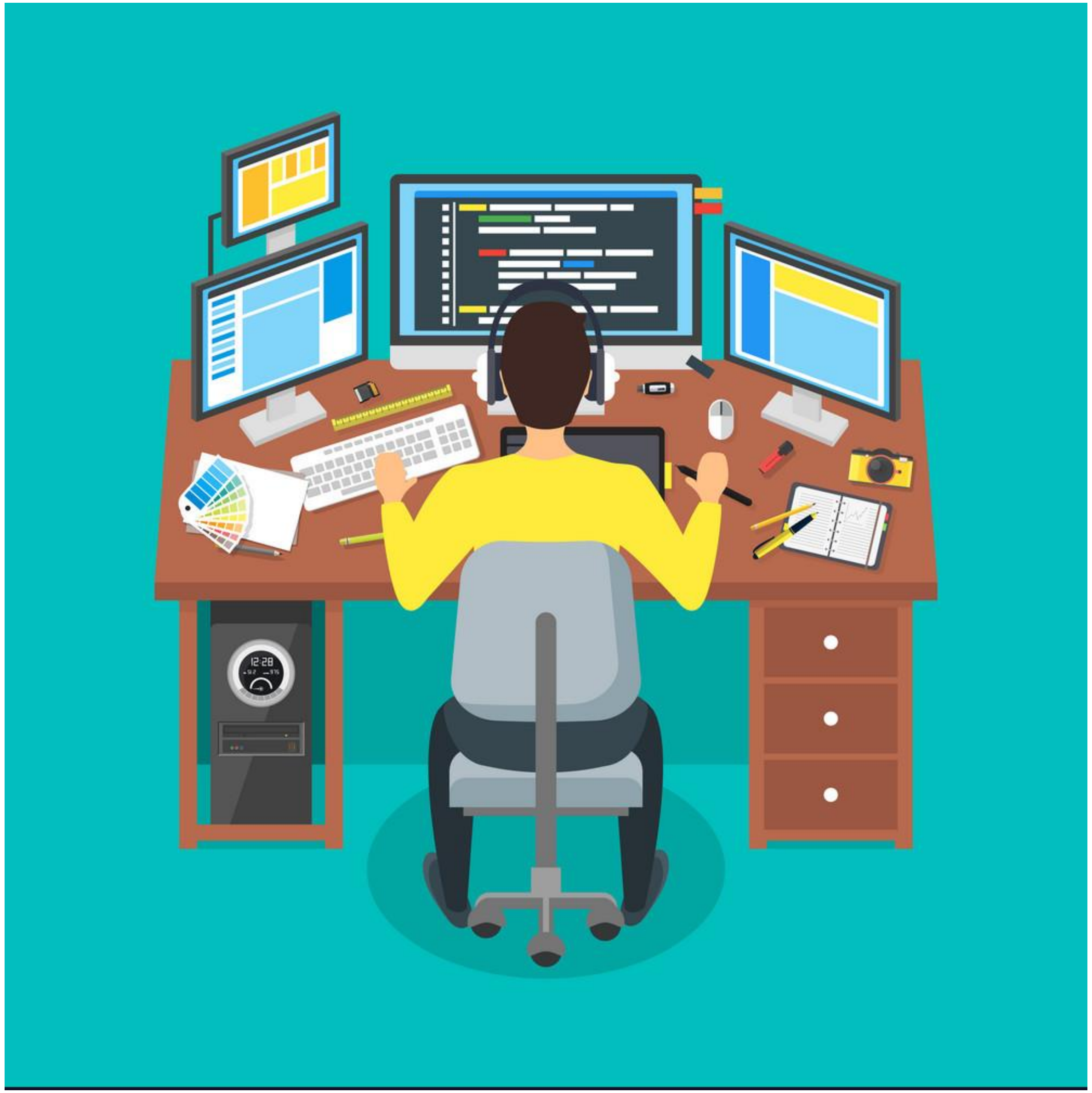


BugRedux: Reproducing Field Failures for In-house Debugging

Wei Jin and Alessandro Orso

Georgia Institute of Technology

Email: {weijin|orso}@gatech.edu

Ming Xiao Yuan

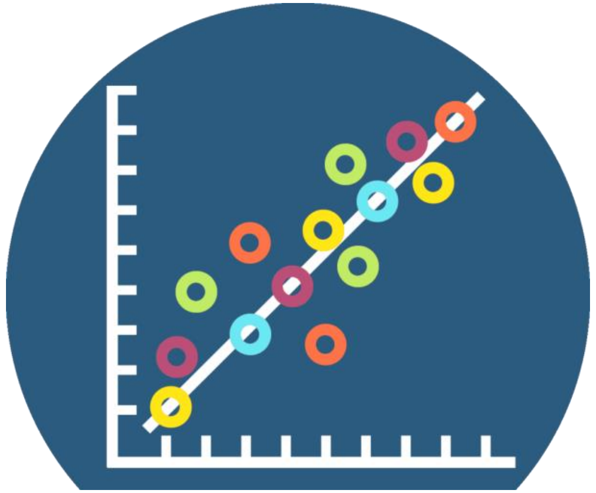
Masters Student at Polytechnique Montreal

The problematic

I can’t reproduce the error!

- Unable to reproduce user’s failures
- Configurability & Portability
- Difficult to foresee & reproduce

Why modern methods are ineffective



- Collect stack traces & register dumps.
- Limited for reproducing failure
- Identify correlations among crashes
- Possible solution: Collect only limited amounts of information like **Sample branch profiles**

Why modern methods are ineffective

- Usefulness ↑ when details ↑
- Novel techniques (richer data)
- Limitations:
  - Expensive to collect
  - Bound to contain sensitive data

Example of the problematic

- Fault at line 30
  - One character skipped & extra character added in the output string
  - Index of output grow larger than the index of the input by one for each \\
- Unlikely achieved by random testing to reach POF

```
1.  char replaceescape(char e) {
2.    switch (e) {
3.      case 'n':
4.        return 10;
5.      case 't':
6.        return 9;
7.      default:
8.        return 0;
9.    }
10. }
11. char uppercase(char l) {
12.   return l-'a'+'A';
13. }
14. void printresult(char* str,int length) {
15.   int i;
