Character String Predicate Based Automatic Software Test Data Generation

Ruilian Zhao
Computer Science Dept.
Beijing University of Chemical Technology
Rlzhao@mail.buct.edu.cn

Michael R. Lyu
Computer Science Dept.
Chinese University of Hong Kong
Lyu@cse.cuhk.edu.hk

Abstract

A character string is an important element in programming. A problem that needs further research is how to automatically generate software test data for character strings. This paper presents a novel approach for automatic test data generation of program paths including character string predicates, and the effectiveness of this approach is examined on a number of programs. Each element of input variable of a character string is determined by using the gradient descent technique to perform function minimization so that the test data of character string can be dynamically generated. The experimental results illustrate that this approach is effective.

1. Introduction

As an important stage to guarantee software quality and reliability, software testing plays an irreplaceable role in the process of software development. Although a large number of software testing approaches have been developed to detect either data flow or control flow faults [1, 2], all current software testing approaches have been limited to programs whose predicates can contain Boolean variables, relational expressions or binary Boolean operators, but not character string variables. This overly reduces software testing approaches for applications in practice since character string predicates are widely used in programming.

To test a program, it is necessary to generate test data from the input domain of the program under test. As a program input domain is usually too large to be exhaustively exercised, the usual way for testing is to select a relatively small subset to represent. Therefore, a key issue in software testing is how to generate adequate test data from the program input domain to detect as many faults as possible with a minimum cost. Obviously, if test data could be automatically generated, the cost of software testing would be significantly reduced.

At present, there are many automatic test data generation approaches, such as random test generation [3], symbolic execution-based test generation [4, 5], rule-based test generation [6], constraint-based test generation [7] and dynamic test generation [8, 9, 10]. Each approach has its own advantages; however, little attention has been paid to the test data generation for programs that contain character string predicates.

In the research reported in this paper, we present a novel approach to automatically generate test data for program paths that include character string predicates, and a corresponding test data generator is developed. Dynamic test data generation is a popular approach for developing test data. It is employed in our test data generation. In essence, the problem of dynamic test data generation can be formulated and reduced to a function minimization problem [9, 10], and gradient descent is considered as a standard function minimization technique [8, 10]. Hence, we use gradient descent to perform function minimization and determine each character element of input variables so that the test data of character string can be dynamically generated. The effectiveness of this approach is examined on a number of programs. The experimental results illustrate that this approach is effective.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes main principle of the automatic test data generation for program paths including character string predicates. Section 4 gives an example study to indicate that the approach is practical. Finally, conclusion is provided in Section 5.

2. An overview of related work

This section will briefly review related work with our test data generation.

2.1 Predicate-based testing

Predicate testing is a common approach to software testing, which requests each predicate in the program under test to be checked. There are a lot of predicate testing strategies [11, 12]. However, they demand that

predicates in tested programs must be numerical predicates. A *numerical predicate* is either simple or compound. A *simple predicate* is a Boolean variable or a relational expression while a *relational expression* is of the following form:

$$E_1 < op > E_2$$

where E_1 and E_2 are arithmetic expressions, and $< op >$ is one of six relational operators $\{<, \leq, >, \geq, =, \neq\}$. A *compound predicate* is a Boolean combination (AND, OR) of two or more simple predicates. In other words, these predicate testing strategies only allow Boolean variables, relational expressions or Boolean operators to appear in a predicate, but non-arithmetic expressions, such as character strings, are not taken into consideration. This overly reduces predicate testing approaches applications in practice.

2.2 Test data generation

As mentioned previously, there are many test data generation approaches. The most often used are random test data generation, symbolic execution based test data generation and dynamic test data generation.

2.2.1 Random test data generation. Random test data generation develops test data at random until a useful input is found [3]. It is easy to implement and commonly involved in literatures, but randomly generated test data have difficulties in satisfying a specific requirement, such as domain testing for a predicate border associated with a chosen path. This test requests that *ON* test point lie on the border, *OFF* test point be placed outside the border, and the *ON-OFF* point pair be very close to each other [13]. In such a case, the number of adequate test data may be very small compared to the total number of inputs, and the probability of selecting an adequate test data randomly can be very low. In fact, random test data generation performs poorly, and is generally considered to be ineffective on realistic programs [5].

