Fine-Grained Code Clone Detection with Block-Based Splitting of Abstract Syntax Tree

Abstract

Code clone detection aims to find similar code fragments and gains increasing importance in the field of software engineering. There are several types of techniques for detecting code clones. Text-based or token-based code clone detectors are scalable and efficient but lack consideration of syntax, thus resulting in poor performance in detecting syntactic code clones. Although some tree-based methods have been proposed to detect syntactic or semantic code clones with decent performance, they are mostly time-consuming and lack scalability. In addition, these detection methods can not realize fine-grained code clone detection. They are unable to distinguish the concrete code blocks that are cloned. In this paper, we design *Tamer*, a scalable and fine-grained tree-based syntactic code clone detector. Specifically, we propose a novel method to transform the complex abstract syntax tree into simple subtrees. It can accelerate the process of detection and implement the fine-grained analysis of clone pairs to locate the concrete clone parts of the code. To examine the detection performance and scalability of *Tamer*, we evaluate it on a widely used dataset BigCloneBench. Experimental results show that *Tamer* outperforms ten state-of-the-art code clone detection tools (*i.e.*, *CCAligner*, *SourcererCC*, *Siamese*, *NIL*, *NiCad*, *LVMapper*, *Deckard*, *Yang2018*,*CCFinder*, and *CloneWorks*).

CCS Concepts

• **Software and its engineering → Software maintenance tools**;

Keywords

Clone Detection, Abstract Syntax Tree, Fine-grained, Splitting

ACM Reference Format:

. 2023. Fine-Grained Code Clone Detection with Block-Based Splitting of Abstract Syntax Tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598040

1 Introduction

Cloning codes is a process of reusing code fragments via copying, pasting, and revising[30, 37]. Since cloning code can save a lot of time, developers tend to clone the code of others rather than write code from scratch when implementing similar methods. However, if programmers clone some vulnerable codes, it will lead to the propagation of vulnerabilities. In other words, although code cloning brings much convenience to software development, it also reduces software security and increases maintenance costs. Due to the presence of these problems, clone detection becomes an active area of software engineering and gradually occupies a pivotal position in this field.

安全隐患

With the development of clone detection technology, many clone detectors have been proposed. According to different code representations, they can be roughly divided into two categories: token-based and intermediate representations-based. They differ in detection capabilities and scalability. Token-based tools [18, 27, 28, 32, 38, 41, 44] directly convert code fragments into text or token sequences and then perform similarity comparison. Although the speed is fast, most of them can only detect clones in text. To solve this problem, researchers propose to apply intermediate representations of code to maintain the syntax and semantics of code. For example, some graph-based tools [16, 29, 31, 43, 53, 54] convert program details into graphs and apply graph analysis to detect complex code clones. However, graph analysis is typically time-consuming, which makes it difficult to be used for large-scale code clone detection. Therefore, to migrate the issue, other methods [14, 24–26, 33, 48] propose a tree representation method to maintain the syntax characteristics of code and conduct tree-matching to detect clones. Although tree analysis of source code is faster than graph analysis, the structure of tree is still complex, and clone detection still takes a long time.

Figure 2 shows the abstract syntax tree of the source code in Figure 1. We can see that a simple method of only ten lines can be transformed into a complex tree of 99 nodes. When a method has more lines of code, the corresponding syntax tree will be more complex, which results in a high overhead for tree analysis. In addition, the existing clone detectors cannot locate similar parts of the clone pair. If the cloned code with a large number of code lines has some security problems in several lines of code, it is difficult to locate and modify. As a result, we need to design a tree-based clone detector that can effectively reduce the time cost and enable fine-grained clone analysis.

In this paper, we implement *Tamer*, an effective and efficient clone detector that can perform fine-grained code clone analysis. *Tamer* is a tree-based clone detector that considers the syntax of source code, thus it can effectively detect clones with high performance. Specifically, we use carefully designed rules to split an *abstract syntax tree* (AST) into a series of subtrees at block granularity and reorganize a relatively simple structure tree to retain an overall feature of the AST. After splitting the original tree, we compute the similarity of two codes at the subtree level instead of the AST level, which can remarkably improve the efficiency of *Tamer*. More importantly, each subtree represents a part of the source code, respectively. Therefore, by calculating the similarity of the corresponding subtrees between two codes, we can distinguish the subtree pairs with high similarity, and then we can locate the similar code blocks in the source code. In this way, we can perform fine-grained analysis of code clones.

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598040

```
public Matrix transpose(int n, int m) {
  if (m == 0 || n == 0)
    return 0;
  Matrix X = new Matrix(n, m);
  double[][] C = X.getArray();
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      C[j][i] = A[i][j];}}
  return X;
}//original
```

```
public double[][] transpose(double[][] a) {
  int am = a.length, an = a[0].length;
  double[][]result = new double[an][am];
  if (a.size() == 0)
    return 0;
  for (int x = 0; x < am; x++) {
    for (int y = 0; y < an; y++) {
      result[y][x] = a[x][y];}}
  return result;
}//MT3 clone
```

Figure 1: A complex code clone pair

To examine the capability of *Tamer*, we conduct comprehensive experiments on a widely used dataset *big clone bench* (BCB) [2]. As for detection performance, our experimental results show that *Tamer* performs better than the other ten state-of-the-art clone detectors which are *CCAligner* [44], *SourcererCC* [39], *Siamese* [36], *NIL* [35], *Nicad* [38], *LVMapper* [49], *Deckard* [25], *Yang2018* [51], *CCFinder* [28], and *CloneWorks* [41]. As for scalability, *Tamer* only requires about 11 minutes and 40 seconds to complete the clone scanning of 10M lines of code, which is faster than most of our comparative detectors. As for fine-grained analysis, the output of *Tamer* can not only report whether two methods are clones, but also give the similarity between different code blocks. In this way, we can know which code blocks in clone pairs are more similar, so as to assist researchers in subsequent security analysis.

In summary, the main contributions of this paper are as follows:

论文的主要内容

- We propose a method to split the complex AST into relatively simple subtrees. Calculating the similarity of subtrees rather than the original tree can greatly reduce the detection cost.
- We implement *Tamer* [11], a scalable tree-based code clone detector. It can not only carry out large-scale clone detection with high performance but also realize fine-grained code analysis to locate specific cloned parts in the source code.
- We evaluate *Tamer* on a widely used dataset namely BigCloneBench. Experimental results indicate that *Tamer* has the best performance and ideal scalability compared with the existing clone detectors. In addition, it can also give fine-grained clone reports.

The remainder of this paper is organized as follows. Section 2 describes the motivation. Section 3 defines code clone types. Section 4 describes *Tamer* in detail. Section 5 gives an overview of our evaluation and presents the results. Section 6 discusses future work. Section 7 reviews related studies. Section 8 concludes this paper.

2 Motivation

To illustrate why we design *Tamer* and how our detection method is implemented, we use a simple but clear example. As shown in Figure 1, we select a clone pair in *BCB* dataset [2], the original method and the cloned method are a clone pair similar in syntactic that implements the matrix assignment. The latter modifies the variable name and adds some statements.

NIL [35] is a state-of-art token-based clone detector. When calculating the similarity of two methods, *NIL* extracts the token sequence of source code and calculates the *longest common sequence* (LCS) of two token sequences, and finally divides the LCS by the minimum of the number of two methods to obtain the similarity.

For the two methods in Figure 2, we find the number of tokens is 99 and 113, respectively. After calculating the LCS of two token sequences, we find the length of their LCS is 30. We can get the similarity between the two methods is 30/99=0.3. However, the default similarity threshold of NIL is 0.7 which means only code pairs with a similarity greater than 0.7 are considered clone pairs. So *NIL* will treat this code pair as a non-clone pair.

To find an approach to determine the two methods as a clone pair, we consider extracting their ASTs. Figure 2 shows brief ASTs of them and they are very similar in structure even though the variable names and the number of statements in the source code are different. Furthermore, if we only use the type name to represent the node, two ASTs will become very similar. Additionally, we can see from Figure 2 that the For statement part of two ASTs is the same. So it is natural to think that if we calculate the similarity of the AST parts of the code blocks, we can get a high similarity.

