq 0.5}}{TP_{d\leq 0.5} + FP_{d\leq 0.5}} \quad (3)$$

$$recall = \frac{TP_{d\leq 0.5}}{TP_{d\leq 1.0}} \quad (4)$$

Since it is unknown how many actual cross-language clones in the projects, we use the "relative" recall (Equation (4)) to reflect the capability that *CLCMIner* detect cross-language clones from the repository logs. The precision and recall for the 10 projects are listed in Table 4. The average precision and recall are 89.1% and 95.0% respectively.

Previously [11], *CLCMiner* does not use *sliding window* and is evaluated on projects Antlr, FpML, Log4j/Log4net, Lucene and Spring. The precision and recall are listed in Column "No Sliding Window" in Table 5. Column "Sliding Window" lists the corresponding precision and recall with *sliding window*. We can see that with the *sliding window*, the average precision is improved greatly from 87.4% to 96.1% and the recall is improved from 93.2% to 93.6%.

4.1.3 Discussion

The results in Table 5 show *CLCMiner with* sliding windows always achieves higher recall (i.e., detecting more clones) than without. In terms of precisions, most results with sliding windows also happen to be higher, except one case for the project Log4j/Log4net. We investigate more clone candidates in Log4j/Log4net manually, and find that there do exist many *diff* pairs whose token streams are greatly different; using sliding windows helps to reduce the calculated distance among various subsequences of the token streams, leading to many more reported clone pairs within the similarity threshold. However, those similar subsequences are significantly

Table 4 Precision and Recall

Projects	Antlr	cordova	DataStax	Factual	FpML	Log4j/4net	Lucene	Spring	ua-parser	zeromq	Average
Precision	87,9%	82.4%	60.0%	85.2%	96.6%	67.8%	95.0%	71.2%	62.5%	83.1%	89.1%
Recall	90.0%	77.8%	96.0%	97.9%	98.8%	81.6%	98.6%	90.8%	83.3%	86.3%	95.0%

Table 5 Improvement

Projects	No Sliding Window		Sliding Window	
	Precision	Recall	Precision	Recall
Antlr	86.3%	89.9%	87.9%	90.0%
FpML	90.3%	96.7%	96.6%	98.8%
Log4j/4net	71.4%	71.4%	67.8%	81.6%
Lucene	90.0%	97.8%	95.0%	98.6%
Spring	68.6%	69.2%	71.2%	90.8%
Average	87.4%	93.2%	96.1%	93.6%

smaller than the whole token streams and are not sufficient in determining whether the whole token streams are similar, leading to more false positives in our reports sometime.

4.2 RQ2. Comparison with Token-Based Clone Detection

The existing clone detection tools aim at single-language clones, but a few token-based clone detection tools [19] can treat inputs as plain texts without language-specific lexical or syntactical information to detect some cross-language clones (*e.g.*, *CCFinder* [7] and *ConQAT* [9]). In this experiment, we compare *CLCMiner* with *CCFinder* and *ConQAT*.

4.2.1 Setup

We set all the *diffs* of 10 projects as the input for the clone detection tools and compare the cross-language clones that they detect from the *diffs*. We configure the tools as follows:

CLCMiner. The threshold distance of *CLCMiner* is set as 0.5. *CLCMiner* is set to report a *diff* and its neighbor as a clone pair if their *diff* distance is equal to or less than 0.5 no matter whether the neighbor is the nearest one or not. To speed up the *diff* matching, *CLCMiner* is set to slide the *sliding window* 10 tokens per step.

CCFinder. The arguments of *CCFinder* is set as default. That is the minimum number of tokens is 50 and that the minimum number of kinds of tokens in code fragments (metric TKS) is 12. The *diffs* are divided into two groups (*i.e.*, Java group and C# group). *CCFinder* is set to detect code clones between *diffs* from the distinct *diff* groups but not to detect code clones between *diffs* in the same group.

