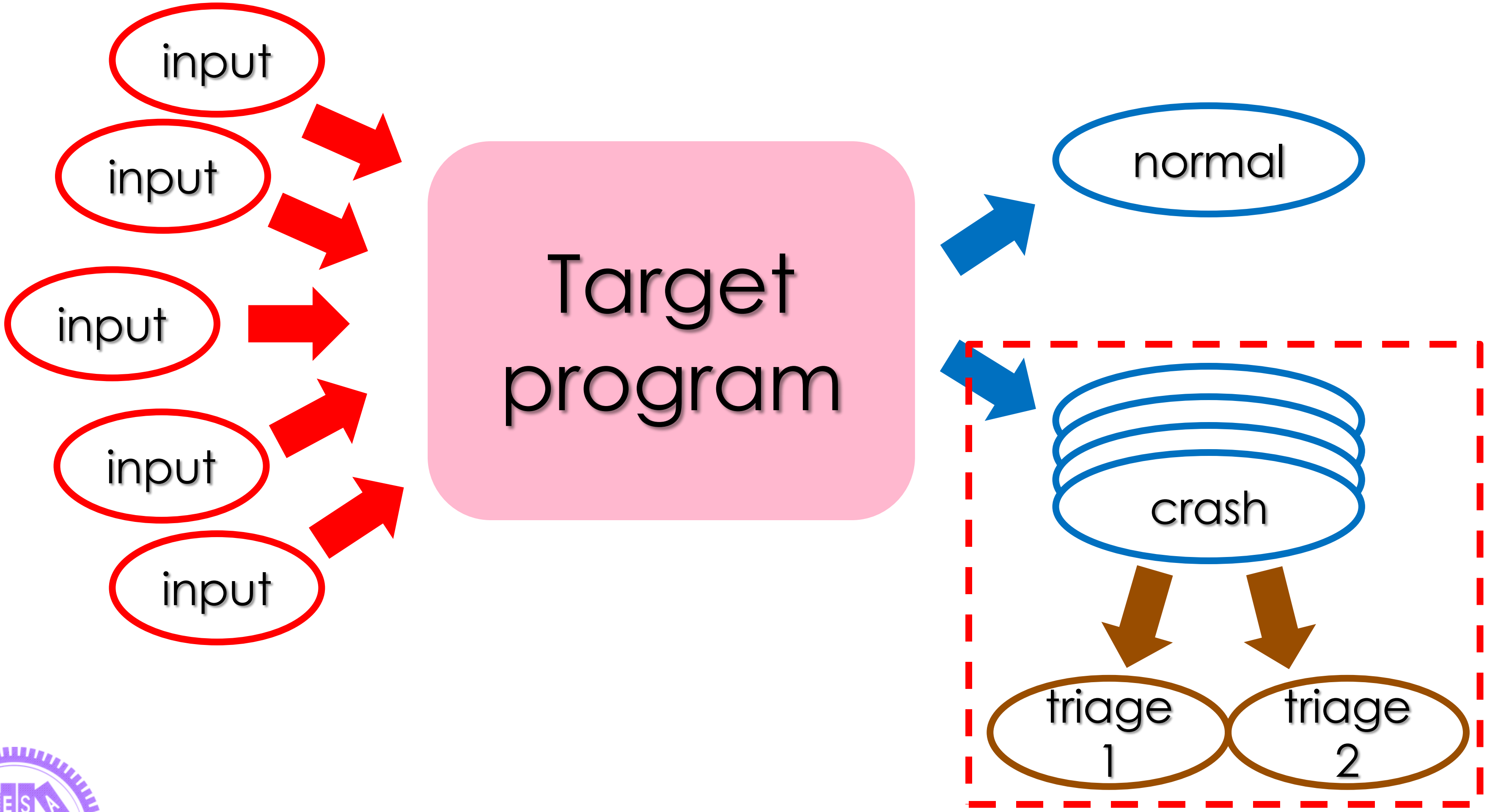
運用程式碼覆蓋範圍分類程式失誤狀況
Using Code Coverage as a Triage Method

Student: Wei-Sheng Chou (周瑋勝)
Advisor: Shih-Kun Huang (黃世昆)
NCTU SQLAB 2015-05-29

Problem Description



input

input

input

input

input

Target program

normal

crash

triage 1

triage 2

Outline

- Motivation
- Background
  - Failure Program
  - Crash Data
  - Fault and Triage
  - Fault localization
- Related Work
  - Fuzzing Tool
  - Stack Trace Triage Method
  - Flaw of Stack Trace Triage Method
- Method
  - Algorithm
  - Case Consideration
  - Research Question
- Results and Evaluation
  - System Architecture
  - Real Program
  - Method Comparison
- Conclusion and Future Work

Outline

- **Motivation**
- Background
  - Failure Program
  - Crash Data
  - Fault and Triage
  - Fault localization
- Related Work
  - Fuzzing Tool
  - Stack Trace Triage Method
  - Flaw of Stack Trace Triage Method
- Method
  - Algorithm
  - Case Consideration
  - Research Question
- Results and Evaluation
  - System Architecture
  - Real Program
  - Method Comparison
- Conclusion and Future Work

Motivation

- Software scalability and functionality is booming
  - Need: High software quality

- Human debugging is ineffective
  - Need: Automated debugging techniques and tools

- Traditional fault triage methods are not accurate
  - Too many triages / Wrong triages
  - Need: A new method

Outline

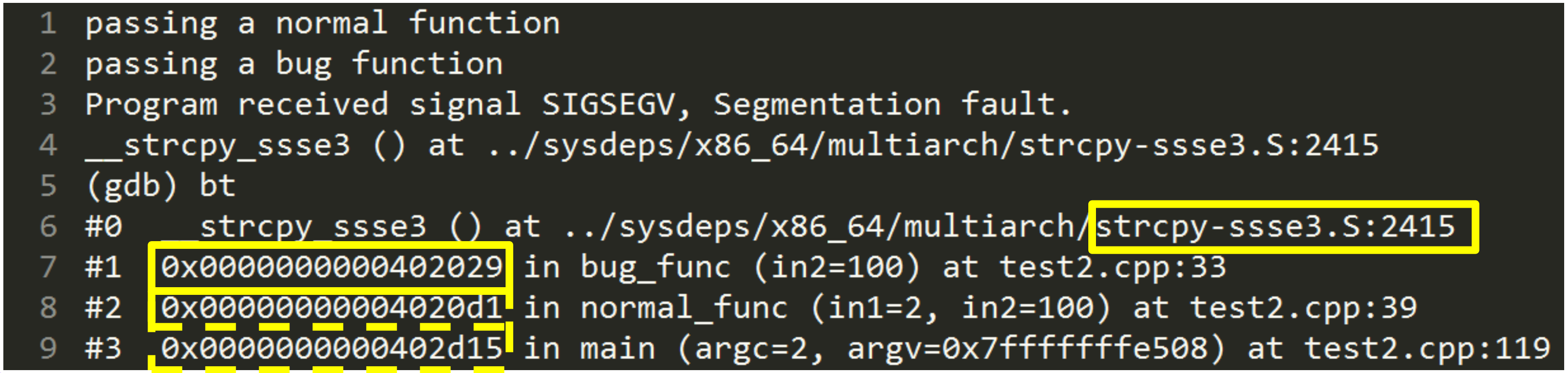
- Motivation
- **Background**
  - **Failure Program**
  - **Crash Data**
  - **Fault and Triage**
  - **Fault localization**
- Related Work
  - Fuzzing Tool
  - Stack Trace Triage Method
  - Flaw of Stack Trace Triage Method
- Method
  - Algorithm
  - Case Consideration
  - Research Question
- Results and Evaluation
  - System Architecture
  - Real Program
  - Method Comparison
- Conclusion and Future Work

Failure Program

- Normal program
  - Execute -> invoke exit()
- Failure program
  - Execute -> send abnormal signal to OS
    - Segmentation fault, Abort…etc
    - Signal can be caught by exception handler
- Why?
  - Human interrupt
  - Wrong OS resource deployment
  - Error manipulation on memory ⟵ We focus on this

Crash Data

- Collected by tools (GDB, Valgrind…etc)
  1. Points of failure
  2. Stack trace
  3. Call sequence
  4. Full executed record

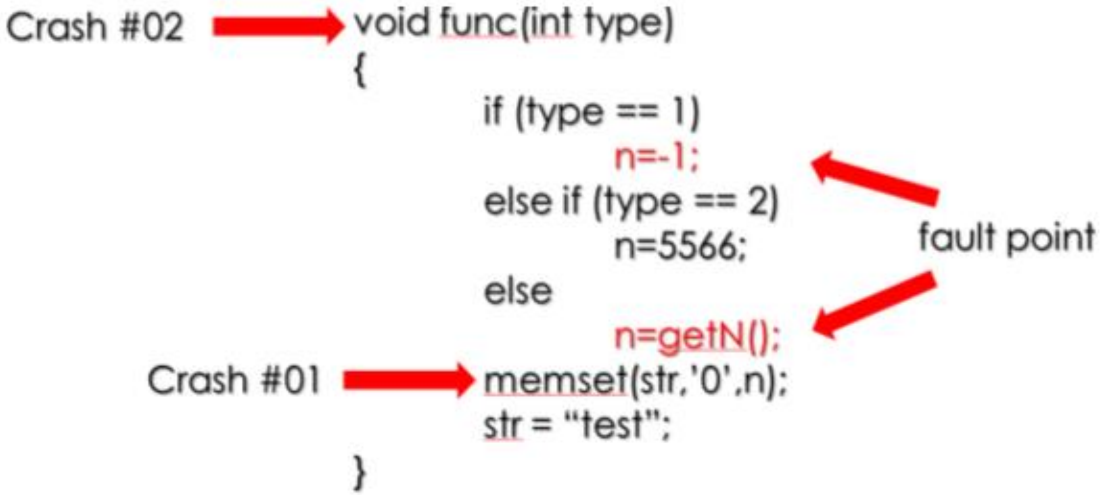


```
passing a normal function
passing a bug function
Program received signal SIGSEGV, Segmentation fault.
__strcpy_ssse3 () at ../sysdeps/x86_64/multiarch/strcpy-ssse3.S:2415
(gdb) bt
#0  __strcpy_ssse3 () at ../sysdeps/x86_64/multiarch/strcpy-ssse3.S:2415
#1  0x0000000000402029 in bug_func (in2=100) at test2.cpp:33
#2  0x00000000004020d1 in normal_func (in1=2, in2=100) at test2.cpp:39
#3  0x0000000000402d15 in main (argc=2, argv=0x7fffffffe508) at test2.cpp:119
```

Program’s backtrace collected by GDB

Fault / Crash(Failure)

- Crash point (Failure point):
  - Where does the program crash?
- Fault point:
  - What causes that program to crash at that point?
- Crash is not usually the same as Fault



```
void func(int type)
{
        if (type == 1)
                n=-1;
        else if (type == 2)
                n=5566;
        else
                n=getN();
        memset(str,'0',n);
        str = "test";
}
```

Difference between fault and crash

Fault Triage

- Fault triage is a technique to classify the input of failure program
- How?
    - Traditional method: Based on stack trace
    - Our new method: Based on code coverage (inspired by fault localization methods)