To illustrate how we divide the different parts of AST, we specifically use different colors in Figure 2. The AST has five parts labeled with five colors. We integrate the three parts of function declaration, variable declaration, and return value into one block, if and for parts into one block, so that we get three blocks from one method, which are if block, for block, and rest block. We first acquire the node-type sequences through *depth-first traverse* (DFS), and then calculate the LCS of the two sequences to get the similarity like *NIL*. Table 1 shows the similarity of each block. We can see that the similarity of For block is 100%, If block is 59%, and the rest block is 53%. They all exceed the similarity calculated by *NIL*. Even if we take the average similarity of three blocks as the final similarity, we can find it is 71% which is higher than the threshold of 70%.

Table 1: The similarity of three blocks

	For block	If block	The rest block	Ave	NIL
Similarity	100%	59%	53%	71%	30%

In practice, we also compute the similarity by directly analyzing the original ASTs, and the code pair in Figure 1 can also be reported as a clone pair. However, clone detection technology needs to both guarantee accuracy and optimize detection efficiency. When we use the original complex AST to perform similar calculations, we find that its calculation time is much higher than the token-based calculation method. For this reason, if a single comparison between two code blocks is time-consuming, the whole comparison time will be hard to estimate.

Since the existing LCS algorithms are mostly $O(N^2)$ complexity algorithms, the time required to calculate LCS between subtrees is far less than the time required to calculate LCS between two large ASTs. To make it more convincing, we conduct an experiment on the three blocks of the original code. The number of nodes of the If block, For block, and the rest block are 12, 48, and 56, respectively. The node number of the entire AST is 116. So the number of computations required to calculate the LCS of the AST is $116 * 116 = 13456$ times. However, the number of computations required to calculate the LCS of the three blocks is $12*12+48*48+56*56 = 5584$ times. The number of computations is reduced nearly twice. With the increase in the code size, the improvement will become more significant, which will be an amazing improvement for the tool's scalability.

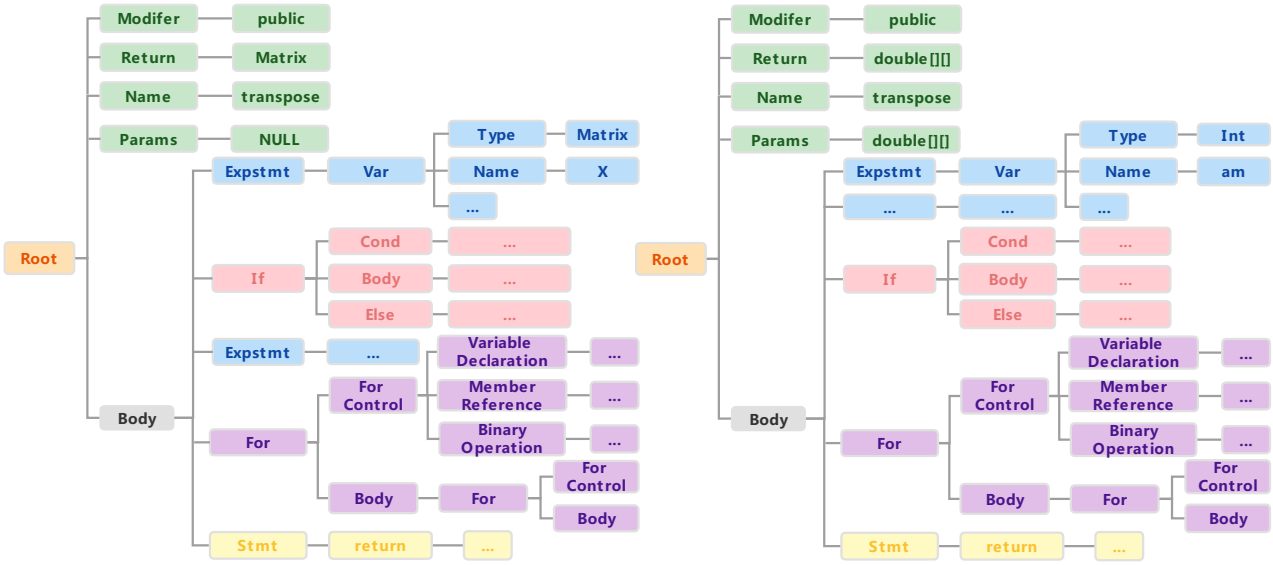

Figure 2: The source code and ASTs of original and clone code

Based on the above research, we propose a method that can greatly improve the performance of clone detection and scalability.

3 Definition

A code fragment is a continuous segment of source code. A code block refers to a code fragment within braces. Since a method can implement a specific functionality, we select it as our detection granularity. Generally, clones can be divided into the following four categories according to their similarity [39, 44]:

- **Type-1 Clone (Textual Similarity)**: The code clone in this category is the same copy, except for some spaces, blank lines, and comments.
- **Type-2 Clone (Lexical Similarity)**: The code clone in this category is a copy different only in variable names, variable types, or some function identifiers.
- **Type-3 Clone (Syntactic Similarity)**: The code clone in this category is a copy with more modifications, and some statements have been modified, deleted, or added.
- **Type-4 Clone (Semantic Similarity)**: The code clone in this category is a copy that has a dissimilar syntactical structure but implements the same functionality.

克隆行为的四种定义

As for Type-4 clones, they are semantic clones and are hard to be detected. Therefore, similar to previous works [35, 39, 44], we mainly focus on detecting the first three types of clones.

4 System

In this section, we will introduce *Tamer* (*i.e.,* a tree-based code clone detector by using N-gram and LCS) in detail.

4.1 System Overview

The overall framework of *Tamer* is shown in Figure 3, which can be divided into three phases: Processing, Locate & Filter, and Verify.

- **Processing**: The purpose of this phase is to extract the ASTs of methods. We use DFS to traverse the AST and record type name of each node to obtain the node sequence. Then we create the inverted index according to the N-grams generated by the node sequence.
- **Locate & Filter**: The purpose of this phase is to use the inverted index to perform a preliminary screening and filtering on all pairs of methods to obtain candidate clone pairs, so as to prepare for the subsequent verify phase.
- **Verify**: The purpose of this phase is to get the true clone pairs. According to the candidate clone pairs obtained in the Locate & Filter phase, we use LCS for similarity calculation to determine whether a candidate clone pair is a true clone.

4.2 Processing

Because *Tamer*'s goal is to get a good performance in detecting Type-3 clones, we need to use AST, an intermediate representation that retains the syntax of the original code. Therefore, the first step of code processing is to obtain AST by static syntax analysis. Because the programming language of the dataset is Java, we use JavaParser [7] to implement the static analysis.

In addition, AST is more complex than the source code, so if we retain the complete AST, it will undoubtedly result in a huge overhead on the space of the detector. So we think about using a single string to store the AST information. We use DFS to traverse the entire AST and record the type name of nodes. Specifically, we perform statistical analysis on the whole *BCB* dataset and we find that there are only 70 types of nodes, so we can convert the name of each node type to a char byte which greatly reduces the memory space to store AST information.

After getting the node sequence of each method, we can get its corresponding N-grams on this basis. An N-gram is a chunk of continuous nodes whose number is N. Figure 4 shows the process of obtaining 3-grams from a node sequence. We traverse the AST node sequences generated by all methods in the dataset, and each method will generate a series of N-grams.

