Journal of Applied Sciences 13 (20): 4270-4277, 2013
ISSN 1812-5654 / DOI: 10.3923/jas.2013.4270.4277

Defect Detection Technique Based on Similarity Comparison of Token

[1]Guo Tao, [1]Dong Guowei, [1]Qin Hu, [2]Long Baolian, [2]Qu Tong and [2]Cui Baojiang
[1]China Information Technology Security Evaluation Center, Beijing, China
[2]School of Computer, Beijing University of Posts and Telecommunications, Beijing, 100876, China

Abstract: Similarity detection technique of code has been widely used. At present, the technique is mainly used for the judgment of plagiarized code. In this study, on the basis of code similarity detection technique, we put forward a static analysis method of defect detection of source code, it adopts lexical analysis and comparison which based of token. Similarity comparison of defective samples can also been achieved in this way. After testing result and comparing with other similar tools, the result demonstrate that the static analysis method based on similarity comparison of token structures defective sample much more easily than others and it has a good expansibility not only about programing language but also in the point of different defect type. The same type defect can be detected thereby help programmer avoid having the same defective code.

Key words: Similarity detection, token-based comparison, defect detection

INTRODUCTION

Similarity detection of code is comparing two source codes and find out the similar or same code Segment (Han *et al*., 2010). Similarity detection technique of code has been widely used for the detection of plagiarized source code (Cui *et al*., 2011). Similarity detection of code achieved by analyze the morphology, syntax and static semantic.

The main method of defect detection of code is static analysis technique, static analysis can analyze the code without modifying, it can completely cover the code, independent of compiler, more convenient and faster than dynamic detection (Gu *et al*., 2008). The static analysis method of defect detection of code can mainly been divided into three types, it includes type inferencing, data flow analysis and constraint analysis etc (Zhou *et al*., 2002). Concrete method just like static detection based on the fault caused by reference null pointer, static detection based on path sensitizing (Xiao *et al*., 2010), static detection based on procedural and cross-process array cross the border, static detection based on memory error, static detection based on path sensitizing.

Software defect detect techniques based on static analysis which we discussed above has their own strong points, at the same time, they all have a clearly disadvantage, that is the defect analysis is restricted to the particular language or particular defect type when the analysis based on variable information, data link information, path information and semantic information. Because of those factors, all the methods have limitation in range of application and expansibility. However, the system we raised can conveniently and effectively detect different code languages and plagiarize types, furthermore, the new plagiarize type can be easily added into our system.

In order to detect defects more effectively, on the basis of importing similarity detection of code, this study put forward a static analysis method base on token similarity comparison. The most obvious feature of the method in this study is the good expansibility about language type and defect type when we detect the defects. Once we structure the corresponding defect sample of code. It will easily detect various languages. For certifying the method, we have an exploratory research in this study, structure defect sample of C, C++ and Java. In the following research, we will broaden it into the other langue and lager number of defect type.

SIMILARITY DETECTION TECHNIQUE OF CODE BASED ON TOKEN

This system proposes a token-based comparison to detect defects of software via compiling lexical grammar rules which could extend to multilingual comparison handily.

The method of similarity detection, a token-based software comparison technique, is breaking source codes which include various function names, variables, parameter, types of identifiers, operators, numbers and keywords, into token via lexical analyzer and then transforming comparison of source codes into comparison

Corresponding Author: Guo Tao, China Information Technology Security Evaluation Center, Beijing, China

Table 1: Lex translate characters into token

Source code character	Translate into token	Example
Identifier↵	Translate into_ID↵	ipszname->_ID↵
Operator↵	Translate into_ID↵	+->_PLUS↵
Number↵	Translate into_NUM↵	78->_NUM↵
Key words↵	Translate into relevant letter↵	Unsigned->UNSIGNED↵

of token sequences. Specific implementation process includes the following main steps.

Pre-processing source codes: This step does relative processes of notes, redefinitions of type, header files containing statements macros, inline functions and so on, in source codes. Since some characters do not affect the semantics, they are recorded empty in pre-processing, such as macro definitions, notes, TABs, enters, spaces, etc. Conditions of redefined types processes according to their semantic, for instance, processing of type redefinition statement “typedef int INTEGER” is replacing INTEGER with int in the file to improve the accuracy of similarity detection.

