

Software Testing Project

SemFix

2016218037 刘莞姝

2016218040 马舒婕

2016218046 滕 飞

2016218055 杨宇杰

2016218041 缪东旭（组长）

May 19, 2017

What are we going to talk about?

Automated program repair / automatic bug fixing

*Automated program repair (automatic bug fixing) is an emerging area of research that focuses on **reducing the cost** of software bug fixing.*

*Automated program repair approaches **automatically** or **semi-automatically** modify buggy program to satisfy given **correctness criteria**. Examples of correctness criteria are **test suite** and **formal specification**. Typical program repair works in the following **three steps**: identifying faulty locations, inferring desired specification, and generating a patch. Existing approaches differ in the underlying techniques used for localization, inference, and patch generation. Roughly, they can be divided into **two groups**: **syntactical** (e.g., GenProg) and **semantical** (e.g., SemFix). Important attributes of automated program repair are scalability, repairability, and reliability of generated patches.*

-- Copied from program-repair.org

Why dose it matter?

It is all about TRADEOFF

Let's assume that:

- code complexity
- human resources
- compiler

The output of automated program repair

*Automated program repair approaches **automatically** or **semi-automatically** modify buggy program to satisfy given correctness criteria.*

- Guidelines?
- Solutions?

Paper we choose

SemFix: Program Repair via Semantic Analysis

Compared with GenProg, SemFix (or Angelix) is:

- newer -- 2012 vs 2016 (first paper 2013)
- better -- comparison in paper
- more open source -- hosted on Github
- less materials -.-

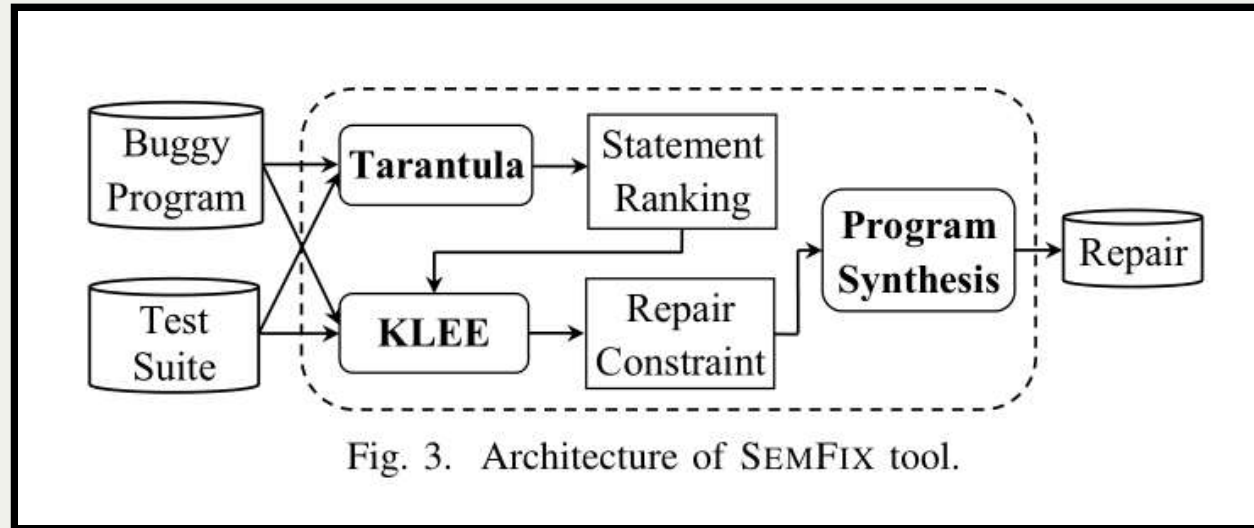
Analysis of chosen paper

*present an automated repair method based on symbolic execution,
constraint solving and program synthesis*

- Statistical Fault Localization

$$susp(s) = \frac{failed(s)/total\ failed}{passed(s)/total\ passed + failed(s)/total\ failed}$$

- Statement-level specification inference
- Program synthesis



Architecture of SEMFIX tool

- Tarantula technique
 - rank statements according to suspiciousness score
- KLEE
 - generate repair constraints
- Z3 SMT (satisfiability modulo theories) solver
 - solve a repair constraint
- Program synthesis
 - provided by the author

error types can be corrected

- if-conditions
- loop-conditions
- assignments
- guards

Examples

Example#1 addOneWhenPositive (I)

```
int addOneWhenPositive(int x) {  
    int r = 0;  
    if(x > 0){  
        r = x - 1;  
    }  
    else{  
        r = x;  
    }  
  
    return r;  
}
```

