String Test Data Generation for Java Programs

Miaomiao Wang[1,3*], Baoquan Cui[2,3*], Jiwei Yan[1,3], Jun Yan[1,2,3†] and Jian Zhang[2,3†]
[1]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3]University of Chinese Academy of Sciences, Beijing, China
Email: wangmiaomiao20@otcaix.iscas.ac.cn, {cuibq, yanjw, yanjun, zj}@ios.ac.cn

2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-5132-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISSRE55969.2022.00033

***Abstract*—Appropriate string test data generation is important for program testing. Complex string APIs combinations are commonly used to handle string parameters. However, the complex combinations make it difficult to express comprehensive string related constraints and generate suitable string data to trigger bugs and cover more branches.**

In this paper, we propose a novel approach to characterize the input strings and their operations (API invocations) with the regular expressions for string test data generation, with insight that they support rich syntax and can express the semantics of various string APIs combinations. We build a set of mapping rules that map 48 string APIs in Java to regular expressions, and design an inference algorithm to generate regular expressions for the complex string APIs combinations. With these regular expressions, more effective string data can be generated in an efficient way. Experiments on multi-type programs from assignments, `LeetCode` platform and open source community show that our approach can increase the branch coverage (17%) and find more bugs (+81) than the existing work. For the basic library JDK, 17 defects have been found, of which 14 are confirmed by the JDK developers and 3 are fixed in new version.

***Index Terms*—String, Test Data Generation, Regular Expression, Test Case Generation**

I. Introduction

Java is one of the most popular programming languages [13]. Strings are widely used in Java programs. A large number of string APIs are provided by JDK to support various operations, and used in many Java programs, such as web applications, mobile device applications, etc. Testing for these programs requires more appropriate string input data.

It is not easy to generate suitable test data for strings automatically, especially for the complex string APIs or even their combinations. For example, the snippet `str.split("#").subString(3,8).lastIndexOf(".")` has some potential exceptions like out of bounds, which is difficult to deal with. One way is to convert constraints into SMT formats. In fact, each string API represents a certain characteristic of its caller. The SMT solvers try to convert the constraints that the caller of the API should satisfy into SMT formats, which is difficult in practice. Although SMT-LIB [3] provides 12 core functions and 5 additional functions to express commonly used string operations, it cannot convert the *split(...)* method and the *lastIndexOf(...)* method into SMT formats by the current syntax. And the automatic conversion from the string API combinations to SMT formats is not supported either. Besides, in the area of test generation, both random testing and search-based testing cannot generate suitable test data for Java programs with complex operations of string APIs like the above snippet. `Randoop` [40] randomly generates regression test cases, and it does not intentionally generate string-related test data. Search-based `EvoSuite` [24] implements seed pools where the seeds are mutated to serve as the data in test cases. One of the seed strategies is dynamic seeding, which uses any values observed during execution as the seeds [44]. It is effective for the single or simple string APIs, but struggling for the complex ones or even their combinations, since the seeds are simple, and the mutation rules are fixed and limited.

String APIs provide support for various operations on strings and each API represents a specific pattern of a string. In other words, if a string tries to invoke a string API and satisfy its semantics, it needs to match some patterns. Our key insight is that regular expressions can characterize a string with rich semantics. They have powerful semantic expression abilities and can express the string APIs that are not supported by the current SMT-LIB. Moreover, The regular expressions can flexibly describe the semantics of various string APIs combination operations. If there is a regular expression to characterize the string parameter which invokes string APIs, it is easy to generate string data for the program testing.

Challenges. To characterize the string parameter with regular expressions, there are two challenges. The first challenge is that for the string APIs with complex semantics like *split(...)* and *lastIndexOf(...)*, it is not easy to map them to the regular expressions directly. Another challenge is that there are a large number of different string API combinations, and generating a regular expression for each API combinations is difficult.

Our Approach. In light of the above, we propose a novel approach to characterize the string parameter with regular expressions. To deal with the first challenge, we construct a mapping from string APIs to regular expressions called API-Regex mapping, which characterizes the string APIs equivalently or approximately based on the JDK specification and the regular expression syntax. Then for the combination of string APIs, we propose an iterative inference algorithm to merge the regular expressions one by one based on the mapping to fit the usage of the API combinations. The inference algorithm is based on slicing analysis which focuses on the path containing at least one string API invocation on the string parameter. Finally, the inferred regular expressions can be used for the string test data generation and test case generation to improve

*These authors contributed equally.
†Corresponding Authors.

2332-6549/22/$31.00 ©2022 IEEE
DOI 10.1109/ISSRE55969.2022.00033

the branch coverage and find string related bugs.
In summary, the contributions in this paper are as follows:

- **API-Regex Mapping.** We build a mapping from 48 string APIs to regular expressions based on their semantics, which are equivalent or approximate.
- **Inference Algorithm.** We design an inference algorithm that can characterize the string parameter with regular expressions under the combination of its string API invocations. It is based on the API-Regex mapping and the slicing analysis.
- **String Test Data Generation Tool.** We have developed an automatic tool `JustinStr` to implement our approach, which outputs the string test data for test cases generation. In a benchmark consisting of various Java programs, `JustinStr` has improved the branch coverage (+17%) and found more bugs (+81) than `EvoSuite`. Among them, 17 bugs are from the JDK, of which 3 are known bugs and 14 bugs have been confirmed by JDK developers.

II. Preliminary

In this section, we will introduce string APIs in Java and the regular expressions. We have conducted a statistical analysis to study the usage frequency of string APIs and their combinations in programs. A motivating example shows the typical problems caused by string operations in the end.

A. String APIs and Regular Expressions

JDK provides many APIs about string operations in the `java.lang.String` class. In terms of semantics, some of them are universal operations, which may be supported in other program languages. Some of them are unique to the JDK, such as the *regionMatches(...)* method which tests whether the specific regions of two strings are equal. They are widely used in Java programs and will be discussed further in section II-B.

Previous work [15], [20] shows that about 40% of Java, JavaScript, Python projects use regular expressions. A regular expression (regex) is a special string consisting of characters or symbols, and defines a search pattern for strings. The basic syntax of the regular expressions is shown on the website [2]. For example, the Kleene star (*) in it is a special mark which is used to indicate infinite numbers of iterations. If the Kleene star is replaced by a specific value, the iteration is finite. The combination of these syntax can effectively constrain strings, which means it is possible to build a mapping set from string APIs to regular expressions.

A regular expression can be used for string data generation. We have investigated 8 tools to figure out whether they are easy to use for generating strings corresponding to regular expressions with friendly Java APIs to support our approach. Among them, these two tools Z3-Strs [12] and CVC4 [8] are excluded because the APIs provided by their Java versions are incomplete, and they always generate characters of **"**a-zA-Z**"** rather than fairer random ones when arbitrariness is required in practice. The other 6 tools are from the open source community [26], [33]. To test which one is more suitable for string data generation, we select four different types of regular expressions that are frequently used in practice and error-prone for the tools. The result in Table I shows that the tool `RgxGen` performs the best, which has friendly APIs and has been selected as the support tool for our work finally.