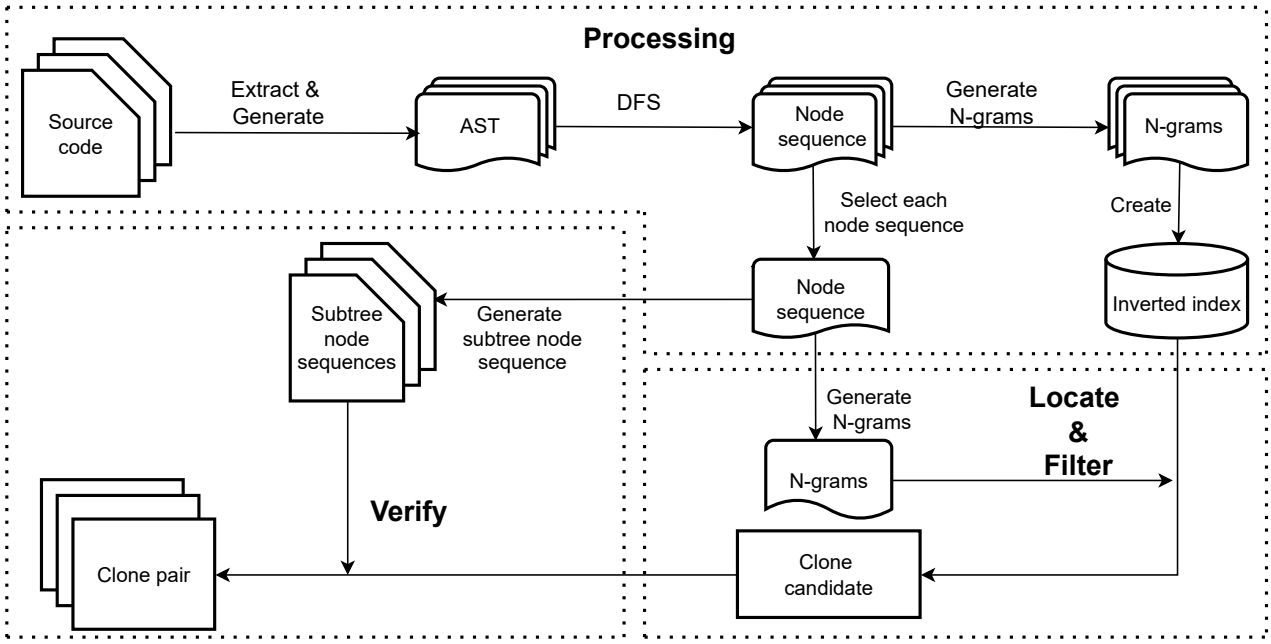

Figure 3: Overview of *Tamer*

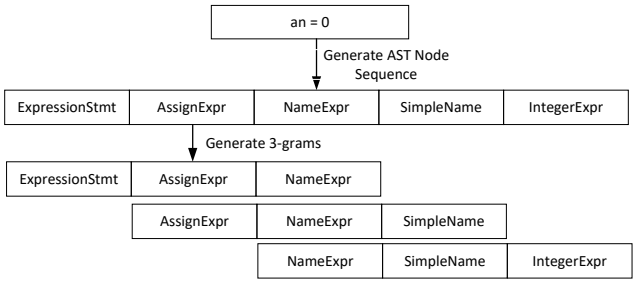

Figure 4: Example of generating N-grams

如何获得inverted index?

Next, *Tamer* creates an inverted index based on the obtained N-grams. The inverted index is a fast information retrieval tool that can quickly retrieve documents containing query words. This idea originates from the situation that we need to find some records according to given values. In *Tamer*, we use the HashMap structure whose key is the hash value of an N-gram, and then we use the ID of methods that also contain the same N-gram as the HashMap value. Therefore, the code blocks containing the same N-gram can be quickly found through the inverted index.

When all N-grams are stored in the HashMap, an inverted index of the dataset is built, which can quickly calculate the number of common N-grams between any two methods.

4.3 Locate & Filter

4.3.1 Locate. After the Processing phase, *Tamer* uses the inverted index obtained from the Processing phase to perform Locate operation. The locating operation mainly obtains candidate clones of the target code block through the inverted index. Its algorithm corresponds to lines 6-11 in Algorithm 1.

Firstly, we get the N-grams generated by the AST node sequence. A node sequence with a length of M can generate M-N+1 N-grams. Next, we calculate the hash value of each N-gram and use the hash value as the query object to query all methods corresponding to the hash value in the inverted index. These methods all have the same N-gram, indicating that their AST node sequences have a part of the same subsequence. Therefore, we add these methods to the candidate clones of the target methods.

4.3.2 Filter. In the Filtering operation, we mainly remove the candidate clone obtained by the Locating operation which cannot be a clone. The algorithm corresponds to lines 12-21 in Algorithm 1. It is necessary to reduce the number of clone candidates because it takes a lot of time to calculate the similarity of node sequences between the two methods in the verification phase. Therefore, for scalable and rapid clone detection, we do not need to detect the code block pairs with few common N-grams.

In this phase, we think about a small trick to reduce the time cost of calculating the number of common N-grams between two methods. We use an array to store the common N-grams number of the target method with the other method. In this way, we only need to traverse the N-grams of the target method once and we can achieve the number of common N-grams between the target method and the other all methods.

To quantitatively describe the similarity of N-grams between the two methods, we use the filter-score defined below.

$$filter\_score = \frac{common\_ngrams(m1, m2)}{max(ngrams(m1), ngrams(m2))} \quad (1)$$

通过N-grams的数量来计算相似度

$$common\_ngrams(m1, m2) = |ngrams(m1) \cap ngrams(m2)| \quad (2)$$

where m1 and m2 are two methods whose node sequence's lengths are |m1| and |m2|, respectively. N-grams (m1) and N-grams (m2)

are the numbers of N-grams produced by m1 and m2. Since the difference of two node sequences' lengths may be very big, we use max in the discriminator to avoid *Tamer* mistakenly regarding them as clone pairs. In many previous studies, min is often used to improve the recall of detection, but this calculation method will greatly reduce the precision of detection at the same time. Since *Tamer* is a tree-based clone detector, its performance will be better than token based detector, so we use max in discriminator to ensure the precision of detection.

Finally, we remove those clone candidate pairs whose value of filter_score is less than threshold θ.

4.4 Verify

In this phase, we verify whether each code block pair in the candidate clone pair is a true clone. The main idea of the algorithm corresponds to lines 22-30 in Algorithm 1.

Firstly, we need to acquire the subtree node sequence of each code block. Because in the processing phase, we have already gotten the AST of each code block, thus we can generate subtrees from it directly through a carefully designed rule. Based on the thought introduced in Section 2, we mainly split the AST according to the different types of statements it has. We perform statistics on all methods in the *BCB* dataset, and we make a sort of the complexity of different types of statements (*i.e.*, Expressionstmt, Ifstmt) in the source code. The complexity of each type statement is calculated by the average number of the corresponding node in the AST. We select the eight most complex statements which are Forstmt, Whilestmt, Trystmt, Dostmt, ForEachstmt, Switchstmt, Synchronizedstmt, and Ifstmt. Additionally, if we only consider the statement subtrees, we may lose the whole structure information of the method. Therefore, it is necessary to generate a structure subtree of the method. In conclusion, our splitting rule is: for relatively simple statements such as variable declaration and assignment, we retain them on the original AST. For more complex statements such as if statements and for statements, we separate them from the original AST tree to form a subtree. Only the root node types of these subtrees are retained on the original AST tree. Finally, the original AST will form a structure subtree, and the separated statement blocks will form a statement tree. In this way, we split the original complex AST into nine simpler subtrees supposing it has all types of statements mentioned above.

The subtrees of the AST in Figure 2 are shown in Figure 5. We can see that a simple method has four blocks distinguished by different colors, and these blocks correspond to a part of the structure tree. These blocks can be represented by a subtree, respectively. As shown in Figure 5, the relationship among source code, blocks, and subtrees is demonstrated. Finally, according to the rule we mentioned above, the complex AST is split into three simpler subtrees. The Ifstmt subtree represents the if block in the source code, the Forstmt subtree represents the for block in the source code, and the Structure subtree records the structure information of AST.