Test cases:

TC#1: (-1;-1)
TC#2: (0;0)
TC#3: (1;2)

```
angelix src test.c oracle 1 2 3 --assert assert.json
```

Patch we got:

```
--- a/test.c  
+++ b/test.c  
@@ -7,7 +7,7 @@  
    int addOneWhenPositive(int x) {  
        int r = 0;  
        if(x > 0){  
-           r = x - 1;  
+           r = (x + 1);  
        }  
        else{
```

```
class(  
  r = x;
```

Example#1 addOneWhenPositive (II)

```
angelix src test.c oracle 1 2 3 --assert assert.json --semfix
```

Patch we got:

```
--- a/test.c
+++ b/test.c
@@ -7,7 +7,7 @@
 int addOneWhenPositive(int x) {
     int r = 0;
     if(x > 0){
-        r = x - 1;
+        r = 2;
     }
     else{
         r = x;
     }
 }
```

And, recap what we learned in our class, there are three conceptions about bug:

- Fault
- Error
- Failure

Example#1 addOneWhenPositive (III)

```
int addOneWhenPositive(int x) {  
    int r = 0;  
    if(x > 0){  
        r = x - 1;  
    }  
    else{  
        r = x;  
    }  
  
    return r;  
}
```

With test cases:

TC#1: (-1;-1)

TC#2: (0;0)

```
angelix src test.c oracle 1 2 3 --assert assert.json --semfix
```

```
INFO      project      configuring validation source  
INFO      project      building json compilation database from validation source  
INFO      testing      running test '1' of validation source  
INFO      testing      running test '2' of validation source  
INFO      project      configuring frontend source  
INFO      transformation instrumenting repairable of frontend source  
INFO      project      building frontend source  
INFO      repair        running positive tests for debugging  
INFO      testing      running test '1' of frontend source  
INFO      testing      running test '2' of frontend source  
INFO      repair        repair test suite: ['1', '2']  
INFO      repair        validation test suite: ['1', '2']  
INFO      localization  No negative test exists
```

SO,

*Automated program repair approaches automatically or semi-automatically modify buggy program to satisfy given **correctness criteria**. Examples of correctness criteria are **test suite** and **formal specification**.*

Build from source & lesson learned

We failed to build it in various ways, if you are interested, you can look more detail in our [write-up](#).

Lesson learned:

- environment variables
- make install
- master branch of codebase

The screenshot shows the GitHub repository page for `mchtaev / angelix`. The file `angelix / activate` is selected, showing its content. The script is a shell script that sets up the environment for the angelix tool. It includes comments and various export statements for paths and directories.

