IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2020.DOI

LVMapper: A Large-variance Clone Detector Using Sequencing Alignment Approach

MING WU[1,2], PENGCHENG WANG[1,2], KANGQI YIN[1,2], HAOYU CHENG[1,2], YUN XU[1,2], AND CHANCHAL K. ROY[3]

[1]School of Computer Science, University of Science and Technology of China, Heifei, Anhui 230027, China (e-mail: {wuming, wpc520, yinkq, chhy}@mail.ustc.edu.cn, xuyun@ustc.edu.cn)
[2]Key Laboratory on High Performance Computing, Anhui Province
[3]Department of Computer Science, University of Saskatchewan, Canada (e-mail: chanchal.roy@usask.ca)

Corresponding author: Yun Xu (e-mail: xuyun@ustc.edu.cn).

This work was partially supported by the National Nature Science Foundation of China under the grant No. 61672480 and the 111 Project 2.0 of China under the grant No. BP0719016.

ABSTRACT To detect large-variance code clones (i.e. clones with many modifications) in large-scale code repositories is difficult because most current tools can only detect almost identical or very similar clones. It has an important impact on downstream software applications such as bug detection, code completion, software analysis, etc. Recently, CCAligner made an attempt to detect the code clones with insertions or deletions in one place, which were called *large-gap clones*. Our contribution is to develop a novel and effective detection approach of large-variance clones to more general cases for not only the concentrated code modifications but also the scattered code modifications. A detector named LVMapper is proposed, borrowing and changing the approach of sequencing alignment in bioinformatics which can find two similar sequences with more differences. The ability of LVMapper was tested on 8 open source projects datasets, and the results show that LVMapper detected more than 5 times of large-variance clones compared with other state-of-the-art tools including CCAligner. Furthermore, our new tool also presents comparable recall for general Type-1, Type-2 and Type-3 clones with precision of 88.5% on the widely used benchmarking dataset BigCloneBench.

INDEX TERMS clone detection, large-variance clone, dynamic threshold, sequencing alignment

I. INTRODUCTION

CLONE code is generated by copying, pasting and modifying code fragments for reuse, which are common in software development [1], [2]. In the past, code clones with many modifications (called *large-variance code clones*) were difficult to be found by existing tools [3] because most of them were suitable for finding the identical or very similar clones. In our experimental observation, large-variance code cloning is ubiquitous. These large-variance clones can be applied to many software applications such as software maintenance, understanding code quality, plagiarism detection, copyright infringement investigation, software evolution analysis, virus detection or detecting bugs. And these tasks do require the extraction of syntactically similar code fragments [4]–[6] which essentially implies that large-variance clones should be detected firstly. For example, programmers write some key statements of code and search for extensions of the partial code in the existing code corpus. The partial code and the extensions are actually large-variance clones. The recent work CCAligner [3] detects large-gap code clones with insertions or deletions in one place which they called *gap*, and the gap in clones often result in large difference in code size. Our study focuses on a more general case that is to detect large-variance code clones that include not only the clones with such concentrated code modifications but also those with scattered code modifications.

It is common that programmers copy code fragments and paste them with many scattered modifications. For example, in software source code evolution, code fragments are modified usually in multiple places rather than only one place where the clones with scattered modifications are generated. Similar to large gap in clones which causes large difference

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

in code size, the scattered modifications can also lead to large difference in size. Fig. 1 shows an example of clones between two different versions of project *Ant 1.6.5* and *1.10.5*. In code B, the code segments of lines 1–2, 6, 8–10, 18–20, 27–28 are the same as those of the lines 1–2, 7, 9–11, 13–15, 18–19 in code A, respectively. Lines 4–5, 7, 14 and 21 in code B are modified from lines 3–5, 8, 12 and 16 in code A. Other lines in code B are new extension part of code A. These modifications in clone codes are scattered and lead to a certain portion of difference in code size.

Existing tools have made attempts but still have more or less limitations in finding Type-3 *(syntactic similarity)*, especially large-variance code clones. For one of the best tools with good performance on Type-3 clone, SourcererCC [7] has to decrease the threshold of similarity to find large-variance clones at the cost of accuracy loss [3]. Another popular detector CCFinderX is good at identifying Type-1 *(textual similarity)* clones and Type-2 *(lexical similarity)* clones, but cannot directly support Type-3 clone detection [8]. iClones identifies the Type-3 clones by merging the nearby small Type1-2 clone fragments, but the recall of Type-3 is low [3] due to the simple strategy. ConQat [9] searches approximate substrings on a suffix tree but the Type-1 recall measured by BigCloneBench was low [10]. NiCad uses a Longest Common Sub-sequence (LCS) algorithm and can tolerate discontinuous subsequences. However, it does not scale and its precision suffers with decreasing thresholds [3]. CCAligner is a good recent attempt in detecting clones with relatively concentrated modifications. It can detect large-gap but misses scenarios where modifications are scattered. Besides, some semantic methods have certain ability to detect variance clones because there is an overlap between semantic clones and syntactical clones. Deckard [11] builds the characteristic vectors from abstract syntax tree (AST) to detect clones, but suffers from low precision and recall rate [7]. Deep learning methods such as Oreo [12] encode software metrics into vectors and achieve good results, but these methods are dependent on the initial training data.

For these considerations, we present a tool aimed at detecting large-variance code clones called LVMapper. Our proposed code clone detector that can find clones with more general variance is based on locate-filter-verify method. Its key idea mainly comes from third-generation sequencing alignment method [13]–[15]. In bioinformatics, the third-generation sequencing alignment based on seed-and-extend strategy performs well with sequence difference up to 30%. LVMapper uses small windows of continuous lines (called *seeds*) with lower costs to locate and filter the candidate pairs of clone codes. In order to verify whether these candidate pairs of codes are cloned, another feature that code clones always have certain proportion of order-preserving code lines is considered. Based on this property, a heuristic algorithm which is more efficient than the Longest Common Subsequence (LCS) algorithm is proposed. Besides, a dynamic threshold that changed with the code size is used for the verification of code clones. It makes LVMapper identify clones with more modifications while guaranteeing certain precision.

To evaluate the large-variance clone detection performance of LVMapper, we compared our tool's performance with NiCad, SourcererCC, CCAligner and Oreo on 4 Java and 4 C projects, respectively. We also used the BigCloneBench [10], [16] to compare and measure the different type clones recall of LVMapper with CCFinderX [8], iClones [17], Deckard [11], NiCad [18], SourcererCC [7], CCAligner [3] and Oreo [12]. Besides, we carried out scalability experiments on datasets ranging from 1M LOC to 250M LOC. The experiments show that LVMapper performed the best in detecting large-variance clones and had comparable recall and precision for general Type-1 to Type-3 clones. It was the fastest for the 1M LOC to 30M LOC datasets and was also scalable to 250M LOC dataset.