2.2.2 Symbolic execution based test data generation. The basic idea in a symbolic execution system is to allow numeric variables to take on symbolic values instead of numeric values. Many existing test data generation systems employ symbolic execution technique, in which symbols are assigned to input variables and subsequent uses of the variables are then expressed in terms of these symbols [4, 5]. However, symbolic execution is very computational intensive and a number of technical problems such as indefinite loops, subprogram calls, array references and so on, are met in practice when symbolic execution is performed [9, 14]. Moreover, if input variables are character string variables, symbolic expression becomes more difficult to apply. For example, let us examine the code fragment in a program shown as below:

```
strncpy(v,x,5);
/* copy initial 5 characters of x to v  */
strupr(v);
/* convert each lowercase character to uppercase */
if (strcmp(v, "LEFT")<0) …;
/* compare v and "LEFT" lexicographically */
```

where *v* is a character string variable, and *x* is an input variable of character string. Then, it is difficult to express the value of variable *v* in terms of the symbolic value of the input variable *x* in the predicate that follows.

2.2.3 Dynamic test data generation. Dynamic test data generation is a popular approach for developing test data. In this paper, we employ dynamic test data generation to derive test data. During dynamic test data generation, if some desired test requirement is not reached, data generated in each test execution is used to identify how close the test input is in meeting the requirement. With the aid of feedback, test inputs are gradually modified until one of them satisfies the requirement. For example, suppose that a program contains the condition statement

$$\text{If } (y \leq 38) \ldots.$$

and the *TRUE* branch of the predicate should be taken. Thus, we must find an input that can make the variable *y* to hold a value smaller than or equal to constant 38 when the condition statement is reached. A simple way to calculate the current value of variable *y* in the predicate is to execute the program up to the condition statement and record the value of *y*.

Each predicate can be transformed to an equivalent form:

$$\Re \quad rel \quad 0$$

where $\Re$ is a real-value function，referred as a *branch function*, and *rel* is one of $\{\leq, <, =\}$, which satisfies

1）positive (or zero if *rel* is <) when the predicate is false,

2）negative (or zero if *rel* is = or ≤) when the predicate is true [10].

Let $y_{condition}(x)$ represent the current value of variable *y* for input *x* when the program is executed up to the condition statement. Then the branch function can be expressed as follows:

$$\Re(x) = y_{condition}(x) - 38$$

The function is minimal when the *TRUE* branch is taken on the condition statement. So, the problem of test

data generation can be reduced to the problem of function minimization. That is, we need to find an input x that can minimize the branch function $\Re(x)$ [9, 15].

The techniques usually used to perform function minimization are gradient descent [8, 10, 14], genetic search [9, 16], and simulated annealing [17]. Some systems developed by applying these techniques can generate test data for common programs; nevertheless, they do not carry out the test data of character strings. Thus, existing systems are restricted to programs whose predicates are numerical predicates.

Gradient descent is thought of a standard function minimization technique, which performs function minimization by only evaluating the branch function values. In general, gradient descent is faster than global optimization algorithms such as genetic search [9], and often used in dynamic test data generation, e.g., ADTEST and TESTGEN system [10, 14]. We also employ gradient descent to perform function minimization during our test data generation. A shortcoming of using gradient descent techniques is that gradient descent algorithms are likely to fail when they meet a local minimum. That is, branch function appears to reach the minimum but it does not. However, our gradient descent algorithm is not subject to the problem (see Section 3.2)

Now we review how the function minimization using gradient descent works. Suppose x_0 is an original input on which the program is executed up to a predicate and the *FALSE* branch of the predicate is taken. A branch function can be constructed for the predicate whose value is positive on input x_0. A new input x' is created by a small amount increment or decrement with respect to x_0 on an input variable while keeping all other input variables constant in order to search a good adjustment direction. The program is executed on input x' and the branch function is evaluated. If both increase and decrease on the input variable do not cause the improvement (decrement) of the branch function, another input variable is selected.

When an appropriate direction is found, i.e., the program execution also reaches the predicate and the branch function is improved, a larger amount adjustment is taken in this direction. Then, the program is executed on the new input, and the branch function is evaluated again. If the input no longer reaches the predicate, or a constraint violation occurs, then an adjustment continues in this direction with a smaller amount. If the branch function is not further decreased, the last value of the branch function is retained, and a new direction is searched on the previous input. If the positive minimum of the branch function is located, an adjustment direction is searched from this minimum for another input variable. The cycle repeats either until the branch function becomes negative, meaning the input x that minimizes the branch function is found or until improvement can not be made for any input variable, meaning there is no input that can make the *TRUE* branch of the predicate to be taken.