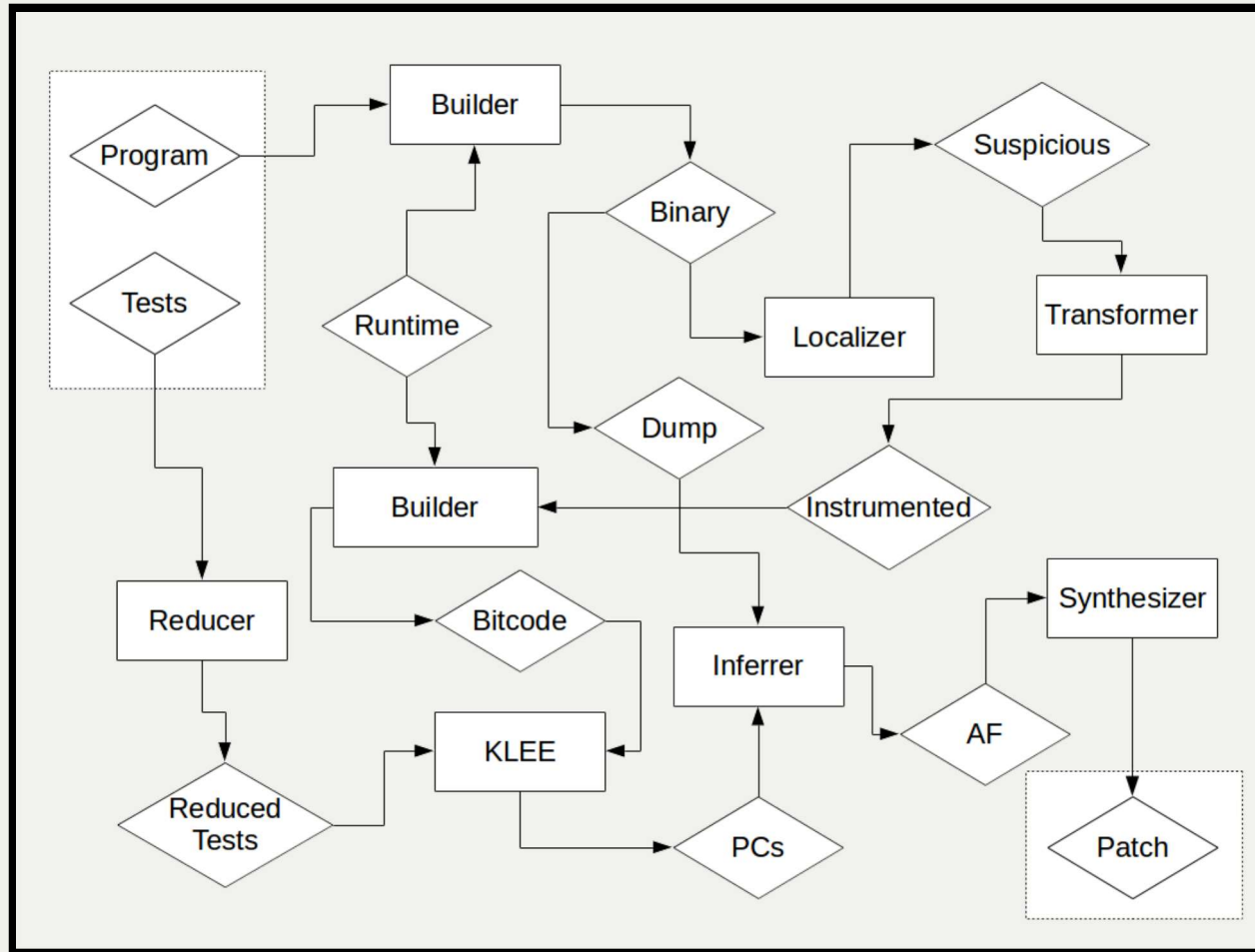
```
1 if [[ `uname -m` != "x86_64" ]]; then
2   echo "Error: Angelix supports only x86_64"
3   exit 1
4 fi
5
6 if [[ "$(lsb_release -si)" == "Ubuntu" ]]; then
7   export LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu:$LD_LIBRARY_PATH
8   export C_INCLUDE_PATH=/usr/include/x86_64-linux-gnu
9   export CPLUS_INCLUDE_PATH=/usr/include/x86_64-linux-gnu
10 else
11   echo "WARNING: add your system libraries to C_INCLUDE_PATH, CPLUS_INCLUDE_PATH and LD_LIBRARY_PATH"
12 fi
13
14 export ANGELIX_ROOT=$( cd $( dirname "${BASH_SOURCE[0]}" ) && pwd )
15
16 # locations of submodules:
17
18 export LLVM_GCC_DIR="$ANGELIX_ROOT/build/llvm-gcc4.2-2.9-x86_64-linux"
19 export LLVM2_DIR="$ANGELIX_ROOT/build/llvm-2.9"
20 export LLVM3_DIR="$ANGELIX_ROOT/build/llvm-3.7.0.src"
21 export CLANG_HEADERS="$LLVM3_DIR/tools/clang/lib/Headers/"
22 export LLVM3_INCLUDE_PATH="$ANGELIX_ROOT/build/include"
23 export STP_DIR="$ANGELIX_ROOT/build/stp"
24 export MINISAT_DIR="$ANGELIX_ROOT/build/minisat"
25 export Z3_DIR="$ANGELIX_ROOT/build/z3"
26 export Z3_JAR="$Z3_DIR/build/com.microsoft.z3.jar"
27 export Z3_2_19_DIR="$ANGELIX_ROOT/build/z3_2_19" # used for semfix
28 export KLEE_DIR="$ANGELIX_ROOT/src/klee"
29 export KLEE_INCLUDE_PATH="$KLEE_DIR/include"
30 export KLEE_LIBRARY_PATH="$KLEE_DIR/Release+Asserts/lib"
31 export KLEE_UCLIBC_DIR="$ANGELIX_ROOT/build/klee-uclibc"
32 export BEAR_DIR="$ANGELIX_ROOT/build/Bear"
33 export MAXSAT_DIR="$ANGELIX_ROOT/build/maxsat-playground"
34 export MAXSAT_JAR="$MAXSAT_DIR/target/scala-2.10/maxsat-playground_2.10-1.1.jar"
35 export SYNTHESIS_DIR="$ANGELIX_ROOT/src/synthesis"
36 export SYNTHESIS_JAR="$SYNTHESIS_DIR/target/scala-2.10/repair-maxsat-assembly-1.0.jar"
37 export NSYNTH_JAR="$ANGELIX_ROOT/src/nsynth/target/nsynth-1.0-jar-with-dependencies.jar"
38 export LOCAL_PERL_ROOT="$ANGELIX_ROOT/build/perl"
39
40 # environment for building and execution:
41
42 export PATH="$LLVM_GCC_DIR/bin:$PATH"
43 export PATH="$LLVM2_DIR/Release+Asserts/bin:$PATH"
44 export PATH="$KLEE_DIR/Release+Asserts/bin:$PATH"
45 export PATH="$Z3_DIR/build:$PATH"
46 export PATH="$Z3_2_19_DIR/bin:$PATH"
47 export PATH="$ANGELIX_ROOT/src/tools:$PATH"
48 export PATH="$ANGELIX_ROOT/build/bin:$PATH"
49
50 export ANGELIX_LIBRARY_PATH_KLEE="$ANGELIX_ROOT/build/lib/klee"
51 export ANGELIX_LIBRARY_PATH_TEST="$ANGELIX_ROOT/build/lib/test"
52 export ANGELIX_RUNTIME_H="$ANGELIX_ROOT/src/runtime/runtime.h"
53
54 # to run tests when linked with runtime:
55 export LD_LIBRARY_PATH="$KLEE_LIBRARY_PATH:$LD_LIBRARY_PATH"
56 # because I didn't manage to build stp statically:
57 export LD_LIBRARY_PATH="$STP_DIR/build/lib:$LD_LIBRARY_PATH"
58
59 # for Z3 bindings:
60 export LD_LIBRARY_PATH="$Z3_DIR/build:$LD_LIBRARY_PATH"
61 export PYTHONPATH="$Z3_DIR/build:$PYTHONPATH"
62
63 ulimit -s unlimited
64
65 mkdir -p $ANGELIX_ROOT/build
66 mkdir -p $ANGELIX_ROOT/build/bin
67 mkdir -p $ANGELIX_ROOT/build/include
68
69 # utilities for testing:
70
71 dump-ast () {
72   "$LLVM3_DIR/build/bin/clang" -Xclang -ast-dump -fdiagnostics-color=never -fsyntax-only "$1"
73 }
74
75 instr-rep () {
76   instrument-repairable "$1" -- "-I$LLVM3_DIR/build/lib/clang/3.7.0/include" -include "$ANGELIX_RUNTIME_H" -D ANGELIX_INSTRUMENTA
77 }
78
79 instr-sus () {
80   ANGELIX_EXTRACTED=./extracted instrument-suspicious "$1" -- "-I$LLVM3_DIR/build/lib/clang/3.7.0/include"
81 }
82
83 # environment for SEMFIX
84 export SEMFIX_ROOT="$ANGELIX_ROOT/src/semfix"
85 export PATH="$ANGELIX_ROOT/build/perl/bin${PATH:+:$PATH}"
86 export PERLSLIB="$ANGELIX_ROOT/build/perl/lib/perl5${PERLSLIB:+:$PERLSLIB}"
87 export PERL_LOCAL_LIB_ROOT="$ANGELIX_ROOT/build/perl${PERL_LOCAL_LIB_ROOT:+:$PERL_LOCAL_LIB_ROOT}"
88 export PERL_MB_OPT="--install_base \"$ANGELIX_ROOT/build/perl\""
89 export PERL_MM_OPT="INSTALL_BASE=$ANGELIX_ROOT/build/perl"
```