The main contributions of our work are as follows:

(1) Goal contribution: CCAligner has advantages in detecting large-gap clones while our work extends the detection approach of large-variance clones to more general cases. It identifies not only the clones with concentrated code modifications but also the clones with the scattered code modifications. We also give a concrete definition of the large-variance clones.

(2) Method contribution: Inspired by the idea of the seed-and-extend method in bioinformatics, we develop a novel tool with locate-filter-verify procedure and it is suited to detect clone with large variance. We propose a heuristic algorithm to rapidly verify the code pairs and use a dynamic threshold to promote the accuracy and recall.

(3) Result contribution: We compared LVMapper with other state-of-the-art detectors on real cases of software projects and the state-of-the-art benchmarks. The results show that LVMapper had the capability of detecting more than 5 times of large-variance clones compared with other state-of-the-art tools with average precision of 88% on 8 open source projects dataset. In addition, our new tool has comparable recall and precision for general Type-1, Type-2 and Type-3 clones.

The rest sections of the paper are organized as follows. Some terminologies and definitions of code clone are introduced in Section II. Section III provides the details of our detection tool. Section IV presents the results of the experiments to evaluate the detection ability of our approach. The threats to validity is described in Section V, and the related work of clone detection is discussed in Section VI. Finally, Section VII concludes the paper and briefly introduces the future work.

II. TERMINOLOGIES & DEFINITIONS

Code block is a statement sequence within braces and usually represents a single function. *Clone pair* is a pair of similar code portions. The *minimum clone size* is the minimum number of lines or tokens that each code of a clone pair should have. The standard minimum clone size is 6 lines or 50 tokens which we also follow in this paper. Four primary

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access


```
1  protected String getPrompt(InputRequest request) {
2    String prompt = request.getPrompt();
3    if (request instanceof MultipleChoiceInputRequest) {
4      StringBuffer sb = new StringBuffer(prompt);
5      sb.append("(");
6      Enumeration e = ((MultipleChoiceInputRequest)request)
         .getChoices().elements();
7      boolean first = true;
8      while (e.hasMoreElements()) {
9        if (!first) {
10         sb.append(",");
11       }
12       sb.append(e.nextElement());
13       first = false;
14     }
15     sb.append(")");
16     prompt = sb.toString();
17   }
18   return prompt;
19 }
```

A

```
1  protected String getPrompt(InputRequest request) {
2    String prompt = request.getPrompt();
3    String def = request.getDefaultValue();
4    if (request instanceof MultipleChoiceInputRequest) {
5      StringBuilder sb = new StringBuilder(prompt).append(" (");
6      boolean first = true;
7      for (String next : ((MultipleChoiceInputRequest) request).getChoices()) {
8        if (!first) {
9          sb.append(", ");
10       }
11       if (next.equals(def)) {
12         sb.append('[');
13       }
14       sb.append(next);
15       if (next.equals(def)) {
16         sb.append(']');
17       }
18       first = false;
19     }
20     sb.append(")");
21     return sb.toString();
22   }
23   else if (def != null) {
24     return prompt + " [" + def + "]";
25   }
26   else {
27     return prompt;
28   }
29 }
```

B

FIGURE 1. Example of a large-variance clone.

clone types are agreed by researchers and the former work [1], [19]:

Type-1 (textual similarity) and *Type-2 (lexical similarity)* clones are syntactically identical code fragments except for variances in white space, layout, comments and variances in identifier names, literal values, white space, layout and comments, respectively. *Type-3 (syntactic similarity)* clones are code fragments which are similar but have statements added, modified and/or removed with respect to each other. *Type-4 (semantic similarity)* clones are code fragments that implement the same functionality but are different in syntax.

Type-3 and Type-4 clones are difficult to partition because there is no clear boundary between syntactically similar Type-3 clone and dissimilar Type-4 clone. Hence, BigCloneBench [16] further divided Type-3 and Type-4 into four types according to the syntactical similarity range: Very Strong Type-3 similarity in range [0.9, 1.0), Strongly Type-3, [0.7, 0.9), Moderately Type-3, [0.5, 0.7), and Weakly Type-3&4, [0.0, 0.5).

Like the the large-gap clones described in [3], for the clones with scattered modifications which lead to large difference in size, we give the definition of large-variance clones quantitatively as follows:

Given two code blocks A and B consisting of $l(A)$ and $l(B)$ pretty-printed lines, respectively. Assuming that $l(A) \leq l(B)$, let $\lambda = l(A)/l(B)$. If code blocks A and B are clone and $\lambda \leq 0.7$, then A and B are called large-variance clone (abbreviated as *LV clones*).

The threshold of λ for large-variance clones is set as 0.7. In intuition, given code A with m lines, if code B is copied from A with *m/2* inserted lines (which are half of the original size), $\lambda = l(A)/l(B) = 2/3 \approx 0.7$. It's difficult for most code clone detection tools or methods to find such clones, even the better tools that are able to detect Type-3 clones cannot find them well.

III. METHOD

Inspired by the seed-and-extend approach which is typically used in sequencing alignment from bioinformatics [13], [14], [20], we proposed a locate-filter-verify procedure for clone detection. In bioinformatics, the seeding step uses the subsequences of the query to quickly locate exact match in reference and the extending step extends and refines the candidate positions by a dynamic programming alignment. Our method includes three phases: locate-filter-verify. The first two phases are designed to seek out the candidate clone pairs with low cost and high recall. In the last phase we design a heuristic algorithm to further eliminate the false clone pairs and improve the accuracy. The uses of dynamic threshold, seeds index and avoiding time-consuming dynamic programming are the keys and innovations of our method. Fig. 2 shows the general procedure. The rest of this section will provide detail descriptions of each phase.

A. LEXICAL ANALYSIS

Lexical analysis for code includes extracting code blocks from source code and tokenizing the code. TXL [21], which is commonly used in previous tools [3], [7], is adapted to extract code blocks from source code files. After obtaining the code blocks, the tokenizing step that mostly based on

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

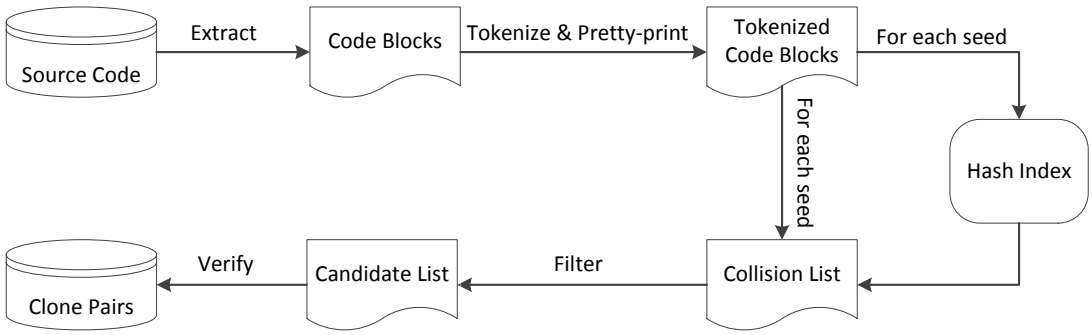

FIGURE 2. General procedure of LVMapper.