After that, we use DFS to traverse subtrees and get the corresponding node sequences. We calculate the LCS of two node sequences as the index of their similarity. To quantitatively describe the degree of similarity, we use the verify-score, defined below, where s_i^1 and s_i^2 are two subtree node sequences whose node lengths

Algorithm 1: Clone Detection

Data: C is a list of node sequence of code blocks $\{c_1, c_2, \ldots, c_n\}$, Inverted index I of C, N for size of N-grams, θ for filter threshold, δ for verify threshold
Result: All clone pairs CP

1 $CP \leftarrow \varphi$;
2 $i = 0, j = 0$;
3 **while** $i <= N$ **do**
4 $i++$;
5 $CC \leftarrow \varphi$;
6 //Location phase, and CC represents clone candidates
  while $j <= c_i.length - N + 1$ **do**
7 $j++$;
8 $n\_gram = strcat(c_i[j], c_i[j+1], \ldots, c_i[j+N-1])$;
9 $key = hash(n\_gram)$;
10 /*acquire is a function that returns values in the hashmap.*/
  CC = CC ∪ acquire(key);
11 **end**
12 //Filter phase
  $j = 0$;
13 **while** $j <= CC.size()$ **do**
14 /*$common\_ngrams$ is a function that returns the number of common N-grams between two given code blocks*/
  $j++$;
15 $comngram = common\_ngrams(c_i, cc_j)$;
16 $maxlen = max(c_i.len, cc_j.len)$;
17 $filter\_score = comngram/(\text{maxlen} - \text{N} + 1)$;
18 **if** $filter\_score < \theta$ **then**
19 CC =CC $-\{cc_j\}$;
20 **end**
21 **end**
22 //Verify phase
  $j = 0$;
23 **while** $j <= CC.size()$ **do**
24 /*LCS is a function that returns the length of LCS of the given two sequences.*/
  $j++$;
25 $sub\_1, sub\_2 = getsubtree(c_i), getsubtree(cc_j)$;
26 $k = 1, verify\_score = 0$;
27 **while** $k <= 9$ **do**
28 $total\_length = sub\_1[k].len + sub\_2[k].len$;
29 $lcs\_length = LCS(sub\_1[k] + sub\_2[k])$;
30 $verify\_score += lcs\_length/(total\_length - lcs\_length)$;
31 $k++$;
32 **end**
33 **if** $verify\_score >= \delta$ **then**
34 $CP = CP \cup \{cc_j\}$;
35 **end**
36 **end**
37 **end**

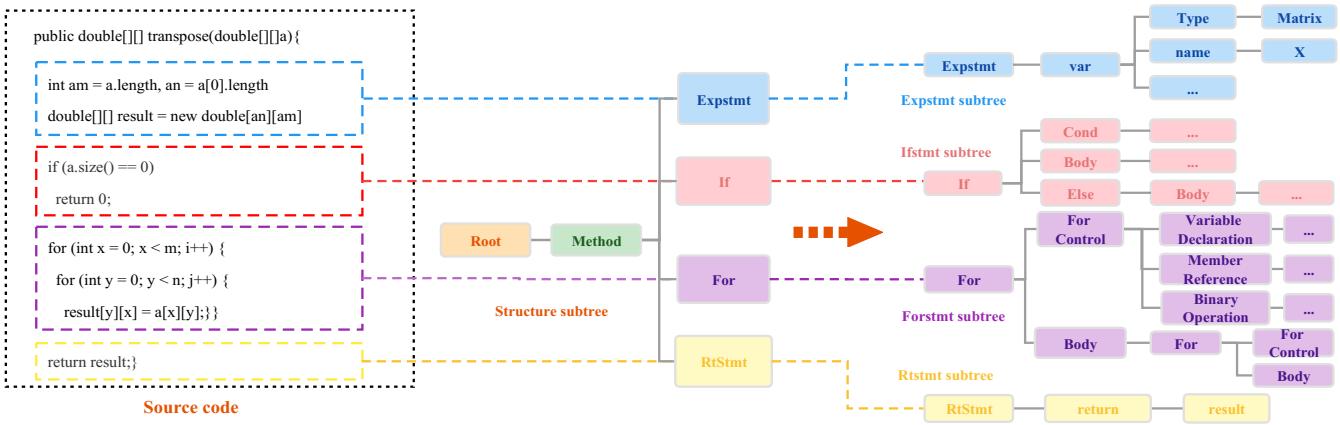

Figure 5: Subtrees of the original method in Figure 1

are $|s_i^1|$ and $|s_i^2|$, respectively. lcs(s_i^1, s_i^2) is the LCS length between two node sequences. We use $|s_i^1| + |s_i^2|$ - lcs(s_i^1, s_i^2) as the denominator, which can avoid the calculation imbalance of subtree_score when the lengths of two node sequences are very different, and can effectively reduce the case that *Tamer* mistakenly judges a non-clone pair as a clone pair. We plus the nine subtree_score together to get verify_score.

$$subtree\_score_i = \frac{lcs(s_i^1, s_i^2)}{|s_i^1| + |s_i^2| - lcs(s_i^1, s_i^2)} \tag{3}$$

$$verify\_score = \sum_{i=1}^{9} subtree\_score_i \tag{4}$$

Finally, we consider the code block pair whose verify_score exceeds threshold δ as a true clone pair.

5 Experiment

In this section, we focus on answering the following five research questions (RQs):

- ***RQ1***: *What is the detection performance of Tamer with different parameters?*
- ***RQ2***: *Can Tamer outperform other state-of-the-art clone detectors?*
- ***RQ3***: *Can Tamer scale to big code?*
- ***RQ4***: *How does Tamer perform fine-grained analysis and locate the specific location of code cloning?*
- ***RQ5***: *What is the advantage of splitting the AST?*

5.1 Experimental Settings

5.1.1 Dataset. We evaluate *Tamer* on a widely used dataset, *BCB* [2]. The clone type of each pair of methods in *BCB* is manually assigned by experts. The *BCB* dataset consists of more than 8,000,000 labeled clone pairs. Due to the unclear boundary between Type-3 and Type-4, these two clone types are further divided into four subcategories by a similarity score measured by line-level and token-level code normalizations, as follows: 1) *very strongly type-3* (VST3), where the similarity is between 90-100%, 2) *strongly type-3* (ST3), where the similarity is between 70-90%, 3) *moderately type-3* (MT3), where the similarity is between 50-70%, and 4) *weakly type-3/type-4* (WT3/T4), where the similarity is between 0-50%. As aforementioned, since T4 code clones are semantic clones and are difficult to be distinguished, we ignore them and pay more attention to the other types.

5.1.2 Implementation. We run all experiments on a standard server with 128GB RAM and 16 cores of CPU. For the implementations of *Tamer*, we mainly use *Javaparser* [7] to complete the static analysis including tree extraction, analysis, and splitting.

5.1.3 Comparison. We compare *Tamer* with ten existing state-of-the-art code clone detectors:

- **CCAligner** [44]: A popular code clone detector by analyzing the code windows and *e* edit distance between methods.
- **SourcererCC** [39]: A popular code clone detector by calculating the number of overlapping tokens between methods.
- **Siamese** [36]: A popular code clone detector by transforming the token sequence into a different presentation.
- **NIL** [35]: A popular code clone detector by calculating the LCS of token sequence of methods.
- **NiCad** [38]: A popular code clone detector by using TXL parser to compute the similarity of methods.
- **LVMapper** [49]: A popular code clone detector by calculating the number of common tokens and dynamic threshold.
- **Deckard** [25]: A tree-based code clone detection with an algorithm based on numerical vectors in the Euclidean space.
- **Yang2018** [51]: A tree-based code clone detection with a hybrid incremental clone detection and live scatterplots technique.
- **CCFinder** [28]: A multilinguistic token-based code clone detection system for large-scale source code.
- **CloneWorks** [41]: a fast and flexible large-scale near-miss clone detection tool using modified Jaccard similarity metric.

5.1.4 Metrics. Since code cloning detection is a binary classification task, we take widely used metrics to measure *Tamer*'s performance. Our measurement metrics are defined as follows:

- **True Positive (TP)**: It means that one or some clone pairs are predicted to be clone code.
- **True Negative (TN)**: It means that one or some non-clone pairs are predicted to be non-clone code.
- **False Positive (FP)**: It means that one or some non-clone pairs are predicted to be clone code.