3. Test data generation based on character string predicate

Our goal of the test data generation is to find a program input on which a chosen program path will be traversed. This problem can be reduced to a sequence of subgoals where each subgoal is solved by performing function minimization using gradient descent. The test data generation approach is considered as a path-oriented testing method, whose major concern is to determine a program execution path that is to be followed. A number of path selection strategies have been reported in the literature [18, 19]. In this paper we focus on how to automatically generate test data for program paths that include character string predicates, leaving out the account for the test paths selection.

3.1 Character string predicate

A *character string predicate* consists of at least one character string variable and one string comparison function such as *strcmp*( ) . Similar to numerical predicates, character string predicates can be simple or compound. A *simple character string predicate* is of the following form

$$strcmp\,(str_1, str_2) \quad op \quad 0$$

where $op \in \{<, =, >\}$. A *compound character string predicate* is a character string predicate that contains at least one Boolean operator such as '*NOT*', '*AND*' or '*OR*'.

Dynamic test data generation can be reduced to the problem of function minimization. As a result, we need to construct a branch function with respect to a character string predicate, and then evaluate the branch function value.

Consequently, we can construct a branch function $\Re$ regarding a character string predicate, e.g., $strcmp(str_1, str_2) > 0$, so that its value is positive for initial input x_0. Namely, let $\Re = str_1 - str_2$ if str_1-str_2 is positive for input x_0; otherwise $\Re = str_2 - str_1$. Then, the program input is adjusted gradually until $\Re$ becomes negative, i.e., the required input has been found. A problem that we must solve first is how to compare two character strings as well as how to evaluate the branch function $\Re$. So we define a function ξ , which maps a character string to a nonnegative integer, satisfying the formula:

$$\xi(str) = \sum_{i=0}^{L-1} str[i] \times w^{L-i-1} \qquad [1]$$

where *str* is a character string, *L* is its length, w^{L-i-1} is a positive weighting factor representing a weighted value

imposed upon each character element of the string, and w is equal to 128.

We propose a theorem to map a character string to a unique nonnegative integer.

Theorem 1: *Suppose S is a set of character strings, N_+ is a set of nonnegative integers. Let $\xi(str)$ is defined in Eq.(1). Then $\xi(str)$ is a one-to-one function from S to N_+.*

By the theorem, a character string can be transformed into a unique nonnegative integer. Therefore, the distance between two strings can be defined as below:

Definition: Let L_1 and L_2 denote the length of string str_1 and str_2, respectively. Suppose $L = \max(L_1, L_2)$, where $\max(L_1, L_2)$ is the maximum of L_1 and L_2. Without loss of generality, let $L = L_2$, $str_1[k] = '\backslash 0', (k < L-1)$. By *the distance between string str_1 and str_2*, represented by $dis(str_1, str_2)$, we mean

$$dis(str_1, str_2) = \left| \xi(str_1) - \xi(str_2) \right|$$
$$= \left| \sum_{i=0}^{L_1-1} str_1[i] \times w^{L-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L-i-1} \right|.$$

By *the distance between the i^{th} characters of string str_1 and str_2*, denoted by $d_i(str_1, str_2)$, we imply

$$d_i(str_1, str_2) = \left| str_1[i] - str_2[i] \right| \quad (i < k)$$

The distance $dis(str_1, str_2)$ uniquely determines a nonnegative integer, which can be used to evaluate the branch function $\Re$ with respect to a character string predicate. Then, the next step is to search an appropriate direction for a character string variable to improve the branch function value. It is easy to see that $\left| str_1[0] - str_2[0] \right| \times w^{L-1} > \sum_{i=1}^{L-1} (\max(str_1[i], str_2[i])) \times w^{L-i-1}$ by the verification of theorem 1 (see Appendix). Since every character element of a string is expressed by its ASCII code (an integer), a practical way is to construct a branch function $\Re_0$ for the 0^{th} character of str_1 and str_2, i.e., let $\Re_0 = \left| str_1[0] - str_2[0] \right|$ so that $\Re_0 \geq 0$. Then, we search an appropriate adjustment direction for the 0^{th} character of an input variable, and adjust the character by gradient descent until $\Re_0 < 0$. As a result, we can find an input that makes the character string predicate to take *TRUE* branch.