Flex [22] begins. Identifiers including variables and function names are replaced by the same token 'id' to tolerate Type-2 changes. The extracted code blocks are pretty-printed. The tokens of each line are concatenated into a single token sequence except the white spaces.

B. SEEDS INDEXING

It is necessary to establish a seed index to speed up the computation for the locating and filtering phases, where the *seeds* are all of the k-line sliding windows (i.e., code fragments of continuous k lines) for a code block. The seeds are basic units for matching instead of tokens. For example, given a code block with 10 lines and sliding windows size $k = 3$, the number of all seeds is obviously 8. In LVMapper, these seeds are converted to hash value and the seed is also regarded as the hash value.

Here are the detailed steps of indexing. LVMapper scans all code blocks, collects all seeds and indexes them in a hash table. The key of the element in hash table is seed's hash and the value is a set of corresponding block *id*s. Like CCAligner, we use MurmurHash hash function [23] in order to guarantee the efficiency with the low collision rate.

C. LOCATING VIA THE SHARED SEED

The locating phase is a preliminary selection for possible code clone pairs. In this phase, the goal is to collect as many candidates as possible without losing real clone pairs. Because the standard minimum clone size is 6 lines [16], [19], CCAligner chooses 6-line or longer windows to match the possible clone pairs. The experiments in CCAligner [3] shows that 6 lines with 1 mismatch windows balanced recall with precision. However, the cost of this approach is still considerably high. We use a lower cost and more efficient way to achieve this. Here, the 3-line sliding windows are chosen as seeds to collect the possible clone pairs that share seed(s).

For any code segments of 6-line with 1 mismatch that can be found by CCAligner, in LVMapper, they can also be identified by two non-overlapping 3-line exact windows. The reason is that the 6 lines window in CCAligner can be covered by two non-overlapping 3-line windows. According to pigeonhole principle, one line modification affects at most one 3-line window and the other one remains unchanged. As a result, the identification ability of our method is better than that of CCAligner.

Algorithm 1 lists the steps of clone detection process, in which *lines 5–13* belong to the locating phase. To retrieve these seeds, we create a hash table for efficiency. Once the index has been built, the candidates of each code clock can be obtained by utilizing this index. Let the current inquiring code block be A. Every sliding overlapping 3-line window in A is the seed, whose hash value is used to find blocks in hash table with the same key (*line 9*). If the block id (denoted as B) is greater than A, B will be added to *CollisionList (A)* — the block list with same hashing seeds of block A (*line 11*). This practice eliminates the duplication of detecting clone for the same two blocks with reverse order. When B is the current inquiring block, block A is not considered anymore because the pair of A and B has been considered before. After the last seed of current block is queried, the positions in the *CollisionList (A)* are sorted according to block id (*line 12*). Every block B that has the collided seed with A will be further filtered and verified (*line 13*).

D. FILTERING VIA THE COMMON SEEDS NUMBER

Through the locating phase, two code blocks that share common seed(s) may be clone pair. Then we take into account the possibility of these *Collision* code blocks being clone. This phase is called the filtering phase and it picks out candidate clone pairs. It brings two benefits. First, the number of candidate code pairs can be reduced significantly. Therefore the processing time and the false positive rate can also be reduced. Besides, the probability of the candidate pairs being the true clone increase significantly. Our idea of the filtering phase is mainly based on considering the number of shared

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

Algorithm 1: Clone Detection

Input: A is a list of tokenized code blocks $\{a_1, a_2,...a_n\}$, Hash Table H of A, window size k, threshold θ for filtering phase, threshold δ for verifying phase
Output: All clone pairs CP

1 $H \leftarrow \varnothing$;
2 $CP \leftarrow \varnothing$;
3 $len =$ number of lines in a_i;
4 **for** *each* a_i *in* A **do**
/* Locating phase */
5 **for** $j = 1; j \leq len - k + 1; j++$ **do**
6 $l_j = a_i.line j$;
7 $win_j =$ CONCAT$(l_j, l_{j+1}, ...l_{j+k-1})$;
8 $key =$ HASH(win_i);
9 $B =$ FIND(H, key);
10 **if** $B > a_i$ **then**
11 $CollisionList_i = CollisionList_i \cup B$;
12 SORT$(CollisionList_i)$;
13 **for** *each* B *in* $CollisionList_i$ **do**
/* Filtering phase */
14 $s =$ number of B in $CollisionList_i$;
15 $L =$ number of lines in B;
16 $SR = s/(L - k + 1)$;
17 **if** $SR \geq \theta$ **then**
/* Verifying phase */
18 $block1 =$block with smaller size between a_i and B;
19 $block2 =$block with larger size between a_i and B;
20 $line1 = 1$;
21 $comm\_lines = 0$;
22 $lastline = 0$;
23 **while** $line1 \neq block1.end$ **do**
24 $k1 =$ HASH$(block1.line1)$;
/* FIND_IN_BLOCK find the first line after $lastline$ in $block2$ and has the same hash value of $k1$ */
25 $line2 =$ FIND_IN_BLOCK$(k1, block2, lastline)$;
26 **if** $line2 \neq NULL$ **then**
27 $seglen = 1$;
28 **while** $block1.line1 + seglen = block2.line2 + seglen$ **do**
29 $seglen++$;
30 **if** $seglen \geq 2$ **then**
31 $comm\_lines+ = seglen$;
32 $lastline = line2 + seglen - 1$;
33 $line1++$;
34 $min\_lines =$ minimum lines of a_i and B;
35 $OS = comm\_lines/min\_lines$;
36 **if** $OS \geq \delta$ **then**
37 $CP = CP \cup (a_1, B)$;
38 return CP;

seeds for two code blocks to measure the possibility of being clone.