Lexical analysis of the source code: This process is achieved by using lexical analysis tool Lex (Lexical compiler) which generates lexical analyzer reading the source codes by characters and then returning Token. The main process of characters shows in Table 1.

This study processes program mainly depending on morphology compiled by lexical analysis tools, Lex, then generates. cpp files and separates the input source files into single strings through executing lexical analysis program. Then source codes are transformed into token sequences to get ready for following token-based comparison.

Table 2 and 3 are instances illustrate the translate process.

Receiving token sequence returned from source codes: First, source codes are turned in to token sequences. Then, middle files of comparison are output and line lists of base files and target files are established which records message digest values of middle files based on line to do comparison, whose results are deposited into same line lists to calculate similarities.

Achieving line lists which records information of files: Definite line list node as following:

```
typedefstruct_line        //record the structure of the line ↵
{↵
DWORD dwLine;             //start byte of the line ↵
DWORD dwStart;            //end byte of the line ↵
DWORD dwHash;             //abstract of the line ↵
Struct_LINE*next;         //next node ↵
Struct_LINE*pre;          //prior node ↵
}LINE, *LPLINE;↵
```

Table 2: Instance of source code

```
int func(in x)↵
{↵
int m;↵
return m;↵
}↵
```

Table 3: Instance of translate into token

```
int func(int x)↵
{↵
int m;↵
m = f1(x)↵
return m↵
}↵
```

This records the position of beginning letter in each line (starting from line 1), the number and the abstract of this line in files.

Doing a quick and unique abstract for each line in codes is a necessary function for a program whether it could process efficient and algorithm of line abstract is adopted in this study. We use line abstract to analyze intermediate file. Ignored blank space and TAB, using token multiply by corresponding default value of it.

Algorithm of line abstract is as following:

$$\text{Hash}(\text{Line}) = \sum_{k=0}^{n} i * \text{Asc}(\text{token}[i]) \tag{1}$$

Asc (token [i]) is the custom value ith token of this line, i is the sequence number of this line. For instance: The token of “int a” is _INT _ID _SEMI and the abstract of this line is 105 (_INT) * 1 + 110 (_ID) * 2 + 116 (_SEMI) * 3 = 673.

This equation could calculate abstract of current line with high speed and no repeat. Due to the comparing item is coding, spaces, TABs which has no effect on program function, this method of abstract is correct and rigorous.

Comparing base files and target files: Same line lists are established and the rows with same message digest values are stored in the same row list via comparing the message digest values of linked lists in base files and target files. The algorithm is as follows.

All nodes reading from target files are stored in Link T and all nodes reading from sample files are stored in Link S. First, the i^{th} code from array T and the j^{th} code from array S are marked as T[i], S[j], i = j = 1and then their Hash value are compared.

If Hash:

$$\{T[i]\}>\text{Hash}\{S[j]\}$$

then i is increased by 1

If Hash:

$$\{T[i]\}<\text{Hash}\{S[j]\}$$

then j is increased by 1

$$\{T[i]\} = \text{Hash}\{S[j]\}$$

then similar codes exist in this two nodes. The information of this two nodes are stored in list Y to be processed next step.

The algorithm circles until i = m and j = n which means the terminal of comparison.

Calculating similarities: Calculating similarities is based on stored information of same line lists among which the unit of similarity is line and the value is the ratio of numbers of similar lines and total lines. Following formula calculates is used to calculate similarity SD of target source codes to sample source codes:

$$SD = \frac{\sum f(StN_x)}{LC} \quad (2)$$

StNx is the same nodes compared in line list Y in 5. LC is total active lines in target source code (excluding comments and blank outside the coding block). Function calculate the numbers of same lines defined as following.

Lines number of two files is not the total sum of same lines achieved by line comparison because simple adding the numberof lines may generate a larger number than actual due to different lines of similarity may overlap. To solve this problem, the system compiles a function f to calculate the number of similar rows. The algorithm is to establish an array of INT type which is same size as the source file as. And then the same line list are checked, the elementin array will be turned to 1 while the corresponding lines are included in list and then the number of similar lines are the numbers of 1.