16.   for (i=0;i<length;i++) {
17.     if (str[i]!=0)
18.       printf("%c",str[i]);
19.   }
20. }
21. void process(char* source, char* dest) {
22.   int out=0;
23.   int in=0;
24.   int srclength = strlen(source);
25.   while (in<srclength) {
26.     if (source[in]>='a'&&source[in]<='z') {
27.       dest[out]=uppercase(source[in]);
28.     } else if (source[in]=='\\') {
29.       dest[out]=replaceescape(source[in+1]);
30.       out++;  // correct version: in++;
31.     } else
32.       dest[out]=source[in];
33.     out++;
34.     in++;
35.   }
36.   printresult(dest,out);
37. }
38. int main(int argc, char *argv[]) {
39.   if (argc!=2)
40.     exit(0);
41.   if (strlen(argv[1])>=256)
42.     exit(0);
43.   char* outputstr=malloc(256);
44.   process(argv[1],outputstr);
45. }
```

Replace character by a new line or tab if \n or \t or null.

Returns the uppercase

Prints the result

If alphabetical and lower case turn it in uppercase

If it's escape run replaceescape

Figure 2. Example of faulty program.

The problematic

Goal of the study

Our goal

- Address the limitations by developing **novel approaches**

- E′ should result in failure F′
- E′ should execute P to produce F′
- Should generate E′ only using P and D
- D should not contain sensitive data

P: Program

E: Execution environment → E′: In-house execution

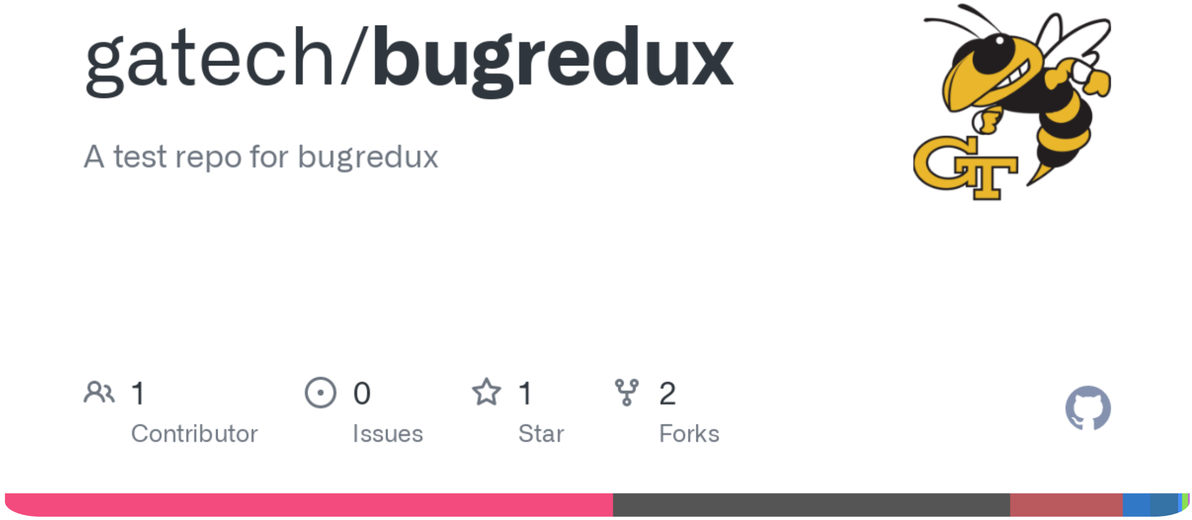
F: Failure of P in E → F′: In-house failure