3.2 Test data generation based on character string predicate

Now we explain in detail how to use gradient descent to perform function minimization in order to generate test data for a path including character string predicates. The current values of variables in the predicates can be calculated or collected by the program instrumentation technique.

Let π be a path in the program under test, x be an adjusted input variable, pr denote a character string predicate (e.g., $strcmp(str_1, str_2) > 0$) on π, and the *TRUE* branch of the predicate should be taken. Suppose that x_0 is an initial input (selected randomly or by hand) on which the program is executed to the predicate pr along path π. However, the *FALSE* branch of pr is taken instead of the *TRUE* branch.

In order to traverse the *TRUE* branch of pr, a branch function with respect to the predicate pr should be constructed. If pr is an inequality predicate, that is, its operator is one of $\{\leq, <, \geq, >\}$, we construct a branch function $\Re_0$ such that $\Re_0 = \left| str_1[0] - str_2[0] \right| \geq 0$. Then, we search an adjustment direction that can improve the branch function $\Re_0$ by modifying the 0^{th} character, denoted by c_0, of input variable x, i.e., let $c_0' = c_0 + 1$ or $c_0' = c_0 - 1$. If c_0' results in a better $\Re_0$ value than c_0, c_0' replaces c_0; otherwise, if there is another input variable, that input variable is selected. If there is no other input variable, the test data generation fails for the path π. For instance, suppose that there is only one input variable x in the program under test, and up to the predicate pr, the program implements the function: str_1 ="2334", str_2 ="abc"+x. In this case, no matter how to adjust the input variable x, it is impossible to make the predicate pr true.

When a good direction is found, the adjustment amount is increased (double) until (1) $\Re_0 < 0$, (2) $\Re_0$ is not improved, (3) constraint violation occurs, or (4) c_0' is outside of 32 and 127. In the last three cases, we reduce (halve) the amount of adjustment and the corresponding input is tried again. In the first case, we find an input that makes the predicate pr to take *TRUE* branch.

If pr is an equality (=) or non-equality ($\neq$) predicates, for instance,

if $(strcmp(str_1, "-ceiling"))$ $\cdots$,

we must adjust every character of an input variable to make str_1 ="$-ceiling$". In this case, we need to construct branch function $\Re_i$ for every unequal character of str_1 and str_2 such that $\Re_i > 0$, $i \in [0, L-1]$, $L = \max(L_1, L_2)$. Then, we compare the corresponding characters of str_1 and str_2 from positions 0 to *L-1*. For example, at position i, if the i^{th} character of str_1 and str_2 is unequal to each other, a branch function $\Re_i = \left| str_1[i] - str_2[i] \right|$ is constructed so that $\Re_i > 0$. Similarly for $\Re_0$, we search an adjustment

direction to improve the branch function $\Re_i$ until $\Re_i \le 0$ or until one of other three cases hold. In the case of $\Re_i \le 0$, we obtain two distinct characters C_{on} and C_{off} such that C_{on} satisfies $\Re_i \le 0$ whereas C_{off} meets $\Re_i > 0$. The two distinct characters are refined gradually so that the distance between them, $d_i(str_1, str_2)$, is the shortest. If d_i is adjusted to 0, the i^{th} character of input variable x is determined, and the next character, $(i+1)^{th}$ character, is considered. Otherwise, the test data generation fails to make the predicate to be *TRUE*, and the cycle terminates. If input variable x ends before $i < L-1$, a space character is added before its terminating position. The comparison continues until $i = L-1$.

Suppose that x_1 makes the predicate *pr* to take *TRUE* branch. Now, either the path π is traversed, or the second subgoal must be solved. In the former case, x_1 is the test input of path π. In the latter case, we must find a program input to satisfy another predicate. The process is repeated until a program input is found on which the path π is traversed, or the subgoal cannot be solved, i.e., the test generation fails to the path.