Storing comparing results into database: The comparing results will be stored into SQLite database for administrative purposes. Storing comparing results into database means that results can be see all the time and similar results for each part of the next step can be performed in-depth research. Comparing flowchart is shown in Fig. 1.

The core base part of similar authentication module of software based on token similarity detection technology is line alignment algorithm. The main idea is to compare token sequences based on line which could compare the totally exact coding blocks and copies of coding blocks with disrupted order. LCS (Longest Common Subsequence) (Yu and Zhao, 2008) is taken to find the same lines in this study.

Procedure to solve the problem of LCS is to use a matrix to record matching solution of the two characters in two different strings. If they are matched, record is 1. Otherwise, record is 0. Then, the longest diagonal 1

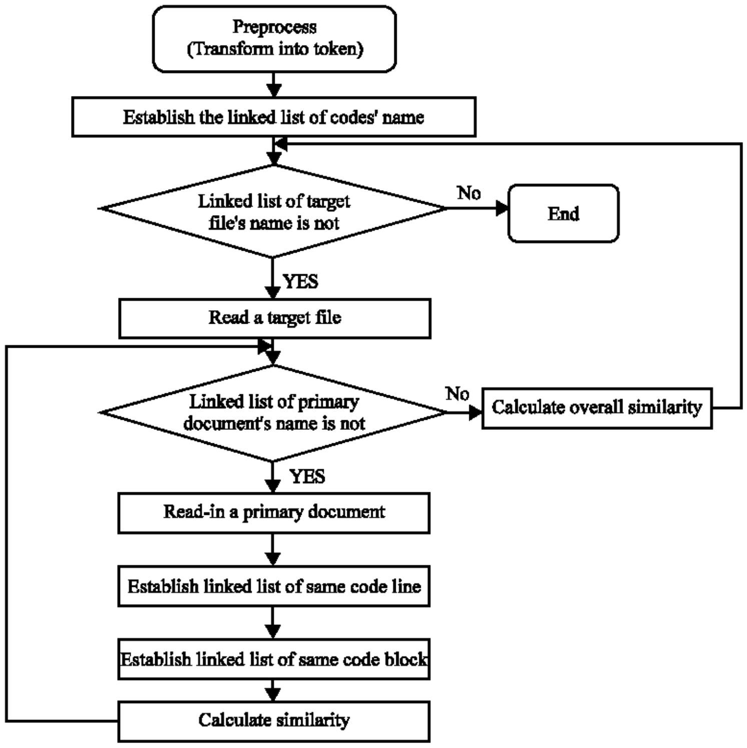
Preprocess (Transform into token)
Establish the linked list of codes' name
Linked list of target file's name is not
No
End
YES
Read a target file
Linked list of primary document's name is not
No
Calculate overall similarity
YES
Read-in a primary document
Establish linked list of same code line
Establish linked list of same code block
Calculate similarity

Fig. 1: Flow chart of comparison based on token

(a)

↵	I	l	o	v	e	C	h	i	n	a
l	0	1	0	0	0	0	0	0	0	0
o	0	0	1	0	0	0	0	0	0	0
v	0	0	0	1	0	0	0	0	0	0
e	0	0	0	0	1	0	0	0	0	0
e	0	0	1	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	1	0	0	0	0
h	0	0	0	0	0	0	1	0	0	0
i	0	0	0	0	0	0	0	1	0	0
n	0	0	0	0	0	0	0	0	1	0
a	0	0	0	0	0	0	0	0	0	1

(b)

↵	I	l	o	v	e	C	h	i	n	a
l	0	1	0	0	0	0	0	0	0	0
o	0	0	2	0	0	0	0	0	0	0
v	0	0	0	3	0	0	0	0	0	0
e	0	0	0	0	4	0	0	0	0	0
e	0	0	1	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	1	0	0	0	0
h	0	0	0	0	0	0	2	0	0	0
i	0	0	0	0	0	0	0	3	0	0
n	0	0	0	0	0	0	0	0	4	0
a	0	0	0	0	0	0	0	0	0	5

Fig. 2(a-b): Description of algorithm (a) Before the improvement and (b) After the improvement

sequence is found, as shown in Fig. 2a which shows the corresponding position of the longest matching substring.