TABLE I
Comparison on Java Tools Generating String Data

Tool	E-prone Regular Expression			
	[\s\S]{1,4}	[^a]{1,3}	(?i)AbC	[\Q*\E]{1,3}
xeger [4]	✗	✔	✗	✗
Generex [39]	✔	✔	✗	E
MutRex [41]	✔	✔	✗	✗
bfgex [22]	✗	✗	E	✗
RgxGen [19]	✔	✔	✗	✔
random-string [9]	E	E	✗	E

✔: the generated string matches the regex; ✗: not matching; *E*: an exception occurs during generation.

B. String API Usage

Although there is already some previous work on the usage of the string APIs, we need to pay attention to the intra-procedural combination usage of these APIs, which is for the unit testing. So we perform a statistical analysis, trying to answer the following questions:

- **EQ1:** What is the usage frequency of string APIs in Java programs, especially for their combinations?
- **EQ2:** How many branching statements are affected by the string APIs' invocations?

Repository. To answer these questions, we have built a repository. It contains the `defect4j` [30] and the release version of `IntelliJ IDEA` [18] (version: 2020-02). The latter one is an IDE program widely used by Java developers. Although it is one program, there are 643 jar files with a total size of 850.87MB, involving a large number of widely used third-party libraries.

Methodology. We pay attention to the usage frequency of string APIs, especially their combinations in a method. We have counted the number of times the string API used on a single path. The loop has been unrolled only once during finding the path. And if a path has no any string API invocation, it will be ignored. In addition, we want to know how many API invocations affect conditional branches. In other words, if the return value of the string API invocation is used in a branch condition expression, it will affect the branch. We ignore the parts that have an indirect effect through complex inter-procedural calls and calculations for the return values of string API invocations.

Result. The string API usage result is shown in Table II. The combination involves multiple APIs, and we have found the combination of at most 5 APIs on a single path in the repository. For example, the fourth row means that there are 873 paths involving 84 combinations of two string APIs, and the 753 of 873 path conditions are affected by the return value of the 75 of 84 two-string-API combination invocations. From the table, we can see that string API invocations are used frequently in Java programs. Among them, the combinations of string APIs make up 8.5% (1068/12604). Subsequent

TABLE II
STRING API USAGE

API	NO.	In Path		In Branch	
		ct.(Total)	ct.(Dedup)	ct.(Total)	ct.(Dedup)
Single	1	11,536	29	8,304	25
Combination	2	**873**	**84**	**753**	**75**
	3	159	58	132	49
	4	32	7	27	6
	5	4	2	3	1
Sum		12,604	180	9,219	156

`Single`: a single string API invocation.
`Combination`: combination of multiple API invocations.
`NO`: the number of API combinations, a single API involves only one API.
`ct.(Total)`: the usage the API combination is used.
`ct.(Dedup)`: the count the API combination is used after deduplication.

experiments will show that they are more difficult to test. Besides, 73.1% (9219/12604) of string API invocations or their combinations have affected the branches in the programs. This is one of our motivations to tackle the challenges in the string test data generation.

C. Motivating Example

Listing 1 shows an example where three string APIs are invoked serially. There are two explicit paths from this code snippet based on the original `if` branch (line 5). The branch conditions in both paths are related to string API invocations. Since the SMT syntax mentioned earlier does not support converting the string API method *split(...)* to an SMT format, the constraint solvers cannot solve the path conditions. `EvoSuite` and `Randoop` can not generate suitable test data to cover them based on the evolutionary algorithms or random testing due to the complexity under the combination of multiple string API invocations.

Listing 1. A Motivating Example

```
1   // str is a method parameter
2   String[] splitArray = str.split("#");
3   String splitStr = splitArray[2];
4   String subStr = splitStr.substring(3, 8);
5   if (subStr.equals("class")) {
6     ... //True branch: omitted
7   } else { ... //False branch: omitted }
```

In addition, there are some implicit paths which may cause defects. The potential `ArrayIndexOutOfBoundsException` may be triggered since there is no judgement on whether the index is out of bounds (line 3). The string API invocations and path conditions in both explicit and implicit paths can help us proactively trigger potential exceptions. To trigger the potential `StringIndexOutOfBoundsException` (line4), one possibility is the length of the *splitStr* is less than 3 and the *str* can be represented by the regular expression `"[#]{2}[^#]{2}"`. The corresponding string **"##aa"** can be generated as the input of *str* which can trigger the exception. Besides, to cover the branch in line 6, the *subStr* should equal **"class"** and the *str* can be represented by `"[#]{2}[^#]{3}class"`. Therefore, the corresponding string **"##abcclass"** can cover the true branch. Other paths can be explored in the same way. The regular expressions above actually represent the set of strings that can trigger an exception or cover the branches. In other words, we not only have found a string that can trigger an exception or cover a branch, but found the characteristics of the input string parameter. One of our purposes is to infer these regular expressions according to the program and use them to generate string data in this paper.

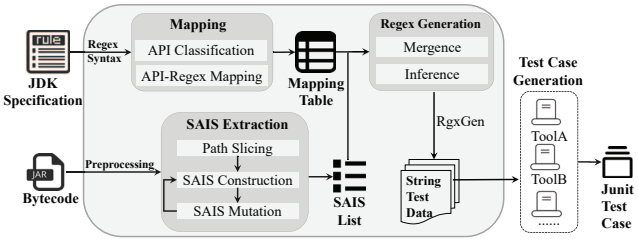

Fig. 1. Overview of Our Approach.

III. STRING INPUT DATA GENERATION

In this section, we characterize the string parameters with regular expressions. Figure 1 shows the overview of our approach. The input is the bytecode jar files, and the specifications for string APIs in JDK. In the mapping module, to deal with the different impacts from the different string APIs for a path, we classify the string APIs into two categories, according to whether their return values can be used in a conditional expression directly. Then for each string API with different conditional expressions, the API-Regex mapping provides a well-defined regular expression wrapper (`RegexWrapper`).

The preprocessing is to obtain the basic properties of the program, such as the class hierarchy and the control flow graph. In order to focus on the string data generation, we extract the string-related path and simplify it into string API invocation sequences (SAIS) for string parameters. Next we mutate the sequences to simulate the string API related exceptional circumstance, and attempt to generate the corresponding string data to trigger the exception. For each string API invocation sequence, an inference algorithm based on the API-Regex mapping is offered to derive the characteristics of the input string parameters represented by a regular expression. After that, the regular expression can be used for the string input data generation and the test case generation. Finally, the test case will be executed to find bugs or cover more branches to improve the quality of the program.

A. String API Invocation Sequence Extraction

A traditional path in the program is represented as a sequence of simple statements and conditional expressions [49], [50]. To analyze the string API invocation, we focus on the string related path with the string parameter and define it as a `StringParaPath`. Besides, as we mainly serve for unit testing, we focus on covering more intra-procedural paths. And the entrances to the analysis are all public methods in the testing program. Based on the `StringParaPath`, we split and simplify it into multiple string API invocation sequences by slicing analysis to deal with the string parameter individually.

Definition III-A1. ***(StringParaPath)*** *A* `StringParaPath` *is the path in a method which contains at least one string API invocation statement on the local variable of the string parameter.*

Definition III-A2. ***(String API Invocation Sequence (SAIS))*** *A* `SAIS` *is a 2-tuple*

$$SAIS = \langle CondExpr, Trajectory \rangle$$

, where the $CondExpr$ *is a conditional expression in the* `StringParaPath`*, and the* $Trajectory$ *is a k-length list of ordered pairs*

$$\langle (stmt_1, map_1), (stmt_2, map_2), ..., (stmt_k, map_k) \rangle$$

where the $stmt_i$ *in each pair is a string API invocation statement related to the conditional expression and the* map_i *maps the variables in the* $stmt_i$ *to their values.*

To construct the $SAIS$, the $StringParaPath$ needs to be sliced to snippets firstly, which purely contains the flow-dependent [1] statements for a single conditional expression. Next the $SAIS$ can be built under the value analysis based on itself. Finally, we make mutations to get more $SAIS$s if it involves the array index operations or the string API method which may throw exception(s).