The selection of candidate clone pairs depends on the similarity between the two code blocks, and the similarty is calculated by the number of seeds they share. In our method, the similarity of code pair *A* and *B* is defined as:

$$SR(B|A) = \frac{s}{t} = \frac{s}{L - k + 1} \tag{1}$$

where *s* is the number of shared seeds, *t* is total seeds number of *B* and *L* is the length in line of *B*. For any pair of the collision code blocks, the higher the $SR(B|A)$ value is, the more likely they are to be a clone pair.

Lines 14–17 in Algorithm 1 belongs to filtering phase. In practice, once we get the candidate blocks list *CollisionList (A)* of current inquiring block *A*, for every candidate block *B* in the list, LVMapper counts the number of *B* in *CollisionList (A)* (*line 14*). As mentioned above, we treat each overlapping *k*-line windows as seed to vote for potential clone blocks, then every position added in the collision list is the block that have the same seed with *A*. In this case, the number of *B* in *CollisionList (A)* is the number of votes that *B* gets from *A*. If *B* gets more votes, then it is more likely to be clone code of *A*. The idea is similar to the idea of seed-and-extend in the sequencing alignment [14]. The threshold for $SR(B|A)$ is θ (*line 17*).

E. VERIFYING VIA THE ORDERED COMMON LINES

Unlike the first two phases, the last phase (called the verifying phase) further measures the clone possibility of the candidate clone pairs output by the filtering phase from another perspective: if two blocks are large-variance clone, an important feature of them is that the common lines of code in them have order preserving property. Actually, previous tool NiCad [18] used similar idea. It is based on a Longest Common Sub-sequence (LCS) algorithm. Not as complex as NiCad, we design a heuristic simple algorithm for this order preserving property. The idea of the heuristic algorithm is to count the order preserving number of two adjacent code lines in one code block.

Similarly, the similarity of the candidate pair is measured by another characteristic quantity: the rate of ordered common lines. This characteristic quantity *OS (A, B)* is defined as:

$$OS(A, B) = \frac{comm\_lines}{min\_lines(A, B)} \tag{2}$$

where *common_lines* is the ordered common lines of *A* and *B*, and *min_lines(A, B)* is the minimum size in line of *A* and *B*.

The method of threshold setting for verifying candidate clone pairs is the key point of our method, and it is also significantly different from other methods. In order to enhance the ability of detecting large variance code clones, here we use a dynamic threshold to verify the code pairs. The function of the threshold is designed as a piecewise function, corresponding to the length of code blocks A and B. As the length of code blocks A and B increases, this threshold can be reduced. Actually, the reason can be explained by an understandable analogy. For example, given real-life conversations, how to judge whether two different conversations belong to the same topic? We have such a consensus that long conversations with lower rate of common sentences could discuss the same subject while short conversations should have higher rate to judge as discussing the same subject. Note that SourcererCC uses the fixed ratio of shared tokens to verify clone pairs and it sets the ratio threshold as 0.7 to guarantee the precision. The detailed setting of filtering and verifying phases will be discussed in Section IV.

We implement the heuristic algorithm as follows and lines 18–38 in Algorithm 1 show the steps. For every candidate pair of block *A* and *B* survived from the locating and filtering phase, assume block *A* is smaller than *B*. The variable *lastline*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

records the last line id in *B* that matches the line in *A* during scanning and it is initialized as 0 (*line 22*). LVMapper scans from the line 1 in *A* to find the first line which also appears in *B* and the matching line's id in *B* is greater than *lastline* (*lines 23–25*). And then it continues to scan and calculates the length of the contiguous matching lines (*lines 26–29*). LVMapper keeps a variable *comm_lines* to record the sum of matching lines. If there are at least 2 contiguous shared lines, then the length is added to *comm_lines* (*lines 30–31*). The *lastline* is also updated according to the length of matching lines (*line 32*). The contiguous lines of *A* and *B* are in order and the segments of the contiguous lines are also in order. After scanning, *OS (A, B)* is calculated (*line 35*) and the threshold δ of *OS (A, B)* to verify candidate pairs (*line 36*) is set according to the minimum size of *A* and *B*.

IV. EVALUATION

A. STUDY DESIGN

In the following subsections, we first introduce the parameter setting of seed length and dynamic threshold of LVMapper. We considered and balanced the precision, recall, execution time and memory use of different seed lengths to find a proper setting in LVMapper.

Then the performance of LVMapper for detecting large-variance clones was evaluated on 8 open source projects dataset. As large-variance clone belongs to Type-3 clone, we chose 4 methods which performed good in Type-3 clone detection for evaluation. We extracted large-variance clones from the reported clones according to the definition and compared the reported large-variance clones' number and precision.

Besides, we also evaluated the performance of LVMapper for general Type-1, Type-2, Type-3 clones detection in the state-of-the-art benchmark dataset BigCloneBench [10], [16]. BigCloneBench is a benchmark which contains different types of manually validated clones in the repository IJaDataset-2.0 [24] and it defines clone types by syntactic similarity as described in Section II. The framework BigCloneEval [25] summarizes recall performance for different clone types of clone detectors automatically and it is widely used in previous work [7], [12].

At last we tested the scalability of LVMapper for large-scale clone detection. We constracted different sizes of dataset ranging from 1M LOC to 250M LOC based on the inter-project Java repository IJaDataset [26]. We were interested in the execution time of LVMapper on different sizes dataset, and whether LVMapper was scalable to the 250M LOC large-scale dataset.

B. PARAMETER SETTING

1) Choice of Seed Length

As the seeds play a major role in locating and filtering phases, the choice of seeds length is important and has an impact on the performance of LVMapper. If the seeds are too long, the recall of our method will be affected. In contrast, if the seeds are too short, the effectiveness of the locating and filtering will be eroded. In Section III we analyzed the choice of seed length theoretically. Here we also used experiments to evaluate the performance of detection for different seed lengths k quantitatively. We used the BigCloneBench [10], [16] to evaluate the detection ability of LVMapper for different seed lengths, because it is not only a benchmark for general clones but also contains large-variance clones. Besides, we considered the memory use and execution time of different seed lengths for Linux kernel dataset.

For evaluation of recall and precision, we configured the BigCloneEval with minimum clone size 6 lines and 50 tokens which are consistent with the standard minimum clone size. The seed length of LVMapper was set as 2-line, 3-line and 4-line, with other parameters fixed. The recall was reported by BigCloneEval. And for each parameter, we measured the precision by randomly validating 400 reported clone pairs.

Table 1 shows the detailed results. Because the recall rate of Weakly Type-3&4 is under 1%, we also provided the number of detected clones in the parentheses. As seen from Table 1, the recall of Type-1, Type-2 and Very Strongly Type-3 were nearly 100% for all seed length. When the seed length became longer, the recall of type-3 with lower similarity decreased. For seed length of 4-line, the recall of Strongly Type-3 and Moderately Type-3 was 77% and 16%, respectively. And the number of Weakly Type-3&4 fell to 16701. However, while the recall for seed length of 2-line was the highest, especially in Weakly Type-3&4, the precision declined. The recall and precision for seed length 3-line strike a balance. The number of Weakly Type-3&4 was over 20000 and the precision was kept at 88.5%.