D: Execution data

What is BugRedux?

"At any point of a symbolic execution, called a symbolic state it is expressed as a path condition".

- A tool that implements a technique that
  - Collect execution data
  - Use the data to synthesize in-house executions
- **Symbolic Executions**: Executes a program using symbolic instead of concrete inputs.
- ***Symbolic State***: Representation of a moment in a symbolic execution
- ***Path condition (PC)***: Conditions on the inputs that cause the execution to reach that point.



What is BugRedux?

- **Symbolic execution** of path $\langle 1,2,3,4,6,7,8,13 \rangle$
- **Symbolic state** at line 13 would be $\{[a \rightarrow a_0], [b \rightarrow b_0], [c \rightarrow c_0], [d \rightarrow a_0 + 4]\}$
- **Path condition (PC)** would be $(a_0 + 4 < b_0) \land (b_0 \leq 5) \land (a_0 < 5) \land (a_0 + 4 < c_0)$

a_0 b_0 c_0

```
function foo(int a, int b, int c) {
 1. int d = a + 4
 2. if (d < b)
 3.    //do something
 4. if (b > 5)
 5.    //do something
 6. else if (a < 5)
 7.   if (d < c)
 8.     //do something
 9.   else
10.     //do something
11. else
12.   //do something
13. return
}
```

The problematic

Goal of the study

Methodology

The Specs

- 4 variations of BugRedux
  - Point of Failures (POF)
  - Stack traces
  - Call sequences
  - Complete program traces
- Implemented in C
- Applied on 17 failures of 15 real world programs

The architecture

- **Instrumenter**: Application → Instrumented application
- **Analyzer**: Crash reports → Test input
- **Software Tester**: Test input → Recreate field failure

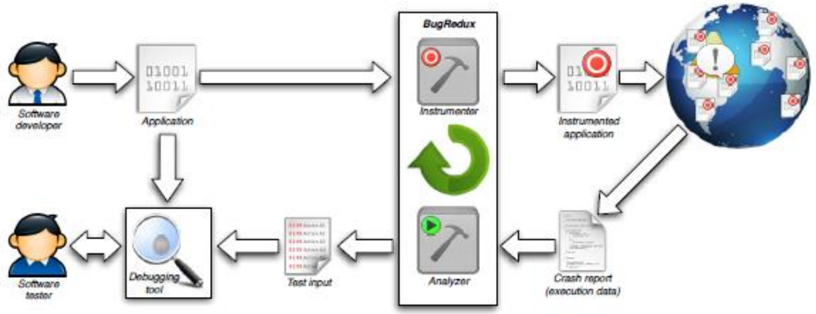


Figure 4. Intuitive high-level view of BUGREDUX.