Slicing. Algorithm 1 shows the slicing process for a `StringParaPath`. It takes a $StringParaPath$ and one of its conditional expressions as the inputs and outputs a statement snippet. The slicing is to record the flow-dependent statements sequence on each conditional expression in the $StringParaPath$. It starts with the conditional expression and searches the assignment statements of its dependent variables backward along the $StringParaPath$ (line 1-6). If it encounters such a statement, the statement will be added into the snippet (line 11). At the same time, the assignment statement may have its own dependent variables which also need to be searched (line 12-13). Thus the searching is an iterative process ending until all dependent variables are handled (line 17-18) or it reaches the start point of the path (line 7). In particular, it uses a boolean variable to mark whether the snippet is the one containing string API calls on the variable localized from a string parameter during the slicing (line 14-15). If it is false, the `SAIS` construction is unnecessary (line 20-21). The the snippet list will be returned as the input of the construction of the SAIS at the end.

Construction. For each snippet generated in the slicing, a $SAIS$ can be constructed based on it as shown in Algorithm 2. For each string API invocation statement in the snippet, the mapping from its variables to the values is built and added into the trajectory of the `SAIS` (line 8-11). If the value of a variable can be inferred as a concrete value, its value in the mapping will be a constant. Otherwise it will be marked with a special mark similar to a symbolic value in the mapping. In particular, for convenience in the inference of the regular expression in the next section, when getting an element of an array with index, we combine the assignment statement of the array and its index expression, and put the variables together

Algorithm 1: Slice

```
   Input: StrPP, condExpr
   Output: snippet
1  containsStringParameter = False;
2  snippet = {};
3  variableSet = {};
4  stmt = condExpr;
5  snippet.insertAtFirst(stmt);
6  variableSet.addAll(getVariables(stmt));
7  while stmt != NULL do
8      for each v ∈ variableSet do
9          if stmt.isAssignStatementOf(v) then
10             variableSet.remove(v);
11             snippet.insertAtFirst(stmt);
12             variables = getVariables(stmt);
13             variableSet.addAll(variables);
14             if isStringParaLocalization(stmt) then
15                 containsStringParameter = true;
16             break;
17     if variableSet.isEmpty() then
18         break;
19     stmt = StrPP.getPredecessorOf(stmt);
20 if containsStringParameter == False then
21     snippet = NULL;
22 return snippet
```

Algorithm 2: Construction

```
   Input: snippet
   Output: SAIS
1  SAIS = {};
2  SAIS.CondExpr = seg.getLastElment();
3  for each s ∈ snippet do
4      if s.contanisStringAPIInvocation() then
5          if s.getReturnType().isArrayType() then
6              expr = getIndexExpr(s);
7              s = conbination(s, expr);
8          variables = getVariables(s);
9          m = getValuesMapping(variables);
10         pair = ⟨s, m⟩;
11         SAIS.Trajectory.add(pair);
12 return SAIS
```

(line 5-7). The following snippet shows an example of the combination.

```
1 array = str.split("#"); element = array[2];//original
2 element = str.split("#")[2];//combination
```

Mutation. After the `SAIS` construction, we also mutate it in order to actively trigger the potential exceptions which may be caused by string-related or array-operation-related operations at runtime. The former involves the string APIs which throw `StringIndexOutOfBoundsException` in their implementations, such as *substring(...)* and *charAt(...)*. The latter contains the string APIs whose return type is array, such as *split(...)*. It is easy to cause an `ArrayIndexOutOfBoundsException` when getting an element in an array. For each statement which is contained in an `SAIS` and may cause exceptions, we mutate the `SAIS` to another one to characterize the string parameter to trigger the potential exceptions. The $Trajectory$ of the mutated `SAIS` can be denoted as

$$\langle (stmt_1, map_1), ..., (stmt_i, map_i), (stmt_N, map_N) \rangle$$

TABLE III
A SAIS AND ITS VARIANT

SAIS	CondExpr:	eq==True	
	Trajectory:	$splitStr$=str.split(v_4)[i_1]	{v_4:"#", i_1:2}
		$substr$=$splitStr$.substring(v_2,v_3)	{v_2:3,v_3:8}
		eq=$subStr$.equals(v_1),	{v_1:"class"}
SAIS$_N$	+CondExpr:	$length$<3	
	Trajectory:	$splitStr$=str.split(v_4)[i_1]	{v_4:"#", i_1:2}
		+ $length$=str.split(v_4).length	{v_4:"#"}

, where the $stmt_i$ is the position of the API invocation, and the $stmt_N$ is the new assignment statement for the length of the array or string caller, and the map_N is the mapping corresponding to the new statement $stmt_N$. At the same time, a new conditional expression will be constructed as the one in the mutated SAIS representing the length judgement.

For the example in Listing 1, an SAIS and its variant are shown in Table III. When there is an array index operation in the SAIS, a length judgement related statement and conditional expression will be added to form another SAIS denoted as SAIS$_N$.

B. API-Regex Mapping

Based on the powerful expressive ability of the regular expression, we have established an API-Regex mapping to represent the string APIs with regular expressions. JDK (version: 8u292) provides 76 string APIs in the class `java.lang.String`. We focus on the 50 string APIs whose modifiers are public and non-static since the static API is used for the class instead of a string instance. Among them, two methods, *hashcode()* and *concat(...)*, are excluded by us because the former one is used for the uniqueness and the latter one has a complex semantic. For each of the 48 remaining string APIs and its related conditional expression, a regular expression is constructed according to its semantics as described by the JDK specification. The construction of regular expressions takes whether the return value of conditional expression is true into account. If it is false, we can generate a string not matching the regular expression. That means the parameter in the SAIS is characterized with a regular expression, which can be used for another SAIS obtained after negating the conditional expression to avoid redundant inference. In fact, it does not pay more attention to identifying which one is true and which one is false when implementing, since there must be another SAIS with a negated conditional expression for each SAIS with a conditional expression. Thus a string matching the regular expression can be used to try to cover a branch and the string not matching it can be used for another one. In addition, the set of the strings corresponding to the generated regular expression for each API is a proper subset (Total: 31) of or is equal (Total: 17) to the one satisfying the API. Because `"[^abc]"` is used to indicate that the string **"**abc**"** is not matched, but it means in regular expression syntax that none of the three letters **"**abc**"** will appear.

Besides, these string APIs play different roles in the path and have different effects on regular expression inference. Different string APIs usually appear in different positions in the SAIS. Therefore, they are classified into two categories as follows.

- **Terminated API.** The API whose return type is `char`, `int`, `boolean`, `char[]` or `byte[]`, that is, primitive type or its array type, is considered as a terminated API (Total: 15). In general, its return value is used directly in the conditional expression,
- **Non-terminated API.** The API whose return type is `String`, `CharSequence` or `String[]` is considered as a non-terminated API (Total: 33). Their return values cannot appear directly in conditional expressions. By default, we use the method *equals(...)* instead of *"=="* to compare strings for equality. Usually, non-terminated APIs appear anywhere in SAIS except at the end.