ConQAT. The gapped ratio of *ConQAT* is set as 0.2, which can make the number of clones reported is twice as that when the gapped ratio is set as 0. The minimum number is set as 5 and the max errors are set as 3. Instead of clone pairs, *ConQAT* reports clone groups, which may include more than two *diffs*. In order to facilitate the comparison, we separate each clone group into clone pairs, in which one is a Java *diff* and the other is C# one.

In addition, we run *CCFinder* and *ConQAT* to detect cross-language clones from files having the same name only as *CLCMiner* does (*cf.* Line 6 in Algorithm 1).

4.2.2 Result

Table 6 lists the number of cross-language clones reported by each tool. We can see that totally *CLCMiner* can detect 1,403,069 pairs of cross-language clones from the 10 projects, while *CCFinder* and *ConQAT* can detect 10,153 and 93,833 pairs respectively. Note that the numbers of the reported clone pairs are much larger than that of the *diffs* because, based on the above tool configurations, each *diff* may appear in more than one clone pair.

The number of clone pairs reported by both *CLCMiner* and *CCFinder* ($\cap_{12}$) is 9,730, which means 9,730 out of 10,153 (95.8%) clones reported by *CCFinder* are also reported by *CLCMiner*. The number of clone pairs reported by both *CLCMiner* and *ConQAT* ($\cap_{13}$) is 91,244, which means 91,244 out of 93,833 (97.2%) clones reported by *ConQAT* are also reported by *CLCMiner*. The number of clone pairs reported by both *CCFinder* and *ConQAT* ($\cap_{23}$) is 5,463 and the number of clone pairs reported by all the three tools ($\cap_{123}$) is 5,418, which means 5,418 out of 5,463 (99.1%) of clone pairs reported both by *CCFinder* and *ConQAT* are also reported by *CLCMiner*. The result indicates that *CLCMiner* can detect the cross-language clones in the *diffs* effectively.

4.2.3 Discussion

Why does *CLCMiner* detect more cross-language clones? Token-based single-language clone detection tools lex each line of source files into token sequence and utilizes certain string matching algorithm to search for similar subsequences, while *CLCMiner* splits each camel case identifier (*e.g.*, variable names and method names) and utilizes the statistical method to calculate the distance between *diffs* and search for similar *diffs*. In this way, *CLCMiner* does finer grained comparison than these tools.

Although *CLCMiner* can detect most of the ones reported by *CCFinder* and *ConQAT*, some are still missed by *CLCMiner*. We investigate those clones missed by *CLCMiner* and summarize potential causes as follows:

- Since the sizes of our *sliding windows* are fixed to be the same as the shorter of the two *diff* token streams, *CLCMiner* can miss clones that are only a small part of the two *diffs*. On the contrary, *CCFinder* and *ConQAT* are able to report any subsequences of the token stream as clones as long as they are bigger than the minimum number of tokens required.
- *CLCMiner* uses the distance threshold 0.5 to report potential clones; it will miss cross-language clones whose distances are large than 0.5 (cf. Figures 3 and 4). *CCFinder* and *ConQAT* may be able to catch some of the *CLCMiner's* false negatives.
- For performance issues, our *sliding windows* are moved forward 10 tokens at each step, which may miss the shortest distance that should have been less than 0.5.

Table 6 No. of Cross-Language Clones Reported

Projects	*CLCMiner*	*CCFinder*	*ConQAT*	$\cap_{12}$	$\cap_{13}$	$\cap_{23}$	$\cap_{123}$
Antlr	63,337	1,099	321	1079	314	81	81
codorva	4,863	290	147	127	144	0	0
DataStax	21,927	70	192	57	168	31	31
Factual	2,228	0	2	0	2	0	0
FpML	43,525	1,587	2,096	1,571	2,051	676	663
Log4j/4net	6,251	12	256	12	90	8	8
Lucene	1,197,108	6,830	88,756	6,631	86,607	4,531	4,506
Spring	32,927	38	753	34	667	16	16
ua-parser	82	0	0	0	0	0	0
zeromq	30,821	227	1,310	219	1,201	120	113
Total	1,403,069	10,153	93,833	9,730	91,244	5,463	5,418