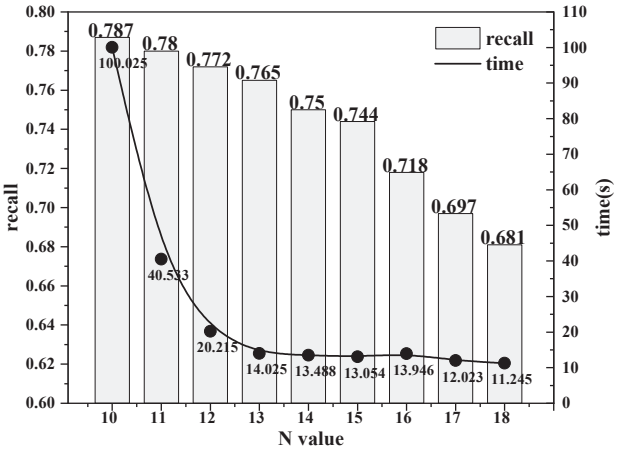

Figure 6: N value select experiment

- **False Negative (FN)**: It means that one or some clone pairs are predicted to be non-clone codes.
- **Precision=TP/(TP+FP)**: The correct rate of detection.
- **Recall=TP/(TP+FN)**: The percentage of clone pairs that are successfully detected.

5.2 RQ1: Parameter Setting

Tamer requires three parameters: the value of *N*, filtration threshold θ, and verification threshold δ. After many experiments, we find that the impact of θ on *Tamer* is not great. Specifically, we perform a lot of experiments and observe that those candidates which can not reach θ are always not real clones. Therefore, the value of θ only has a slight impact on scalability. In detail, we consider previous research (*e.g., NIL* [35] and *LVMapper* [49]) and our experimental results together to set θ to 0.15.

However, the value of N needs to be carefully selected, because it has a great impact on detection performance. If the N value is set to a small value, the recall will be high while the system execution time will increase dramatically. If the N value is set to a large value, although the system execution time will decrease, the recall will be low. Therefore, to select the most appropriate N value, we conduct an experiment with N ranging from 10-18 and measure the recall and execution time when θ=0.7. Figure 6 shows the results of each N value. It can be seen that when N<15, the execution time increases significantly with the drop of the N value, while when N>15, execution time tends to be flat. In addition, when N>15, the recall decreases significantly. Therefore, considering the balance of these two factors, we finally choose N=15 as the optimal parameter.

Next, we experiment to select the optimal value of the verification threshold δ. We calculate recall and precision of δ from 0.6 to 0.8 and the step is 0.05. The experimental results are shown in Figure 7. It can be seen that when δ>0.65, recall starts to decrease significantly, but precision does not improve significantly with only minor changes. Considering that precision is not so good when δ=0.60, we finally choose δ=0.65 as the best parameter.

5.3 RQ2: Comparative Performance

We measure the recall of different types of clones on the *BCB* dataset under the parameters selected in RQ1. At the same time, we also calculate precision which is the same as the previous studies [39, 44, 49].

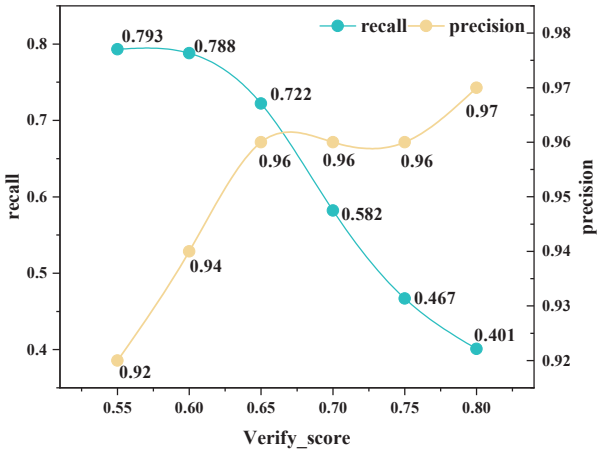

Figure 7: Verify_score select experiment

Table 2 shows the recall and precision results of each tool tested on the *BCB* dataset. We can see that *Tamer* has the highest recall in all types of clones. Among them, the recall of the ST3 clone exceeds the *LVMapper* which has the highest recall in the remaining ten tools by 17%, and the recall of the MT3 clone exceeds *Yang2018* which has the highest recall in the remaining ten tools by 26%. The experimental results indicate that *Tamer* has a significant improvement in clone detection.

After analyzing the reason behind this phenomenon, we find that it is difficult for the eight token-based clone detectors to detect Type-3 clones. The reason is that the Type-3 clone has different lexical structures but similar syntactic structures. However, tree representation of source code can retain its syntax information, so the tree-based detector can detect Type-3 clones, which is why *Tamer* has such a high recall value compared with other clone detectors. Besides, even though *Deckard* and *Yang2018* use the tree-presentation of source code to perform clone detection and they indeed defeat the other token-based methods, their recall is still lower than *Tamer*. This is because although these two methods use the tree representation of the source code, they only perform a shallow analysis of the AST, and finally use the entire AST to calculate the similarity, and the algorithm is not significantly improved. However, *Tamer* does not just use AST, it also proposes a new algorithm to cut the tree for more fine-grained code cloning analysis, and therefore more sensitive to Type-3 cloning, so it has a higher recall.

The precision of *Tamer* exceeds that of *CCAligner*, *NIL*, *LVMapper*, *Deckard*, *Yang2018*, and *CCFinder*, and is slightly lower than that of *SourcererCC*, *Siamese*, and *NiCad*. This is because they sacrifice detection performance to improve their precision. It can be seen that their recall is very low. Therefore, compared with existing tools, *Tamer* has far better detection performance and equivalent precision. The experimental results prove our detection method is very effective.

5.4 RQ3: Scalability Evaluation

We use different sizes of the codebases to evaluate *Tamer*'s scalability and compare the execution time with existing tools. We use *IJaDataset* [1], a large inter-project Java dataset, as done in the prior studies [39, 44, 49]. We use *CLOC* [4] to measure the number

Table 2: Detection performance of Tamer, CCAligner, SourcererCC, Siamese, NIL, NiCad, LVMapper, Deckard, Yang2018, CCFinder, and CloneWorks on BCB

Tools		Tamer	CCA	Sou	Sia	NIL	NiCad	LVM	Dec	Yang2018	CCF	Clon
Recall	Type-1	**100**	100	94	100	99	98	99	60	100	100	100
	Type-2	**100**	100	78	96	97	84	99	52	100	93	98
	VST3	**100**	99	54	85	88	97	98	62	99	62	88
	ST3	**99**	65	12	59	66	52	81	31	73	15	64
	MT3	**53**	14	1	14	19	2	19	12	25	1	15
Precision		96	61	100	98	86	99	59	35	95	72	96

Table 3: Runtime overhead of Tamer, CCAligner, SourcererCC, Siamese, NIL, NiCad, LVMapper, Deckard, Yang2018, CCFinder, and CloneWorks when addressing different sizes of code

Tools	Tamer	CCA	Sou	Sia	NIL	NiCad	LVM	Dec	Yang2018	CCF	Clon
1K	1s	1s	3s	4s	1s	1s	1s	1s	5s	2s	1s
10K	1s	2s	5s	14s	1s	3s	1s	4s	16s	5s	2s
100K	3s	6s	7s	45s	3s	36s	4s	32s	2m7s	10s	6s
1M	13s	11m52s	37s	45m1s	11s	6m13s	34s	27m12s	1h45m3s	39s	43s
10M	11m40s	29m48s	12m21s	14h11m	1m3s	2h10m	22m10s	error	error	6m30s	10m37s

of rows in the data set, thereby dividing the 1K, 10K, 100K, 1M, and 10M LOC data sets. Our experiments are run on a quad-core CPU and 12GB of memory, as done in previous studies [39, 44].