TABLE 1. Recall per Clone Type and Precision Measured for BigCloneBench with Different Seed Lengths

k	2	3	4
Type-1	100	100	100
Type-2	99	99	99
Very Strongly Type-3	98	98	98
Strongly Type-3	82	82	77
Moderately Type-3	20	19	16
Weakly Type-3&4 (Num)	0.3 (27043)	0.3 (23923)	0.2 (16701)
Precision	86.5	88.5	93

Besides, we took into consideration the memory use and execution time of different seed lengths. The Linux kernel 4.18 was used as the target source code and it has 25782 files with 12964738 lines of code (LOC) measured by cloc [27]. As shown in Table 2, the execution time of 2-line method was as much as 8.4 times compared to the execution time of 3-line method and the memory use increased about 548MB. The configuration of seed length 3-line had the least memory use and medium execution time. The execution time of 4-line method was the shortest, but it had more memory requirement. Note that the configuration of seed length 3-line has the least memory use. The reason is that for the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

method using 4-line, larger window space allows greater variation of seed, resulting in greater hash index space. For the method using 2-line, more position values (i.e., the block id and line id corresponding to the seeds) occupy the storage space. Taken together, the seed length of 3-line balanced not only the recall and precision, but also the space and time. Therefore, we selected 3-line as our default configuration.

TABLE 2. Execution Time and Memory Space with Different Parameterizations for Linux 4.18

k	2	3	4
Time	52m 14s	6m 12s	3m 53s
Memory	1394 MB	846 MB	851 MB

2) Threshold for Filtering and Verifying Phase

In the filtering phase, the threshold of $SR(B|A)$, i.e. θ, is set to 0.1 empirically.

In the verifying phase, we use dynamic threshold for judgment of results. The threshold is defined according to the block size in order to be better adapted to code clone judgment.

For the ratio of ordered common sequences in the verifying phase, assume the smaller block of the candidate pair is block A. We adapt a more-refined piecewise function to define the threshold δ, which is used in the verifying phase, according to *A.size l*:

$$\delta = \begin{cases} 0.7 & \text{if } 6 < l \leq 10, \\ g(l) & \text{if } 10 < l \leq 20, \\ 0.4 & \text{if } l > 20. \end{cases} \tag{3}$$

In Equation (3), g(l) = -α·*l*+β relies on size of A. In our implementation, we set α = 0.03, β = 1 empirically. For the smaller blocks whose length is smaller than 10 lines, the ratio of minimum matching continuous lines is 0.7, because in our observation the precision will be slashed when $\delta < 0.7$. For medium size blocks with length from 10 to 30 lines, the ratio of ordered common sequences linearly decreases with the smaller blocks length. As big blocks are more likely to be modified in code clone, the lower limit of δ is 0.4 which allows large-variance for big code blocks and ensures certain accuracy. When the code size is small, there are plenty of statements that have similar forms. So LVMapper filters the smaller candidate block with stricter standards. The use of the dynamic threshold utilizes the characteristic of source code.

C. LARGE-VARIANCE CLONE DETECTION

To test the large-variance clone detection ability, we first compared LVMapper with others in eight open source projects dataset.

Here we evaluated the large-variance clone detection ability and studied the existence and pervasiveness of large-variance clones. For all the methods in the experiments, we calculated the number of reported clone pairs according to the definition of large-variance clones in Section II.

To validate the precision, for each project, if the reported clones were more than 100 pairs, we randomly selected 100 samples from the results to validate whether they were true clone pairs or not. If the reported clone pairs were less than 100, we validated all the pairs. Two judges manually validated the pairs according to the definitions of different clone types. One of the judges was the author of this paper and the other was not. They were kept blind for the pairs from different tools. If there were conflicts, they discussed with each other to resolve the conflicts.

In order to compare the detection ability of LVMapper with the state-of-the-art tools, we selected two clone detection tools SourcererCC and CCAligner, which performed well in Type-3 and large-gap clone detection [3], respectively. The results data of SourcererCC and CCAligner were taken from that study straightforwardly. And we also compared LVMapper with the recent tool Oreo which was based on machine learning. Oreo was executed with the default configuration. We did not provide the result of NiCad here, because NiCad detected almost none of clones with largely different sizes or variances. For all the experiments using these 8 projects, we considered the clones with minimum length of 10 lines, which is consistent with that of the experiments in paper of CCAligner.

The detecting number of large-variance clones (shorted as *LV*) and the precision (shorted as *Prec*) in 8 projects are shown in Table 3. Because Oreo only supported clone detection with Java code, the results of Oreo in 4 C projects were denoted as "–". The number of large variance clones detected by LVMapper was markedly more than that detected by SourcererCC, CCAligner and Oreo. In project JDK1.2.2, the large-variance clones reported by LVMapper are 949 while Oreo reported 194, CCAligner only reported 15 and SourcererCC only reported 4. Oreo were the second best in large-variance clone detection but the portions of large-variance clones that reported by Oreo and LVMapper were 20% or lower. CCAligner performed better than SourcererCC at detecting the clones with largely different sizes, which was in fact the target of CCAligner. For all projects we tested on, the precisions of LVMapper were all above 80% and the average precision was 88%. Among the reported pairs of LVMapper, we found that many clone pairs has scattered modifications and insertions which were missed by Oreo and CCAligner.

We summarized the number of different types clones and the proportion of *LV* clones detected by LVMapper in Table 4. We classified the clone pairs reported by LVMapper to Type-1&Type-2, Type-3 and *LV* clones. As we can see from Table 4, the majority of reported pairs belong to the clones of Type-3. The proportion of the large-variance clones LVMapper reported ranges from 18% to 54% in these projects. There is a high proportion of large-variance clones in open source projects and they should not be overlooked.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

TABLE 3. Large-variance Clone Evaluation Results for 8 projects

Project	LVMapper		Oreo		CCAligner		SourcererCC	
	LV	Prec	LV	Prec	LV	Prec	LV	Prec
JDK 1.2.2	949	88%	194	90%	15	93.3%	4	100%
Ant 1.10.1	447	87%	74	97%	87	88.6%	13	100%
Maven 3.5.0	398	89%	18	100%	217	86.2%	38	97.4%
Opennlp 1.8.1	2779	86%	38	100%	221	92.3%	5	100%
Cook 2.34	651	95%	–	–	63	96.8%	14	100%
Redis 4.0.0	142	88%	–	–	22	90.9%	7	100%
PostgreSQL 6.0	793	90%	–	–	219	94.1%	38	100%
Linux 1.0	620	82%	–	–	27	96.3%	12	91.7%