In particular, we have taken getting the length of an array as a special terminated API to analyze the path that throws an exception, like *arr.length* especially for the mutated SAIS. The trajectory of a SAIS contains one terminated API invocation and possible multiple non-terminated ones. When involving multiple APIs, the regular expression corresponding to each API needs to be merged one by one to get a final regular expression. We design a data structure named `RegexWrapper` to describe the regular expression and the information to use when merging.

Definition III-B1. ***(RegexWrapper)*** *A `RegexWrapper` is a 4-tuple*

$$\mathcal{W} = \langle R, L, S, L_{min} \rangle$$

, where

- *R is the regular expression according to the string API and the expression on its return value;*
- *L is the length of the string that the R allows to merge and –1 indicates that there is no limit for the length;*
- *S is a special regular expression with which to replace during inference if any character regular expression is in the suffix of the R;*
- *L_{min} is the minimum length of a string corresponding to the R;*

Definition III-B2. ***(API-Regex Mapping)*** *A `API-Regex Mapping` is a 2-tuple*

$$\mathcal{M} = \langle APIPair, RegexWrapper \rangle$$

, where $APIPair$ is a pair including the string API invocation and its related conditional expression, and $RegexWrapper$ is defined above.

We can get one `RegexWrapper` by one `APIPair` from the `API-Regex Mapping`. For the terminated APIs which appear at the end of the SAIS and have the conditional expression, the *S* and *L* in a `RegexWrapper` are always `NULL` and –1, since they do not need to merge another one. And for the non-terminated APIs, their conditional expressions are always `NULL`, because the SAIS does not end with them. Table IV shows some API-Regex mappings (access anonymous link[1] for all). The generated regular expression for each API is used to represent the caller of that API. For example, the

[1]https://github.com/suoyi123wang/JustinStr

TABLE IV
PARTIAL API-REGEX MAPPING

APIPair		RegexWrapper			
API Invo. (caller: *str* | *arr*)	CondExpr	R	L	S	L_{min}
r=str.startswith(v)	r==T	v[\s\S]*	–1	NULL	Len(v)
r=str.length()	r==l	[\s\S]{l}	–1	NULL	l
r=str.substring(i_1,i_2)	NULL	[\s\S]{i_1}	i_2-i_1	NULL	i_1
r=str.split(v)[i]	NULL	[v]{i}	–1	[^v]	i
r=arr.length	r >i	[\s\S]{i+1,}	–1	NULL	i+1

substring(...) method (row 5) returns a new string (r) that is the substring of the caller (str) from the begin-index (i_1) to the end-index ($i_2 - 1$). The characters before the begin-index in the caller (str) will be cut and not affect the subsequent statements. Thus they will be characterized by any strings with a length of i_1. So the element R in the `RegexWrapper` is the regular expression `"[\s\S]{`i_1`}"` and its minimum length L_{min} is the value of i_1. Because the *substring(...)* is a non-terminated method, other string APIs will be called by the return variable r later. Therefore, a subsequent regular expression characterizing the return variable r needs to be merged into the current regular expression. And the length of subsequent regular expression is restricted to ($i_2 - i_1$), represented by the element L. The length of the caller (str) is constrained to be i_2, that is, an arbitrary string of length i_1 and another arbitrary string of length i_2-i_1 will be merged later denoted as $\boldsymbol{str}_{len:i_2} = \boldsymbol{R}_{len:i_1} + \boldsymbol{R}^r_{len:(i_2-i_1)}$. Moreover, it has no extra limitation for the API invocation, so the element S is `NULL`. The next section will introduce the inference algorithm to characterize the caller in a sequence.

C. Regex Generation

The trajectory in an `SAIS`, which is an ordered pair sequence with API invocations and their mappings, contains multiple non-terminated APIs and a terminated one. It can be denoted briefly as

$$traj = \langle N_1, M_1\rangle, ..., \langle N_i, M_i\rangle, ..., \langle N_{k-1}, M_{k-1}\rangle, \langle T, M_k\rangle$$

, where N_i is the non-terminated API invocation statement, T is the terminated API invocation one following with a related conditional expression and M_i is the mapping from the variables of the statement to their values. For each pair in a trajectory, the $RegexWrapper$ corresponding to it can be obtained from the API-Regex mapping. For the inference of an $SAIS$ with the multiple string API invocations, the inference between two $RegexWrapper$s called merge function needs to be introduced firstly before the $SAIS$ can be inferred.

Mergence. The implementation of the merge function is described in Algorithm 3. It takes two $RegexWrappers$ (w_2 and w_1) as inputs and outputs a new one ($\mathcal{W}$). If there is a restriction for any character in w_2, it needs to replace the regular expression `"[\s\S]"` in w_1 with the restriction expression in w_2 (line 3). In addition, if w_2 has a length limitation for the merged string, the Kleen star (**"*"**) in the element R of w_1 needs to be concretized into a certain one ($w_2.L - w_1.L_{min}$) and the minimum length of w_1 has updated to $w_2.L$ (line 5-7). After that, it concatenates the regular expression R of w_1

Algorithm 3: Merge

```
   Input: w2, w1
   Output: W
1  ANY_REGEX = "[\s\S]";
2  if w2.S != NULL then
3      w1.R = w1.R.replace(ANY_REGEX, w2.S);
4  if w2.L != -1 then
5      newLen = w2.L − w1.Lmin;
6      w1.R = w1.R.replace("*", "{" + newLen + "}");
7      w1.Lmin = w2.L;
8  W.R = concat(w2.R, w1.R);
9  W.L = -1;
10 W.S = NULL;
11 W.Lmin = w1.Lmin + w2.Lmin;
12 return W;
```

Algorithm 4: Inference

```
   Input: SAIS
   Output: regex
1  trajectory = SAIS.trajectory;
2  condEpr = SAIS.condExpr;
3  length = trajectory.length;
4  i = length- 1;
5  do
6      pair = trajectory[i − −];
7      if i == (length - 1) then
8          w = getWrapperFromMapping(pair, condEpr);
9      else
10         W = getWrapperFromMapping(pair, NULL);
11         w = Merge(W, w);
12 while i ≥ 0;
13 regex = w.R
14 return regex;
```

to the one of w_2 to get a new regular expression assigned to the element R of the new `RegexWrapper` ($\mathcal{W}$) without the restrictions for its length and suffix (line 8-10). Its minimum length is ($w_1.L_{min} + w_2.L_{min}$) (line 11). Finally it is returned for continued inference.

Inference. Based on the merge function, an inference can be conducted for a $SAIS$ as shown in Algorithm 4. It takes a $SAIS$ as the input and outputs a regular expression $regex$. It starts from the terminated API invocation point of the trajectory in the $SAIS$. And the `RegexWrapper` corresponding to it is based on the values of its variables and the conditional expression (line 8). The merge process iteratively executes backwards along the trajectory and terminates after its starting point has also been iterated (line 5-12). During the iterating, the conditional expression of the non-terminated API invocation is `NULL` (line 10). In the end, the regular expression element R in the final $RegexWrapper$ is returned as the result. In other words, the regular expression has characterized the string parameter under the $SAIS$. Then it can be used for the string test data generation and test case generation.