The algorithms

- *GenerateInputs*: Symbolic execution to identify set of intermediate goals.
- *SelNextState*: Heuristic based on distance to select states

Algorithm 2: SelNextState

```
Input  : icfg : ICFG for program P
         states_set: set of symbolic states
         curr_goal: next goal
Output: ret_state: candidate state for exploration

begin
    mindis ← +∞
    ret_state ← null
    foreach State_i ∈ states_set do
        if State_i.goal == curr_goal then
            if State_i.loc can reach curr_goal in ICFG then
                nd ← shortest distance from State_i.loc to
                curr_goal in ICFG
                if nd < mindis then
                    mindis ← nd
                    ret_state ← State_i
                end
            end
        end
```

Algorithm 1: GenerateInputs

```
Input  : icfg : ICFG for program P
         goals_list : an ordered list of statements G_0, ...G_n
Output: input_f: candidate input for synthesized run

begin
    sym_state_0 ← initial symbolic values of program inputs
    states_set ← (icfg.entry, true, sym_state_0, G_0)
    curr_goal ← G_0
    while true do
        curr_state ← null
        while curr_state == null do
            curr_state ←
            SelNextState(icfg, states_set, curr_goal)
            if curr_state == null then
                if curr_goal ≠ G_0 then
                    curr_goal ← previous goal in goals_list
                    continue
                else
                    return null
                end
            end
        end
        if curr_state.cl == curr_goal then
            if curr_goal == G_n then
                input_f ← solver.getSol(curr_state.pc)
                if input_f is found then
                    return input_f
                else
                    remove(curr_state, states_set)
                    continue
                end
            else
                curr_goal ← next target in goals_list
                curr_state.goal ← curr_goal
            end
        else
            if curr_state.cl ∈ goal_list then
                remove(curr_state, states_set)
                continue
            end
        end
        if curr_state.cl is a conditional statement then
            curr_state.pc ←
            addConstr(curr_state.pc, pred, true)
            curr_state.cl ← getSucc(curr_state.cl, true)
            if solver.checkSat(curr_state.pc) == false then
                remove(curr_state, states_set)
            end
            false_pc ←
            addConstr(curr_state.pc, pred, false)
            false_cl ← getSucc(curr_state.cl, false)
            if solver.checkSat(curr_state.pc) == true then
                new_state ←
                (false_cl, false_pc, curr_state.ss, curr_state.goal)

                insert(new_state, state_set)
            end
        else
            curr_state.ss ←
            symEval(curr_state.ss, curr_state.cl)
            curr_state.cl ← getSucc(curr_state.cl)
        end
    end
end
```