TABLE 4. Proportion of Large-variance Clones Detected by LVMapper

Project	Type-1&2	Type-3	All	LV	LV/ALL
JDK 1.2.2	1201	3956	5157	949	18.4%
Ant 1.10.1	130	1458	1588	447	28.1%
Maven 3.5.0	484	1220	1704	398	23.4%
Opennlp 1.8.1	191	4898	5089	2779	54.6%
Cook 2.34	138	1751	1889	651	34.5%
Redis 4.0.0	46	460	506	142	28.0%
PostgreSQL 6.0	135	1805	1940	793	40.8%
Linux 1.0	145	1539	1684	620	36.8%

D. GENERAL CLONE DETECTION

Apart from the evaluation of detection ability for large-variance clones above, we also compared the detection performance of LVMapper for general clones (i.e. from Type1 to Type3) with other clone detectors. We used BigCloneEval [25] to test the recall of tools on BigCloneBench [16]. The configuration of NiCad was minimum length 6 lines, similarity threshold 70%, blind renaming and literal abstraction. We used the default configuration of LVMapper, which has seed length of 3-line, and the threshold is described previously. The configuration of SourcererCC was minimum one token and similarity threshold 70%. We set CCAligner with minimum clone size of 6 lines, window size q = 6, edit distance e = 1, and similarity threshold of 60%. The result of Deckard [11], iClones [17] and CCFinderX [8] and Oreo were from [7]. The number of Weakly Type-3&4 of Deckard was estimated by the recall rate in [7]. As iClones and CCFinderX did not perform well in detecting Moderately Type-3 clones, we did not run them for the performance of Weakly Type-3&4 (denoted as "–" in Table 5). We evaluated the clone pairs in BigCloneEval with the setting of considering minimum clone size pretty-printed 6 source lines and minimum clone size 50 tokens.

For the measurement of precision, as a common practice in [3], [7], [12], we randomly picked 400 pairs from the reported clones of each tool and two judges manually validated the true clone pairs as metioned in Section IV-C.

The results for general clone detection performance in BigCloneBench listed in Table 5 has two parts: the last line is the precision and the rest are the recall. The precision of LVMapper is 88.5%, and the recall of LVMapper for Type-1, Type-2 and Very Strongly Type-3 were 100% or nearly 100%. For Strongly Type-3, NiCad performed the best, followed by Oreo and LVMapper. But Nicad's recall declined rapidly for next types with lower similarities. For Moderately Type-3 and Weakly Type-3&4, although Deckard detected the most pairs in Weakly Type-3&4, it had poor recall for other types of clones and the precision was only 34.8%. Oreo performed better than LVMapper on Moderately Type-3 and Weakly Type-3&4 because its results included some semantic clones [12] that existed in BigCloneBench. Through data analysis on the results, we found that more than 50% large-variance clones results reported by LVMapper could not be found by Oreo.

E. SCALABILITY

To test the scalability of LVMapper, we selected 1M LOC, 10M LOC, 20M LOC, 30M LOC and 250M LOC from the inter-project Java repository IJaDataset [26] as the target files to detect clones. We used an Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz machine with 28 cores and 512GB of memory. We limited the memory use of each tool to 12GB as described in [12]. As CCAligner, SourcererCC and Oreo had relative good scalability in recent studies [3], [12], we compared the execution time of LVMapper with those of CCAligner, SourcererCC and Oreo. We asked the authors for the proper configurations of these tools.

The execution time across different scales of datasets are listed in Table 6. LVMapper was the fastest for 1M LOC to 30M LOC and it scaled to 250M LOC dataset with the execution time of 52 hours and 29 minutes. CCAligner scaled to 10M LOC and it failed for the 20M LOC, 30M LOC and 250M LOC inputs with the out of memory error (denoted as "–" in Table 6). SourcererCC took 16 hours and 41 minutes, and Oreo took 76 hours and 21 minutes for clone detection of 250M LOC dataset. Although SourcererCC has good scalability but the large-variance detection ability of SourcererCC is limited.

V. THREATS TO VALIDITY

To estimate the precision of clone detection tools, judges usually manually validate the clone pairs. A threat is that the judges may not give the correct judgment. We mitigated this threat by using two judges that one was the author and one was not. They validated the code pairs according to the definition of different types of clone and they discussed with each other when they had different opinions until resolving the conflict.

The performance of clone detection is influenced by the parameter setting of methods. In Section IV-B we quantitatively analyzed different settings for the recall, precision, time and memory usage and selected a proper setting for LVMapper. For other methods in our evaluation, we used the configurations which had good performance in previous

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

TABLE 5. Recall Per Clone Type and Precision Measured for BigCloneBench

Type	LVMapper	CCAligner	Oreo	NiCad	SourcererCC	Deckard	iClones	CCFinderX
Type-1	100	100	100	100	100	60	100	100
Type-2	99	99	99	100	98	58	82	93
Very Strongly Type-3	98	97	100	100	93	62	82	62
Strongly Type-3	82	70	89	95	61	31	24	15
Moderately Type-3	19	10	30	1	5	12	0	1
Weakly Type-3&4 (Num)	0.3 (23923)	0.2 (12540)	0.7 (57273)	0 (12)	0 (1892)	1 (77293)	–	–
Precision	88.5	78.8	89.5	94.5	98.8	34.8	91	72

TABLE 6. Execution time for different LOC

LOC	1M	10M	20M	30M	250M
LVMapper	11s	5m 25s	19m 9s	1h 33m 34s	52h 29m 18s
CCAligner	54s	48m 36s	–	–	–
Oreo	4m 22s	24m 12s	1h 7m 45s	2h 11m 30s	76h 21m 25s
SoucererCC	5m 40s	27m 52s	1h 3m 59s	1h 48m 34s	16h 41m 35s

studies and also contacted the authors to set the proper configurations, where available.

VI. RELATED WORK

There are many code clone detection tools proposed in the literature. More descriptions of these tools and methods can be found in [1], [2], [19], [28]–[33]. At present, the code clone detection of Type-3 is still a difficult task, especially for large-variance code clones. According to the types of clone similarity, the clone detection methods can be divided into two categories. One is the non-semantic (lexical and syntactic similarity) method, and the other is the semantic (functional and semantic similarity) method. Our method belongs to the former.