Fault localization

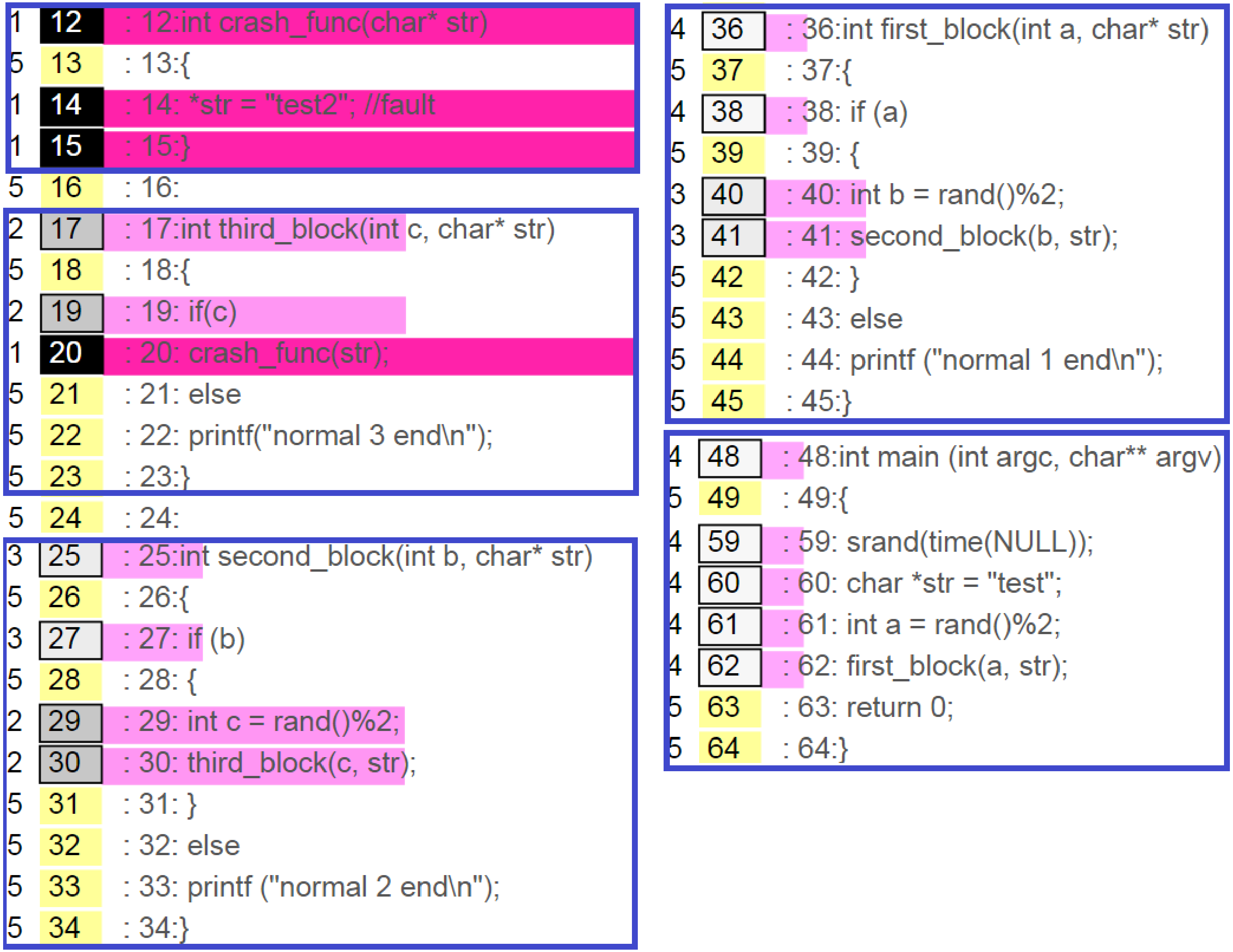
- Fault localization is a technique to locate the possible fault point of failure program

- How?
  - A huge dataset (statistical method)
  - A suspiciousness rank list
  - Using "branch" to evaluate

Fault localization - DStar

- DStar is a coefficient-based fault localization method
- Parameter:
    - Covered Success
    - Uncovered Success
    - Covered Failure
    - Uncovered Failure
    - A weighted star

$$Suspiciousness\ Value = \frac{Covered\ Failure^{*}}{Uncovered\ Failure + Covered\ Success}$$



```
1 12 : 12:int crash_func(char* str)
5 13 : 13:{
1 14 : 14: *str = "test2"; //fault
1 15 : 15:}
5 16 : 16:
2 17 : 17:int third_block(int c, char* str)
5 18 : 18:{
2 19 : 19: if(c)
1 20 : 20: crash_func(str);
5 21 : 21: else
5 22 : 22: printf("normal 3 end\n");
5 23 : 23:}
5 24 : 24:
3 25 : 25:int second_block(int b, char* str)
5 26 : 26:{
3 27 : 27: if (b)
5 28 : 28: {
2 29 : 29: int c = rand()%2;
2 30 : 30: third_block(c, str);
5 31 : 31: }
5 32 : 32: else
5 33 : 33: printf ("normal 2 end\n");
5 34 : 34:}
```

```
4 36 : 36:int first_block(int a, char* str)
5 37 : 37:{
4 38 : 38: if (a)
5 39 : 39: {
3 40 : 40: int b = rand()%2;
3 41 : 41: second_block(b, str);
5 42 : 42: }
5 43 : 43: else
5 44 : 44: printf ("normal 1 end\n");
5 45 : 45:}
4 48 : 48:int main (int argc, char** argv)
5 49 : 49:{
4 59 : 59: srand(time(NULL));
4 60 : 60: char *str = "test";
4 61 : 61: int a = rand()%2;
4 62 : 62: first_block(a, str);
5 63 : 63: return 0;
5 64 : 64:}
```

Implementation by D3JS

http://people.cs.nctu.edu.tw/~wschou/d3js/dataset/loadjson.html

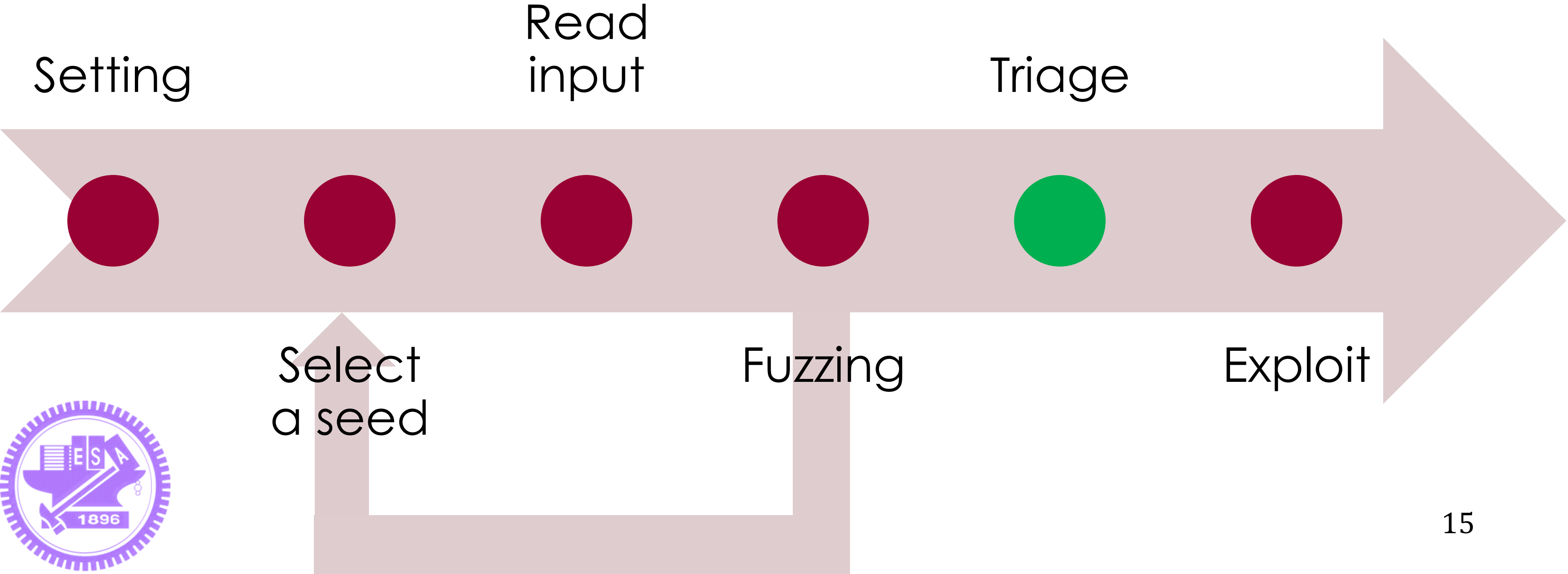
Outline

- Motivation
- Background
  - Failure Program
  - Crash Data
  - Fault and Triage
  - Fault localization
- **Related Work**
  - **Fuzzing Tool**
  - **Stack Trace Triage Method**
  - **Flaw of Stack Trace Triage Method**
- Method
  - Algorithm
  - Case Consideration
  - Research Question
- Results and Evaluation
  - System Architecture
  - Real Program
  - Method Comparison
- Conclusion and Future Work

Fuzzing Tool

- Fuzzing tool is using for finding exploitable possibilities of target programs
- **Triage** is an important phase of fuzzing

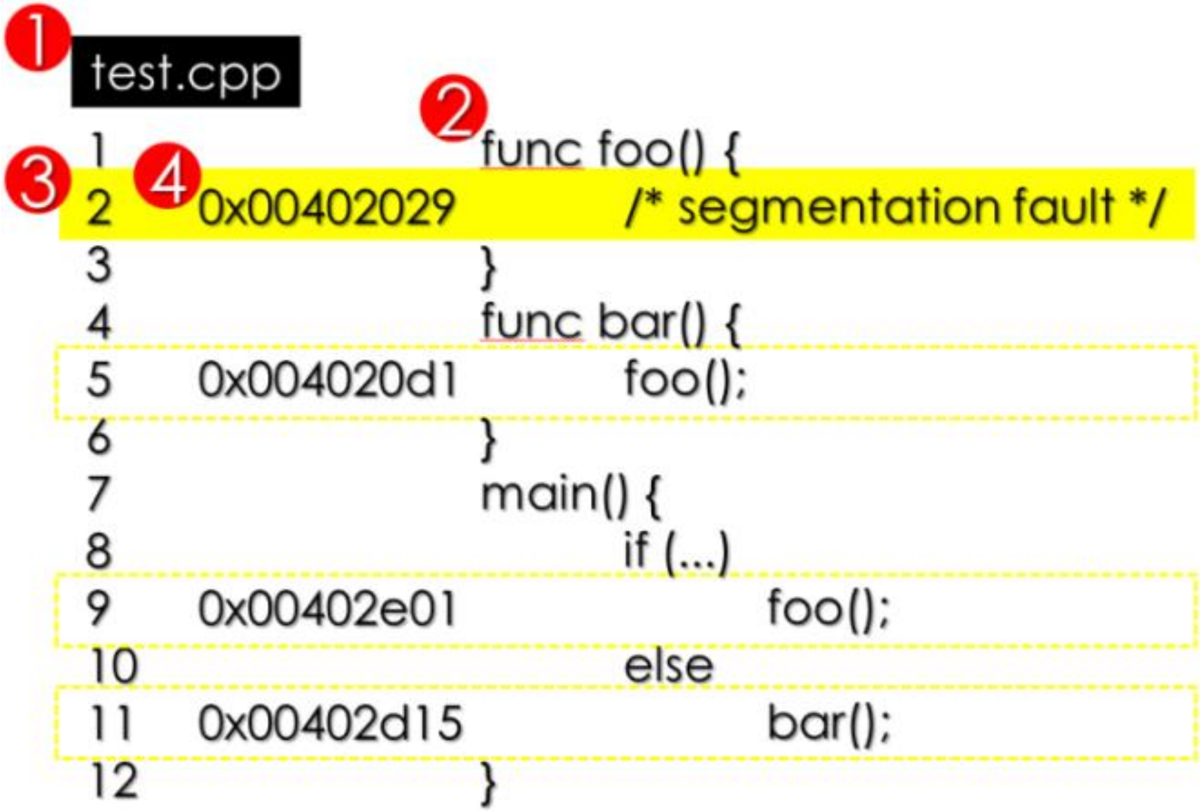
Steps:



Fuzzing Tool (cont.)

