1 EXAMPLE PROGRAM

This supplemental material gives the details of the running example in our paper.

The toy program in Listing 1 is designed to identify and replace certain words from the input string, and then output the modified string and the log message. Specifically, if an input string contains "*wordNone*" or "*wordNtwo*", these two words will be replaced with "**1**" and "**2**", respectively. The log message records the operation of the program.

Listing 1: An example program

```
1  public static String process(String s){
2      if(s.contains("*1*") || s.contains("*2*")){
3          return "";}
4      int sign = 0;
5      int sum_1 = 0;
6      sum_1 = s.contains("wordNone") ? 1 : 0;
7      sign += sum_1;
8      s = s.replaceAll("wordNone", "?1?");                // Fault1: "?1?" should be "*1*"
9      int sum_2 = 0;
10     sum_2 = s.contains("wordNtwo") ? 2 : 0;
11     sign += sum_2;
12     s = s.replaceAll("wordNtwo", "*2*");
13     if(sign == 3){
14         return "both pattern recognized";}
15     String msg = sign == 1 ? "wordNone recognized" : "pass";
16     msg = sign > 2 ? "wordNtwo recognized" : msg;       // Fault2: "> 2" should be "== 2"
17     return s + "//" + msg;}
```

We give a test suite that comprises twelve test cases to debug this program, as shown in Table 1. Due to the distinction between the expected output and the actual output, $t_1 \sim t_6$ are determined as "$failed$" (also referred to as $f_1 \sim f_6$ hereafter). Among them, $f_1 \sim f_4$ are triggered by $Fault_1$, while $f_5 \sim f_6$ are triggered by $Fault_2$. A promising failure indexing technique should satisfy two points, i.e., correctly predicting the number of faults, and properly indexing failures to their root causes. In the context of this example, two fault-focused groups, i.e., $\{f_1, f_2, f_3, f_4\}$ and $\{f_5, f_6\}$, are expected.

Table 1: Test suite

Test case	Input	Expected output	Actual output	Result
t_1	"$speak\ wordNone$"	"$speak\ *1*//wordNone\ recognized$"	"$speak\ ?1?//wordNone\ recognized$"	failed
t_2	"$wordNone$"	"$*1*//wordNone\ recognized$"	"$?1?//wordNone\ recognized$"	failed
t_3	"$wordNonecontained$"	"$*1*contained//wordNone\ recognized$"	"$?1?contained//wordNone\ recognized$"	failed
t_4	"$wwwwordNoneeee$"	"$www*1*eee//wordNone\ recognized$"	"$www?1?eee//wordNone\ recognized$"	failed
t_5	"$has\ wordNtwo$"	"$has\ *2*//wordNtwo\ recognized$"	"$has\ *2*//pass$"	failed
t_6	"$wordNtwo$"	"$*2*//wordNtwo\ recognized$"	"$*2*//pass$"	failed
t_7	""	"$//pass$"	"$//pass$"	passed
t_8	"$midd*1*le$"	""	""	passed
t_9	"$*1*2*$"	""	""	passed
t_{10}	"$a\ normal\ sentence$"	"$a\ normal\ sentence//pass$"	"$a\ normal\ sentence//pass$"	passed
t_{11}	"$wordnonewordNtw$"	"$wordnonewordNtw//pass$"	"$wordnonewordNtw//pass$"	passed
t_{12}	"$wordNone\ and\ wordNtwo$"	"$both\ pattern\ recognized$"	"$both\ pattern\ recognized$"	passed

2 FAILURE INDEXING BY RECLUES

As mentioned in Section 4 in the original paper, ReClues involves four phases, i.e., breakpoint determination, variable information collection, distance measurement, and clustering. Here we give a step-by-step elaboration on these four steps based on the example program.

2.1 Phase-1: Breakpoint determination

We execute the twelve test cases against the program, and record the coverage information, as shown in the column "Coverage Information" in Table 2, where "1" denotes that a test case covers a statement, and "0" otherwise. We further reorganize the coverage information into program spectrum to facilitate spectrum-based suspiciousness calculation, as shown in the column "Spectrum Information" in Table 2. Specifically, the notations "N_{CF}" and "N_{UF}" denote the numbers of failed test cases that cover and do not cover a statement, respectively. Similarly, "N_{CS}" and "N_{US}" denote the numbers of passed test cases that cover and do not cover a statement, respectively. To evaluate the