The algorithm of test data generation for an equality or non-equality character string predicate is shown as follows:

```
For the i^th character of adjusted input variable
   Initialize ℜ_i
   Search a good adjustment direction dir
   If (dir is not found) test generation fails, exit
   Else initialize AMOUNT⇐1
        AMOUNT ⇐ AMOUNT × 2
        // Enlarge adjustment amount. //
        Repeat
            c_i = c_i dir AMOUNT
            executing the program under test
            Evaluate ℜ_i
            If (ℜ_i ≤ 0)
                Obtain distinct characters at position i
                Repeat refining the distinct characters
                Until d_i is shortest
                If (d_i=0)
                   i^th character is determined
                Else test generation fails
                Exit
                Endif
            Else If (ℜ_i decrease)
                      AMOUNT ⇐ AMOUNT × 2
                  Else AMOUNT ⇐ AMOUNT / 2
                  // Lessen adjustment amount. //
                  Endif
            Endif
        Until (AMOUNT <initial value)
   Endif
Endfor
```

A shortcoming of using gradient descent to perform function minimization is that gradient descent algorithms can fail if a local minimum is encountered. Fortunately, our gradient descent algorithm would not encounter the problem. We note that minimizing a branch function is very difficult if str_1 and str_2 are all involved in an adjusted input variable. In most cases, one of them is not associated with the adjusted input variable. Without loss of generality, we assume that str_2 is not related to the adjusted input variable, and c_i represents the i^{th} character of the adjusted input variable. Then, at position i, we have $\Re_i = \left| str_1[i] - str_2[i] \right|$. In fact, $str_1[i]$ is a function of c_i, denoted as $\vartheta(c_i)$. $str_2[i]$ has nothing to do with c_i, so it can be thought of as a constant, represented by M. Accordingly, the branch function $\Re_i$ can be expressed as $\Re_i = \left| \vartheta(c_i) - \mathrm{M} \right|$. The relationship of $\Re_i$ and $\vartheta(c_i)$ is shown in Fig.1. It can be seen that the branch function $\Re_i$ is a monotonic increasing or decreasing function, i.e., the adjustment for each character is not restricted to a localized region of $\Re_i$. The branch function $\Re_i$ can reach its minimum so that each character of the adjusted input variable is determined in turn. Therefore, the function minimization by using gradient descent does not suffer from the local minimum problem.

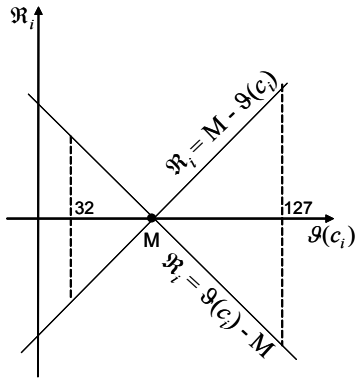

Fig.1 Branch function $\Re_i$

4. Experimental results

We describe how to automatically generate test data for a path including character string predicate by a ***Max*** program, shown in Fig.2**,** which is a variation of the program taken from [20].

Max program has two input variables. One is an integer variable *argc*, and the other is character string variable *argv*. Although the program is made up of only dozens of statements, it contains lots of structures, such as numerical predicates, character string predicates, compound predicates, *IF-THEN-ELSE* statements and *FOR* loop statements, etc. The current values of variables in the predicates can be calculated or collected by the

program instrumentation technique. An instrumented version of ***Max*** program is shown in Fig.3, in which the instrumentation statements are shown in italics.

```
 int  max(int argc, char ** argv)
 {
1    argc--;
2    argv++;
3    if ((argc>0)&&('-'==**argv))
4    {   if (!strcmp(argv[0],"-ceiling"))
5       {    strncpy(ceiling,argv[1],BUFSIZE);
6            argv++;  argv++;
7            argc--;   argc--;
        }
        else
8       {    fprintf("Illegal option %s.\n",argv[0]);
9            return(2);
        }
     }
10   if(argc==0)
11   {    fprintf("At least one arguments.\n");
12       return(2);
     }
13   for(;argc>0;argc--,argv++)
14   {   if(strcmp(argv[0],result)>0);
15           strncpy(result,argv[0],BUFSIZE);
     }
16   if (strcmp(ceiling,result)<=0)
17       printf("\n max:%s",ceiling);
     else
18       printf("\n max:%s",result);
19   return(0);
 }
```

Fig.2 *Max* program

Considering that the *FOR* loop is executed zero time, one time and two times, there are 31 paths in ***Max*** program. We design 50 program inputs at random, which are used as original input to the test data generation for these 31 paths. If the test data generation fails for a path on these 50 inputs, the path is taken as an infeasible path. As a result, 16 test inputs are generated by the test data generation approach. That is to say, 16 out of 31 paths are feasible paths.