	Fuzzer	Triage Method
BFF	Zuff	Stack trace (5)
FOE	Zuff	Stack trace (5)
COVERSET	Zuff	Stack trace (5)
Catchconv	Smartfuzz	Stack trace (3)
Microsoft VPM	unknown	Stack trace (1 + neighbor(n))
Our method	Zuff	**Code coverage**

Stack Trace Triage Method

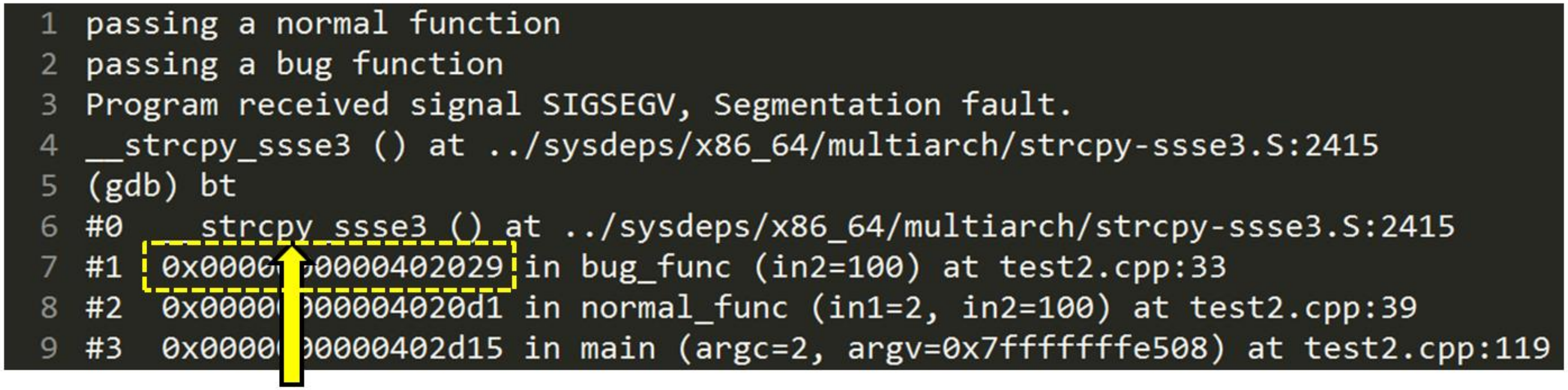


```
test.cpp
1                       func foo() {
2   0x00402029              /* segmentation fault */
3                       }
4                       func bar() {
5   0x004020d1              foo();
6                       }
7                       main() {
8                           if (...)
9   0x00402e01                  foo();
10                          else
11  0x00402d15                  bar();
12                      }
```

hash value =
    Hash(1 filename, 2 function_name,
        3 crash_point_line_number, 4 backtrace)

Stack Trace Triage Method (cont.)

- Real tool: “Observing more than one backtrace”



```
passing a normal function
passing a bug function
Program received signal SIGSEGV, Segmentation fault.
__strcpy_ssse3 () at ../sysdeps/x86_64/multiarch/strcpy-ssse3.S:2415
(gdb) bt
#0  __strcpy_ssse3 () at ../sysdeps/x86_64/multiarch/strcpy-ssse3.S:2415
#1  0x0000000000402029 in bug_func (in2=100) at test2.cpp:33
#2  0x00000000004020d1 in normal_func (in1=2, in2=100) at test2.cpp:39
#3  0x0000000000402d15 in main (argc=2, argv=0x7fffffffe508) at test2.cpp:119
```

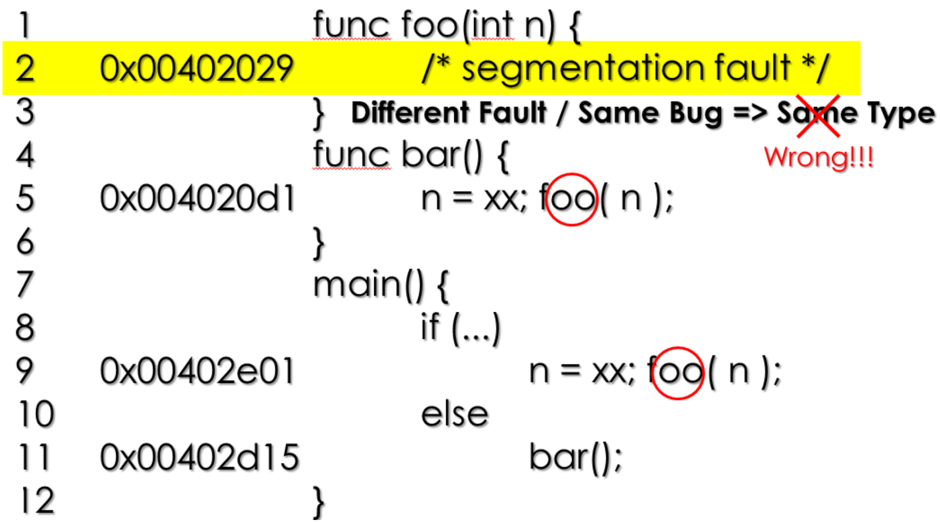
Hash=((test2.cpp, bug_func, 33, 0x0….402029)

(test2.cpp, normal_func, 39, 0x0….4020d1)

(test2.cpp, main, 119, 0x0….402d15))

Stack Trace Triage Method (cont.)

- Why not observe only one backtrace?
  - Triage wrong



```
                 func foo(int n) {
0x00402029           /* segmentation fault */
                 }
                 func bar() {
0x004020d1           n = xx; foo( n );
                 }
                 main() {
                     if (...)
0x00402e01               n = xx; foo( n );
                     else
0x00402d15               bar();
                 }
```

Different Fault / Same Bug => ~~Same~~ Type

Wrong!!!

- Different faults
  - main() => **alter n** => foo() => failure
  - main() => bar() => **alter n** => foo() => failure

Observing enough backtrace

Observing only one backtrace.

Stack Trace Triage Method (cont.)

- How about observing too many backtrace?
  - main() => a() … z() => **alter n** =>
  - main() => a() … z() => bar() => **alter n** =>

Hard to find Fault point

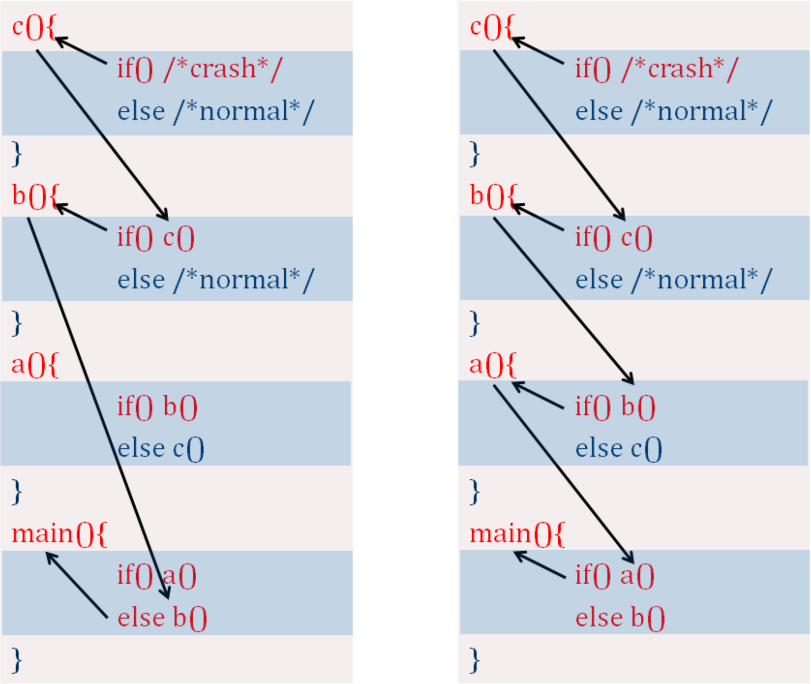
Fault point

  - foo() => failure
  - foo() => failure

Crash point

Flaw of Stack Trace Triage Method

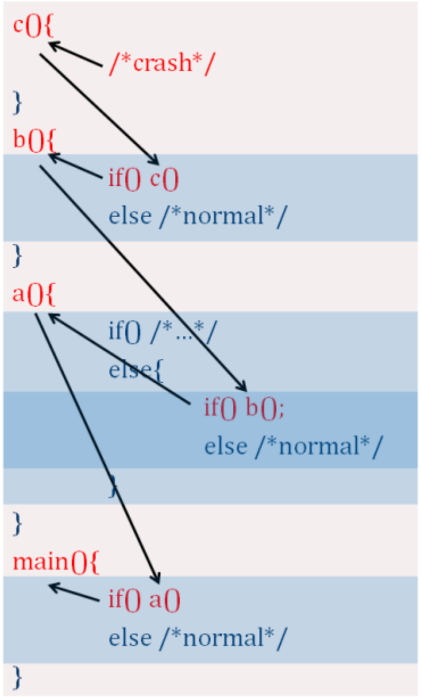
- Q1. Evaluating by basic block, however, unit is stack trace.



```
c(){
        if() /*crash*/
        else /*normal*/
}
b(){
        if() c()
        else /*normal*/
}
a(){
        if() b()
        else c()
}
main(){
        if() a()
        else b()
}
```

```
c(){
        if() /*crash*/
        else /*normal*/
}
b(){
        if() c()
        else /*normal*/
}
a(){
        if() b()
        else c()
}
main(){
        if() a()
        else b()
}
```

Ideal
(one function with one basic block)



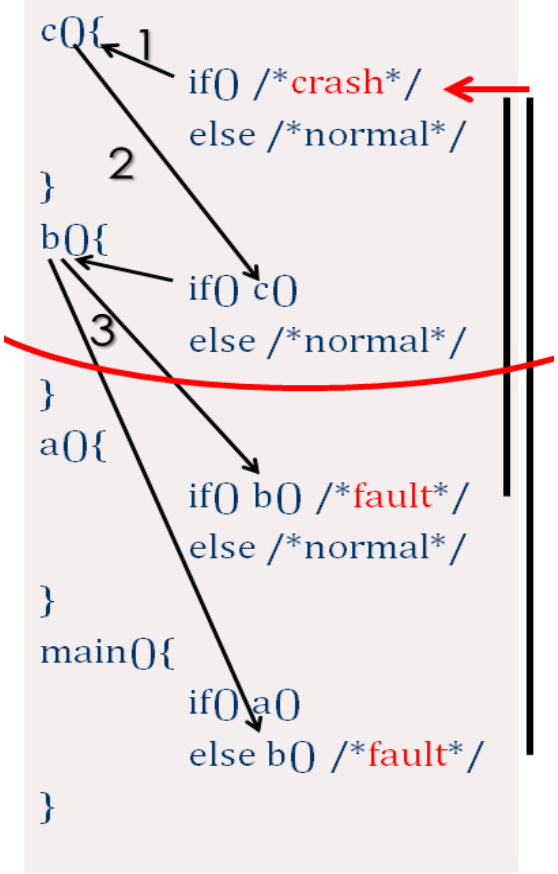
```
c(){
        /*crash*/
}
b(){
        if() c()
        else /*normal*/
}
a(){
        if() /*...*/
        else{
                if() b();
                else /*normal*/
        }
}
main(){
        if() a()
        else /*normal*/
}
```