Table 2: Breakpoint Selection process

S	Program	Coverage Information												Spectrum Information				Risk	Ranking
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	N_{CF}	N_{UF}	N_{CS}	N_{US}		
s_1	public static String process(String s){	1	1	1	1	1	1	1	1	1	1	1	1	6	0	6	0	6	14
s_2	if(s.contains(“*1*”) || s.contains(“*2*”)){	1	1	1	1	1	1	1	1	1	1	1	1	6	0	6	0	6	14
s_3	return “”;}	0	0	0	0	0	0	0	1	1	0	0	0	0	6	2	4	0	16
s_4	int sign = 0;	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_5	int sum_1 = 0;	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_6	sum_1 = s.contains(“wordNone”) ? 1 : 0;	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_7	sign += sum_1;	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_8	s = s.replaceAll(“wordNone”, “?1?”); //Fault1: “?1?” should be “*1*”	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_9	int sum_2 = 0;	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_{10}	sum_2 = s.contains(“wordNtwo”) ? 2 : 0;	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_{11}	sign += sum_2;	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_{12}	s = s.replaceAll(“wordNtwo”, “*2*”);	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_{13}	if(sign == 3){	1	1	1	1	1	1	1	0	0	1	1	1	6	0	4	2	9	4
s_{14}	return “both pattern recognized”;}	0	0	0	0	0	0	0	0	0	0	0	1	0	6	1	5	0	16
s_{15}	String msg = sign == 1 ? “wordNone recognized” : “pass”;	1	1	1	1	1	1	1	0	0	1	1	0	6	0	3	3	12	1
s_{16}	msg = sign > 2 ? “wordNtwo recognized” : msg; //Fault2: “>2” should be “==2”’	1	1	1	1	1	1	1	0	0	1	1	0	6	0	3	3	12	1
s_{17}	return s + “//” + msg;}	1	1	1	1	1	1	1	0	0	1	1	0	6	0	3	3	12	1

likelihood of a statement being faulty, the spectrum information is fed into a risk evaluation formula (for example, DStar, and * = 2), as defined in Formula 1.

$$suspiciousness_{\text{Dstar}} = \frac{N_{CF}^{*}}{N_{UF} + N_{CS}} \tag{1}$$

The calculated suspiciousness is given in the column “Risk” in Table 2. Sorting all statements in descending order by the suspiciousness, we can get a ranking list: {s_{15}, s_{16}, s_{17}, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_1, s_2, s_3, s_{14}}. Notice that if several statements share the same value of suspiciousness, we adopt the widely-used solution by ranking the tied statements in the ascending order of line numbers.

Then, we determine the Top-x% (e.g., Top-10%) riskiest statements as breakpoints. As such, s_{15} and s_{16} are selected.

2.2 Phase-2: Variable information collection

In this phase, we collect the variable information at preset breakpoints during the execution of the six failed test cases. Revisiting the definition in Section 4.2.2 in the original paper, we can represent failed test cases $f_1 \sim f_6$ as the run-time variable information $VI_1 \sim VI_6$, respectively. Here we take f_1 as an example. The proxy for f_1 is $VI_1 = \{bp_1\text{: } V_1^1, bp_2\text{: } V_2^1\}$. V_1^1 is the variable information collected at bp_1 (s_{15}) during the execution of f_1, which is a dictionary with variable names as *key* and variable values as *value*. Similarly, V_2^1 is those collected at bp_2 (s_{16}) when executing f_1. V_1^1 and V_2^1 are given as follows[1]:

$V_1^1 = V_2^1$ = {*s*: “speak ?1?”, *sign*: “1”, *sum_1*: “1”, *sum_2*: “0”, *msg*: “wordNone recognized”}

The run-time variable information collected during executing all of the six failed test cases is given in Table 3. V_1^1 is shown in this table with blue highlighting, while V_2^1 is shown with green highlighting. To put it another way, each of the failed test cases can be represented as the corresponding row in Table 3.

2.3 Phase-3: Distance measurement

A failed test case has been converted to the run-time variable information in Phase-1 and Phase-2. Therefore, the distance measurement between two failures equals that between two sets of variable information. Here we give an example to show how to calculate the distance between f_1 (triggered by $Fault_1$) and f_5 (triggered by $Fault_2$), i.e., $Distance_{f_1, f_5}$, according to Figure 3 in the original paper.

[1]During executing f_1, the variable information collected at the two breakpoints is identical, since the scale and complexity of the used toy program are very low.