Code coverage has been considered to be an important metric for software testing and software reliability measurement, and many software product companies require 85% coverage to achieve [21,22]. We measure the coverage of generated test data using ATAC (***A****utomatic* ***T****est* ***A****nalysis for* ***C***) tool [23]. The results are show in Fig.4. It is seen that 15 out of 17 P-Uses (88%) are coved, and all blocks, decisions and C-Uses have been covered. In addition, test data can be developed by using the gradual descent approach and the random-number approach to perform function minimization. In the gradual descent approach, each character is adjusted, one by one, from 32 to 127, while in the random-number approach; each character is generated by a random-number generator.

```
record (argc,0,'>',"&&");
record('-',**argv, '=');
if ((argc>0)&&('-'==**argv))
{  record(argv[0],"-ceiling", '!');
   if (!strcmp(argv[0],"-ceiling"))
   …;
}
record(argc,0,'=',"");
if(argc==0)
…;
record(argc,0,'>',"");
for(;argc>0;argc--,argv++)
{  record(argv[0],result, '>', "");
   if (strcmp(argv[0],result)>0)
     …;
   record(argc,0,'>',"");
}
record(ceiling,result, '-', "");
if (strcmp(ceiling,result)<=0)
     …;
```

Fig.3 Instrumented *Max* program

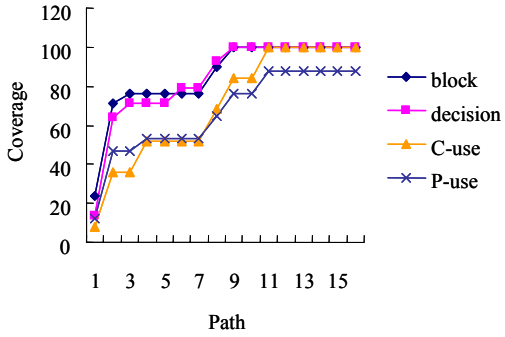

Fig.4 Coverage of the test data

Dynamic test data generation is a heuristic process. When a new input is created, the program has to be executed again in order to evaluate its branch function. Thus, the cost of dynamic test data generation depends mainly on the cost of executing the program. In fact, the number of program executions is the evaluation number of branch function in the test data generation for a path. We compare the evaluation number of branch function in the gradient descent, the gradual descent and the random-number test data generation approaches under the same coverage. The results are shown in Fig.5. Obviously, the gradient descent test data generation approach is more economical than the gradual descent and the random approaches. However, the gradual descent approach is not better than the random-number approach.

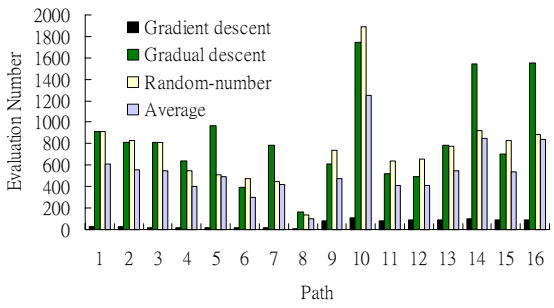

Fig.5 The comparison of evaluation number

5. Conclusion

The objective of software testing is to detect faults in programs. Nevertheless, all current software testing strategies have been limited to programs in which character string predicates are not taken into consideration. In this paper, we present a test data generation approach for program paths including character string predicates.

To our knowledge, this is the first automatic test data generation approach based on character string predicates. The preliminary experimental results show that the methodology is effective.

Appendix

This appendix shows the verification of Theorem 1.

Proof: Suppose $str_1, str_2 \in S \wedge str_1 \neq str_2$. Let L_1 and L_2 denote the length of string str_1 and str_2, respectively. Then, by the definition of $\xi(str)$, we have

$$\xi(str_1) = \sum_{i=0}^{L_1-1} str_1[i] \times w^{L_1-i-1} \in N_+$$

$$\xi(str_2) = \sum_{i=0}^{L_2-1} str_2[i] \times w^{L_2-i-1} \in N_+$$

where w=128.