Real

Flaw of Stack Trace Triage Method

- Q2. Fault point is far away from crash point

- Crash point: c()
- Useless info: b()
- Fault point: a() & main()



```
c(){
        if() /*crash*/
        else /*normal*/
}
b(){
        if() c()
        else /*normal*/
}
a(){
        if() b() /*fault*/
        else /*normal*/
}
main(){
        if() a()
        else b() /*fault*/
}
```

***# of** backtrace should be large **enough**, otherwise getting **wrong** triage

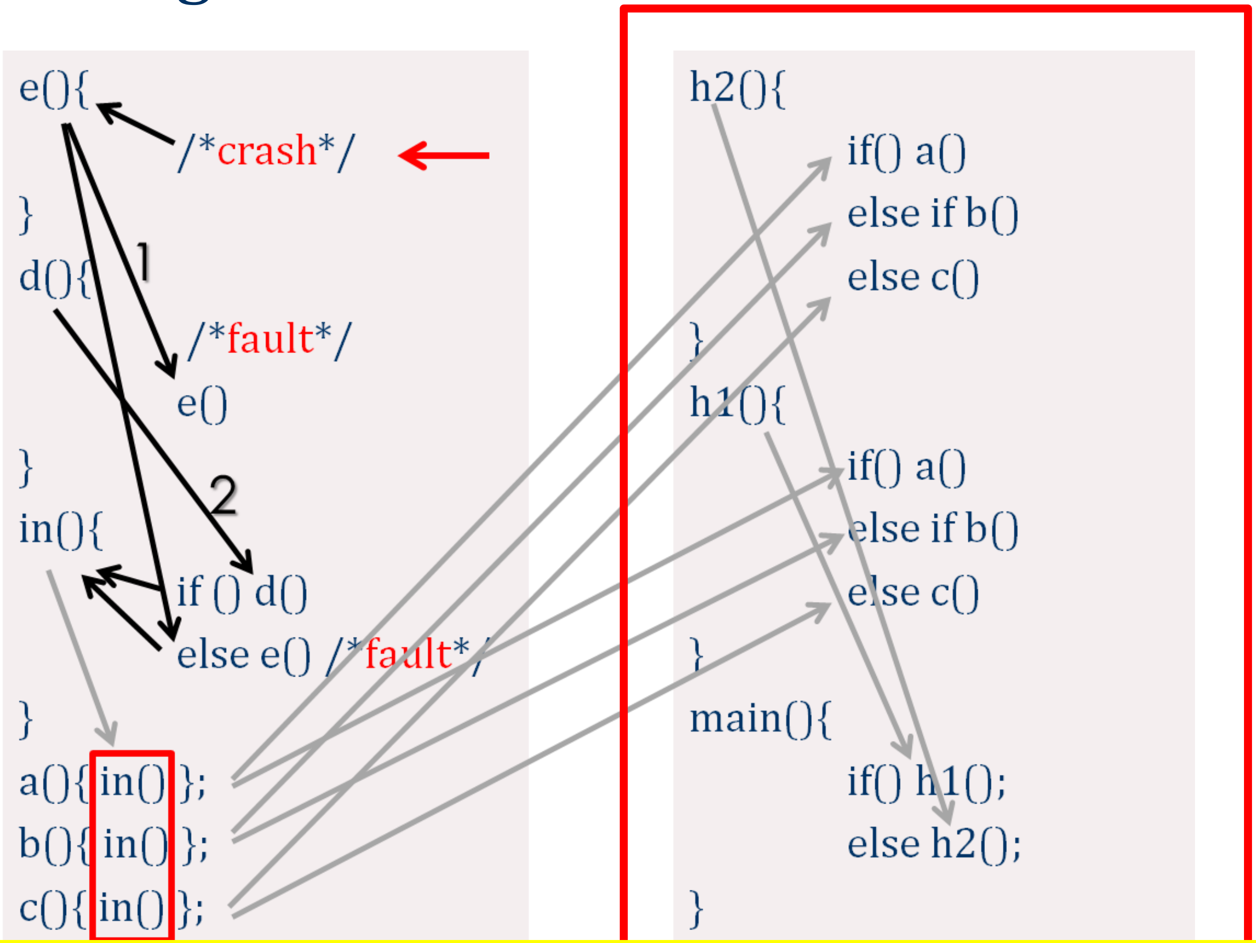
Flaw of Stack Trace Triage Method

- Q3. Over triage

Theory:
2 triages
(1 and 2)

Actual:
Possible
6 triages
(2 * 3)

Have no relationship
with crash point



```
e(){
        /*crash*/
}
d(){
        /*fault*/
        e()
}
in(){
        if () d()
        else e() /*fault*/
}
a(){in()};
b(){in()};
c(){in()};
```

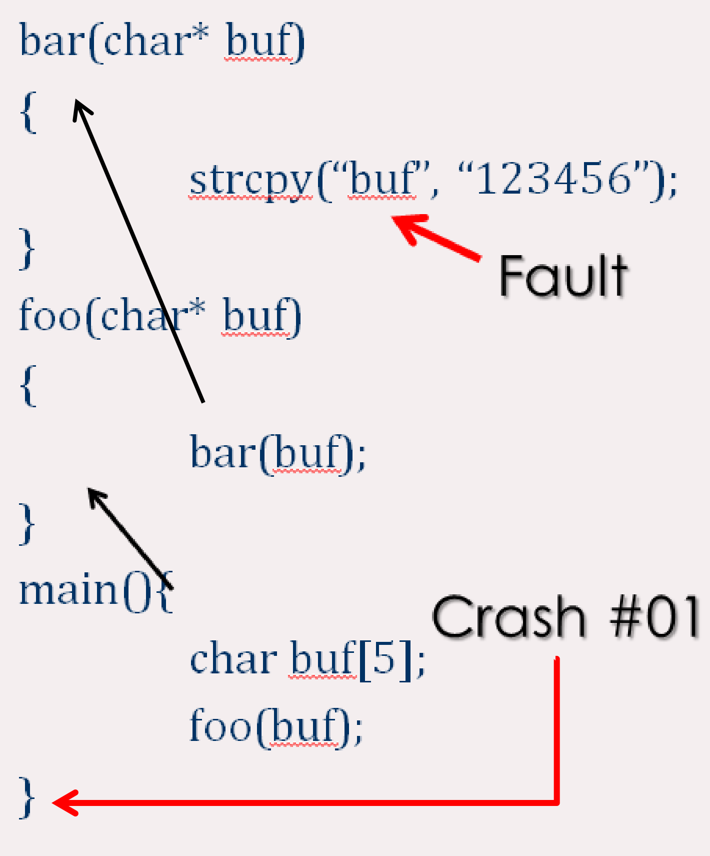
```
h2(){
        if() a()
        else if b()
        else c()
}
h1(){
        if() a()
        else if b()
        else c()
}
main(){
        if() h1();
        else h2();
}
```

*__# of__ backtrace should be __smaller__, otherwise getting too __many__ triages

Flaw of Stack Trace Triage Method

- Q4. Untraceable fault point

- Fault point:
  - strcpy(...)
- Crash point:
  - At the end of main()



```
bar(char* buf)
{
        strcpy("buf", "123456");
}
foo(char* buf)
{
        bar(buf);
}
main(){
        char buf[5];
        foo(buf);
}
```

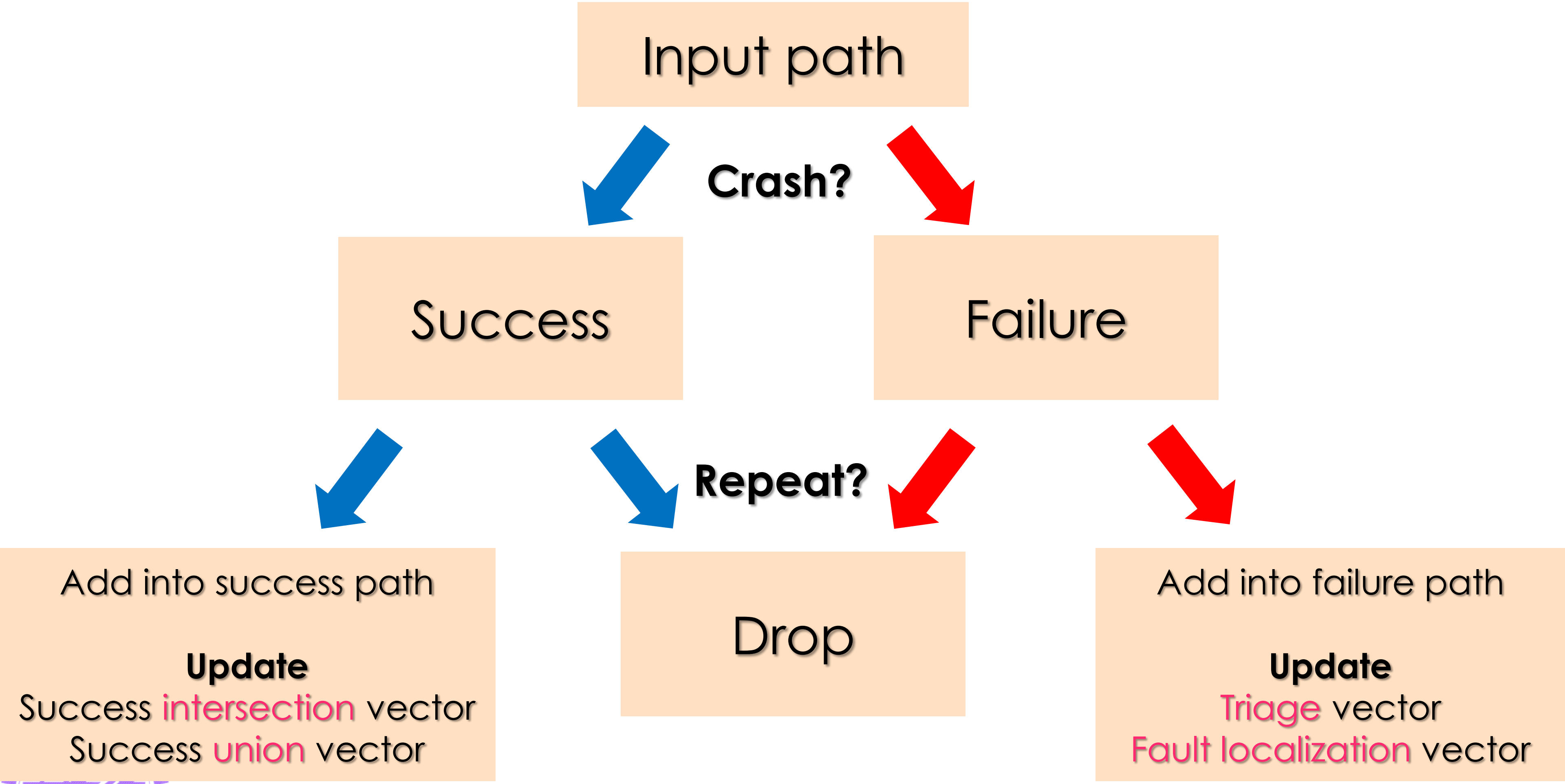
Outline

- Motivation
- Background
  - Failure Program
  - Crash Data
  - Fault and Triage
  - Fault localization
- Related Work
  - Fuzzing Tool
  - Stack Trace Triage Method
  - Flaw of Stack Trace Triage Method
- **Method**
  - **Algorithm**
  - **Case Consideration**
  - **Research Question**
- Results and Evaluation
  - System Architecture
  - Real Program
  - Method Comparison
- Conclusion and Future Work

Research Question

- RQ1. Is Basic Block (BBL) a suitable benchmark for our method?
- RQ2. Can our method resolve the problem of Q2?
- RQ3. Can our method resolve the problem of Q3?
- RQ4. Can our method observe untraceable fault point?