Figure 2 shows the inferring process for the `SAIS` in Table III. It starts from the terminated API invocation point and ends with the starting one of the sequence. By looking up the API-Regex mapping table three times and two rounds of the mergence iteration, the regular expression is finally obtained with the value `"[#]{2}[^#]{3}class"` characterizing the input parameter str in the sequence.

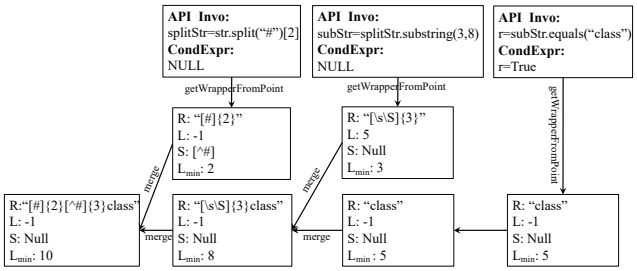

Fig. 2. Inference Example for the SAIS in Table III

Complexity Analysis. In the process of the SAIS extraction, we need to find paths for each public method on its CFG [38] first. We unroll each loop only once, so this is a problem of finding linearly independent paths which can be represented by the cyclomatic complexity. And the cyclomatic complexity (M) of the method (or program) is shown as follows :

$$M = E - N + 2P$$

, where E is the number of edges of the CFG, N is the number of nodes of the CFG, and P is the number of connected components in the CFG. At this time, not all paths are needed to be stored, but only those StringParaPaths. Therefore, the upper bound on the space complexity equals to the cyclomatic complexity M. The analysis of the slicing, construction and mutation for an SAIS is linear. These three steps will greatly reduce the complexity of subsequent analysis which reduces both the number of paths to be analyzed and the length of the sequence. In fact, the sequence involves at most 5 APIs according to Section II-B. When generating regular expressions, the inference contains the mergence operation. For some points in the sequence whose context needs to be explored, the worst time complexity is $O(P^2)$ where P is the number of points in the sequence, which is much smaller than N (P^2 is much smaller than N too) and is not greater than 5 in practice as mentioned before. And the space complexity is linearly related to P, which is $O(P)$. In summary, the time complexity of the whole process is

$$O(M) + O(M) + O(P^2) = O(M)$$

. And the space complexity is also $O(M)$. In other words, the time and space complexity of the whole process is linearly related to the cyclomatic complexity of the program.

IV. Evaluation

Based on our approach, we have developed an automatic tool named JustinStr to implement the regular expression inference for the programs. It has been built on the top of Soot [47] and the intermediate representation Shimple. And it has integrated the RgxGen to automatically output strings based on the inferred regular expressions. For those regular expressions that contain symbolic values, we ignore their string generation. In this section, we will evaluate JustinStr by answering the following three questions.

- **RQ1:** How effectively does JustinStr characterize the input string parameter?
- **RQ2:** Is the string data generated by JustinStr helpful to improve the branch coverage in programs?
- **RQ3:** How does JustinStr perform on finding bugs?

A. Experiment setup

All experiments below have been executed in a docker container, where the operating system is Ubuntu 9.4.0 (Linux version 5.4.0-107-generic) with 46 cores (Intel(R) Xeon(R) E5-2680) and 50G RAM. In addition, the Java environment is JDK-1.8 and the JUnit environment is JUnit4.

Setup for RQ1. Ensuring that regular expressions are semantically correct is difficult. And testing is a common way to ensure the correctness of regular expressions [51]. In order to test the characterization ability of JustinStr for the input string parameter, we have constructed a data set based on the programs in Section II-B. We obtain a total of 85 combinations of API usage, but they distribute in different projects and mix with many other execution environment requirements. For a more pure evaluation, we exclude the interference of other factors and extract these 85 combinations and their corresponding real parameters to build new methods. For each API combination, we select 5 groups of real parameters (and expression parameters if necessary) randomly. If the number of an API combination usage is less than 5, the parameter involved will be used multiple times. And an if-statement including the expression of the last string API is added in the end of each new method with its true and false branches.

Therefore, we have constructed a data set with a total of 425 methods. There are two branches (two paths) for each method and both of them are affected by the same string API invocation or their combination. JustinStr infers two regular expressions for each method, characterizing the input parameter for the true branch and the false one respectively. 10,000 strings are generated by RgxGen for each regular expression to eliminate randomness, which are used as the method inputs to test whether the branch will be executed. During the experiment, we have found that the upper limit of the length for the generated string has a little impact on the results, so we also conduct experiments for that. If a branch is executed under the input string data generated by JustinStr, the regular expression corresponding to it will be recorded as an effective one. It indicates that the input parameter is effectively characterized by JustinStr.

Benchmark for RQ2 and RQ3. To answer RQ2 and RQ3, we have collected three datasets of programs from developers with different experiences. **(1) Classroom Assignments.** The first dataset is the student assignments in a programming practice class for the beginners. 26 assignments have been selected from 26 students, 3 of which cannot be compiled successfully and are excluded. **(2) LeetCode Solutions.** Moreover, another dataset is the official solutions from the LeetCode [43] platform which is popular with interviewees. There are a total of 600 questions labeled with "string" in the platform, of which 79 questions are viewable to members only. We have obtained 290 official solutions from 521 questions, where the solution code match the regular

TABLE V
STATISTICS OF THREE DATASETS

Program	#Cls	#LOC	M_1	M_2	#RE
23 Assginments	406	9667	45	42	98
289 Solutions	289	4529	217	183	677
closure-compiler-20220202	8,162	218,939	194	153	475
commons-lang3-3.12.0	505	9,600	73	39	211
commons-io-2.11.0	240	5,544	21	17	81
mysql-connector-java-8.0.29	1,345	42,726	56	44	163
poi-ooxml-5.2.2	1,529	47,176	45	31	84
OpenJDK-8u292	25,574	697,752	1,513	1,250	6,697

expression "```Java[\s\S]*?```". Besides, one official solution is excluded because its local compilation fails. Finally 289 official solutions have been put into the dataset, whose average number of views and average number of comments are 65,112 and 123, respectively. **(3) Open Source Programs.** 5 popular Java programs have been selected from the open source community and Defect4j benchmark, since they contain abundant string operations. The average usage numbers on the maven repository [25] for these 5 projects in Table V are 10,933. In addition, considering that JDK provides basic support for the execution of Java programs, we add three common modules of JDK to the benchmark, which are *tools.jar*, *nashorn.jar*, and *rt.jar*. The basic information of the programs is shown in Table V, where $\#Cls$ indicates the number of classes and the number of lines of code is $\#LOC$. The column M_1 represents the number of methods, each of which is contained in at least one *StringParaPath*. The column M_2 indicates the number of methods in which the regular expression without any symbolic value can be inferred by `JustinStr`. The methods in M_2 are a subset of the methods in M_1. The column $\#RE$ represents the number of expressions inferred by `JustinStr` without symbolic values.

Setup for RQ2. To test whether the generated string test data can improve branch coverage, we have applied the data into `EvoSuite` based on [23]. `EvoSuite` implements a seed pool, where the constants that appear in a class are put in, and the seeds are mutated to generate test cases. To reduce the mutating effect from `EvoSuite`, three strings are generated for each inferred regular expression, and put into the seed pool of a class after deduplication. If the string data has not appeared in the test case before output, the test case will be regenerated once again. Test cases have been generated 10 times with default configuration to reduce randomness.