The disadvantage of this algorithm is to store the adjacency matrix which will take up a lot of memory when the two string length is large. Therefore, LCS algorithm is improved in this study. While characters are matched, we are not simply assigned to the corresponding element by one but endowed with its upper left corner of the element's value plus one. We use two markers marking the positions of the max variable values in the matrix elements to determine whether the current value of the element generated is the greatest in generation process of the matrix, according to which we changes the value of marked variables. Then the position and length of the longest substring matching has come out when the matrix is completed, as shown in Fig. 2b.

A sample of testing

Model construction of defected coding: This study constructs more than 1500 testing samples of common types of defect for three languages: C, C++ and JAVA.

Including

Type mismatch: Signed to Unsigned, whose declaration is returning an unsigned value. However, a signed value is returned.

Poor style: Value Never Read. Variable assignments are not used, thus they become dead stores. Redundant Null Check. The program may indirect reference a null pointer, thereby it causes segmentation fault. Uninitialized Variable. Program may use a variable before initialization.

Often misused: Exception Handling Enter Critical Section() which would throw an exception and result in the collapse.

Often misused: Strings. Conversion functions between multi-byte and Unicode character could easily cause buffer overflow. Dead Code. Instruction will never be executed. Heap Inspection. Never use realloc () to adjust the size of the buffer store with sensitive information, because the function might leave sensitive information in memory which cannot be covered in the memory; Missing Check against Null. Program will indirectly reference a null pointer, because it does not check the return values of functions which possible returning Null; Out-of-Bounds Read. The program reads data from outside bounds of the allocated memory; Poor Style: Redundant Initialization. Variable assignment does not use and becomes a dead storage. All the testing samples are divided into 68 kinds of categories and subdivided into 90 specific species.

Construction of testing samples: In the construction of testing samples, we first find out a piece of code including the type of the defect and then estimate whether it is context related. If it is context related, the similarity test has some limitations which needs improvements in future. If it is not context related, Token processing is used on a removal complete block to make the fragment have a better universal. The data obtained at this time is the construction of this sample.

Take uninitialized variable as an instance: Stack variables of C and C ++ language is uninitialized by default. Their initial value depends on what happened in their stack when functions are called. Uninitialized variable should never be used in program. Programmers will usually use uninitialized variables in code to handle errors or some special and unusual circumstances. Warnings of uninitialized variables are sometimes able to identify typographical errors existing in the code.

Hence, a complete block of uninitialized variable is as following:

```
swith (ctl) {↵          aN = i;↵                  break ↵
case –1:↵               bN = i ↵                  default:↵
aN = 0; bN = 0 ↵        break;↵                   aN = -1 ↵
break;↵                 case 1:↵                  aN = -1;↵
case 0:↵                aN = I+NEXT_SZ;↵          break;↵
                        bN = I-NEXT_SZ;↵          }↵
```

In the sample above, switch statement attempts to assign values of variables aN and bN, however, programmers will accidentally assign the value of aN twice and ignore bN in default cases.

Most problems of uninitialized variable are caused by reliability of software. If attackers can intentionally trigger the use of an uninitialized variable, then they can trigger crashes to launch a denial of service attack. Attackers can affect the value of the stack before calling a function to control an uninitialized variable value under appropriate circumstances.

Token fragments of the above procedures give the test samples as follows:

1: 59 Reserved Token↵	0.286806↵	9: 79 ID, name = bN↵
swit↵		
1: 101(↵	6: 133 /n↵	9: 98 Operator↵
1: 79 ID, name = ctl↵	6:79 ID, name = I↵	9: 86 Operator↵
1: 102)↵	6:79 ID, name = I↵	9: 86 Operator↵
1: 105 {↵	0.31875↵	0.44375↵
2: 133 \n↵	6:79 ID, name = bN↵	10:133 \n↵
2: 9 Reserved Token:↵	6:98 Operator↵	10: 6 Reserved Token:↵
break↵		
case↵		
2: 86 Operator↵	6: 79 ID, name = i↵	11: 133 \n↵
2: 80 NUM, val = 1↵	0.31875↵	11:15 Reserved Token;↵
defa↵		
0.175694↵	7: 133 \n↵	0.550694↵
3: 133 \n↵	7: 6 Reserved Token↵	12: 133 \n
3: 79 ID, name = aN↵	brea↵	12: 79 ID, name = aN↵
3: 98 Operator↵	0.360417↵	12: 98 Operator↵
3: 80 NUM, val = 0↵	8: 133 \n↵	12: 86 Operator↵
0.19375↵	8: 9 Reserved Token↵	12: 80 NUM, val = 1↵
3: 79 ID, name = bN↵	case↵	0.56875↵
3: 98 Operator↵	8:80 NUM, val = 1↵	12: 79 ID, name = aN↵
3: 80 NUM, val = 0↵	0.411806↵	12:98 Operator↵
0.19375↵	9: 133 \n↵	12: 86 Operator↵
4: 133 \n	9:79 ID, name = aN↵	12: 80 NUM, val = 1↵
4: 6 Reserved Token↵	9:98 Operator↵	0.56875↵
brea↵	9:79 ID, name = i↵	13:133 \n↵
0.235417↵	9:85 Operator↵	13: 6 Reserved Token:↵
brea↵		
5: 133 \n↵	9:79 ID, name = NEXT↵	0
EOF		
5: 9 Reserved Token↵	0.44375↵	
case↵		
5: 80 NUM, val = 0↵	14: 106 }↵	
5: 80 NUM, val = 0↵	0.717361↵	
14: 133 \n↵		

Now, a competed testing sample is constructed which can be used for defected codes testing of Uninitialized Variable type.

RESULTS

Similarity analysis using the defect results from defected sample database test: The key of token-based similarity analysis system used in defect detection is to construct testing samples. Constructed results above can detect risk of defects within codes to the point. The following constructed 1005 test samples, for C and C++ language, covering 38 kinds of defect types. Tools of code defect detection, Code Compare which is a token-based similarity analysis technique, is developed to analyze defects of source codes, based on constructed defected samples, of eMule, vlc, iometer and other software. The number of detected defects of software is shown in Table 4.