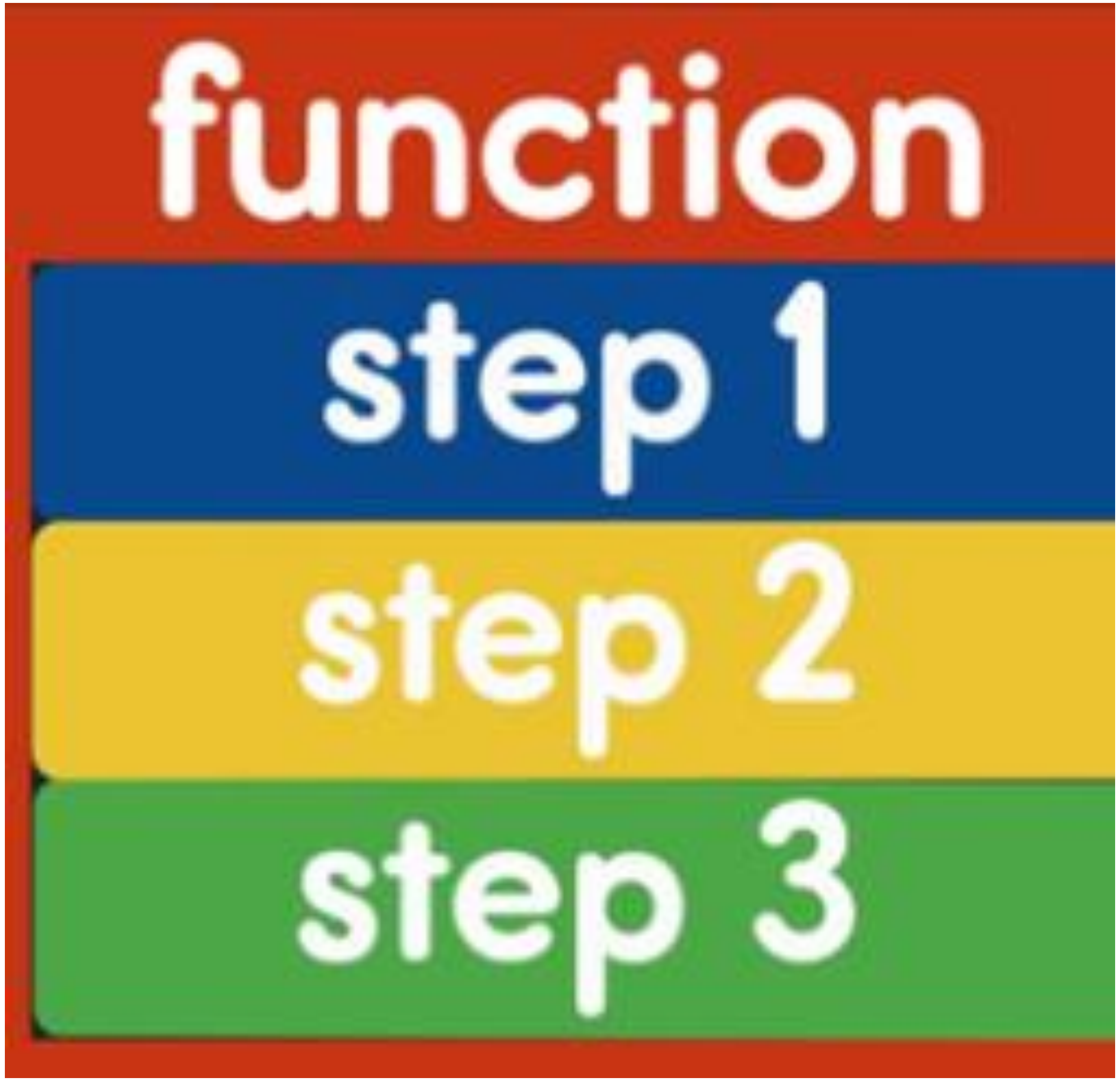
The Research Questions

RQ1: Can BugRedux synthesize executions that are able to reproduce field failures starting from a set of execution data?

RQ2: If so, which types of field data provide the best tradeoffs in terms of cost benefit?

The Data

- 4 variations of BugRedux
    - POFs (no modif)
    - Call Stacks (no modif)
    - Call Sequences (instruments call sites)
    - Complete program traces (instruments all branches)
- POFs: Single entry
- Call Stacks: Entry for each function
- Call Sequences: *goals_list* would contain entry for each call
- Complete traces*: goal_list* would be an entry per branch



Methodology

- 3 public repositories:
  - 3 from SIR
  - 2 from BugBench
  - 9 from exploit-db
- Chose because of past researches and representativeness

1. Ran test cases until fail
2. Run all the failing 3 times
   2.1 Ran on unmodified program
   2.2 Ran on BugRedux to collect Call Sequences
   2.3 Ran on BugRedux to collect complete traces

- Collect time and size of execution data generated
- Use 4 variants of BugRedux to synthesize candidate input
- See if inputs can reproduce the original failure
- Timeout of 72 hours for generation

Table I
SUBJECT PROGRAMS USED IN OUR STUDY.

Name	Repository	Description	Size (kLOC)	# Faults
sed	SIR	stream editor	14	2
grep	SIR	pattern-matching utility	10	1
gzip	SIR	compression utility	5	2
ncompress	BugBench	(de)compression utility	2	1
polymorph	BugBench	file system "unixier"	1	1
aeon	exploit-db (CVE-2005-1019)	mail relay agent	3	1
iwconfig	exploit-db (CVE-2003-0947)	wireless tool	11	1
glftpd	exploit-db (OSVDB-ID#16373)	FTP server	6	1
htget	exploit-db (CVE-2004-0852)	file grabber	3	1
socat	exploit-db (CVE-2004-1484)	multipurpose relay	35	1
tipxd	exploit-db (OSVDB-ID#12346)	IPX tunneling daemon	7	1
aspell	exploit-db (CVE-2004-0548)	spell checker	0.5	1
exim	exploit-db (EDB-ID#796)	message transfer agent	241	1
rsync	exploit-db (CVE-2004-2093)	file synchronizer	67	1
xmail	exploit-db (CVE-2005-2943)	email server	1	1

The problematic

Goal of the study

Methodology

Results

Results

Table II Overhead by BugRedux

TIME OVERHEAD IMPOSED ON THE SUBJECT PROGRAMS.