Setup for RQ3. `JustinStr` implements a built-in module to apply the string test data corresponding to the inferred regular expressions to test cases quickly and directly. Besides, `JustinStr` supports generating test cases for all projects, while `EvoSuite` whose implementation is based on JDK, is internally set not to generate test cases for JDK to prevent confusion when generating. Moreover, `JustinStr` supports dealing with string data, but `EvoSuite` can also handle other types. Therefore, we choose these two tools with different characteristics to generate test cases with default configurations. We focus on the methods denoted as M_2 in Table V, where the regular expressions are generated without symbolic values. Test cases from `JustinStr` which cannot be successfully compiled are filtered out.

TABLE VI
HIT RATE OF REGULAR EXPRESSIONS GENERATED BY JUSTINSTR

Length	Branch	Single (%)	Combination (%)			
			2	3	4	5
L=10,20 (30,40,50)	True	100.00	100.00	100.00	100.00	100.00
L=10	False	99.77	99.41	99.33	98.09	100.00
L=20	False	99.55	99.14	99.02	99.11	100.00
L=30	False	99.35	98.84	98.72	99.35	99.99
L=40	False	99.14	98.47	98.50	99.49	99.99
L=50	False	98.94	98.20	98.18	99.65	99.98

B. Experimental Results

Result for RQ1. As shown in Table VI, the 85 string API combinations have been divided into 5 categories, which is consistent with the classification in Table II. The first column ($Length$) represents the maximum length for the string generation. The regular expressions we generate for the true branch are 100% covered for all 5 generation lengths, which are either equivalent to the strings satisfying the true branch or are the proper subset of them. For each API combination, about 99% of the false branches are covered on average since we generate a string which is the complement of the strings corresponding to the true branch.

Answer to RQ1. In summary, `JustinStr` can effectively characterize the input string parameter with a regular expression based on its API-Regex mapping and the inference algorithm for both single string API invocation and their combination usage.

Result for RQ2. The methods in any `StringParaPath` for the datasets assignments and solutions involve more single string API invocations and fewer combination usages than that in open source programs. Therefore the branch coverage for them is high by `EvoSuite` and the improvement for them is 0%. Table VII shows the branch coverage information improved by `JustinStr` for the five open source programs. The column $\mathrm{M}_{Cov100\%}$ represents the number of the methods whose branch coverage is 100% already after the execution of test cases generated by `EvoSuite` in 10 generations. We have ignored these methods and focused on the methods whose branch coverage is not 100%. And the column $\mathrm{M}_{Cov}(\%)$ shows the number of the methods whose branch coverage is improved by `JustinStr` and their percentages of the methods with any branch uncovered. The last two columns (Max_{Cov}) and (Ave_{Cov}) show the largest branch coverage improvement within a method and the average branch coverage of the methods whose branch coverage has been improved.

Answer to RQ2. `JustinStr` is effective for the improvement of the branch coverage by about 22%(40/186) of the methods, which can be improved by up to 57%, and 17% on average.

Result for RQ3. Table VIII shows the comparison of bugs found by `JustinStr` and `EvoSuite`. The second column is the number of bugs found by `JustinStr` and the next one is the number of bugs found by both `JustinStr` and `EvoSuite`. The fourth column represents the number

TABLE VII
BRANCH COVERAGE IMPROVED BY JUSTINSTR

Program	Seeds	$M_{Cov100\%}$	M_{Cov}(%)	Max_{Cov}	Ave_{Cov}
closure-compiler	972	58	+18(19%)	+50%	+21%
commons-lang3	441	10	+10(34%)	+57%	+11%
commons-io	128	4	+5(38%)	+43%	+17%
mysql-connector-java	301	16	+5(18%)	+34%	+11%
poi-ooxml	125	10	+2(10%)	+50%	+28%

TABLE VIII
COMPARISON OF BUGS FOUND BY JUSTINSTR AND EVOSUITE

Program	JustinStr	Common	Δ	Time (s)	
				JustinStr	EvoSuite
Assignments	38	27	11	339	1627
Solutions	72	60	12	14	12,453
closure-compiler	33	21	12	72	2,394
commons-io	11	1	10	16	80
commons-lang3	15	3	12	15	709
mysql-connector-java	7	3	4	23	306
poi-ooxml	3	0	3	40	-
OpenJDK	17	0	17	10,228	-
Total	196	115	81	-	-

of bugs that JustinStr can find but EvoSuite cannot. Totally, JustinStr has found 196 bugs, of which 81 can not be found by EvoSuite, and EvoSuite has found 287 bugs. JustinStr does not cover all bugs because it has not paid attention to input parameters of other types which may affect the result, while EvoSuite does. And the generation time consumed by JustinStr is much less than EvoSuite. Therefore JustinStr can be used as an effective supplementary bug finding tool, which can find 28% (81/287) more bugs with an extreme low cost of time. The 196 bugs found by JustinStr, are caused by ArrayIndexOutOfBoundsException (28%), StringIndexOutOfBoundsException (23%), NullPointerException (20%), NumberFormatException (9%) and others (20%). Besides, we have reported the JDK bugs we have found, of which 3 are known and have been fixed in the new version, and 14 have been confirmed by the JDK developers. The Issue ID and Commit ID of them are shown in Table IX. There are more flaws in solutions that only address a certain problem. But the three different data sets all have quite a few defects which show the string-related problem is ubiquitous. This reminds developers that they should pay more attention to checking for null pointers, array out-of-bounds, and string lengths for input parameters or intermediate variables during string operations regardless of experience.

Additionally, for these 115 bugs found by both JustinStr and EvoSuite, we have counted the number of bugs triggered by the test cases according to their orders. The earlier the order of the test cases that trigger the bug, the more efficient the test cases are. As shown in Figure 3, the bars represent the number of bugs triggered by the test case at current order and the poly-lines represent the number of accumulated bugs triggered by the test cases before and within the order. The number of bugs triggered by the first test case in JustinStr accounts for 49%(56/115) and the number of bugs triggered by the second test case accounts for 25%(29/115). The first two test cases in JustinStr can trigger 74% of the bugs,

TABLE IX
BUGS CONFIRMED OR FIXED BY THE JDK DEVELOPERS

Type: ArrayIndexOutOfBoundsException (2)
Issue ID: I4MWI1 (Commit ID), 8279422
Type: StringIndexOutOfBoundsException (14)
Issue ID: 8278186, 8279128, 8279129, 8279198, 8279218, 8279336, 8279341, 8279342, 8279362, 8279423, **21212bd18(Commit ID), 8279424, **411a404a9(Commit ID), **8baba7d11(Commit ID)
Type: Infinite Loop (1), **Issue ID**: 8278993

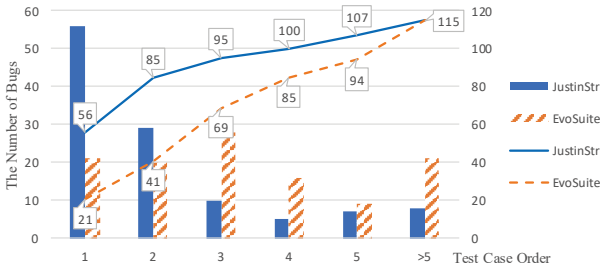

Fig. 3. Comparison on Number of Bugs Found by JustinStr and EvoSuite under Test Case Order. Bars represent the number of bugs triggered by the test case at current order and poly-lines represent the number of accumulated bugs triggered by the test cases before and within the order.

while EvoSuite only triggers 36%. In other words, the string data generated by JustinStr have represented the string parameter effectively, especially for the buggy path.