In the second section, we have illustrated how *Tamer* performs subtree splitting and code fine-grained analysis. In this section, we will use experimental data to further explain. Table 3 shows the execution time of each tool for different-size datasets. It can be seen that *Tamer* has surpassed almost all detection tools under various sizes of datasets. *NiCad*, *CCAligner*, and *Siamese* use complex calculation formulas and intermediate code expressions to calculate the similarity of the two methods. *Deckard* and *Yang2018* are tree-based clone detection tool. They use tree presentation to calculate the similarity without optimization means which is undoubtedly complex and time-consuming, so their scalability performance is poor. *LVMapper* and *SourcererCC* both calculate the number of common tokens between two code blocks, so the algorithm complexity is low and the time consumption is relatively small. However, the complexity of calculating the public common token is O (N^2), so when comparing code blocks with a large number of two tokens, it will lead to a large increase in detection time.

In consideration of this problem, *Tamer* innovatively proposes to split the original complex AST into different types of subtrees, so that the subtree node-type sequence will also be shortened, significantly reducing the detection time. The experimental data also shows that our scalability is better than *LVMapper* and *SourcererCC*. However, we find that our detection time is slower than *NIL*, *CCFinder*, and *CloneWorks* when detecting 10M datasets. To take *NIL* as an example to illustrate the reason, we analyze its source code and find that *NIL* reduces the time-consuming steps such as code standardization in the pre-processing step. Although its detecting execution time is short, the detection performance decreases a lot. Besides, it can hardly detect Type-3 clones. In the real-world scenario, there are many Type-3 clones, so *NIL* is difficult to be applied to practical large-scale clone detection. *CCFinder* and *CloneWorks* both have this problem. *Tamer* has overcome the problem of low detection rate of Type-3 type cloning, and also has a very good performance in execution time, so we say *Tamer* has excellent scalability.

5.5 RQ4: Fine-Grained Clone Analysis

To better answer this question, we draw a part of the report produced by *Tamer*, as shown in Figure 8. For non-clone pairs, we do not report. For clone pairs, we not only give the results but also form the fine-grained analysis report. Take the source codes in Figure 8 as an example. We can see that (c1, c2) is regarded as a non-clone pair because the similarity of their subtrees is low. (c2, c3) is regarded as a clone pair because the similarity of their subtrees is high. In the report, we give a fine-grained analysis of the clone pair. We can see that the similarity of For block in c2 and c3 is 100%, and the corresponding code lines are 6-8 and 6-8 respectively. Because we abstract the source code and use node type to represent the node, the node sequence of For block is completely the same. The similarity of the If block in c2 and c3 is 59%, and the corresponding code lines are 2-3 and 3-4 respectively. After we analyze the source code, we find the judge condition of two If blocks are different, so their similarity is only 59%. As for the Structure subtree similarity, we can find it is only 53%. This is because the parameters of interface methods are different, and the way of local variables declaration is different. Besides, the position of the If block is different. As a result, the similarity of their structure subtree is only 53%.

It can be seen that our method of similarity analysis between subtrees can fully and accurately reflect the similarity relationship of corresponding parts of the source code. Therefore, we can use this method to locate the cloning parts of clone pairs and perform fine-grained analysis of clone pairs through the report produced by *Tamer*. The report above is only a simple example to illustrate this functionality of *Tamer*. Our real reports are more detailed.

5.6 RQ5: Impact of Splitting the AST

We attribute Tamer's advantage to the breakdown of the AST, and we further research the impact of splitting the AST. We perform an ablation experiment in this part. Specifically, we design another

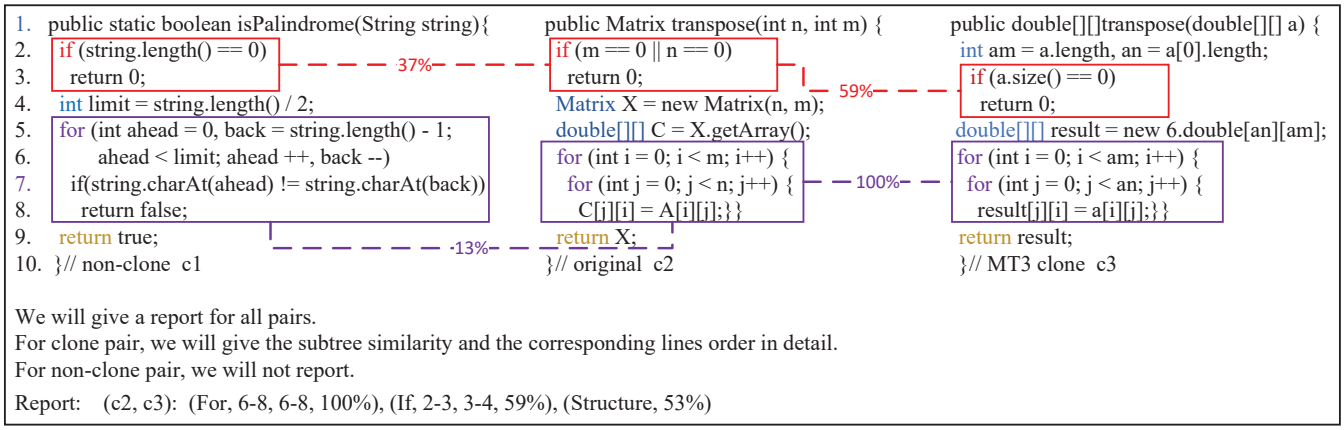

Figure 8: Fine-grain analysis report

Table 4: The ablation experiment of Tamer and Tamer without splitting (Tamer-ws)

Recall	Tamer	Tamer-ws	Runtime	Tamer	Tamer-ws
Type-1	100	100	1K	1s	1s
Type-2	100	100	10K	1s	1s
VST3	100	99	100K	3s	4s
ST3	99	96	1M	13s	18s
MT3	53	35	10M	11m40s	21m31s

tool namely *Tamer-ws* (**w**ithout **s**plitting) by directly processing the entire AST.

Table 4 shows the recall and execution time of two tools on the *BCB* dataset. As for scalability, it takes about 13 seconds for *Tamer* to detect clones from 1M lines of code, while *Tamer-ws* consumes about 18 seconds to complete the procedure. In addition, *Tamer-ws'* runtime overhead of the 10M dataset is nearly twice as much as *Tamer's*. As for effectiveness, *Tamer-ws* has an MT3 recall of 35% which is 18% lower than *Tamer*. Such results suggest that decomposing the AST can indeed accelerate the speed of detection and improve the effectiveness of detection. Therefore, we can find that splitting the AST can not only greatly improve the inspection speed of the system as mentioned in Section 2, but also improve the effectiveness of *Tamer*.

As for why splitting can enhance the effectiveness, we observe that each subtree obtained after decomposing the AST can represent a sub-behavior of the method. Some clone methods only have some sub-behaviors that are similar, while the other sub-behaviors are not similar which may result in the difference of their entire AST. Therefore, if we analyze the entire AST directly, other dissimilar sub-behaviors may become noise to interfere with clone detection. When we adopt more fine-grained sub-behaviors (subtree) matching, these similar sub-behaviors are easier to detect, allowing us to detect more clones.

6 Discussion

6.1 Threats to Validity

There are three main threats. Firstly, it is known that the performance and execution time of a clone detector is greatly influenced by its parameter settings and it is difficult to find the most suitable ones. To mitigate it, we conduct a series of exhaustive experiments to select the most suitable parameters. In detail, we select N-value from 10 to 18, and verification score δ from 0.55 to 0.80, and we finally choose 15 for N-value and 0.65 for δ. Secondly, because the physical state of the computer is different and the CPU runs at different times, the measurement of *Tamer's* execution time will be biased. To mitigate the threat, we repeat *Tamer* five times and then average it as a measurement to guarantee the validity and accuracy of the results. Thirdly, there is no clear standard for determining whether a pair of methods is a clone currently. So different researchers will have different results for manual verification and calculation of precision. To minimize errors, we scramble the clone pairs during the verification and it is not known in advance which tool detected these clones, so there is no bias in the evaluation.