Let $\chi = \xi(str_1) - \xi(str_2)$. Now, we prove $\xi(str_1) \neq \xi(str_2)$, i.e., $|\chi| > 0$. Two cases need to be discussed:

(1) $L_1 = L_2 = L$

Without loss of generality, we assume $str_1[j] = str_2[j]$, $j = 0, 1, \cdots, l$, $(l < L-1)$ and the $(l+1)^{th}$ character of string str_1 and str_2 are not equal to each other, i.e., $str_1[l+1] \neq str_2[l+1]$. If $(l+1) = L-1$, it is easy to see that $|\chi| = |\xi(str_1) - \xi(str_2)| = |str_1[l+1] - str_2[l+1]| > 0$; otherwise $l+1 < L-1$.

Here $|\chi| = \left| \sum_{i=l+1}^{L-1} (str_1[i] - str_2[i]) \times w^{L-i-1} \right|$, and the weighting factor of $(l+1)^{th}$ character is $w^{L-(l+1)-1}$. If we can prove $|str_1[l+1] - str_2[l+1]| \times w^{L-(l+1)-1} > \sum_{i=l+2}^{L-1} (\max(str_1[i], str_2[i])) \times w^{L-i-1}$, we can derive $|\chi| > 0$.

The ASCII characters that are commonly used in programming are between 32 and 127. We consider the worst case, i.e., 127 is regarded as the ASCII code of every character and $|str_1[l+1] - str_2[l+1]| = 1$. Then the problem can be converted to verify $w^{\eta-1} > \sum_{k=0}^{\eta-2} 127 \times w^k$, where $\eta = L - l - 1$. In fact, $w^{\eta-1} = \sum_{k=0}^{\eta-2} 127 \times w^k + 1$. We confirm the proposition by induction.

i). In the case of $\eta = 2$, $w^{\eta-1} = 128$, and $\sum_{k=0}^{\eta-2} 127 \times w^k = 127$. So the proposition holds.

ii). Suppose for $\eta = n$, $w^{n-1} = \sum_{k=0}^{n-2} 127 \times w^k + 1$ is true. Following shows for $\eta = n+1$, $w^{(n+1)-1} = \sum_{k=0}^{(n+1)-2} 127 \times w^k + 1$ is also true.

By the hypothesis, we have

$$w^{n-1} \times w = \left(\sum_{k=0}^{n-2} 127 \times w^k + 1\right) \times w,$$

then, $w^{(n+1)-1} = \sum_{k=0}^{n-2} 127 \times w^{k+1} + w$

$$= \sum_{k=1}^{n-1} 127 \times w^k + 127 \times w^0 + 1$$

$$= \sum_{k=0}^{(n+1)-2} 127 \times w^k + 1$$

Thus, $w^{\eta-1} > \sum_{k=0}^{\eta-2} 127 \times w^k$ holds.

Therefore, in the case of $L_1 = L_2$, if $str_1 \neq str_2$, we derive $|\chi| > 0$.

(2) $L_1 \neq L_2$

In this case, we assume $L_1 > L_2$. Then, $|\chi| = |\xi(str_1) - \xi(str_2)| = \left| \sum_{i=0}^{L_1-1} str_1[i] \times w^{L_1-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L_2-i-1} \right|$

$$= \left| str_1[0] \times w^{L_1-1} + \sum_{i=1}^{L_1-1} str_1[i] \times w^{L_1-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L_2-i-1} \right|.$$

By the above proposition, we have $str_1[0] \times w^{L_1-1} > \sum_{i=1}^{L_1-1} 127 \times w^{L_1-i-1} >$ $\left| \sum_{i=1}^{L_1-1} str_1[i] \times w^{L_1-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L_2-i-1} \right|$. So $|\chi| > 0$. holds.

This completes the proof of the theorem.

Acknowledgement

The work described in this paper was supported by the Hong Kong Research Grants Council, under Project No. CUHK 4360/02E and Young Science Foundation of BUCT, China, under Project No. QN0312.

Reference

[1] B. Beizer. "Software Testing Techniques," *International Thomson Publishing Inc.*, 2nd edition, 1990.

[2] P. C. Jorgensen. "Software Testing: A Craftsman's Approach". CRC Press LLC. 2002.

