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 \(<, \leq, \geq, =, \neq, \approx\) . 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:

```c
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

\[ y \leq 38 \]

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:

\[ \forall rel \ 0 \]

where \( \forall \) 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 \( y_{\text{condition}}(x) \) 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:

\[ \forall(x) = y_{\text{condition}}(x) - 38 \]

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 minimization by only evaluating the branch function 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

\[
\text{strcmp}(str_1, str_2) \ x \ 0
\]

where \( x \in \{<,=,>\} \). 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., \( \text{strcmp}(str_1, str_2) > 0 \), so that its value is positive for initial input \( x_0 \). Namely, let \( \Re = str_1 - str_2 \) if \( str_1 \neq str_2 \) 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 \( \xi \), which maps a character string to a nonnegative integer, satisfying the formula:

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

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, \( N_s \) 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 \( N_s \).

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 \( L = L_2 \), \( str_2[k] = 0 \), \( k < L - 1 \). By the distance between string \( str_1 \) and \( str_2 \), represented by \( \text{dist}(str_1, str_2) \), we mean

\[
\text{dist}(str_1, str_2) = \left| \sum_{i=0}^{L-1} \text{str}[i] \times w^{L-i-1} - \sum_{i=0}^{L-1} \text{str}_2[i] \times w^{L-i-1} \right|
\]

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

\[
d_i(str_1, str_2) = \left| \text{str}_1[i] - \text{str}_2[i] \right| \quad (i < k)
\]

The distance \( \text{dist}(str_1, str_2) \) uniquely determines a nonnegative integer, which can be used to evaluate the branch function \( \mathcal{R} \) 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

\[
\sum_{i=1}^{L-1} \left( \text{max}(\text{str}[i], \text{str}_2[i]) \right) \times w^{L-i-1}
\]

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 \( \mathcal{R} \) for the \( 0 \)th character of \( str_1 \) and \( str_2 \), i.e., let \( \mathcal{R}_0 = \left| \text{str}[0] - \text{str}_2[0] \right| \) so that \( \mathcal{R}_0 \geq 0 \). Then, we search an appropriate adjustment direction for the \( 0 \)th character of an input variable, and adjust the character by gradient descent until \( \mathcal{R}_0 < 0 \). As a result, we can find an input that makes the character string predicate to take \( \text{TRUE} \) 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 \( \pi \) be a path in the program under test, \( x \) be an adjusted input variable, \( pr \) denote a character string predicate (e.g., \( \text{strcmp}(str_1, str_2) > 0 \)) on \( \pi \), and the \( \text{TRUE} \) 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 \( \pi \). However, the \( \text{FALSE} \) branch of \( pr \) is taken instead of the \( \text{TRUE} \) branch.

In order to traverse the \( \text{TRUE} \) 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 \( \mathcal{R}_0 \) such that \( \mathcal{R}_0 = \left| \text{str}[0] - \text{str}_2[0] \right| \geq 0 \). Then, we search an adjustment direction that can improve the branch function \( \mathcal{R}_0 \) by modifying the \( \theta \)th character, denoted by \( c_k \), of input variable \( x \), i.e., let \( c_k = c_0 + 1 \) or \( c_k = c_0 - 1 \). If \( c_0 \) results in a better \( \mathcal{R}_0 \) value than \( c_k \), \( c_0 \) replaces \( c_k \); 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 \( \pi \).

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: \( str_1 = \text{"abc"} + x \). 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) \( \mathcal{R}_0 < 0 \), (2) \( \mathcal{R}_0 \) is not improved, (3) constraint violation occurs, or (4) \( c_k \) 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 \( \text{TRUE} \) branch.

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

\[
\text{if} \left( \text{strcmp}(str_1, \text{\"ceiling\"}) \right)
\]

we must adjust every character of an input variable to make \( str_1 = \text{\"ceiling\"} \). In this case, we need to construct branch function \( \mathcal{R}_i \) for every unequal character of \( str_1 \) and \( str_2 \) such that \( \mathcal{R}_i > 0 \), \( i \in [0, L-1] \), \( L = \max(L_1, L_2) \). 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 \( \mathcal{R}_i = \left| \text{str}_1[i] - \text{str}_2[i] \right| \) is constructed so that \( \mathcal{R}_i > 0 \). Similarly for \( \mathcal{R}_0 \), we search an adjustment
direction to improve the branch function \( R_i \) until \( R_i \leq 0 \) or until one of other three cases hold. In the case of \( R_i \leq 0 \), we obtain two distinct characters \( C_{\text{on}} \) and \( C_{\text{off}} \) such that \( C_{\text{on}} \) satisfies \( R_i \leq 0 \) whereas \( C_{\text{off}} \) meets \( R_i > 0 \). The two distinct characters are refined gradually so that the distance between them, \( d_i(\text{str}_1, \text{str}_2) \), 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 \( x_i \) makes the predicate \( pr \) to take TRUE branch. Now, either the path \( \pi \) is traversed, or the second subgoal must be solved. In the former case, \( x_i \) is the test input of path \( \pi \). 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 \( \pi \) 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 \( \text{AMOUNT} \leftarrow 1 \)

\( \text{AMOUNT} \leftarrow \text{AMOUNT} \times 2 \)

// Enlarge adjustment amount. //

Repeat

\( c_i = c_i, \ dir = \text{AMOUNT} \)

executing the program under test

Evaluate \( R_i \)

If (\( R_i \leq 0 \))

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)

\( \text{AMOUNT} \leftarrow \text{AMOUNT} \times 2 \)

Else \( \text{AMOUNT} \leftarrow \text{AMOUNT} / 2 \)

// Lessen adjustment amount. //

Endif

Endif

Until (\( \text{AMOUNT} < \text{initial value} \))

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 \( \text{str}_1 \) and \( \text{str}_2 \) 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 \( \text{str}_2 \) 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 \( R_i = | \text{str}_1[i] - \text{str}_2[i] | \). In fact, \( \text{str}_1[i] \) is a function of \( c_i \), denoted as \( \mathcal{H}(c_i) \). \( \text{str}_2[i] \) has nothing to do with \( c_i \), so it can be thought of as a constant, represented by \( M \). Accordingly, the branch function \( R_i \) can be expressed as \( R_i = | \mathcal{H}(c_i) - M | \). The relationship of \( R_i \) and \( \mathcal{H}(c_i) \) is shown in Fig.1. It can be seen that the branch function \( R_i \) is a monotonic increasing or decreasing function, i.e., the adjustment for each character is not restricted to a localized region of \( R_i \). The branch function \( R_i \) 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 \( R_i \)](image)

4. Experimental results

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

\textit{Max} program has two input variables. One is an integer variable \textit{argc}, and the other is character string variable \textit{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, \textit{IF-THEN-ELSE} statements and \textit{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.