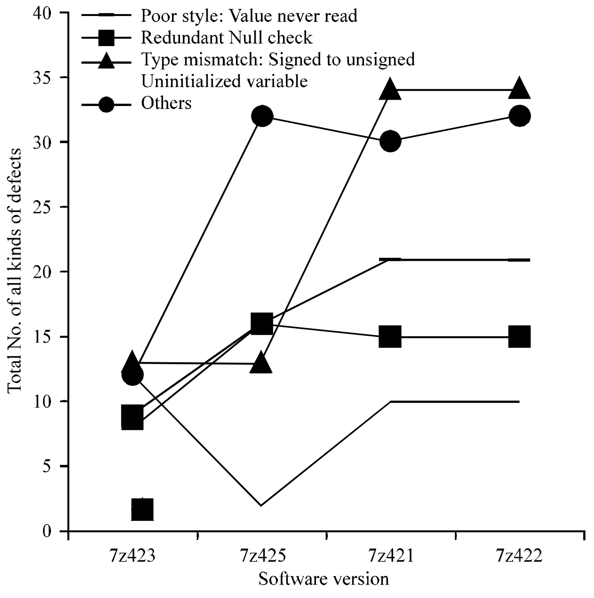

Fig. 3 Solution of defect detect 7zp

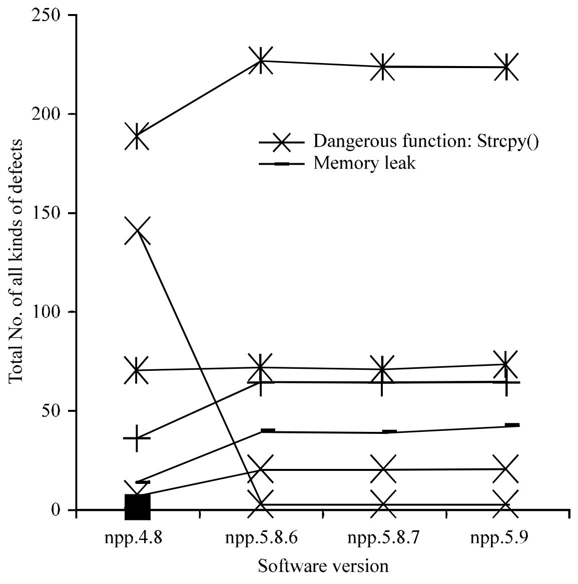

Fig. 4: Solution of defect detect of npp

As can be seen from above testing results, the number of defect which is tested out of five software above, is 5959. Following is the defect analysis for the same software of different versions. This study uses 7zip and npp. The results by Code Compare analysis tools for defect detection of 7zip of different versions are shown in Fig. 3.

The results of software flaw test for notepad ++ of different versions by Code Compare tools are shown in Fig. 4.

Table 4: Amount of defects detected in various softwares

ID	Catogory	eMule 0.50a	vlc0.8.6e	iometer20060727	optipng-0.5	optipng-0.6
1	Buffer overflow: Format string	3	5	0	0	0
2	Buffer overflow: Off-by-One	0	4	0	0	0
3	Buffer overflow: Signed comparison	0	1	0	0	0
4	Code correctness: Memory free on stack variable	1	0	0	0	0
5	Command injection	25	0	1	0	0
6	Dangerous function: Strcpy()	60	2	63	9	20
7	Dead code	80	4	10	10	10
8	Double free	0	9	0	7	0
9	Format string	971	0	68	0	2
10	Format string: Argument type Mismatch	6	0	2	0	0
11	Heap inspection	3	139	4	1	1
12	Insecure compiler optimization	0	0	0	1	1
13	Insecure randomness	41	3	0	0	0
14	Memory leak	400	28	45	40	51
15	Memory Leak: Reallocation	0	124	0	0	0
16	Missing Check against Null	21	171	1	1	1
17	Null Dereference	6	6	0	1	1
18	Obsolete	24	5	3	0	0
19	Obsolete: Inadequate pointer validation	4	0	0	0	0
20	Often misused: Authentication	11	0	4	0	0
21	Often misused: Exception Handling	4	2	0	0	0
22	Often misused: File system	15	0	0	0	0
23	Often misued: Strings	1	0	0	0	0
24	Out-of-bounds read	0	1	1	0	0
25	Out-of-bounds read: Off-by-one	2	0	0	0	0
26	Password management: Password in comment	14	2	0	0	0
27	Poor style: Redundant initialization	41	37	1	0	0
28	Poor style: Value never read	104	25	5	6	24
29	Poor style: Variable never used	145	0	0	0	0
30	Process control	21	5	1	0	0
31	Race condition: File system access	1	0	0	0	11
32	Redundant Null check	19	2	1	26	16
33	String termination error	0	0	0	9	11
34	Type mismatch: Signed to unsigned	44	0	1	7	5
35	Unchecked return value	621	25	24	1	3
36	Uninitialized variable	11	0	22	0	4
37	Unreleased resource	2	2	0	0	0
38	Use after free	0	4	0	10	0
39	Total	2701	606	257	129	161

As can be seen via analysing the experimental results of the comparison above: The same test sample can be used to detect different software or different versions of similar defects in code; the same buggy code likely continue to occur in different versions of the same software; different software types of defects that appear are quite different; appearance of the same risks can be avoided in the next version via code analysis on different versions of the same defected software.

Comparison results with other static analysis codes of defect detection software.

Take Java source code for instance. Comparisons are made among Code Compare which based on token similarity comparison technique and other defect detection tools like FindBugs and Lapse+. This experiments use544 test samples for aTunes, freecol, freemind, jstock, MegaMek, robocode and other software source, whose codes are analyzed. Results of similarity analysis system are shown in Table 5.

The experimental results of code defect analysis by Java code defect analysis tool FindBugs (version 2.0) are shown in Table 6.