6.2 Why Tamer Performs Better?

Tamer performs better than the other clone detectors on *BCB* datasets. The main reason is that the AST node sequence extracted by *Tamer* can retain the complete syntactic information of the source code to a great extent. For scalability, *Tamer* is also better than most current efficient methods (*i.e., LVMapper*), even if they are token-based detection tools. There are two main reasons for this. First, we do not directly use the complex AST to calculate the similarity. Instead, we traverse the entire AST and use the node type name to record the traversed nodes to get the node sequence. In this way, LCS can be used to calculate the similarity between sequences and we do not need to use the high time-cost tree similarity algorithm. In addition, we use a reasonable method to split AST into multiple types of statement subtree and a structure subtree, and then we use subtrees to calculate code block similarity.

6.3 Comparative Tools

In our experiments, we mainly choose token-based clone detectors to compare rather than intermediate representation-based clone detectors. The main reason is that most of the intermediate representation-based clone detectors use machine learning or deep learning methods to detect clones, but such detection methods have poor scalability. Because machine learning or deep learning tools require training sets, the larger the training set, the better the

training model. However, the labeled datasets of *BCB* are unrepresentative and not large enough, resulting in poor generalization of these models. So they can only perform well on the same dataset. For example, those models trained on *BCB* can only perform well on *BCB*, but the performance is poor when they are used to detect other datasets, such as *GCJ* [6]. Therefore, it is unnecessary to compare with these detectors.

6.4 Application of Tamer

Tamer, like most of the tools we compare, is primarily suitable for clone detection of general types. In fact, in addition to *CCAligner*, *LVMapper*, and *NIL*, the remaining seven tools are only suitable for general-type clone detection and not for large variance clone detection. This is because if a method with just a dozen lines of code is similar to a small part of a method with hundreds of lines of code, they are still considered a large variance clone. However, *Tamer* uses the intersection of the lengths of the two methods as the denominator, rather than *NIL* using the smallest of the two methods as the denominator, so it is hard for *Tamer* to detect the large variance clone.

Tamer determines whether a pair of methods is a clone based on the similarity between the blocks of the AST, and the relative position relationship between the blocks does not affect the detection results. Therefore, the effect of *Tamer's* detection of clones has nothing to do with the order in which the cloned code is written in the code, and the large order within the code is a feature of the large variance clone. So we can speculate if we use the minimum length of two methods as the denominator, *Tamer* may also achieve good performance in large variance clone detection. However, it will harm the precision of *Tamer* in general type clone detection, and how to make *Tamer* suitable for the detection of large variance is also our future work.

6.5 Limitation and Future Work

The main limitation of *Tamer* is that it can not implement the clone detection of the 100MLOC standard dataset on the 12GB memory limit. This is because *Tamer* needs to build an inverted index for fast searching of all files during the preprocessing phase, and the inverted index is extremely memory-consuming. Through our test, using the parameters mentioned in the paper to perform the clone detection of the 100MLOC dataset requires nearly 40GB of memory space. If the N value is set below 8, the clone detection of the 100MLOC dataset with the limit of 12GB memory can be realized, but this will greatly prolong the detection time. In addition, the configuration environment of the tool will also affect the detection performance [45]. We follow the environment configuration method previously studied, which seems to be standard [39].

In our paper, we mainly use AST as the code representation. Because we get the node sequence through the DFS of AST, the node sequence loses code information to some extent, and it is difficult to detect Type-4 clones. In the future, we will continue to research more reasonable methods of computing the similarity between trees, and consider conducting N-gram experiments based on graphs to detect clones of WT3 and even Type-4 clones, and further improve *Tamer*'s detection performance.

In addition, at present, *Tamer* only supports clone detection of Java files, but in fact, *Tamer* has high scalability. We only need to replace the Java source code parsing tool we currently use, Javaparser, with the parser tool of other programming languages to implement clone detection of other programming languages (*i.e.*, we can use *pycparser* [9] to parse C language and implement clone detection of C language code).

7 Related Work

7.1 Scalable Clone Detection

With the development of software engineering, the code size has become larger and larger. Many studies [12, 17, 23, 34, 41] have also confirmed that we need large-scale clone detection.

At present, most scalable detection tools are text-based or token-based. Their main idea is to calculate the similarity between code blocks in text or token sequences. The text-based [18, 27, 28, 32, 38, 44] methods directly converted the source code to a string for similarity calculation, while the token-based [19, 22, 35, 39, 49] methods used a lexical analyzer to analyze the source code to obtain the token sequence for similarity calculation.

CCFinder [28] converted source code into token sequences and standardized variables, then used the suffix tree algorithm to calculate the similarity between token sequences. Ishihara et al. proposed a method-level clone detection technology [23]. This tool calculated the hash value of each function after the function was standardized. Those with the same hash value were considered clones. Their method could detect the 360MLOC super large dataset within 3.5 hours.

In addition, some detection methods using GPU were proposed. Solutions for GPGPU programming include Nvidia's CUDA [8] and AMD's CTM [20]. *SAGA* [42] proposed an efficient detection method based on subsequent arrays. [40] used GPU to accelerate the dynamic programming algorithm. It improved the speed of clone detection. Using GPU was a good idea to improve detector scalability, and the algorithm of *Tamer* to calculate LCS between subtrees was also implemented by using the algorithm of dynamic programming. Therefore, we believe that using GPU to accelerate *Tamer* is also a part of our future work.

Some commercial tools and methods for clone detection of code repositories have also been proposed. Commercial tools such as *FOSSID* [5], *BlackDuck* [3], and *Scantist* [10] could handle clone code scanning for more than 10 million lines of code, but semantic clone detection was not yet supported.

In general, although the detection time of the tools mentioned above is very short, most of them perform poorly in Type-3 clone detection. Compared with these tools, *Tamer* is an excellent tool with both high detection performance and good scalability in detecting general code clones.

7.2 Complex Code Clone Detection

These years, with the popularity of learning algorithms in various computer research fields, some researchers start using learning algorithms in the software engineering field. Many tools combine tree or graph representations of code and learning algorithms have been proposed.

For tree-based detection methods, most of them are based on deep learning or machine learning. *ASTNN* [52] converted a complex AST into a small statement tree and sent it to the deep learning

model to overcome the long-term dependency problem caused by complex AST. *CDLH* [48] first normalized AST, then encoded the tree and converted it into a vector, and finally used the tree-based LSTM method for clone detection. *HELoC* [47] designed a comparative learning network, which could learn more specific information in the AST hierarchy to optimize detection performance. *InferCode* [13] deconstructed AST into the form of a subtree, and then extracted features from each node according to a tree-based convolutional neural network and coded them for similarity detection.

For graph-based detection methods [29, 31, 43, 50, 53, 54], most of them were also based on deep learning or machine learning. They were generally divided into *control flow graph* (CFG) and *program dependency graph* (PDG) according to the different graph types. Most detection methods used graph matching algorithms to detect the similarity of code blocks [29, 31]. Krinke was the first to propose using PDG to detect clones. He mainly analyzed whether there were isomorphic parts between different PDGs to determine whether the two code blocks were clones. He reported PDG-based clone detection method had high recall and precision. *FA-AST* [46] proposed a method to build a new AST by adding edges to control flow and data flow graphs. *FCDetector* detector [15] used a comprehensive intermediate to train the deep learning model. These graph-based algorithms had a large time cost, so *CCSharp* proposed two methods to reduce the time cost, namely, by modifying the structure of the graph and extracting feature vectors to reduce the complexity of the graph. *DeepSim* [53] extracted data flow and control flow of code to train a DNN-based detection model. *FCCA* [21] implemented an advanced graph-based clone detection tool by using hybrid code representations. *SCDetector* [50] regarded CFG as a social network and extracted the centrality of each code block for similarity detection.