$\cap_{12}$: intersection results of *CLCMiner* and *CCFinder*;
$\cap_{13}$: intersection results of *CLCMiner* and *ConQAT*;
$\cap_{23}$: intersection results of *CCFinder* and *ConQAT*;
$\cap_{123}$: intersection results of *CLCMiner*, *CCFinder* and *ConQAT*

4.3 RQ3. Impact of More Attributes of Diffs

For matching *diffs*, BOW as used in Section 3.3.2 may not be the only choice. We identify the following attributes that may have an impact on the similarity among *diffs* too: *commit author* (CA), *commit date* (CD), and *commit message* (CM)) of the *diffs*. In this subsection, based on the sampled and labelled clone candidates, we analyze the potential impact of these attributes and discuss how to improve the effectiveness of matching cross-language clones in future work.

4.3.1 Setup

Intuitively, the attributes CA, CD and CM of *diffs* tend to have some correlations with the *diff* similarity. As a developer may have a programming style that may persist even across different languages, a pair of similar *diffs* from different language versions of a project may be more likely to committed by the same developer. 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 interval. As a commit message often summarizes the changes in the commit, a pair of similar *diffs* may be more likely to share similar commit messages. Based on the intuitions, here we aim to test the (in)validity of our *null hypotheses* as follows:

- H_{a0}**:** Similar *diffs* and dissimilar *diffs* have the same probability to be committed by the same author.
- H_{d0}**:** The interval between the commit dates of similar *diffs* is likely to be as long as that of dissimilar ones.
- H_{m0}**:** The distances between the commit messages of the similar *diffs* is likely to be as large as that of the dissimilar ones.

To investigate these hypotheses, we look into the labels for the clone reports of the 10 projects sampled in the way mentioned in Section 4.1.1. For each hypothesis, we build two variables: one is a label (l) indicating whether the *diffs* are similar, and the other is the value of the corresponding attribute (ca, cd or cm). For variable l, $l = 1$ means the pair of *diffs* is similar and $l = 0$ means it is not. For variable ca, $ca = 1$ means the pair of *diffs* is committed by the same author and $ca = 0$ mean it is not. Variable cd is the interval between the commit dates of two *diffs*. Variable cm is the distance between commit messages of two *diffs*.

Table 7 T-test Result

Hypothesis	H_{a0}		H_{d0}		H_{m0}	
Label (l)	0	1	0	1	0	1
Observations	1,553	1,541	1,553	1,541	1,553	1541
Mean	0.077	0.296	790	585	0.806	0.623
t-statistic	-16.272		-8.211		18.452	
p-value	< 0.0001		< 0.0001		< 0.0001	

4.3.2 Result

Table 7 lists the *t-test* statistics of the three hypotheses. Based on the *t-statistic* for each hypothesis, all the null hypotheses are rejected (p-value < 0.0001). This means the attribute CA, CD, CM have some correlation ship with the similarity of their corresponding *diff* pair.

We find that about 29.6% pairs of similar *diffs* are committed by the same author, but only 7.7% pairs of dissimilar *diffs* are committed by the same author. Therefore, pairs of similar *diffs* tend to be committed by the same author. For H_{d0}, pairs of similar *diffs* are committed 585 days after one another on average, while pairs of dissimilar *diffs* are committed 790 days after one another on average. Therefore, similar *diffs* tend to be committed between a shorter period of time. For H_{m0}, the distances between the commit messages of similar *diffs* tend to be shorter.