A. NON-SEMANTIC METHODS

These methods or tools determine whether the code pairs are clones or not base on the similarity of code words and code sentences. These clone detection methods mainly include the text based, the token based, the tree and graph based and the metrics based methods. Among these methods, some researchers classified the latter two as the semantic method.

For the text based tools [6], [18], [34], two code blocks are compared in the form of text or strings. Johnson [6] proposed a fingerprinting technique to identify similar source code and to speed up processing speed. Ducasse [34] developed a line based comparison detection tool. NiCad [18] is based on a two phases process, viz., identification of potential clones and code comparison using longest common subsequences. Compared with LVMapper, it adopts similar locating and verifying strategy. NiCad can detect Type-3 clones, but did not perform well in the test of detection ability for clones with large difference [3].

For the token based tools [7], [17], [35], tokens are firstly extracted from the source code by lexical analysis, and it is better than simple keyword matching since it tolerates different identifiers. CCFinder [35] is a popular tool based on token, but it does not support Type-3 clone detection. In their work, they used suffix tree to find identical subsequences and increase the threshold to filter small clones. Essentially, these operations are equivalent to the indexing and filtering technology in LVMapper. ConQat [9] did an edit distance based traversal of a suffix tree which is also investigated extensively in bioinformatics, but the Type-1 recall measured by BigCloneBench was low [10]. iClones [17] and SourcererCC [7] are also influential representatives of such tools. Göde and Koschke [17] developed the incremental tool iClones by merging neighboring Type-1/Type-2 clones to big clones or Type-3 clones. However, iClones can only detect Type-3 clones with small variance. Sajnani [7] developed a fast clone detection tool SourcererCC which uses tokens composition to verify clones, but it is constrained to the identification ability of token granularity. CCAligner [3] has good performance in detecting clones with relatively concentrated modifications but it misses scenarios where modifications are scattered.

For the tree and graph based tools [11], [36]–[39], abstract syntax tree (AST) is frequently used as the representation of source code, and program dependency graph (PDG) is used to represent the control and data flow dependencies in source code. Yang [36] and Deckard [11] proposed AST approaches for finding the syntactic differences between two programs. Duplix [37] and PDG-DUP [38] are PDG-based tools which use program slicing to find isomorphic subgraphs. These tree and graph based tools suffer from large execution time and poor scalability. To this end, CCSharp [39] improves the time performance and accuracy of the PDG-based method, but it still cannot achieve good performance in large scale dataset. Besides, these tools will fail to detect large-variance clones since structure of tree and graph may be changed during the extension and modification of the code.

For the metrics based tools [40]–[42], some metrics and characteristic features for tree and graph of source code can be used for code clone detection. Both Mayrand [40] and Balazinska [41] extracted metrics from an AST representation of source code and used the metrics for clone identification. Patenaude [42] used the metrics of source code that

can be divided into five categories, viz., classes, coupling, methods, hierarchical structure and clones. These methods extract some features from tree or graph or source code to verify the semantic similarity of two code blocks. They have similar limitations to that of the tree and graph based tools for large-variance clones.

B. SEMANTIC METHODS & MACHINE LEARNING METHODS

Apart from the tree and graph based and the metrics based tools mentioned above, these kind of clone detection tools include the semantic space mapping based tools [43], the software behavior based tools [44]–[46] and so on. Substantially, the tools based on semantics adopt semantic abstraction or modeling for source code rather than abstraction of lexical and syntactic similarity. Due to overlap of semantic clones and large-variance clones, however, the methods based on semantics can also find a small part of large-variance clones.

Machine learning is always an effective way to deal with complex problems, including code clone detection especially for semantic clone. With the spread of deep learning method, the clone detection using deep learning technologies is an emerging area. White [47] presented an unsupervised deep learning approach to detect clones, which can automatically learn discriminating features of source code. Wei [48] proposed a method to detect clones by learning representations and Hamming distance of code fragments. Zhao [49] encoded code control flow and data flow into a semantic matrix for detecting semantic clones. Recently, Saini [12] put forwarded a machine learning based method called Oreo, which can find the code clones in the overlap between syntactic and semantic zone. Oreo used the clones that are almost identical or very similar to train the deep learning model and the large-variance clone detection ability of Oreo is limited. Machine learning methods always face the issues of dependency on the initial training data. The experiment in Section IV-C shows that the machine learning method could not detect the large-variance clones well.

Overall, the clone detection tools previously discussed have different application conditions and are still limited in detecting large-variance clones for the following reasons:

- Existing text-based and token-based methods don't have specific strategy and solution for large-variance clones because they mainly detect clones that are almost identical or very similar. They have to set the threshold of similarity lower to detect large-variance clones, which harms the precision.
- Tree-based and PDG-based methods suffer from large execution time and poor scalability. Besides, large-variance in code will change the structure of tree and graph, which make it difficult for these methods to detect large-variance clones. Metric-based and semantic methods have similar limitation as they extract features from tree or graph or source code to verify the semantic similarity of two code blocks.
- Machine learning methods always face the issues of dependency on the initial training data. Besides, they are usually costly in deploying and training.

VII. CONCLUSION & FUTURE WORK

The large-variance code clones are generated by homologous modification and can be used in software development and other applications. Our experiments found that these clones were widespread in Type-3 clones, even in some datasets up to half or more. And the large-variance code clone changes the past clone detection methods that focus on finding almost identical or very similar code pairs. Therefore, the research on large-variance code clone is important and meaningful. In this paper, we proposed a novel concrete definition and a detector LVMapper borrowing from the idea of sequencing alignment in bioinformatics for large-variance code clones. Effective and innovative technologies such as dynamic threshold, avoiding time-consuming dynamic programming and seeds index are designed in our method. A series of testing on real cases of software projects and the state-of-the-art benchmarks showed the large-variance clone detection performances of LVMapper are much better than the other state-of-the-art tools, and it has comparable recall and precision for general Type-1 to Type-3 clones. And it will be important work to do research on software engineering applications such as code recommendation and completion, refactoring and bug propagation for large-variance clones in the future.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's School of Computing TR, vol. 541, no. 115, pp. 64–68, 2007.

[2] R. Koschke, "Survey of research on software clones," in Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[3] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in Proceedings of the 40th International Conference on Software Engineering (ICSE'18). ACM, 2018, pp. 1066–1077.

[4] A. Walenstein, R. Koschke, and E. Merlo, "06301 summary – duplication, redundancy, and similarity in software," in Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006, 2006.

[5] M. Fowler, Refactoring - Improving the Design of Existing Code, ser. Addison Wesley object technology series. Addison-Wesley, 1999.

[6] J. H. Johnson, "Substring matching for clone detection and change tracking," in Proceedings of International Conference on Software Maintenance, vol. 94, 1994, pp. 120–126.