This block continues the content of the `activate` script from the previous block, showing lines 41 through 89. It includes environment setup for various tools and paths.

```
41
42 export PATH="$LLVM_GCC_DIR/bin:$PATH"
43 export PATH="$LLVM2_DIR/Release+Asserts/bin:$PATH"
44 export PATH="$KLEE_DIR/Release+Asserts/bin:$PATH"
45 export PATH="$Z3_DIR/build:$PATH"
46 export PATH="$Z3_2_19_DIR/bin:$PATH"
47 export PATH="$ANGELIX_ROOT/src/tools:$PATH"
48 export PATH="$ANGELIX_ROOT/build/bin:$PATH"
49
50 export ANGELIX_LIBRARY_PATH_KLEE="$ANGELIX_ROOT/build/lib/klee"
51 export ANGELIX_LIBRARY_PATH_TEST="$ANGELIX_ROOT/build/lib/test"
52 export ANGELIX_RUNTIME_H="$ANGELIX_ROOT/src/runtime/runtime.h"
53
54 # to run tests when linked with runtime:
55 export LD_LIBRARY_PATH="$KLEE_LIBRARY_PATH:$LD_LIBRARY_PATH"
56 # because I didn't manage to build stp statically:
57 export LD_LIBRARY_PATH="$STP_DIR/build/lib:$LD_LIBRARY_PATH"
58
59 # for Z3 bindings:
60 export LD_LIBRARY_PATH="$Z3_DIR/build:$LD_LIBRARY_PATH"
61 export PYTHONPATH="$Z3_DIR/build:$PYTHONPATH"
62
63 ulimit -s unlimited
64
65 mkdir -p $ANGELIX_ROOT/build
66 mkdir -p $ANGELIX_ROOT/build/bin
67 mkdir -p $ANGELIX_ROOT/build/include
68
69 # utilities for testing:
70
71 dump-ast () {
72   "$LLVM3_DIR/build/bin/clang" -Xclang -ast-dump -fdiagnostics-color=never -fsyntax-only "$1"
73 }
74
75 instr-rep () {
76   instrument-repairable "$1" -- "-I$LLVM3_DIR/build/lib/clang/3.7.0/include" -include "$ANGELIX_RUNTIME_H" -D ANGELIX_INSTRUMENTA
77 }
78
79 instr-sus () {
80   ANGELIX_EXTRACTED=./extracted instrument-suspicious "$1" -- "-I$LLVM3_DIR/build/lib/clang/3.7.0/include"
81 }
82
83 # environment for SEMFIX
84 export SEMFIX_ROOT="$ANGELIX_ROOT/src/semfix"
85 export PATH="$ANGELIX_ROOT/build/perl/bin${PATH:+:$PATH}"
86 export PERLSLIB="$ANGELIX_ROOT/build/perl/lib/perl5${PERLSLIB:+:$PERLSLIB}"
87 export PERL_LOCAL_LIB_ROOT="$ANGELIX_ROOT/build/perl${PERL_LOCAL_LIB_ROOT:+:$PERL_LOCAL_LIB_ROOT}"
88 export PERL_MB_OPT="--install_base \"$ANGELIX_ROOT/build/perl\""
89 export PERL_MM_OPT="INSTALL_BASE=$ANGELIX_ROOT/build/perl"
```