In addition, the Pearson's correlation coefficient between l and ca is 0.28, which indicates that the *diffs* committed by the same author are more likely to be clones than those committed by different authors. The coefficient between l and cd is -0.14, which indicates that *diffs* committed between a shorter period of time are more likely to be clones than those committed between a longer period of time. The coefficient between l and cm is -0.31, which indicates that *diffs* with similar commit messages are more likely to be clones than those with dissimilar commit messages.

4.3.3 Discussion

The above statistics are aggregated from 10 projects which can differ from one to another. The correlations between the attributes and the *diff* similarity are weak, indicating none of the attributes is a deciding factor for *diff* pairs to be clones. Whether a *diff* pair is clones could be a combined effect of all the attributes and even some contexts beyond *diff*s.

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 in code change histories.

5. Discussion and Future Work

We realize that our approach is subject to various threats to validity in its algorithm design, experimental settings, and generalizability. In the following, we discuss several of such threats and propose possible mitigations.

Mapping clones in change histories to the latest revision. Since *CLCMiner* detects clones in *diffs* and the code corresponding to some of the *diff* clones may be changed repeatedly or even deleted along software evolution, it will cause some deleted code to be reported as clones or the same piece of code to be reported repeatedly in many different *diff* clone pairs. On one hand, clones occurred repeatedly or deleted in history may still be useful for the purpose of studying code evolution, refactoring, and consistency (e.g., [12–15]); on the other hand, developers working on the latest revision of a project may not need deleted or overlapping clones, and removing such clones from the clone reports can improve the usefulness of *CLCMiner* for such developers. To address the concern on different use cases for *diff* clones, we further track them to check whether they still exist in the latest revision. We compare the historical file containing each *diff* clone with the file in the latest revision, by the *diff* tool, to build an existence mapping for all lines of code in the *diff* clone. Based on the mapping, we classify *diff* clones into 5 categories: 1) the clones whose containing files no longer exist in the latest revision (**NF**); 2) the clones none of whose lines of code exists in the latest revision ($\mathbf{R}_1$); 3) the clones less than 50% of whose lines of code are still in the latest revision ($\mathbf{R}_2$); 4) the clones over 50% but not all of whose lines of code are still in the latest revision ($\mathbf{R}_3$); 5) the clones all of whose lines of code are still in the latest revision ($\mathbf{R}_4$).

We categorize all the clones reported in Table 6 into Table 8. It shows that, along software evolution, on average 65.8% (NF and R_1) of the *diff* clones no longer exist in the latest revision, while 34.2% (R_2, R_3 and R_4) still exist or partially exist. In particular, over 50% of lines of code in 7% (R_3 and R_4) of the *diff* clones still exist.

Furthermore, totally the 10 projects have 3,602,358 lines in their latest revisions (including comments), 805,960 (about 22.4%) of which can be mapped from *diff* clones, which may be used as an alternative way to detect many clones in the latest revisions. Column "LR LOC" shows the number of lines in the latest revision that can be mapped from at least one *diff* clone, and Column "DC LOC" shows the total number of lines from the *diff* clones in R_2, R_3 and R_4. We use the ratio (OR= $\frac{\text{"DC LOC"}}{\text{"LR LOC"}}$) as an estimate for the average number of reported *diff* clones (6.3 on average) that overlap with each clone pair mapped to the latest revision.

In brief, many of detected *diff* clones remain and can help find similar code in different programming languages in the latest revisions; many others are deleted or overlapped with others, but can be useful for tasks related clone changes.

Using comments in code. In diff normalization (Section 3.2), code comments were removed as we hypothesized that comments in natural language may be too high-level and appear similar even for non-clones and thus are not accurate enough for clone detection. However, during the manual labelling of the sampled *diff* pair reports, we noticed that many clone pairs either contain quite different comments for different parts of the two code fragments in the pair or contain almost exactly the same comments (which may indicate an actual copying-pasting operation). In our future work, we plan to more systematically investigate how comments in code are related with clones.