```c
int max(int argc, char ** argv)
{
    argc--;
    argv++;
    if ((argc>0) && (argv[0][0] == 'c'))
    {
        strncpy(ceiling, argv[1], BUFSIZE);
        argv++; argv++;
        argc--; argc--;
    }
    else
    {
        fprintf("Illegal option %s.\n", argv[0]);
        return(2);
    }

    if(argc==0)
    {
        fprintf("At least one arguments.\n");
        return(2);
    }

    for(;argc>0;argc--, argv++)
    {
        if (strcmp(argv[0], result)>0);
        strncpy(result, argv[0], BUFSIZE);
    }

    if (strcmp(ceiling, result)<=0)
    {
        printf("\n max:%s", ceiling);
    } else
    {
        printf("\n max:%s", result);
    }
    return(0);
}
```

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 (Automatic Test Analysis for C) tool [23]. The results are show in Fig.4. It is seen that 15 out of 17 P-Uses (88%) are covered, 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 generated by a random-number generator.

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.
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 based on 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 \( str_1, str_2 \in S \land str_1 \neq str_2 \). 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 \( str_1[j] = str_2[j], j = 0, 1, \ldots, l \) (\( l < L - 1 \)) 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

\[
|\chi| = \xi(str_1) - \xi(str_2) = |str_1[l+1] - str_2[l+1]| > 0 ; \quad \text{otherwise} \quad l + 1 < L - 1.
\]

Here \( |\chi| = | \sum_{i=0}^{L_1-1} (str_1[i] - str_2[i]) \times w^{L_1-i-1} | \), and the weighting factor of \((l+1)\)th character is \( w^{L-(l+1)} \). 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 \( w^{n-1} > \sum_{k=0}^{n-2} 127 \times w^k \), where \( n = L - l - 1 \). In fact, \( w^{n-1} = \sum_{k=0}^{n-2} 127 \times w^k + 1 \). We confirm the proposition by induction.

i). In the case of \( n = 2 \), \( w^{n-1} = 128 \), and

\[
\sum_{k=0}^{n-2} 127 \times w^k = 127 .
\]

So the proposition holds.

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

\[
w^{(\eta+1)-1} = \sum_{k=0}^{(\eta+1)-2} 127 \times w^k + 1.
\]

By the hypothesis, we have

\[
w^{\eta-1} \times w = \sum_{k=0}^{\eta-2} 127 \times w^k + 1 \times w,
\]

then,

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

Thus, \( w^{\eta-1} > \sum_{k=0}^{n-2} 127 \times w^k \) 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, \( |\chi| = \xi(str_1) - \xi(str_2) = \sum_{i=0}^{L_1-1} str_1[i] \times w^{L_1-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L_2-i-1} \)

\[=|str_1[0] \times w^{L_1-1} - \sum_{i=1}^{L_1-1} str_1[i] \times w^{L_1-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L_2-i-1}|.
\]
By the above proposition, we have
\[
str_1[0] \times w^{L-1} > \sum_{i=1}^{L-1} 127 \times w^{L-1-i} > \\
\sum_{i=1}^{L-1} str[i] \times w^{L-1-i} - \sum_{i=0}^{L-1} str[i] \times w^{L-1-i}.
\]
So \( |z| > 0 \) 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