Name	POF	Call stack	Call sequence	Complete trace
sed.fault1	0	0	4.5%	27.2%
sed.fault2	0	0	12.5%	87.5%
grep	0	0	47%	182%
gzip.fault1	0	0	10.3%	72%
gzip.fault2	0	0	12%	308%
ncompress	0	0	2%	16%
polymorph	0	0	1%	8%
aeon	0	0	50%	1066%
iwconfig	0	0	7%	128%
glftpd	0	0	9%	45%
htget	0	0	9%	287%
socat	0	0	21%	110%
tipxd	0	0	2%	36%
aspell	0	0	18.8%	143%
exim	0	0	17.4%	389%
rsync	0	0	3%	66%
xmail	0	0	22.6%	290%

Depends on program size and execution length

Table III Function call low but loops high

SIZE OF EXECUTION-DATA COLLECTED (KB).

Name	POF	Call stack	Call sequence	Complete trace
sed.fault1	0.8	0.8	5.8	54.4
sed.fault2	0.9	0.9	10.2	261.9
grep	0.7	0.7	3.4	716.1
gzip.fault1	0.8	0.8	2.0	176
gzip.fault2	0.8	0.8	2.5	1784.6
ncompress	0.7	0.7	0.9	33.1
polymorph	0.5	0.5	0.7	1.5
aeon	1	1	1.1	3
iwconfig	1	1	3	594
glftpd	1.5	1.5	3.2	130
htget	0.7	0.7	2.7	2814
socat	0.8	0.8	9.6	451
tipxd	0.6	0.6	0.7	19
aspell	0.6	0.6	30.5	566
exim	0.9	0.9	100.7	14897
rsync	1	1	11.4	521
xmail	0.8	0.8	84.8	2361

Same (Extract from crash reports together)

Results

Leaves more degree of freedom
& ↑ chance of success

Table IV
EFFECTIVENESS AND EFFICIENCY OF BUGREDUX IN SYNTHESIZING EXECUTIONS STARTING FROM COLLECTED EXECUTION DATA.

Name	POF	Call stack	Call sequence	Whole trace
sed.fault1	N/A	N/A	98s	N/A
sed.fault2	N/A	N/A	17349s	N/A
grep	N/A	16s	48s	N/A
gzip.fault1	3s	18s	11s	N/A
gzip.fault2	20s	28s	25s	N/A
ncompress	155s	158s	158s	N/A
polymorph	65s	66s	66s	N/A
aeon	1s	1s	1s	1s
rysnc	N/A	N/A	88s	N/A
iwconfig	5s	5s	5s	N/A
glftpd	5s	5s	4s	N/A
htget	53s	53s	9s	N/A
socat	N/A	N/A	876s	N/A
tipxd	27s	27s	5s	N/A
aspell	5s	5s	12s	N/A
xmail	N/A	N/A	154s	N/A
exim	N/A	N/A	N/A	N/A

Provide more information

1) POF are close to entry
2) Failures triggered by reaching POFs

Table V
EFFECTIVENESS OF SYNTHETIC EXECUTIONS IN REPRODUCING OBSERVED FAILURES.

Name	POF	Call stack	Call sequence	Complete trace
sed.fault1	N	N	Y	N
sed.fault2	N	N	Y	N
grep	N	N	Y	N
gzip.fault1	Y	Y	Y	N
gzip.fault2	N	N	Y	N
ncompress	Y	Y	Y	N
polymorph	Y	Y	Y	N
aeon	Y	Y	Y	Y
rsync	N	N	Y	N
iwconfig	Y	Y	Y	N
glftpd	Y	Y	Y	N
htget	N	N	Y	N
socat	N	N	Y	N
tipxd	Y	Y	Y	N
aspell	N	N	Y	N
xmail	N	N	Y	N
exim	N	N	N	N

Now for the questions…

- **RQ1: Can BugRedux synthesize executions that are able to reproduce field failures starting from a set of execution data?**
- Yes
- **RQ2: If so, which types of field data provide the best tradeoffs in terms of cost benefit?**
- Call Sequences (acceptable time & space overhead and hides confidential information about an execution)
- Downside of POF and Call stack: POF distant from the fault, which makes POF and Call stack irrelevant.
- Downside of Complete Trace: Providing too much guidance may be just as problematic as providing too little. (Sweet spot)

What we learned

Richest data, beside being expensive to collect, is not necessarily the most useful when used for synthesizing executions.

Best cost-benefit ratio is Call Sequences (recreates 16 out of 17 failures)

To conclude

- Recreating failures in the field is arduous
- To address this problem, we presented BugRedux
  - Collecting data about program runs in the field
  - Extract the data collected to mimic the failing execution & reproduce failure
- We found that using Call Sequences is the most efficient way to reproduce failures