Table 8 Distributions of *diff* Clones in the Latest Revision

Projects	NF	R_1	R_2	R_3	R_4	LR LOC (Java + C#)	DC LOC (Java + C#)	OR
Antlr	2.4%	0.7%	49.1%	32.3%	15.4%	55,295	198,649	3.6
codorva	100%	0	0	0	0	0	0	NA
DataStax	25.5%	9.2%	56.2%	7.1%	1.9%	14,949	96,337	6.4
Factual	5.9%	3.3%	66.2%	18.6%	5.9%	3,096	11,849	3.8
FpML	22.2%	2.3%	36.9%	19.1%	19.6%	54,724	247,839	4.5
Log4j/4net	26.9%	5.0%	51.8%	12.9%	3.4%	22,160	114,209	5.2
Lucene	40.9%	31.1%	24.5%	2.8%	0.7%	570,582	3,944,655	6.9
Spring	55.9%	3.5%	26.7%	10.2%	3.8%	65,397	345,958	5.3
ua-parser	100%	0	0	0	0	0	0	NA
zeromq	32.7%	12.7%	52.2%	2.0%	0.4%	19,757	140,594	7.1
Ave.	38.6%	27.2%	27.2%	4.9%	2.1%	805,960	5,100,087	6.3

NF: No Files; $\mathbf{R}_1$: [0, 0]; $\mathbf{R}_2$: (0, 50%); $\mathbf{R}_3$: [50%, 100%); $\mathbf{R}_4$: [100%, 100%];

Relaxing filenames. Diff matching (Section 3.3.2) used a requirement that potentially matched diffs should be from files of the same name, and thus all code in every reported clone pair has the same file name. However, cross-language clones can appear in files with different names, especially if they are from different projects. The setting was added based on the heuristic that implementations of similar functionalities in different languages *within the same project* are likely to be in files of the same name and to reduce the pair-wise comparison time for projects involving too many commits; it is a trade-off between detection efficiency and recall. In the future work, we will optimize our matching algorithm and analyze how the file names impact cross-language clones that may be from different projects.

Detecting clone groups and change propagation. *CLCMiner* matches a *diff* in one language to its nearest neighbors in another language only, as we focus on the feasibility of using *diffs* for detecting cross-language clones. We can change the setting to return all the neighbors of a *diff* whose distance is within a small threshold, which can enable us to detect cross-language clone groups, in addition to pairs. Also, by linking clone groups based on clone transitivity within a threshold and complemented with a single-language detector, we will be able to study how changes are propagated even through different languages, extending similar studies within the same language [20].

Detecting clones beyond revision histories. *CLCMiner* is based on revision histories; it is limited to detect cross-language clones that have been changed in the past in the same project. For clones that are never changed, we can explore more language attributes that can identify clone relations (e.g., using deep learning to build vector representation of programs [21]) across languages. We also believe this limitation can be compensated by a single-language detector that can detect cross-project and same-language clones based on certain clone transitivity across projects and languages.

Crossing more languages. Increasing demands for cross-platform mobile applications (*e.g.*, iOS and Android) raise the need for quick development that can reuse code across more diverse kinds of languages (*e.g.*, Objective-C, Swift, and Java). Since functionalities implemented in one programming language can be used as a reference for the

implementation in another language, code fragments implementing similar functionalities in different languages would be changed in a similar way. We believe there exist some alignments between the changed code as long as the changes in different languages use similar lexical features, such as identifier names. In our future work, we plan to adapt *CLCMiner* to more languages and explore more attributes that can identify similar changes and be used to detect clones and facilitate code reuse across different languages.

Handling false positives. Although the precisions of the results reported by *CLCMiner* are relatively high, there is still space for improvement. We investigated the false positives and found they may have various characteristics causing "accidental similarity" among *diffs*: 1) a short method is defined in one *diff* but invoked in the other *diff*; 2) the *diffs* contain code that handles exceptions or errors; 3) the *diffs* contain a number of same string constants used differently; 4) the *diffs* contain a number of different numeric values which were excluded by our normalizing step; 5) the *diffs* contain code that uses the same set of library functions (e.g., File I/O, HttpHeaders) in different ways. In our future work, we will refine *CLCMiner* to handle such cases.