However, these graph or tree-based detection methods require a long execution time and learning algorithms have the problems mentioned in Section 6.3. So they have poor scalability. On the one hand, the current graph-based extraction tools need to compile the source code first, and then get the graph. However, most of the code in the dataset can not be compiled at present, the compilation process is inconvenient and difficult to implement. On the other hand, both tree-based and graph-based methods are limited to complex source code representations, which naturally leads to a huge time overhead. To address the challenges, we propose to split the original tree into subtrees and design a scalable, effective, and fine-grained code clone detector namely *Tamer*.

8 Conclusion

In this paper, we propose *Tamer*, a scalable tree-based code clone detector with ideal performance. We first extract the AST of the source code, and then get the node sequence through depth-first traversal. On this basis, we use N-grams and inverted index ideas to perform locate and filter operations to obtain candidate clone pairs. Then we split the AST into different types of subtrees and calculate LCS between subtrees. Finally, we obtain the similarity between two code blocks, which is the basis for judging whether candidate clone pairs are true clones. The experimental results show that *Tamer* is far superior to the other ten clone detectors (*i.e., CCAligner* [44], *SourcererCC* [39], *Siamese* [36], *NIL* [35], *Nicad* [38], *LVMapper* [49], *Deckard* [25], *Yang2018* [51], *CCFinder* [28], and *CloneWorks* [41]) in detection performance, and also superior to most token-based detection tools in scalability.

Acknowledgement

Thanks to the anonymous reviewers for their insightful comments. This work was supported by the National Key R&D Plan of China (No. 2022YFB3103403) and the Hubei Province Key R&D Technology Special Innovation Project (No. 2021BAA032).

References

[1] 2022. Ambient Software Evolution Group: IJaDataset 2.0. http://secold.org/projects/seclone.
[2] 2022. BigCloneBench. https://github.com/clonebench/BigCloneBench.
[3] 2022. Blackducks. https://www.blackducksoftware.com.
[4] 2022. CLOC: Count lines of code. http://cloc.sourceforge.net.
[5] 2022. Fossid. https://fossid.com.
[6] 2022. Google Code Jam. https://code.google.com/codejam/contests.html.
[7] 2022. Javaparser. https://github.com/javaparser/javaparser.
[8] 2022. Nvidia. https://www.nvidia.com..
[9] 2022. Pycparser. https://github.com/eliben/pycparser.
[10] 2022. Scantist. https://scantist.com.
[11] 2023. Tamer. https://github.com/CGCL-codes/Tamer.
[12] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. 2005. Cloning by Accident: An Empirical Study of Source Code Cloning across Software Systems. In *Proceedings of the 6th International Symposium on Empirical Software Engineering (ESEM'05)*. 10–pp.
[13] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised Learning of Code Representations by Predicting Subtrees. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. 1186–1197.
[14] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. 2012. XIAO: Tuning Code Clones at Hands of Engineers in Practice. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. 369–378.
[15] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In *Proceedings of the 29th SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. 516–527.
[16] Jeanne Ferrante, Karl J Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.
[17] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from Here, Some from There: Cross-project Code Reuse in Github. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*. 291–301.
[18] Nils Göde and Rainer Koschke. 2009. Incremental Clone Detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. 219–228.
[19] Yaroslav Golubev, Viktor Poletansky, Nikita Povarov, and Timofey Bryksin. 2021. Multi-threshold Token-based Code Clone Detection. In *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*. 496–500.
[20] Mark Harris, Shubhabrata Sengupta, and John D. Owens. 2007. Parallel Prefix Sum (scan) with CUDA. *GPU gems* 3, 39 (2007), 851–876.
[21] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2020. Fcca: Hybrid Code Representation for Functional Clone Detection Using Attention Networks. *IEEE Transactions on Reliability* 70, 1 (2020), 304–318.
[22] Yu-Liang Hung and Shingo Takada. 2020. CPPCD: A Token-Based Approach to Detecting Potential Clones. In *Proceedings of the 14th International Workshop on Software Clones (IWSC'20)*. 26–32.
[23] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2012. Inter-project Functional Clone Detection toward Building Libraries-an empirical Study on 13,000 Projects. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*. 387–391.
[24] Yue Jia, David Binkley, Mark Harman, Jens Krinke, and Makoto Matsushita. 2009. KClone: A Proposed Approach to Fast Precise Code Clone Detection. In *Proceedings of the 3rd International Workshop on Detection of Software Clones (IWSC'09)*.
[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 96–105.
[26] Young-Bin Jo, Jihyun Lee, and Cheol-Jung Yoo. 2021. Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network. *Applied Sciences* 11, 14 (2021), 6613.

[27] Toshihiro Kamiya. 2021. Ccfinderx: An Interactive Code Clone Analysis Environment. (2021), 31–44.

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

[29] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Static Analysis Symposium (SAS'01)*. 40–56.

[30] Rainer Koschke. 2007. Survey of Research on Software Clones. In *Proceedings of the 5th Dagstuhl Seminar Proceedings (DROPS'07)*.

[31] Jens Krinke. 2001. Identifying Similar Code With Program Dependence Graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*. 301–309.

[32] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A Deep Learning-based Clone Detection Approach. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME'17)*. 249–260.

[33] Hongliang Liang and Lu Ai. 2021. AST-path Based Compare-Aggregate Network for Code Clone Detection. In *Proceedings of the 28th International Joint Conference on Neural Networks (IJCNN'21)*. 1–8.

[34] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2017. Does cloned code increase maintenance effort?. In *Proceedings of the 11th International Workshop on Software Clones (IWSC'17)*. 1–7.

[35] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: Large-scale Detection of Large-variance Clones. In *Proceedings of the 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'21)*. 830–841.

[36] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: Scalable and Incremental Code Clone Search via Multiple Code Representations. *Empirical Software Engineering* 24, 4 (2019), 2236–2284.

[37] Chanchal Kumar Roy and James R. Cordy. 2007. A Survey On Software Clone Detection Research. *Queen's School of computing TR* 541, 115 (2007), 64–68.

[38] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate Detection of Near-miss Intentional Clones Using Flexible Pretty-printing and Code Normalization. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*. 172–181.

[39] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. [n. d.]. SourcererCC: Scaling Code Clone Detection to Big Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'15)*. 1157–1168.

[40] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-lehman Graph Kernels. *Journal of Machine Learning Research* 12, 9 (2011).

[41] Jeffrey Svajlenko and Chanchal K. Roy. 2017. CloneWorks: A Fast and Flexible Large-scale Near-miss Clone Detection Tool. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. 177–179.

[42] Jeffrey Svajlenko and Chanchal K. Roy. 2019. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Transactions on Software Engineering* 47, 5 (2019), 1060–1087.

[43] Min Wang, Pengcheng Wang, and Yun Xu. 2017. CCSharp: An Efficient Three-phase Code Clone Detector Using Modified PDGs. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. 100–109.

[44] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: A Token Based Large-gap Clone Detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 1066–1077.

[45] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE'13)*. 455–465.

[46] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-augmented Abstract Syntax Tree. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. 261–271.

[47] Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. Heloc: Hierarchical contrastive learning of source code representation. In *Proceedings of the 30th International Conference on Program Comprehension (ICPC'22)*. 354–365.

[48] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conferences on Artificial Intelligence Organization (IJCAI'17)*. 3034–3040.

[49] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K. Roy. 2020. LVMapper: A Large-variance Clone Detector Using Sequencing Alignment Approach. *IEEE Access* 8 (2020), 27986–27997.

[50] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'20)*. 821–833.

[51] Yanming Yang, Zhilei Ren, Xin Chen, and He Jiang. 2018. Structural Function Based Code Clone Detection Using a New Hybrid Technique. In *Proceedings of the 42nd Annual Computer Software and Applications Conference (COMPSAC'18)*, Vol. 1. 286–291.

[52] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 783–794.

[53] Gang Zhao and Jeff Huang. 2018. Deepsim: Deep Learning Code Functional Similarity. In *Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*. 141–151.

[54] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: A PDG-based Code Clone Detector with Approximate Graph Matching. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'20)*. 931–942.

Received 2023-02-16; accepted 2023-05-03