export PATH by providing a profile file
<https://github.com/mchtaev/angelix/blob/master/activate>

workflow of the program



workflow

Limits

- Generalization of the fix
 - rely on the test suite
- Other statistical debugging metrics

$$susp(s) = \frac{failed(s)/total\ failed}{passed(s)/total\ passed + failed(s)/total\ failed}$$

-

A statement exercised by more failing tests and fewer passing tests will have a higher suspiciousness score.

- Other

Example#3 semfix inc

```
// Change from the original semfix inc
// https://github.com/mechtaev/angelix/blob/master/tests/semfix/src/test.c
```

```
#include <stdio.h>

#ifdef ANGELIX_OUTPUT
#define ANGELIX_OUTPUT(type, expr, id) expr
#endif

int inc(int i) {
    return i - 1; // +
}

int main(int argc, char *argv[]) {
    int x, n;
    x = atoi(argv[1]);
    n = inc(x);
    printf("%d\n", ANGELIX_OUTPUT(int, n, "n"));
    return 0;
}
```

```
TC#1: (1;2)
TC#2: (2;3)
TC#3: (3;4)
```

```
angelix src test.c oracle 1 2 3 --assert assert.json --semfix --synthesis-level variables
```

```
--- a/test.c
+++ b/test.c
@@ -11,7 +11,7 @@
 int main(int argc, char *argv[]) {
     int x, n;
     x = atoi(argv[1]);
-    n = inc(x);
```

```
+  n = (1 + x);  
    printf("%d\n", ANGELIX_OUTPUT(int, n, "n"));  
    return 0;  
}
```