[7] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE'16). IEEE, 2016, pp. 1157–1168.

[8] T. Kamiya, "The official ccfinderx website," http://www.ccfinder.net/ccfinderx.html, 2008.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in Proceedings of the IEEE/ACM 31th International Conference on Software Engineering (ICSE'09). IEEE, 2009, pp. 485–495.

[10] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'15). IEEE, 2015, pp. 131–140.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society, 2007, pp. 96–105.

[12] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Oreo: Detection of clones in the twilight zone," in Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18). ACM, 2018, pp. 354–365.

[13] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," Journal of Molecular Biology, vol. 215, no. 3, pp. 403–410, 1990.

[14] B. Liu, D. Guan, M. Teng, and Y. Wang, "rhat: fast alignment of noisy long reads with regional hashing," Bioinformatics, vol. 32, no. 11, pp. 1625–1631, 2015.

[15] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," Briefings in Bioinformatics, vol. 11, no. 5, pp. 473–483, 2010.

[16] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14). IEEE, 2014, pp. 476–480.

[17] N. Göde and R. Koschke, "Incremental clone detection," in Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09). IEEE, 2009, pp. 219–228.

[18] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08). IEEE, 2008, pp. 172–181.

[19] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Transactions on Software Engineering, vol. 33, no. 9, 2007.

[20] H. Cheng, H. Jiang, J. Yang, Y. Xu, and Y. Shang, "Bitmapper: an efficient all-mapper based on bit-vector computing," BMC Bioinformatics, vol. 16, no. 1, p. 192, 2015.

[21] J. R. Cordy, "The txl source transformation language," Science of Computer Programming, vol. 61, no. 3, pp. 190–210, 2006.

[22] V. Paxson et al., "Flex–fast lexical analyzer generator," Lawrence Berkeley Laboratory, 1995.

[23] A. Appleby. (2016) Murmurhash hash functions. [Online]. Available: https://github.com/aappleby/smhasher/

[24] A. S. E. Group. (2013) Ijadataset 2.0. [Online]. Available: http://secold.org/projects/seclone

[25] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'16). IEEE, 2016, pp. 596–600.

[26] A. S. E. Group. (2011) Ijadataset 1.0. [Online]. Available: http://secold.org/projects/seclone

[27] Cloc. (2015) Count lines of code. [Online]. Available: http://cloc.sourceforge.net/

[28] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation. IEEE, 2002, pp. 36–43.

[29] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, vol. 74, no. 7, pp. 470–495, 2009.

[30] F. V. Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in Proceedings of the 19th IEEE international conference on Automated software engineering. IEEE Computer Society, 2004, pp. 336–339.

[31] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," ISRN Software Engineering, vol. 2013, 2013.

[32] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," Information and Software Technology, vol. 55, no. 7, pp. 1165–1199, 2013.

[33] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," International Journal of Computer Applications, vol. 137, no. 10, pp. 1–21, 2016.

[34] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in Proceedings of International Conference on Software Maintenance. IEEE, 1999, pp. 109–118.

[35] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654–670, 2002.

[36] W. Yang, "Identifying syntactic differences between two programs," Software: Practice and Experience, vol. 21, no. 7, pp. 739–755, 1991.

[37] J. Krinke, "Identifying similar code with program dependence graphs," in Proceedings of the 8th Working Conference on Reverse Engineering. IEEE, 2001, pp. 301–309.

[38] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in Proceedings of the International Static Analysis Symposium. Springer, 2001, pp. 40–56.

[39] M. Wang, P. Wang, and Y. Xu, "Ccsharp: An efficient three-phase code clone detector using modified pdgs," in Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17). IEEE, 2017, pp. 100–109.

[40] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in Proceedings of the International Conference on Software Maintenance, vol. 96, 1996, p. 244.

[41] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in Proceedings of the 6th International Software Metrics Symposium. IEEE, 1999, pp. 292–303.

[42] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë, "Extending software quality assessment techniques to java systems," in Proceedings of the 7th International Workshop on Program Comprehension. IEEE, 1999, pp. 49–56.

[43] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE'01). IEEE, 2001, pp. 107–114.

[44] S. Choi, H. Park, H.-i. Lim, and T. Han, "A static api birthmark for windows binary executables," Journal of Systems and Software, vol. 82, no. 5, pp. 862–873, 2009.

[45] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in Proceedings of the 18th International Symposium on Software Testing and Analysis. ACM, 2009, pp. 81–92.

[46] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011, pp. 301–310.

[47] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 2016, pp. 87–98.

[48] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17), 2017, pp. 3034–3040.

[49] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2018, pp. 141–151.

MING WU received the B.S. degree in computer science and technology from University of Science and Technology of China (USTC), China, in 2015. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, University of Science and Technology of China. Her current research interests include code clone detection and code completion.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2971545, IEEE Access

PENGCHENG WANG received the B.S. degree in computer science and technology from University of Science and Technology of China (USTC), China, in 2014, and the Ph.D. degree in computer science and technology from the University of Science and Technology of China (USTC), in 2019. His research interests include code clone detection, parallel computing and big data analysis.

KANGQI YIN received the B.S. degree in computer science and technology from University of Science and Technology of China (USTC), China, in 2014. And he received the B.S. degree in electronic information engineering from Hefei University of Technology, China, in 2016. And he received the M.S. degree in computer science and technology from University of Science and Technology of China, Hefei, China, in 2019. His main research interest is code completion.

HAOYU CHENG received the Ph.D. degree in computer science and technology from University of Science and Technology of China (USTC), China, in 2019. He is currently working at the Dana-Farber Cancer Institute, Harvard Medical School, Boston, MA, USA. His main research interests include high performance computing and bioinformatics.

YUN XU is a professor at Department of Computer Science, University of Science and Technology of China (USTC), Hefei, China. He received Ph.D. in Computer Science from USTC in 2002. He has published over 80 refereed papers in areas of high-performance computing, software engineering, bioinformatics and so on. His research has been supported by the National Nature Science Foundation of China and Project of Ministry of Education of China etc. He has advised over 50 PhD and MS students. Details of his research and related information can be found at http://staff.ustc.edu.cn/~xuyun/

CHANCHAL K. ROY is a professor of Software Engineering/Computer Science at the University of Saskatchewan, Canada. While he has been working on a broad range of topics in Computer Science, his chief research interest is Software Engineering. In particular, he is interested in software maintenance and evolution including clone detection and analysis, program analysis, reverse engineering, empirical software engineering, and mining software repositories. He served or has been serving in the organizing and/or program committee of major software engineering conferences, and has been a reviewer of major Computer Science journals. He received his Ph.D. at Queen's University in 2009.