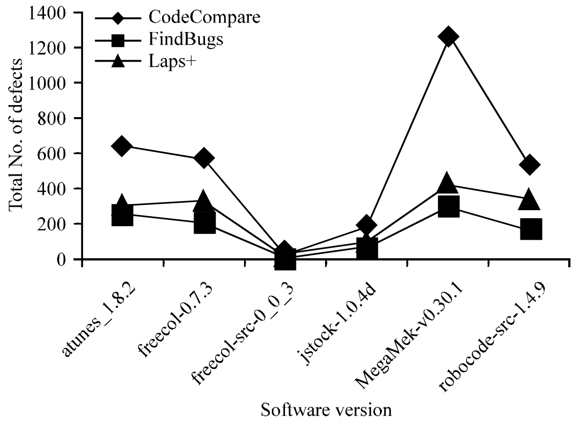

Fig. 5: Total No. of defects line chart

The defect detection results of analysis code by Java static analysis tools Lapse + (version 2.8.1) are shown in Table 7.

Figure 5 shows the total defected numbers of testing by three kinds of detection tools for six software. This shows that defected numbers by matching technique tool

Table 5: Amount of the defects of six software detected by code compare

ID	Catogory	aTunes 1.8.2	freecol-0.7.3	freemind-src-0_0_3	jstock-1.0.4d	MegaMek-v0.30.1	Robocode-1.4.9
1	Code Correcness	160	6	1	18	82	31
2	Command injection	0	1	0	0	0	10
3	Dead code	6	11	1	13	36	6
4	Denial of service	39	11	0	0	81	34
5	Insecure randomness	6	49	0	1	3	18
6	J2EE bad practices	133	108	2	41	53	81
7	Missing check against Null	4	3	0	1	3	15
8	Missing check for Null parameter	0	0	0	0	3	0
9	Missing security manager check	0	0	0	0	0	21
10	Missing XML validation	1	11	0	1	0	0
11	Null dereference	0	8	0	1	91	1
12	Object model violation	0	3	0	1	15	1
13	Often misused	0	4	0	0	8	0
14	Password management	11	0	0	16	5	0
15	Poor error handling	110	119	8	41	122	138
16	Poor logging practice	25	162	10	20	560	112
17	Poor style	16	19	9	16	84	5
18	Race condition: Format flaw	1	0	0	9	7	2
19	Redundant Null check	0	31	0	3	24	0
20	System information leak	110	0	0	0	0	0
21	Unchecked return value	3	6	0	0	49	26
22	Unreleased resource: Streams	13	20	4	2	35	17
23	Weak cryptographic hash	1	0	0	0	0	0
24	Weak encryption	4	0	0	0	0	0
25	Weak security mangager check: Overridable method	0	0	0	0	0	13
26	Total	643	572	35	184	1261	531

Table 6: Amount of the defects of six softwares detected by FindBugs

ID	Catogory	aTunes _1.8.2	freeco l-0.7.3	freemind-src-0_0_3	jstock-1.0.4d	MegaMek-v0.30.1	Robocode-1.4.9
1	Call to equals () comparing different types	17	13	1	9	28	19
2	Check for oddness that won't work for negative numbers	11	20	1	8	35	22
3	Class defines non-transient non-serializable instance field reader TypeInfo						
4	Load of known null value, improper use of null	11	17	1	5	27	17
5	Method may fail to close database resource	9	8	0	5	15	11
6	Method might ignore exception	12	6	0	3	13	7
7	Nullcheck of value perviously dereferenced	11	5	0	5	11	3
8	Possible null pointer dereference	6	4	0	2	11	3
9	Comparison of packaging should use "eueqls", to compare the value type, you need to mandatory conversion before use	27	3	0	1	8	2
10	Do not use new string ()defines the empty string	34	27	1	11	27	28
11	Method naming convention, the firdt letter lowercase	16	58	2	10	49	27
12	Inner class does not reference an external class properties/methods, it should as a static inner class						
13	A primitive type value after box immediately unbox	21	31	1	9	28	21
14	Empty check needs to be done before refrence	48	4	0	2	11	2
15	Total	257	211	7	75	302	167