Answer to RQ3. Our approach can effectively characterize string parameters with regular expressions. The test cases generated based on it has efficiently found 81(+28%) more defects with an extreme low cost of time. Besides, the first two test cases in JustinStr have triggered 74% of the bugs, while in EvoSuite have triggered 36%. JustinStr has found 14 new bugs on JDK which are all confirmed by the JDK devepolers.

C. Case Study

Listing 2. Case Study in JDK

```
// Case 1: StringIndexOutOfBoundsException
public static String parseIdFromSameDocumentURI(String
     uri) {
   if (uri.length() == 0) {
      return null;
   }
   String id = uri.substring(1);
   if (id != null && id.startsWith("xpointer(id(")) {
      int i1 = id.indexOf('\'');
      int i2 = id.indexOf('\'', i1+1);
      id = id.substring(i1+1, i2);
   }
   return id;
}
//Case 2: Infinit Loop
static public String sansArrayInfo (String name) {
   int index = name.indexOf ('[');
   if (index >= 0) {
      String array = name.substring (index);
      name = name.substring (0, index);
      while (!array.equals ("")) {
         name = name + "[]";
         array = array.substring(array.indexOf(']') + 1);
      }
   }
   return name;
}
```

We will study cases in JDK to demonstrate the effectiveness of our approach. Listing 2 shows a typical bug caused

by `StringIndexOutOfBoundsException` in JDK. The method *parseIdFromSameDocumentURI(...)* in the class `org.jcp.xml.dsig.internal.dom.Utils` will throw a `StringIndexOutOfBoundsException` if the value of i_1+1 is less than 0 (line 9). One of the generated regular expressions is `"[\s\S]{1}xpointer\(id[\s\S]*"`. A generated string corresponding to the regular expression is **"**0xpointer(id(G6**"**, which will make the values of the variables i_1 and i_2 to be -1. Therefore, the `StringIndexOutOfBoundsException` will be triggered. Listing 2 shows another serious bug in the class `com.sun.tools.corba.se.idl.toJavaPortable` caused by the infinite loop found by `JustinStr`. It has generated a regular expression `"[[\s\S]*"`. When the corresponding string is **"[**", the loop will never break (line 20).

V. Discussion

Threat to Validity. `JustinStr` has been built on the top of Soot and its IR Shimple which may throw error during execution in practice. And it uses the tool `RgxGen` to generate the string data which involves randomness and so does `EvoSuite`. Although we have conducted multiple times to reduce the effect of randomness, they may prevent a complete reproduction of the experiment. Moreover, the effectiveness of regular expressions in RQ1 is carried out by testing, which cannot guarantee the correctness of the regular expressions. There is currently no more efficient solution, although some APIs have been proven to be equivalent to the regular expressions. So the loss and impact on accuracy of each iteration in the inference process are difficult to be evaluated. Furthermore, it is unknown how many other conditional expressions ignored by us affect the $SAIS$.

Extensive Scenario. The regular expressions in the experiments are the ones where the values of the variables can be obtained as concrete values after analysis. According to our statistics based on the programs in Section II-B, this accounts for 46% (5811/12604). For the remaining 54%, some symbolic values in them can be specified to generate a combination of inputs, if they are also input parameters. In addition, the string parameter mentioned in this paper is a single parameter, but the approach can be used to characterize any string variable (local or field one) in the program for the object-oriented instance construction and the inter-procedural string variable analysis. Besides, other programming languages also provide string APIs themselves, such as C++, Python, and our approach can also be transplanted for the testing of programs written in those languages.

VI. Related Work

Test Data Generation. `EvoSuite` [24] is one of state-of-the-art tools to generate unit test case automatically, and it adopts a search-based strategy to generate test cases. One of the seed strategies in `EvoSuite` is dynamic seeding, which uses any values observed during execution as the seeding [44]. This strategy can convert some API methods and their parameters into dynamic seeds. Randoop [40] is another tool for automatic test case generation, which uses a random strategy to generate regression tests. Both tools cannot generate appropriate test data when involving complex string API invocation or their combinations. In contrast to them, our approach can handle both simple and single APIs as well as complex ones and even their combinations.

String Constraint Solving. There are currently three types of approaches for string solving constraints based on a survey [6]. The first one usually is based on the finite state automata like [16], [17], [27], [32], [48]. The second one is based on word equivalence and mainly uses the SMT solvers to solve such as [5], [11], [12], [36], [46]. The last approach is an expansion-based approach, which simplifies the string as consecutive character elements like [21], [31], [35]. They are more for general-purpose string operations than for those in the JDK. Although the solver [16] claims to support more syntax than that in SMT-LIB, they support more syntax about regular expressions, and the syntax about the string API is the same as that in [3]. Thomé *et al.* [45] studied string constraint solving for detecting vulnerability in web applications. They proposed a search-driven constraint solving technique based on Ant Colony Optimization (ACO), which complements general-purpose SMT-solvers. Compared with them, the purpose of our approach is not to find an exact solution, but to efficiently characterize string variables to generate strings for testing.

Static Checking and Verification. Several researches focus on the semantic checking of the regular expressions. [28] focus on the regular expressions in XML documents and [10], [14] check the sub-type with both inputs and outputs. [34] implements 11 checkers for the regular expressions in pattern matching. In addition, [37] proposes a template in which the users need to describe the regular expression's semantics in natural language and they generate strings to check whether they can satisfy both the regular expression and its natural language in the template. The other analysis tools like [7], [29], [42] can find some simple string-related bugs but doesn't pay special attention to those bugs. Our approach is to use regular expressions to characterize string variables, generate strings to dynamically trigger bugs to reduce false positives or improve coverage.

VII. Conclusion

String APIs are commonly used in Java programs, yet error-prone. In this paper, we propose a novel approach to characterize the string parameter with regular expressions for string data generation and test case generation. We build an API-Regex mapping and offer an inference algorithm to fit the usage of both single string API and their combination. Evaluation shows that the mapping and the inference algorithm are effective for the string parameter characterization. And it also shows that our approach is powerful in both the branch coverage improvement (+17%) and bug finding (+28%). Finally we have found 14 new JDK bugs which are all confirmed by the JDK developers. In the future, we will focus on the regular expressions with the symbolic value and take the other types of parameters into account.

ACKNOWLEDGEMENT

Thanks to Zhen Tang for helping us with the Classroom Assignments in our datasets, to Yajun Zhu for comments on earlier drafts of this paper, and to the anonymous reviewers for their helpful comments and suggestions. This work is supported in part by the National Natural Science Foundation of China (Grant No. 62132020 and Grant No. 62102405).

REFERENCES

[1] Dependence analysis. https://en.wikipedia.org/wiki/Dependence_analysis, n.d.

[2] Regular expression. https://en.wikipedia.org/wiki/Regular_expression, n.d.

[3] Smt-lib. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml, n.d.

[4] xeger. https://code.google.com/archive/p/xeger/, n.d.

[5] ABDULLA, P. A., ATIG, M. F., CHEN, Y.-F., HOLÍK, L., REZINE, A., RÜMMER, P., AND STENMAN, J. Norn: An smt solver for string constraints. In *International conference on computer aided verification* (2015), Springer, pp. 462–469.