[3] J. Duran and S. Ntafos. "An Evaluation of Random Testing." *IEEE Transactions on Software Engineering,* 10 (4), 1984, pp. 438-444.

[4] M. R. Girgis. "An Experimental Evaluation of a Symbolic Execution System," *Software Engineering Journal*, July 1992, pp. 285-290.

[5] P.D. Coward. "Symbolic Execution and Testing." *Information and Software Technique,* Vol. 33, No. 1, 9991, pp. 229-239.

[6] W. H. Deason, D.B. Brown, K.H. Chang and J.H. Cross, II. "A Rule-based Software Test Data Generator," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 3 No.1, March 1991, pp. 108 –117.

[7] A. Gotlieb, B. Botella, and M. Rueher. "Automatic Test Data Generation using Constraint Solving Techniques." *International Symposium on Software Testing and Analysis,* 1998, pp. 53-62.

[8] A. Hajnal and I. Forgacs. "An Applicable Test Data Generation Algorithm for Domain Errors," *ISSTA'98, Proceedings of ACM SIGSOFT International symposium on Software Testing and Analysis, Florida*, USA, March 2-5, 1998, pp. 63-72.

[9] C. C. Michael, G. McGraw, and M. A. Schatz. "Generating Software Test Data by Evolution," *IEEE Transactions on Software Engineering*, Vol.27, No.12, Dec. 2001, pp. 1085-1110.

[10] B. Korel. "Automated Software Test Data Generation", *IEEE Transactions on Software Engineering,* Vol.16, No.8, Auguest.1990, pp. 870-879.

[11] A. Paradkar, K. C. Tai, and M. A. Vouk. "Automatic Test-Generation for Predicates," *IEEE Transactions on Reliability*, VOL, 45, NO 4, 1996 December, pp. 515-530.

[12] K. C. Tai. "Theory of Fault-based Predicate Testing for Computer Programs." *IEEE Transactions on Software Engineering*, Vol. 22, August 1996, pp. 552-562.

[13] B. Jeng and E. J. Weyuker, "A Simplified Domain-Testing Strategy." *ACM Trans. Software Engineering and Methodology,* Vol.3, No.3, July 1994, pp. 254-270.

[14] M. J. Gallagher and V. L. Narasimhan. "Adtest: A Test Data Generation Suite for Ada Software System." *IEEE Transactions on Software Engineering,* Vol. 23, No. 8, Aug. 1997, pp.473-484.

[15] R. Ferguson, and B. Korel. "The Chaining Approach for Software Test Data Generation," *ACM Transactions on Software Engineering and Methodology*, 1996, Vol.5, No.1, pp. 63-86.

[16] B. F. Jones, H. H. Sthamer and D. E. Eyres. "Automatic Structural Testing using Genetic Algorithms." *Software Engineering Journal,* Sept. 1996, pp. 299-306.

[17] N. Tracey, J. Clark, and K. Mander. "Automated Program Flaw Finding Using Simulated Annealing," *Proceedings of International Symposium on Software Testing and Analysis, Software Engineering, Notes*, March 1998, pp.73-81.

[18] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, Vol.15, No. 11, Nov. 1989, pp. 1318–1332.

[19] L. M. Peres, S. R. Vergilio, M. Jino and J .C. Maldonado, "Path Selection in the Structural Testing: Proposition, Plementation and Application of Strategies, *Proceedings of International Conference of the Chilean, Computer Science Society*, 2001. SCCC '01, Chilean, pp. 240 –246

[20] B. Marick. "The Craft of Software Testing," *PTR Prentice Hall*, NJ, 1995.

[21] T. W. Williams, M. R. Mercer, J. P. Mucha and R. Kapur. "Code Coverage, What Does it Mean in Terms of Quality?" *Proceedings of Annual on Reliability and Maintainability Symposium,* 2001, pp. 420-424.

[22] M. Chen, M. R. Lyu, and E. Wong, "Effect of Code Coverage on Software Reliability Measurement", *IEEE Transactions on Reliability*, Vol. 50, No. 2, June 2001, pp.165-170.

[23] M. R. Lyu, J. R. Horgan, and S. London, "A Coverage Analysis Tool for the Effectiveness of Software Testing", *IEEE Transactions on Reliability*, Vol. 43, No. 4, December 1994, pp. 527-535.