Algorithm Flow Chart



Input path

Crash?

Success

Failure

Repeat?

Add into success path

Update
Success intersection vector
Success union vector

Drop

Add into failure path

Update
Triage vector
Fault localization vector

Algorithm

```
CRAX Triage Algorithm procedure (PATH):
```

Input PATH

```
begin
```

normal

```
do if program_failure_flag = 0
    for i:=0 to SV.size
    do
    if SV[i] == PATH then
        exit
    fi;
    done
```

Path exist in Success Vector

```
    SV.push_back(PATH)
    S = calc1D(SV, INTERSECTION)
    SS = calc2D(SV, UNION)
```

Add Path Calculate S & SS

failure

```
do if program_failure_flag = 1
    for i:=0 to FV.size
    do
    if FV[i] == PATH then
        exit
    fi;
    done
```

Path exist in Failure Vector

```
    FV.push_back(PATH)
    calc2D(PATH,S,FV,TV,INTERSECTION)
    calc2D(PATH,SS,FV,FLV,UNION)
```

Add Path Triage & FL

Algorithm (cont.)

TV = {complements(Failure_input, intersection(SV))}

  - TV is the triage vector / SV is the success vector

- ex:
  - FV {1 2 3 | 12 13 14 15 | 19 20 21 22 23 | 28}
    - FV is the failure vector
  - S { | 12 13 14 15 | 19 20 21 22 | 28}
    - S is the intersection of success vector
  - new TV: {…, {1 2 3 | 23}}
    - {1 2 3 | 23} is one triage result
  - In Line 12~15, Line 19~22 and Line 28
    - When the PATH passing , the program must be success
    - Those lines don't have relationship with fault

Algorithm (cont.)

FLV = {complements(Failure_input, union(SV))}

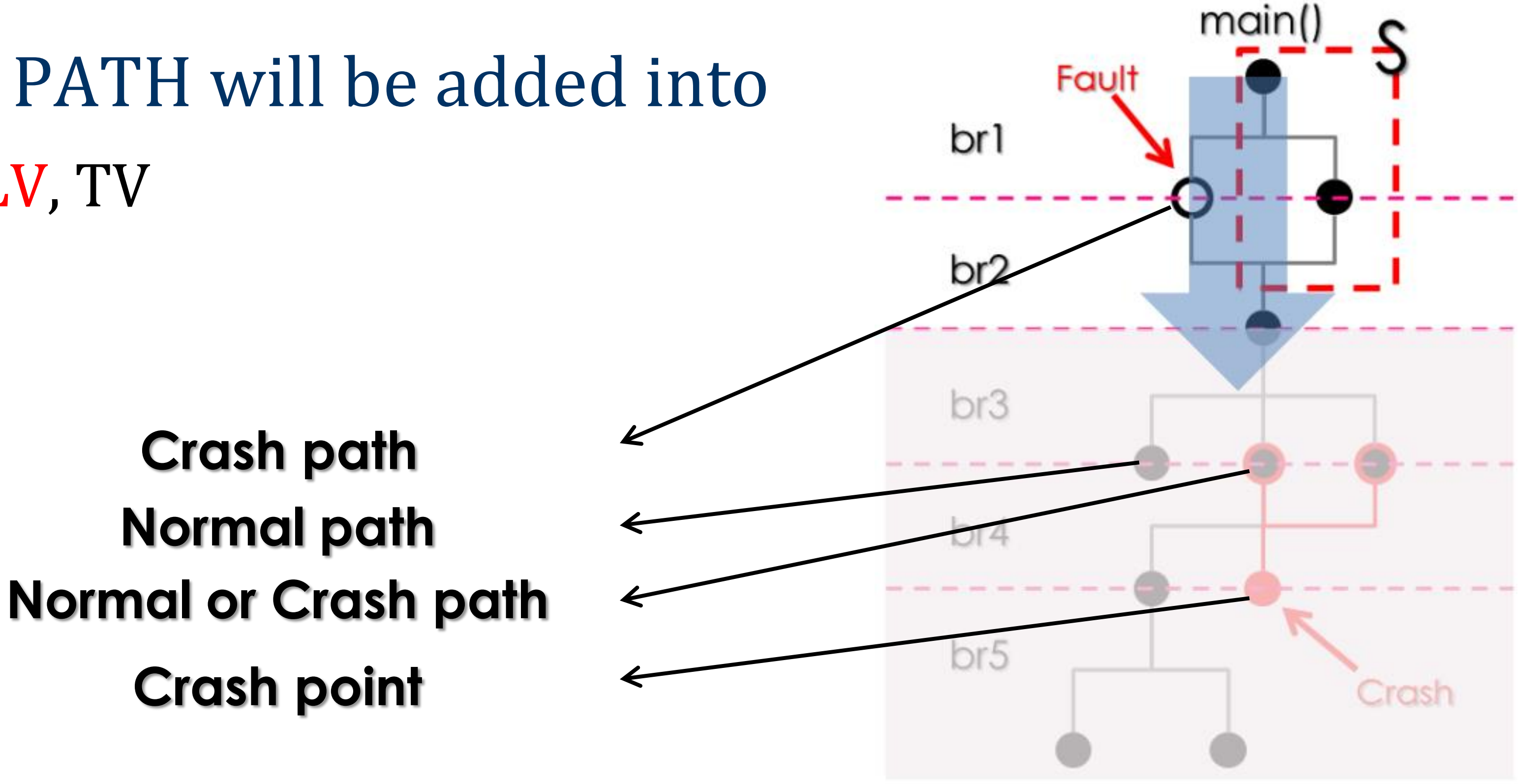
- FLV is the fault localization vector

- ex:
  - FV {1 2 3 | 12 13 14 15 | 19 20 21 22 23 | 28}
  - SS {1 2 3 | 12 13 14 15 | 19 20 21 22   24 25 26 27 28}
    - SS is the union of success vector
  - new FLV: {…, “23”}
  - In Line 23
    - When the PATH passing , the program must be failed
    - That means line:23 suspiciousness will be enhanced

Case Consideration

Case 1. the PATH “only” makes program failed

- This PATH will be added into
  - FLV, TV



Crash path

Normal path

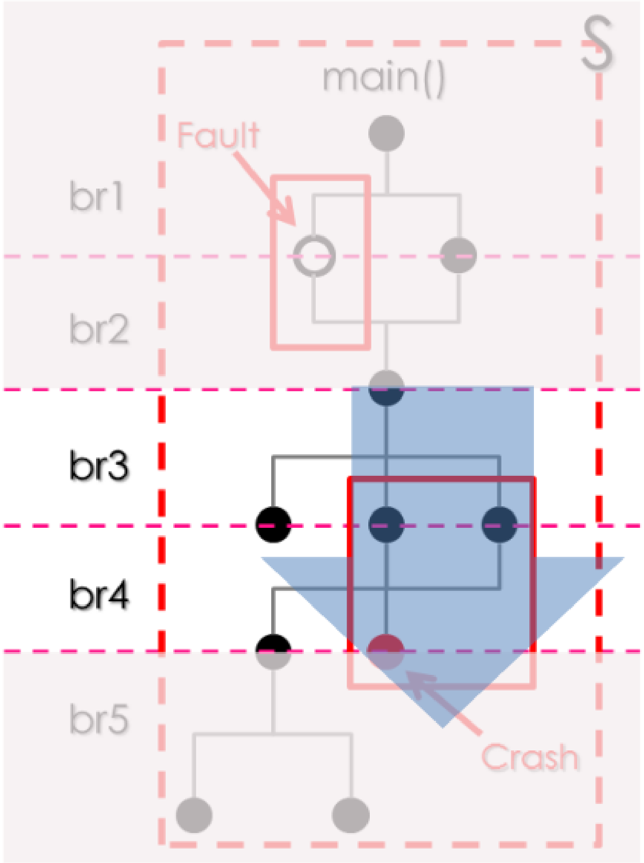
Normal or Crash path

Crash point

Case Consideration

Case 2. the fault point is occurred after br4…

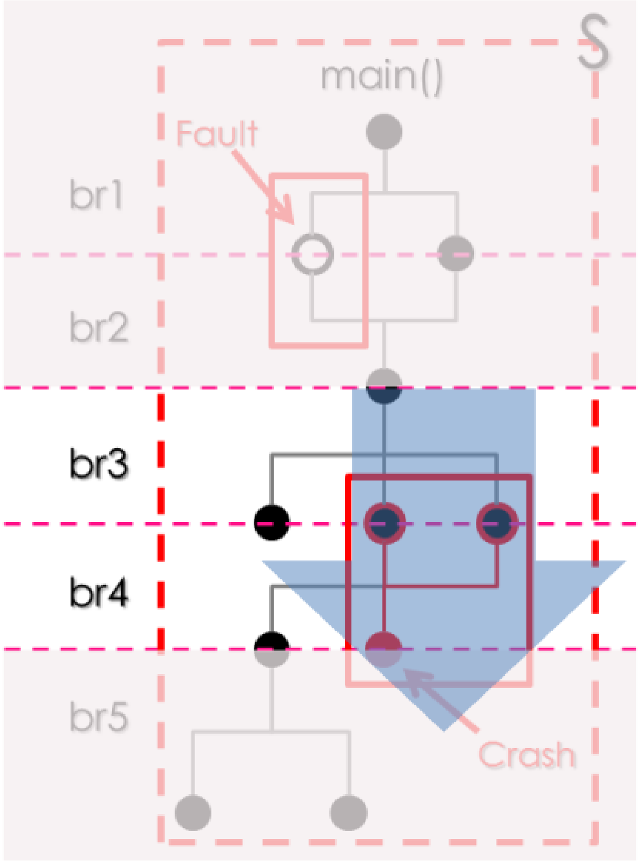
- Wrong triage result
  - The correct triage is only one
  - But two triage results, because…
    - Two PATHs make program crash



Case Consideration

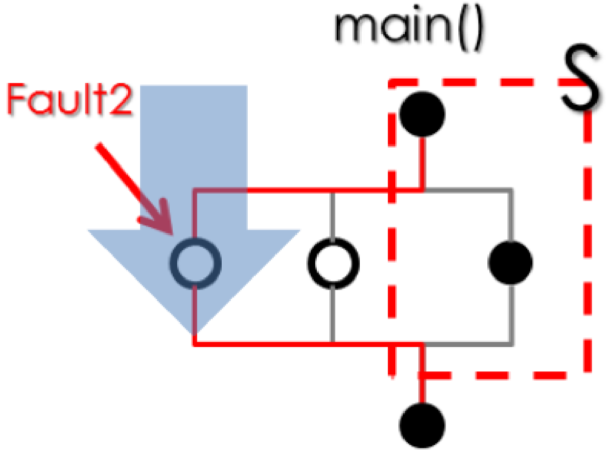
Case 3. the fault point is occurred between br3 and br4…