Table 7: Amount of the defects of six software detected by lapes+

ID	Catogory	aTunes 1.8.2	freeco l-0.7.3	freemind-src-0_0_3	jstock-1.0.4d	MegaMek-v0.30.1	Robocode-1.4.9
1	Ignores exceptional return value of javai.o.File.mkdirs()	67	49	4	13	61	43
2	Method may fail to close stream	82	7	0	4	13	10
3	Possible null pointer dereference in method on exception path	23	4	0	2	11	2
4	Invocation of to string on values	13	61	4	20	81	60
5	Method concatenates strings using+in a loop	82	11	0	5	23	14
6	Dead store to new status record	12	67	8	25	91	90
7	Write to static field from instance method	13	3	0	1	7	2
8	Comparison of string objects using = = or ! =	15	124	10	31	121	112
9	Method call passes null for nonnull parameter	8	11	0	5	24	15
10	Total	315	337	26	106	432	348

CodeCompare based on token similarity analysis is higher than others, because Code Compare can construct more safe missing defected samples easily and the number of types of defect which can be tested is higher than others which is shown in Fig. 6. Hence, the number of types of defect for six software is the highest.

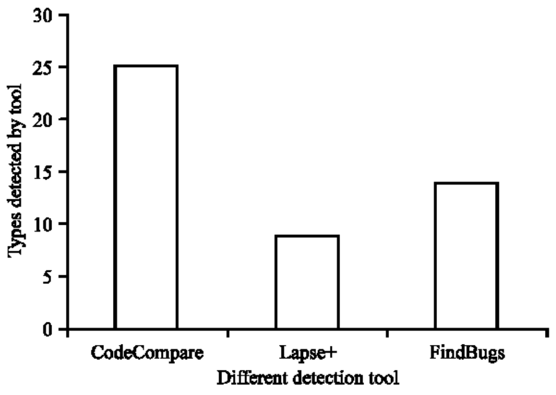

Fig. 6: Amount of defect detected by three tools

Figure 5 shows the general trend of testing by three kinds of detection tools for six software is the same which indicates the similarity of CodeCompare with Findbugs, lapse + and other tools, all of which have the function to detect defected codes.

As can be seen by comparing, matching technique tool CodeCompare based on token similarity analysis, compared with other tools like Lapse+, FindBugs and so on, could easily detect more types of defects via constructing testing samples. Hence, CodeCompare supports more types of defects and constructs testing samples more handly. Above three tools have different emphases and more defect types can be detect by CodeCompare.

CONCLUSION

According to the analysis above. Similarity detection technique based on token can be applied to static defect detection of source code. Compared with other detection method, it is more convenient structure and expansion samples used for defect detection, specifically, compare with Lapse+ and FindBugs, it supports more types of code. For the different versions of a soft ware, it can effectively find the same type defect and avoid it occurring in the subsequent software. Admittedly, this technique has misinformation sometimes, this need to be solved in the following study.

ACKNOWLEDGMENTS

This study was supported by National Natural Science Foundation of China (No. 61170268, 61100047 and 61272493); International S and T Cooperation Special Projects of China (No. 2013DFG72850); The National Basic Research Program of China (973 Program) (No.2012CB724400).

REFERENCES

Cui, B., J. Guan, T. Guo, L. Han, J. Wang and Y. Ji, 2011. Code syntax-comparison algorithm based on type-redefinition-preprocessing and rehash classification. J. Multimedia, 6: 320-328.

Gu, K., C. Liu and M. Jin, 2008. Customizable defect rules of research and implementation of the c + + code defect detection tool. Proceedings of the National Conference on Adaptive Hardware and Systems, June 22-25, 2008, Netherlands.

Han, L., B. Cui, J.X. Wang and Y. Hao, 2010. Type redefinition plagiarism detection of token-based comparison. Proceedings of the 2nd International Conference on Multimedia Information Networking and Security, November 4-6, 2010, Nanjing, China, pp: 351-355.

Xiao, Q., C. Yang and Y. Gong, 2010. Improve precision of the static defect detection methods. Comput. Aided Design Graph. J., Vol. 22,

Yu, H. and J. Zhao, 2008. The longest common subsequence algorithm application in program code similarity metric. J. Inner Mongolia Univ., 39: 255-229.

Zhou, D., N. Li and Y. Yang, 2002. Detecion and deletion on vicious pocedures. Microelectron. Comput., 1: 11-14.