[6] AMADINI, R. A survey on string constraint solving. *ACM Computing Surveys (CSUR) 55*, 1 (2021), 1–38.

[7] AYEWAH, N., PUGH, W., MORGENTHALER, J. D., PENIX, J., AND ZHOU, Y. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2007), pp. 1–8.

[8] BARRETT, C., CONWAY, C. L., DETERS, M., HADAREAN, L., JOVANOVIĆ, D., KING, T., REYNOLDS, A., AND TINELLI, C. Cvc4. In *International Conference on Computer Aided Verification* (2011), Springer, pp. 171–177.

[9] BASTIAN BEHRENS, B. S. random-string. https://www.npmjs.com/package/random-string, n.d.

[10] BENZAKEN, V., CASTAGNA, G., AND FRISCH, A. Cduce: an xml-centric general-purpose language. *ACM SIGPLAN Notices 38*, 9 (2003), 51–63.

[11] BERZISH, M. Z3str4: A solver for theories over strings.

[12] BERZISH, M., GANESH, V., AND ZHENG, Y. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)* (2017), IEEE, pp. 55–59.

[13] CASS, S. Top programming languages 2021. https://spectrum.ieee.org/top-programming-languages/.

[14] CASTAGNA, G., COLAZZO, D., AND FRISCH, A. Error mining for regular expression patterns. In *Italian conference on Theoretical Computer Science* (2005), Springer, pp. 160–172.

[15] CHAPMAN, C., AND STOLEE, K. T. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016* (2016), ACM, pp. 282–293.

[16] CHEN, T., FLORES-LAMAS, A., HAGUE, M., HAN, Z., HU, D., KAN, S., LIN, A. W., RUEMMER, P., AND WU, Z. Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proceedings of the ACM on Programming Languages 6*, POPL (2022), 1–31.

[17] CHEN, T., HAGUE, M., HE, J., HU, D., LIN, A. W., RÜMMER, P., AND WU, Z. A decision procedure for path feasibility of string manipulating programs with integer data type. In *International Symposium on Automated Technology for Verification and Analysis* (2020), Springer, pp. 325–342.

[18] COMMUNITY, I. Jetbrains intellij idea. https://github.com/JetBrains/intellij-community, 2020.

[19] COMMUNITY, R. Rgxgen. https://github.com/curious-odd-man/RgxGen, n.d.

[20] DAVIS, J. C., COGHLAN, C. A., SERVANT, F., AND LEE, D. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018* (2018), ACM, pp. 246–256.

[21] DAY, J. D., EHLERS, T., KULCZYNSKI, M., MANEA, F., NOWOTKA, D., AND POULSEN, D. B. On solving word equations using sat. In *International Conference on Reachability Problems* (2019), Springer, pp. 93–106.

[22] DOUGLAS RODRIGO, ARTHUR HIRATA, N. L. D. L. A. P. A. K. bfgex. https://github.com/six2six/bfgex, n.d.

[23] FRASER, G. A tutorial on using and extending the evosuite search-based test generator. In *International Symposium on Search Based Software Engineering* (2018), Springer, pp. 106–130.

[24] FRASER, G., AND ARCURI, A. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 416–419.

[25] FRODRIGUEZ. mvnrepository. https://mvnrepository.com/, n.d.

[26] HARMEL-LAW, A. Using regex to generate strings rather than match them. https://stackoverflow.com/questions/22115/using -regex-to-generate-strings-rather-than-match-them , n.d.

[27] HOLÍK, L., JANK, P., LIN, A. W., RÜMMER, P., AND VOJNAR, T. String constraints with concatenation and transducers solved efficiently. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 1–32.

[28] HOSOYA, H., AND PIERCE, B. C. Xduce: A statically typed xml processing language. *ACM Transactions on Internet Technology (TOIT) 3*, 2 (2003), 117–148.

[29] HOVEMEYER, D., AND PUGH, W. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2007), pp. 9–14.

[30] JUST, R., JALALI, D., AND ERNST, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), pp. 437–440.

[31] KIEZUN, A., GANESH, V., ARTZI, S., GUO, P. J., HOOIMEIJER, P., AND ERNST, M. D. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology (TOSEM) 21*, 4 (2013), 1–28.

[32] KRINGS, S., SCHMIDT, J., SKOWRONEK, P., DUNKELAU, J., AND EHMKE, D. Towards constraint logic programming over strings for test data generation. In *Declarative Programming and Knowledge Management*. Springer, 2019, pp. 139–159.

[33] LANGKEMPER, S. Generating-random-strings-from-a-regular-expression. http://www.linuxonly.nl/docs/69/168_Generating_random_strings_from_a_regular_expression.html, n.d.

[34] LARSON, E. Automatic checking of regular expressions. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2018), IEEE, pp. 225–234.

[35] LI, G., AND GHOSH, I. Pass: String solving with parameterized array and interval automaton. In *Haifa Verification Conference* (2013), Springer, pp. 15–31.

[36] LIANG, T., REYNOLDS, A., TINELLI, C., BARRETT, C., AND DETERS, M. A dpll (t) theory solver for a theory of strings and regular expressions. In *International Conference on Computer Aided Verification* (2014), Springer, pp. 646–662.

[37] LIU, X., JIANG, Y., AND WU, D. A lightweight framework for regular expression verification. In *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)* (2019), IEEE, pp. 1–8.

[38] MCCABE, T. J. A complexity measure. *IEEE Trans. Software Eng. 2*, 4 (1976), 308–320.

[39] MIFRAH YOUSSEF, MYK KOLISNYK, P. H. A. N. Generex. https://github.com/mifmif/Generex, n.d.

[40] PACHECO, C., AND ERNST, M. D. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* (2007), pp. 815–816.

[41] PAOLO ARCAINI, A. G. Mutrex. https://foselab.unibg.it/mutrex/, n.d.

[42] PĂSĂREANU, C. S., AND RUNGTA, N. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (2010), pp. 179–180.

[43] PLATFORM, L. Leetcode. https://leetcode.cn/, n.d.

[44] ROJAS, J. M., FRASER, G., AND ARCURI, A. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability 26*, 5 (2016), 366–401.

[45] THOMÉ, J., SHAR, L. K., BIANCULLI, D., AND BRIAND, L. Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 198–208.

[46] TRINH, M.-T., CHU, D.-H., AND JAFFAR, J. Model counting for recursively-defined strings. In *International Conference on Computer Aided Verification* (2017), Springer, pp. 399–418.

[47] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 2010, pp. 214–224.

[48] WANG, H.-E., TSAI, T.-L., LIN, C.-H., YU, F., AND JIANG, J.-H. R. String analysis via automata manipulation with logic circuit representation. In *International Conference on Computer Aided Verification* (2016), Springer, pp. 241–260.

[49] ZHANG, J. Constraint solving and symbolic execution. In *Working conference on verified software: Theories, tools, and experiments* (2005), Springer, pp. 539–544.

[50] ZHANG, J., AND WANG, X. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering 11*, 02 (2001), 139–156.

[51] ZHENG, L.-X., MA, S., CHEN, Z.-X., AND LUO, X.-Y. Ensuring the correctness of regular expressions: A review. *International Journal of Automation and Computing 18*, 4 (2021), 521–535.