- Correct triage result
  - The correct triage is two
  - Two PATHs make program crash

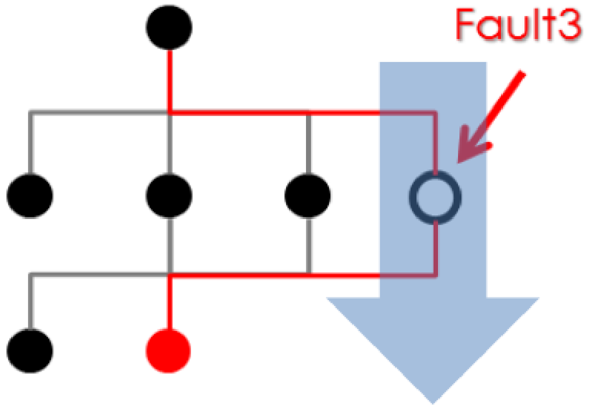


Case Consideration

Case 4. a new faulty PATH is encountered



- We always obtain failing runs
  - This PATH will be added into
    - FLV, TV



- We sometimes obtain failing runs
  - This PATH will be added into
    - TV

Research Question

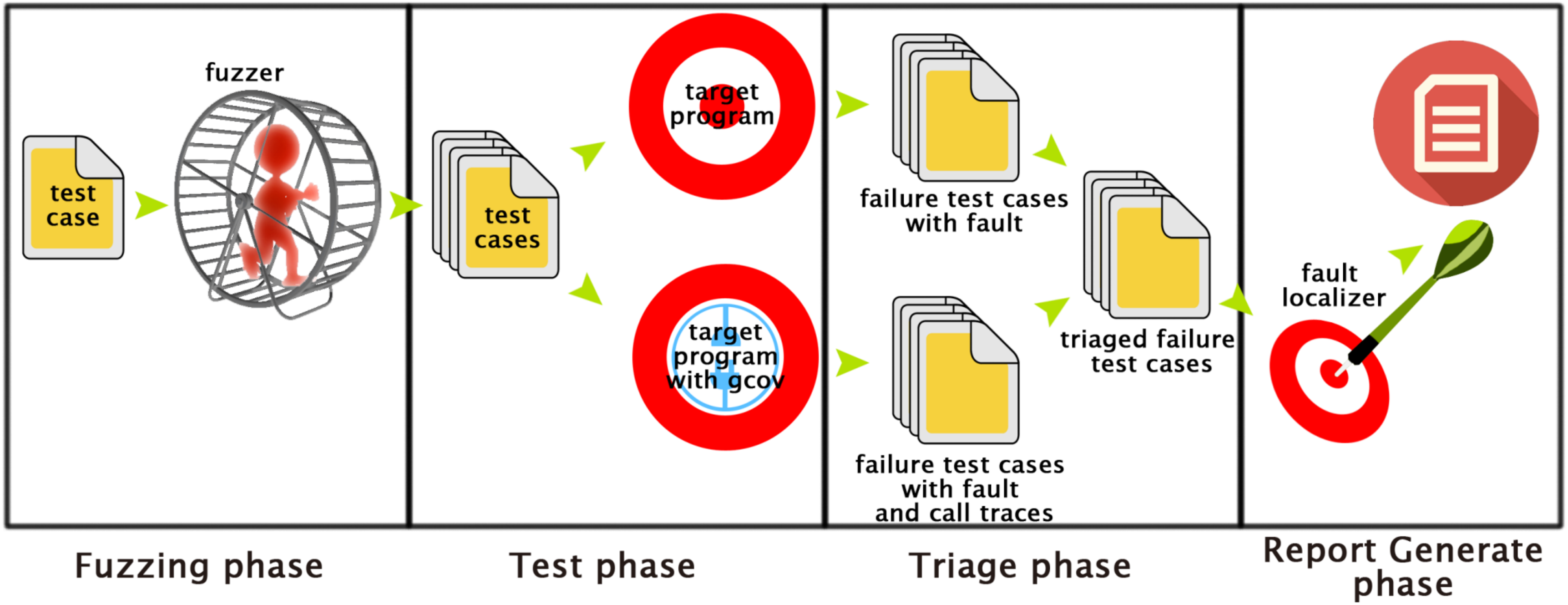
- RQ1. Is BBL a suitable benchmark for our method?
  - Sol1: Yes, the unit of our method is “statement“ , which is smaller than basic block
- RQ2. Can our method resolve the problem of Q2?
  - Sol2: Yes, Sol1 + considering whole code coverage
- RQ3. Can our method resolve the problem of Q3?
  - Sol2: Yes, Sol2 + considering fault relevant code
- RQ4. Can our method observe untraceable fault point?
  - Sol3: Yes, Sol2 + Sol3

Outline

- Motivation
- Background
  - Failure Program
  - Crash Data
  - Fault and Triage
  - Fault localization
- Related Work
  - Fuzzing Tool
  - Stack Trace Triage Method
  - Flaw of Stack Trace Triage Method
- Method
  - Algorithm
  - Case Consideration
  - Research Question
- **Results and Evaluation**
  - **System Architecture**
  - **Real Program**
  - **Method Comparison**
- Conclusion and Future Work

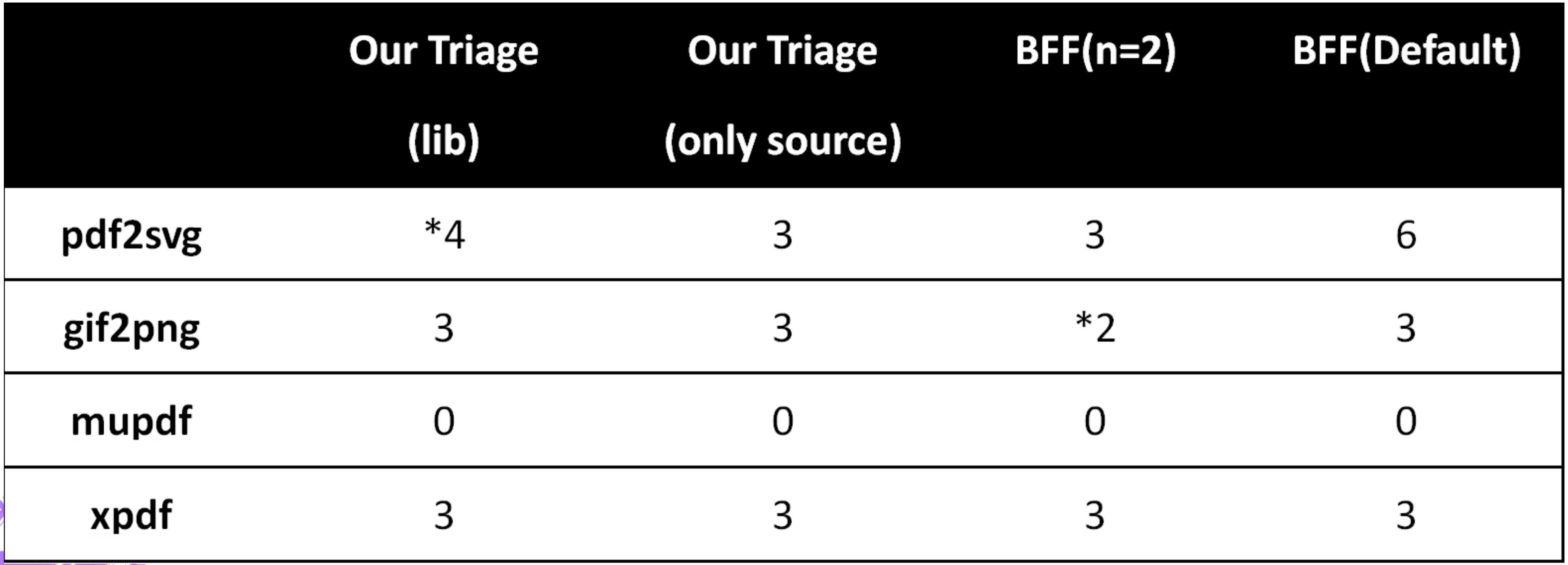
System Architecture

- Ability to integrate into existing tools



Real Case

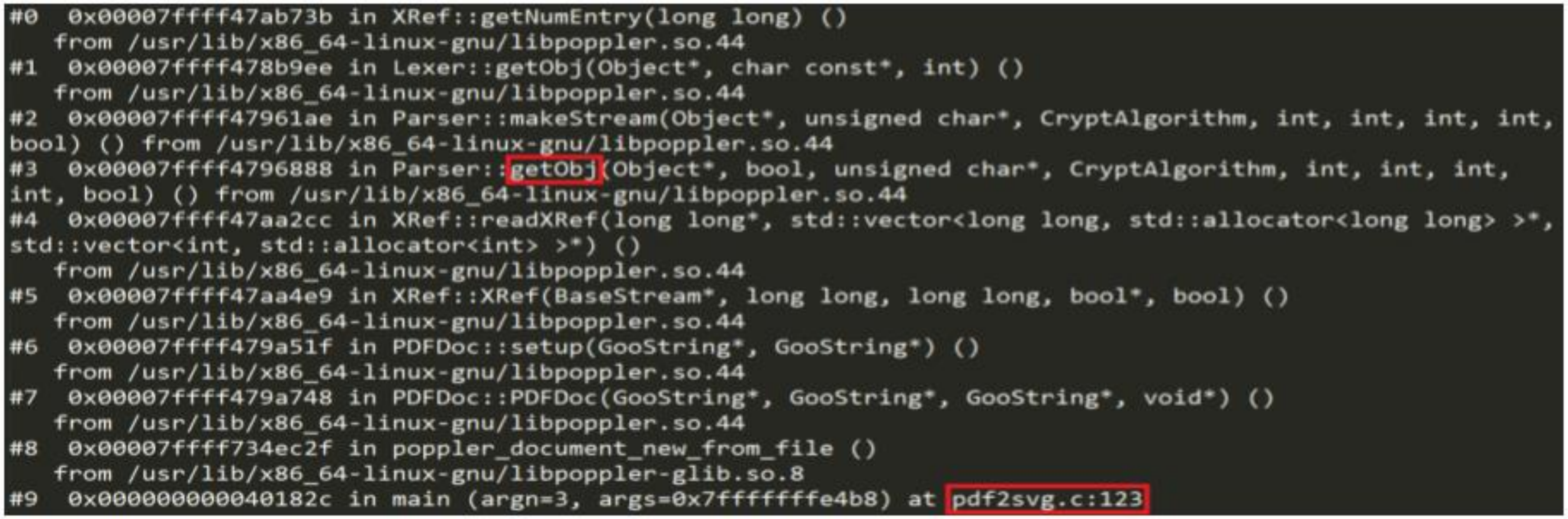
1. Traditional methods may get wrong triage results
2. Our method and traditional method get wrong triage result on special case