6. Related Work

Cross-language clone detection. The number of various software systems implemented in multiple languages is increasing considerably [22], but cross-language clone detection is limited. Kraft *et al.* [3] 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.* [10] 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. Nakamura *et al.* [23] detect *interlanguage clones* that are clones whose code may be in more than one programming languages (*e.g.*, a web page containing both HTML and Javascript), while each of our cross-language clones is still code in one language only.

Vocabulary similarity. Vocabulary similarity is an efficient way for semantic similarity. Marcus *et al.* [24] apply Latent Semantic Indexing (LSI) to source code and its associated internal documentation (*e.g.*, comments) and could detect high-level concept clones with low costs. Kuhn *et al.* [25] introduce Semantic Clustering, which is also based on LSI to group source artifacts that use similar vocabulary. Semantic clustering captures topics regardless of class hierarchies, packages, and other structures. Lucia *et al.* [26] leverage information retrieval techniques such as VSM, LSI, and LDA to pick terms from specific parts of source code and comments for source code labeling. They can efficiently identify and cluster topics in the source code. Since it is difficult to analyze *diffs* for different programming languages by traditional program analysis tools, our approach applied vocabulary similarity to measure the *diff* similarity.

Data mining in VCS. There are considerable studies of data mining in Version Control Systems (VCS). Zimmermann *et al.* [27] apply data mining on version histories to recommend related syntactic changes. Gîrba *et al.* [28] apply concept analysis on VCS to identify groups of co-changes. McIntosh, *et al.* [29] mine source and test code for accompanying build changes. We apply data mining on VCS for a different purpose, detecting cross-language clones.

7. Conclusion

This paper proposes a novel approach, *CLCMiner*, that detects cross-language clones without common intermediate languages. Our key new idea is to utilize *diff* similarity. We have implemented and evaluated its prototype on 10 open source projects. The results show that our approach can detect many cross-language code clones that appear in *diff*s in the revision histories with a high precision of 89.1% and a high recall 95% on average.

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 5.

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.

References

[1] "Antlr." `http://www.antlr.org`.
[2] "Lucene." `http://lucene.apache.org`.
[3] N.A. Kraft, B.W. Bonds, and R.K. Smith, "Cross-language clone detection," Proc. SEKE, pp.54–59, 2008.
[4] R. Fanta and V. Rajlich, "Removing clones from the code," J. of Software Maintenance, vol.11, no.4, pp.223–243, 1999.
[5] C.J. Kapser and M.W. Godfrey, ""cloning considered harmful" considered harmful: Patterns of cloning in software," Empirical Softw. Engg., vol.13, no.6, pp.645–692, Dec 2008.
[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," ICSE, pp.96–105, 2007.
[7] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," TSE, vol.28, no.7, pp.654–670, 2002.
[8] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," Proc. ISSTA, pp.81–92, 2009.
[9] E. Jürgens, F. Deissenboeck, and B. Hummel, "CloneDetective - A workbench for clone detection research," ICSE, pp.603–606, 2009.

[10] F. Al-Omari, I. Keivanloo, C.K. Roy, and J. Rilling, "Detecting clones across microsoft .net programming languages," Proc. WCRE, pp.405–414, 2012.

[11] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "Mining revision histories to detect cross-language clones without intermediates," ASE, pp.696–701, 2016.

[12] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," IWSC, pp.7–13, 2011.

[13] M. Kim, V. Sazawal, D. Notkin, and G.C. Murphy, "An empirical study of code clone genealogies," ESEC-FSE, pp.187–196, 2005.

[14] J. Krinke, "A study of consistent and inconsistent changes to code clones," WCRE, pp.170–178, 2007.