Table 3: Run-time variable information

Failed test cases	Run-time Variable Information									
	*Breakpoint*1 : s_{15}					*Breakpoint*2 : s_{16}				
	s	**sign**	**sum_1**	**sum_2**	**msg**	**s**	**sign**	**sum_1**	**sum_2**	**msg**
f_1	"speak ?1?"	1	1	0	"wordNone recognized"	"speak ?1?"	1	1	0	"wordNone recognized"
f_2	"?1?"	1	1	0	"wordNone recognized"	"?1?"	1	1	0	"wordNone recognized"
f_3	"?1?contained"	1	1	0	"wordNone recognized"	"?1?contained"	1	1	0	"wordNone recognized"
f_4	"www?1?eee"	1	1	0	"wordNone recognized"	"www?1?eee"	1	1	0	"wordNone recognized"
f_5	"has *2*"	2	0	2	"pass"	"has *2*"	2	0	2	"pass"
f_6	"*2*"	2	0	2	"pass"	"*2*"	2	0	2	"pass"

The proxy for f_1 has been given previously. The proxy for f_5 is $VI_5 = \{bp_1: V_1^5, bp_2: V_2^5\}$, and V_1^5 and V_2^5 are as follows:

$V_1^5 = V_2^5 = \{s$: "has *2*", $sign$: "2", sum_1: "0", sum_2: "2", msg: "pass"$\}$

The evident distinction between the variable information of f_1 and that of f_5 exhibits that these two failures have different dataflows, despite the fact that they have the same coverage (as shown in Table 2).

The distance metric of ReClues is performed at two levels, i.e, the breakpoint level and the variable level.

At the breakpoint level. Considering Formula 1 in the original paper, we can get the value of $\sum_j^q BPCount_j$ to be 2 (according to Formula 4 and Formula 3 in the original paper), since both two breakpoints are covered by f_1 and f_5. While the numerator, i.e., $Distance_{bp_1}$ + $Distance_{bp_2}$, need to be further calculated at the variable level (according to Formula 2 and Formula 3 in the original paper).

At the variable level. Taking $Distance_{bp_1}$ as an example. $Distance_{bp_1}$ is set to $Distance_{var}^1$ according to Formula 2 in the original paper, which will call Formula 5 in the original paper. Considering Formula 5 in the original paper. We can get the union of the variables' names collected at bp_1 during the execution of f_1 and f_5, i.e., $\hat{V}_1^1 \cup \hat{V}_1^5$, is {**s**, *sign*, *sum_1*, *sum_2*, *msg*}. Thus, the value of the denominator, i.e., $|\hat{V}_1^1 \cup \hat{V}_1^5|$, is 5. And the numerator, i.e., $\sum_z^{|\hat{V}_j^\alpha \cup \hat{V}_j^\beta|} dis_z$, is the sum of the normalized Jaccard distances of these five variables' values. Combining Formula 6, Formula 7, and Formula 8 in the original paper, we can get the value of $Distance_{var}^1$ to be:

$$\left[\text{Jacc}\left(val_1^1, val_1^5\right) + \text{Jacc}\left(val_2^1, val_2^5\right) + \ldots + \text{Jacc}\left(val_5^1, val_5^5\right)\right] \ / \ 5. \tag{2}$$

Putting the specific values to it. For example, $\text{Jacc}(val_1^1, val_1^5)$ is the Jaccard distance between the values of the first variable ***s*** collected at bp_1 during the execution of f_1 and f_5, which is actually Jacc("*speak* ?1?", "*has* $*$ 2 $*$ "). Similarly, the distances of the values of the remaining four variables can be calculated.

As such, we can immediately get the distance between f_1 and f_5, i.e., $Distance_{f_1, f_5}$, is 0.8. We give the distances between each pair of failures in Table 4.

Table 4: Distance information

	f_1	f_2	f_3	f_4	f_5	f_6
f_1	0	0.2	0.2	0.2	0.8	1
f_2	0.2	0	0.2	0.2	1	1
f_3	0.2	0.2	0	0.2	0.8	1
f_4	0.2	0.2	0.2	0	1	1
f_5	0.8	1	0.8	1	0	0.2
f_6	1	1	1	1	0.2	0

2.4 Phase-4: Clustering

The clustering part of ReClues involves faults number estimation and clustering. Based on the distance information, the number of faults, i.e., 2, can be correctly predicted. And the clustering process delivers two groups, i.e., $\{f_1, f_2, f_3, f_4\}$ and $\{f_5, f_6\}$, which meet the expectations.