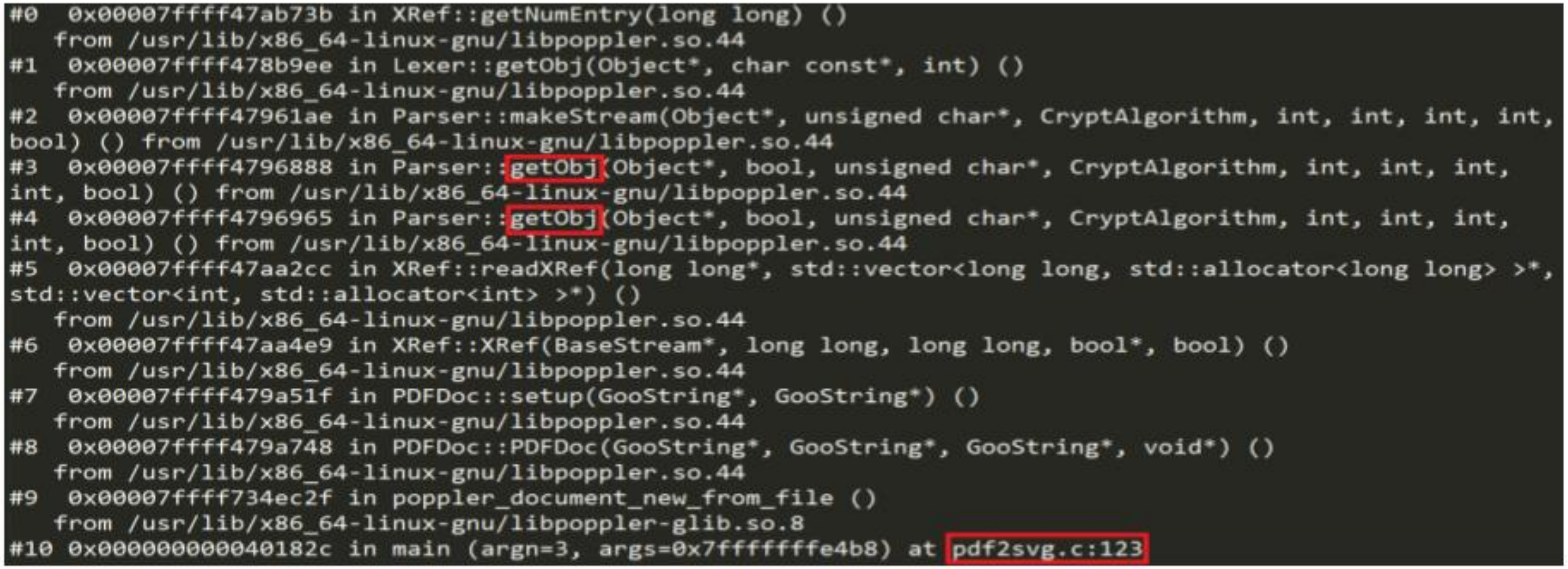
	Our Triage (lib)	Our Triage (only source)	BFF(n=2)	BFF(Default)
pdf2svg	*4	3	3	6
gif2png	3	3	*2	3
mupdf	0	0	0	0
xpdf	3	3	3	3

Real Case (cont.)

traditional method get wrong result



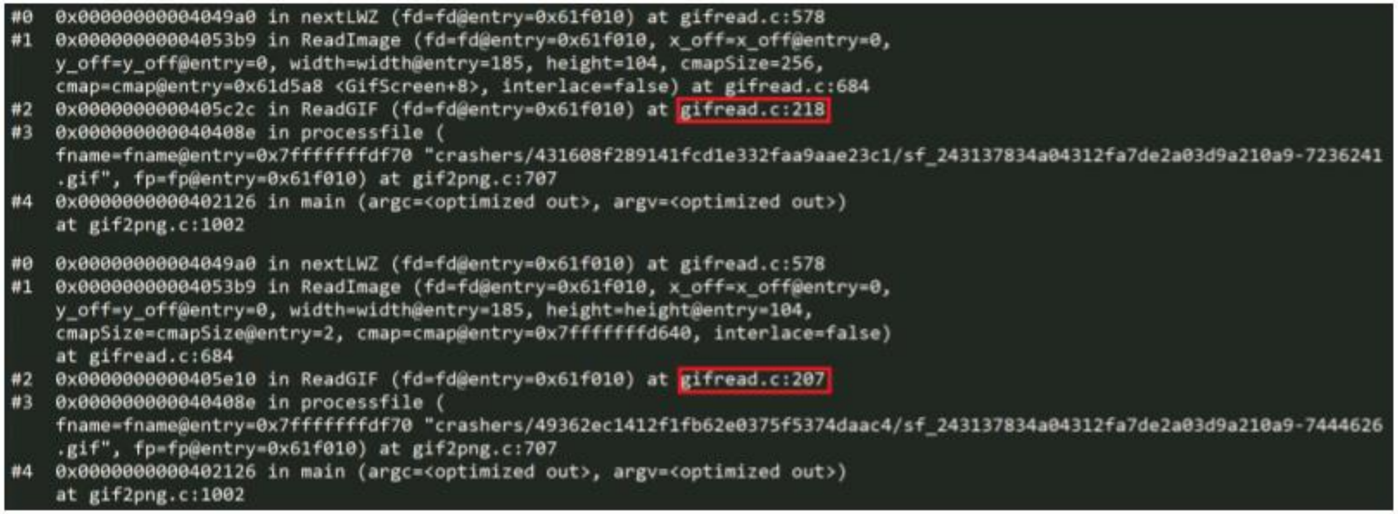
```
#0  0x00007ffff47ab73b in XRef::getNumEntry(long long) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#1  0x00007ffff478b9ee in Lexer::getObj(Object*, char const*, int) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#2  0x00007ffff47961ae in Parser::makeStream(Object*, unsigned char*, CryptAlgorithm, int, int, int, int,
bool) () from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#3  0x00007ffff4796888 in Parser::getObj(Object*, bool, unsigned char*, CryptAlgorithm, int, int, int,
int, bool) () from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#4  0x00007ffff47aa2cc in XRef::readXRef(long long*, std::vector<long long, std::allocator<long long> >*,
std::vector<int, std::allocator<int> >*) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#5  0x00007ffff47aa4e9 in XRef::XRef(BaseStream*, long long, long long, bool*, bool) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#6  0x00007ffff479a51f in PDFDoc::setup(GooString*, GooString*) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#7  0x00007ffff479a748 in PDFDoc::PDFDoc(GooString*, GooString*, GooString*, void*) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#8  0x00007ffff734ec2f in poppler_document_new_from_file ()
   from /usr/lib/x86_64-linux-gnu/libpoppler-glib.so.8
#9  0x000000000040182c in main (argn=3, args=0x7fffffffe4b8) at pdf2svg.c:123
```



```
#0  0x00007ffff47ab73b in XRef::getNumEntry(long long) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#1  0x00007ffff478b9ee in Lexer::getObj(Object*, char const*, int) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#2  0x00007ffff47961ae in Parser::makeStream(Object*, unsigned char*, CryptAlgorithm, int, int, int, int,
bool) () from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#3  0x00007ffff4796888 in Parser::getObj(Object*, bool, unsigned char*, CryptAlgorithm, int, int, int,
int, bool) () from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#4  0x00007ffff4796965 in Parser::getObj(Object*, bool, unsigned char*, CryptAlgorithm, int, int, int,
int, bool) () from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#5  0x00007ffff47aa2cc in XRef::readXRef(long long*, std::vector<long long, std::allocator<long long> >*,
std::vector<int, std::allocator<int> >*) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#6  0x00007ffff47aa4e9 in XRef::XRef(BaseStream*, long long, long long, bool*, bool) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#7  0x00007ffff479a51f in PDFDoc::setup(GooString*, GooString*) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#8  0x00007ffff479a748 in PDFDoc::PDFDoc(GooString*, GooString*, GooString*, void*) ()
   from /usr/lib/x86_64-linux-gnu/libpoppler.so.44
#9  0x00007ffff734ec2f in poppler_document_new_from_file ()
   from /usr/lib/x86_64-linux-gnu/libpoppler-glib.so.8
#10 0x000000000040182c in main (argn=3, args=0x7fffffffe4b8) at pdf2svg.c:123
```

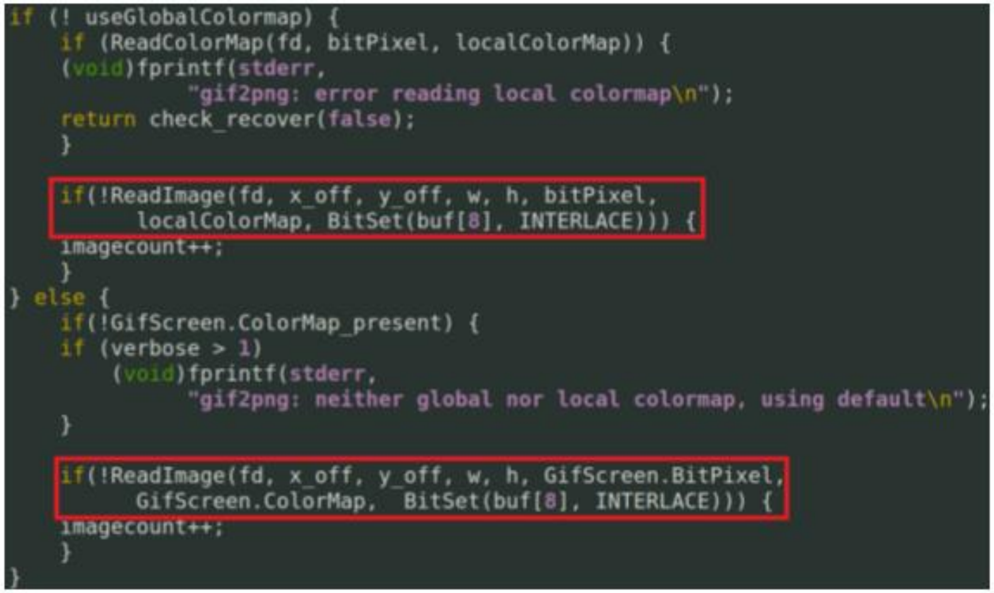
Real Case (cont.)