[15] F. Zhang, S.C. Khoo, and X. Su, "Predicting consistent clone change," Proc. ISSRE, 2016 (to appear).

[16] C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. Germán, and P.T. Devanbu, "The promises and perils of mining git," MSR, pp.1–10, 2009.

[17] Z.S. Harris, "Distributional structure," Word, vol.10, no.2-3, pp.146–162, 1954.

[18] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Trans. Software Eng., vol.33, no.9, pp.577–591, 2007.

[19] C.K. Roy and J.R. Cordy, "A survey on software clone detection research," Queen's Univ., SoC, vol.541, no.115, pp.64–68, 2007.

[20] S. Wang, D. Lo, and L. Jiang, "Understanding widespread changes: A taxonomic study," 17th CSMR, pp.5–14, 2013.

[21] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," Knowledge Science, Engineering and Management (KSEM), pp.547–553, 2015.

[22] K. Kontogiannis, P.K. Linos, and K. Wong, "Comprehension and maintenance of large-scale multi-language software applications," Proc. ICSM, pp.497–500, 2006.

[23] Y. Nakamura, E. Choi, N. Yoshida, S. Haruna, and K. Inoue, "Towards detection and analysis of interlanguage clones for multilingual web applications," Proc. IWSC, pp.17–18, IEEE, 2016.

[24] A. Marcus and J.I. Maletic, "Identification of high-level concept clones in source code," Proc. ASE, pp.107–114, 2001.

[25] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," Information & Software Technology, vol.49, no.3, pp.230–243, 2007.

[26] A.D. Lucia, M.D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: an empirical study," ESEM, vol.19, no.5, pp.1383–1420, 2014.

[27] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," ICSE, pp.563–572, 2004.

[28] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and D. Ratiu, "Using concept analysis to detect co-change patterns," Proc. ESEC/FSE, pp.83–89, 2007.

[29] S. McIntosh, B. Adams, M. Nagappan, and A.E. Hassan, "Mining co-change information to understand when build changes are necessary," Proc. ICSME, pp.241–250, 2014.

Xiao Cheng is currently a Ph.D. candidate in Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests includes code clone detection, code clone synchronization, software mining, software maintenance and code search.

Zhiming Peng received the Bachelor's degree in Computer Science from Zhejiang University. Currently he works as a Research Engineer in Singapore Management University. His research interests includes machine learning, computer vision and complex networks.

Lingxiao Jiang received the Ph.D. degree in the Department of Computer Science at University of California, Davis (2003-2009). He worked as a test strategist at Nvidia for half a year before He joined the faculty of School of Information Systems at Singapore Management University in November 2009. His research interests are mainly in software engineering and program analysis.

Hao Zhong received the Ph.D. degree from Peking University. After receiving his Ph.D., he joined Institute of Software, Chinese Academy of Sciences as an assistant professor in 2009. He was promoted as an associated professor in 2011. He was a visiting scholar with University of California, Davis (2012-2014). In 2014, he joined the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include program analysis, software maintenance, and software mining.

Haibo Yu received the Ph.D. degree in computer science from Kyushu University, Japan, in 2009. She joined the School of Software at Shanghai Jiao Tong University in March, 2010 as an assistant professor. Her research interests include information retrieval, web application systems and software engineering.

Jianjun Zhao received the Ph.D. degree in Computer Science from Kyushu University (Japan) in 1997. After receiving his Ph.D., he joined the Department of Computer Science and Engineering, Fukuoka Institute of Technology (Japan) as an Assistant Professor, and then was promoted to be an Associate Professor in 2000. In November 2005, he joined the School of Software, and then the Department of Computer Science and Engineering, Shanghai Jiao Tong University (China) as a Professor. Since April 2016, he has been with the Department of Advanced Information Technology, Kyushu University (Japan) as a Professor. His research interests include program analysis for software engineering and compiler optimization.