Example#4 for-loop (I)

```
// The original for-loop
// https://github.com/mechtaev/angelix/blob/master/tests/for-loop/src/test.c

#include <stdio.h>

#ifdef ANGELIX_OUTPUT
#define ANGELIX_OUTPUT(type, expr, id) expr
#endif

int main(int argc, char *argv[]) {
    int n;
    n = atoi(argv[1]);
    for (n = n - 1; n > 0; n--) { // >=
        printf("%d\n", ANGELIX_OUTPUT(int, n, "n"));
    }
    return 0;
}
```

Test cases:

```
TC#1: (2;[1, 0])
TC#2: (3;[2, 1, 0])
TC#3: (4;[3, 2, 1, 0])
```

```
angelix src test.c oracle 1 2 3 --assert assert.json --klee-max-forks 5 --defect loop-conditions
```

```
--- a/test.c
+++ b/test.c
@@ -7,7 +7,7 @@
 int main(int argc, char *argv[]) {
     int n;
     n = atoi(argv[1]);
-    for (n = n - 1; n > 0; n--) { // >=
```

```
+   for (n = n - 1; (n >= 0); n--) { // >=

        printf("%d\n", ANGELIX_OUTPUT(int, n, "n"));
    }
    return 0;
```


Example#4 for-loop (II)

However, if we change the start point

```
#include <stdio.h>

#ifdef ANGELIX_OUTPUT
#define ANGELIX_OUTPUT(type, expr, id) expr
#endif


int main(int argc, char *argv[]) {
    int n;
    n = atoi(argv[1]);
    for (n = n - 2; n >= 0; n--) { // >=
        printf("%d\n", ANGELIX_OUTPUT(int, n, "n"));
    }
    return 0;
}
```

Same oracle, assert.json and command as before, but

```
[...]
INFO      inference      found 0 angelic paths for test '1'
INFO      repair        no patch generated in 8s
FAIL
```

What's Next?

There are many other research topics and tools for automated program repair, if you are interested and want to have a try, please refer to program-repair.org for more info.




Program repair


Community-driven effort to facilitate discovery, access and systematization of data related to automated program repair research


[Contribute](#)[Mailing list](#)

What one would like ideally [...] is the automatic detection and correction of bugs
— R. J. Abbot, 1990

Resources
Up-to-date references to publicly available resources on automated program repair, related fields, and commercial applications

Bibliography

Tools

Benchmarks

HomeBibliographyToolsBenchmarksAbout

DeepFix

DeepFix: A Deep Learning-based Program Repair Framework

DeepFix: A Deep Learning-based Program Repair Framework

ACS

Automatic Code Synthesis for Program Repair

Automatic Code Synthesis for Program Repair

CoderAssist

CoderAssist: A Deep Learning-based Program Repair Framework

CoderAssist: A Deep Learning-based Program Repair Framework

Astor

Astor: A Deep Learning-based Program Repair Framework

Astor: A Deep Learning-based Program Repair Framework

HistoricalFix

HistoricalFix: A Deep Learning-based Program Repair Framework

HistoricalFix: A Deep Learning-based Program Repair Framework

Angelix

Angelix: A Deep Learning-based Program Repair Framework

Angelix: A Deep Learning-based Program Repair Framework

Prophet

Prophet: A Deep Learning-based Program Repair Framework

Prophet: A Deep Learning-based Program Repair Framework

SearchRepair

SearchRepair: A Deep Learning-based Program Repair Framework

SearchRepair: A Deep Learning-based Program Repair Framework

LeakFix

LeakFix: A Deep Learning-based Program Repair Framework

LeakFix: A Deep Learning-based Program Repair Framework

QACrashFix

QACrashFix: A Deep Learning-based Program Repair Framework

QACrashFix: A Deep Learning-based Program Repair Framework

That's all, thanks!

Good luck & have fun

Team member:

2016218037 刘莞姝

2016218040 马舒婕

2016218041 缪东旭

2016218046 滕 飞

2016218055 杨宇杰