Both method get wrong result



```
#0  0x00000000004049a0 in nextLWZ (fd=fd@entry=0x61f010) at gifread.c:578
#1  0x00000000004053b9 in ReadImage (fd=fd@entry=0x61f010, x_off=x_off@entry=0,
    y_off=y_off@entry=0, width=width@entry=185, height=104, cmapSize=256,
    cmap=cmap@entry=0x61d5a8 <GifScreen+8>, interlace=false) at gifread.c:684
#2  0x0000000000405c2c in ReadGIF (fd=fd@entry=0x61f010) at gifread.c:218
#3  0x000000000040408e in processfile (
    fname=fname@entry=0x7fffffffdf70 "crashers/431608f289141fcd1e332faa9aae23c1/sf_243137834a04312fa7de2a03d9a210a9-7236241
    .gif", fp=fp@entry=0x61f010) at gif2png.c:707
#4  0x0000000000402126 in main (argc=<optimized out>, argv=<optimized out>)
    at gif2png.c:1002

#0  0x00000000004049a0 in nextLWZ (fd=fd@entry=0x61f010) at gifread.c:578
#1  0x00000000004053b9 in ReadImage (fd=fd@entry=0x61f010, x_off=x_off@entry=0,
    y_off=y_off@entry=0, width=width@entry=185, height=height@entry=104,
    cmapSize=cmapSize@entry=2, cmap=cmap@entry=0x7fffffffd640, interlace=false)
    at gifread.c:684
#2  0x0000000000405e10 in ReadGIF (fd=fd@entry=0x61f010) at gifread.c:207
#3  0x000000000040408e in processfile (
    fname=fname@entry=0x7fffffffdf70 "crashers/49362ec1412f1fb62e0375f5374daac4/sf_243137834a04312fa7de2a03d9a210a9-7444626
    .gif", fp=fp@entry=0x61f010) at gif2png.c:707
#4  0x0000000000402126 in main (argc=<optimized out>, argv=<optimized out>)
    at gif2png.c:1002
```

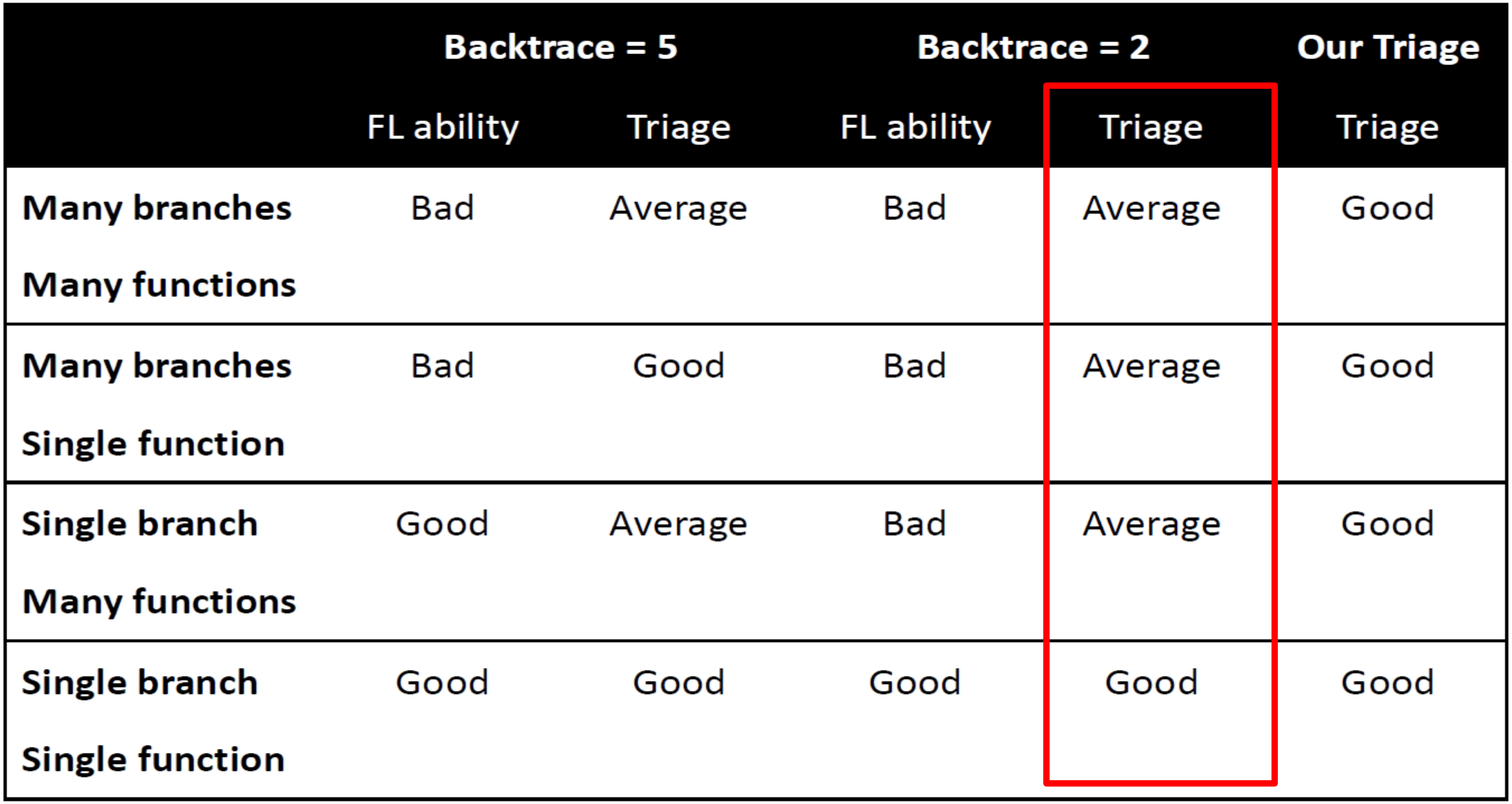


```
if (! useGlobalColormap) {
    if (ReadColorMap(fd, bitPixel, localColorMap)) {
    (void)fprintf(stderr,
             "gif2png: error reading local colormap\n");
    return check_recover(false);
    }

    if(!ReadImage(fd, x_off, y_off, w, h, bitPixel,
          localColorMap, BitSet(buf[8], INTERLACE))) {
    imagecount++;
    }
} else {
    if(!GifScreen.ColorMap_present) {
    if (verbose > 1)
        (void)fprintf(stderr,
             "gif2png: neither global nor local colormap, using default\n");
    }

    if(!ReadImage(fd, x_off, y_off, w, h, GifScreen.BitPixel,
          GifScreen.ColorMap,  BitSet(buf[8], INTERLACE))) {
    imagecount++;
    }
}
```

Method Comparison

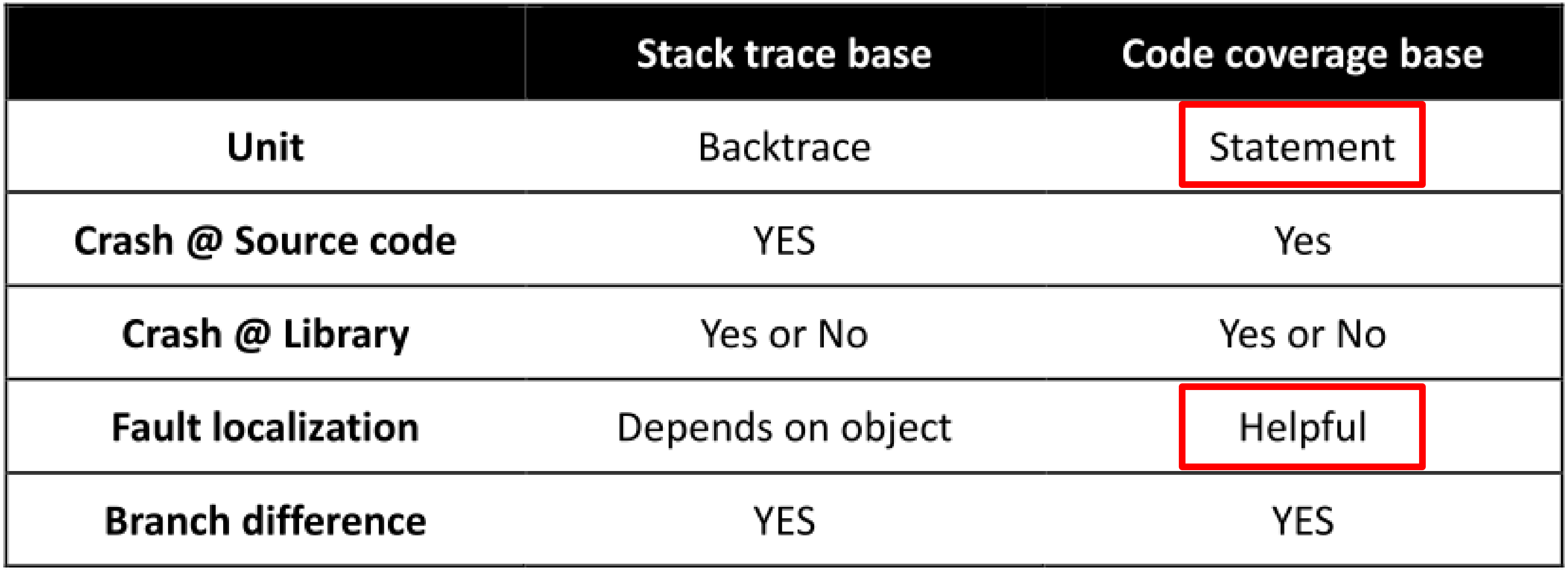


	Backtrace = 5		Backtrace = 2		Our Triage
	FL ability	Triage	FL ability	Triage	Triage
Many branches **Many functions**	Bad	Average	Bad	Average	Good
Many branches **Single function**	Bad	Good	Bad	Average	Good
Single branch **Many functions**	Good	Average	Bad	Average	Good
Single branch **Single function**	Good	Good	Good	Good	Good

1. BT=5 or Our method have almost same trend, our method is better

2. BT=2 usually get reversely results, but sometimes is correct (e.g. gif2png)

Method Comparison (2)



	Stack trace base	Code coverage base
Unit	Backtrace	Statement
Crash @ Source code	YES	Yes
Crash @ Library	Yes or No	Yes or No
Fault localization	Depends on object	Helpful
Branch difference	YES	YES

The only different between our method and traditional method is “UNIT”
Hence, FL ability of traditional method is depends on Object.

Outline

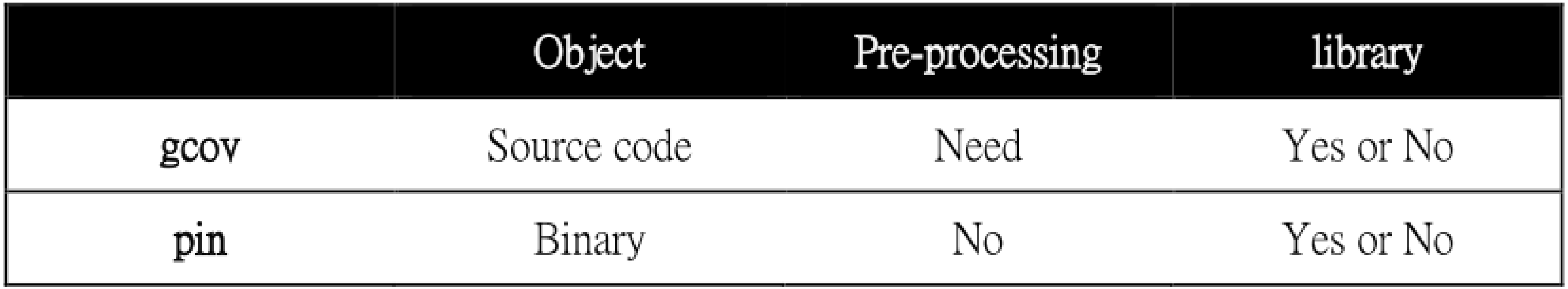
- Motivation
- Background
  - Failure Program
  - Crash Data
  - Fault and Triage
  - Fault localization
- Related Work
  - Fuzzing Tool
  - Stack Trace Triage Method
  - Flaw of Stack Trace Triage Method
- Method
  - Algorithm
  - Case Consideration
  - Research Question
- Results and Evaluation
  - System Architecture
  - Real Program
  - Method Comparison
- **Conclusion and Future Work**

Conclusion

- Our method
  - Based on Code Coverage (inspired by fault localization method)
  - Classify the fault triage type incrementally
- Contributions
  - Identify the drawbacks of the stack trace triage method
  - Resolve issues of traditional triage method

Future Work

1. Implementation on binary files
2. Integration of existing tools
   - Better triage for Fuzzer
   - Providing useful info for Fault Localizer



	Object	Pre-processing	library
gcov	Source code	Need	Yes or No
pin	Binary	No	Yes or No

DEMO

https://youtu.be/bKJtygkpJMs

Q&A

Thank you ☺