Using Likely Invariants for Automated Software Fault Localization

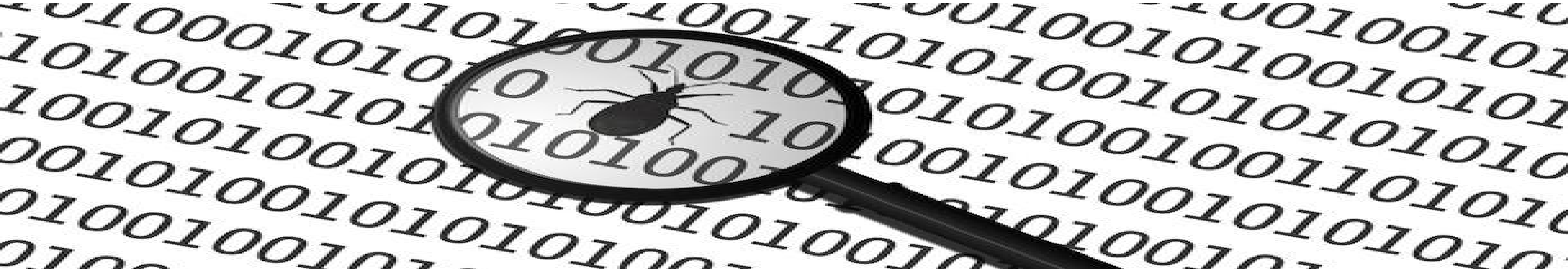
Swarup Kumar Sahoo, John Criswell Chase, Geigle Vikram Adve

slide author names omitted for FERPA compliance

Bug Diagnosis and Fault Localization

Given: A program’s source code and a failing test input

Goal: Identify a set of program locations which may be causing the failure



Currently, mostly a manual process; existing automated tools are not widely used

Research Questions

1. How can we perform automated bug diagnosis, specifically fault localization?
2. How to find root causes of program failure without including many false positives or eliminating the true root cause from the list of candidates?

Contributions

1. A “a novel invariant-based approach for fault localization” which provides more precise fault location candidates than other methods.
2. A set of heuristics for pruning false positive fault location candidates from the diagnoses results list.
3. An evaluation comparing the system with existing approaches, using real bugs from a larger and more diverse set of applications than in previous studies.

Approach:

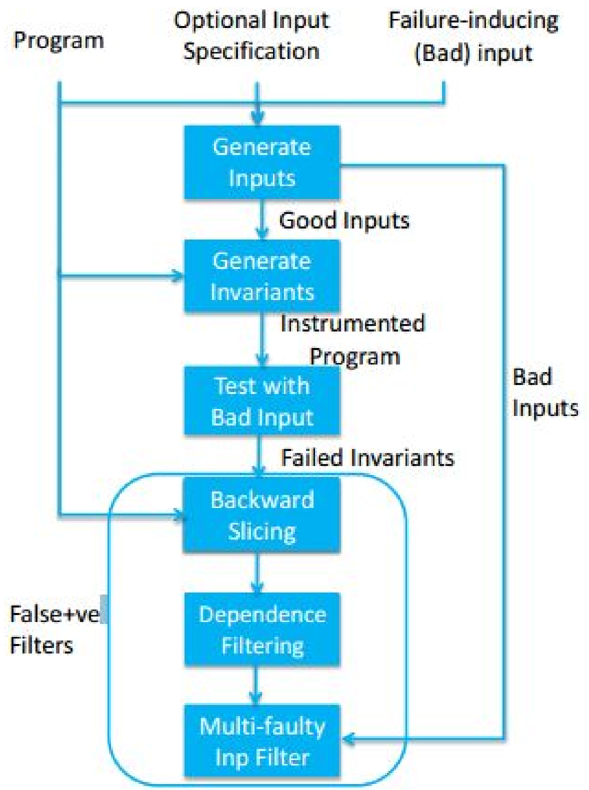


Figure 1. Diagnosis Tool Architecture

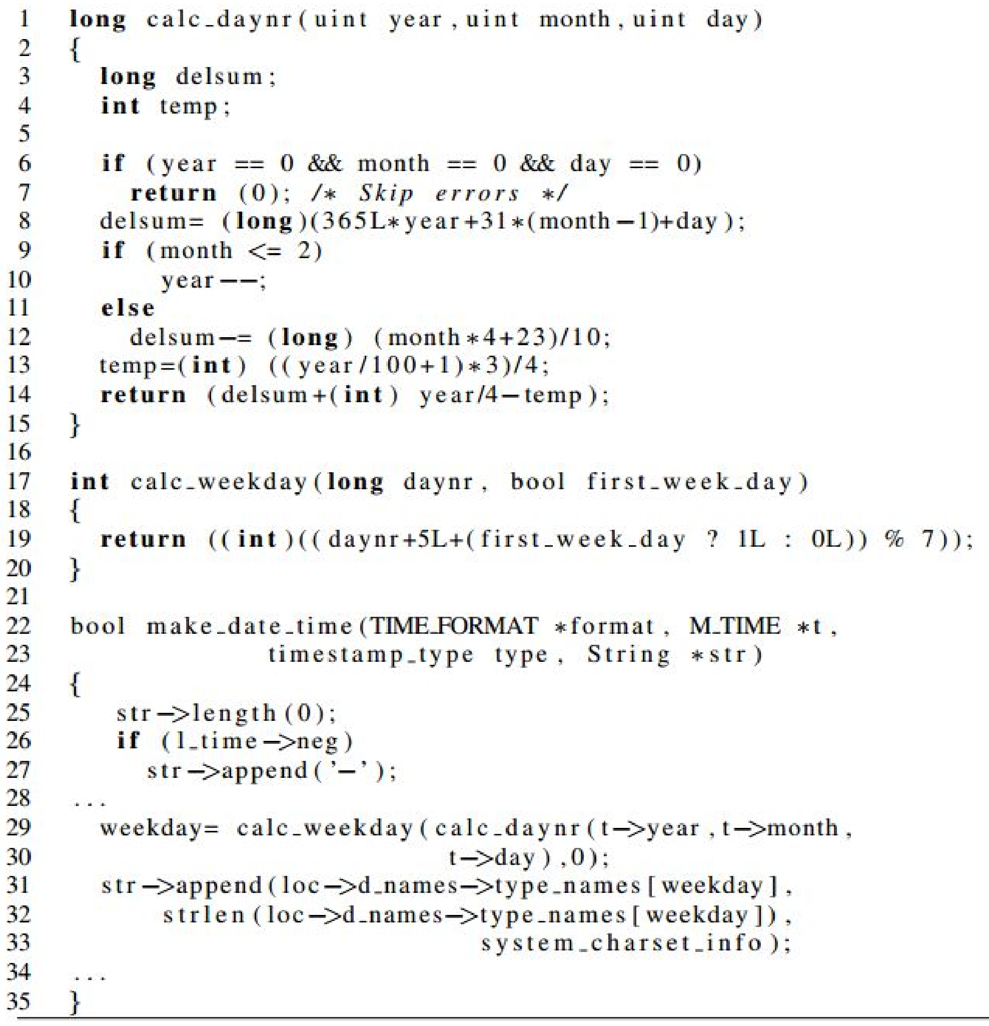
Key Ideas:

Combine **delta debugging** with **likely invariant analysis** to see which likely invariants are invalidated on failing inputs

1. Generate likely invariants using training inputs ***close to failing input***
2. Use ***filtering heuristics*** to reduce the number of false positives, while making sure that the true root cause is not eliminated

Example

- Error uint year equal to 0 is decremented
- large negative number when cast as a int in line 14.
- Error propagates, causing a buffer overflow at line 31.
- While line 31 causes the crash, it isn’t the **root cause** of the problem.
- We can see that program crashes when year=0 and month <=2



```
1  long calc_daynr(uint year,uint month,uint day)
2  {
3    long delsum;
4    int temp;
5
6    if (year == 0 && month == 0 && day == 0)
7      return (0); /* Skip errors */
8    delsum= (long)(365L*year+31*(month-1)+day);
9    if (month <= 2)
10       year--;
11   else
12     delsum-= (long) (month*4+23)/10;
13   temp=(int) ((year/100+1)*3)/4;
14   return (delsum+(int) year/4-temp);
15 }
16
17 int calc_weekday(long daynr, bool first_week_day)
18 {
19   return ((int)((daynr+5L+(first_week_day ? 1L : 0L)) % 7));
20 }
21
22 bool make_date_time(TIME_FORMAT *format, M_TIME *t,
23                     timestamp_type type, String *str)
24 {
25     str->length(0);
26     if (l_time->neg)
27       str->append('-');
28 ...
29   weekday= calc_weekday(calc_daynr(t->year,t->month,
30                                    t->day),0);
31   str->append(loc->d_names->type_names[weekday],
32       strlen(loc->d_names->type_names[weekday]),
33                             system_charset_info);
34 ...
35 }
```

Listing 1. Source Code Example

Input Generation

Given source code, failing input and (optional) specification containing valid tokens:

- Generate lexicographically close “good,” non-failing program inputs
- “inp-gen-delete” and “inp-gen-replace” algorithms
- Both use the ddmin algorithm
- Can also use to generate some “bad” inputs
- **Use the good inputs to generate likely invariants, bad inputs to see which ones fail (failing invariants = candidate bug locations)**

Step 1:Likely program invariants

- Using ddmin, take bad input *t-> [year=0, month=2, day=20]
- Produce similar but non failing inputs, such as *t->[year=1,month=4,day=21]
- Uses inputs to find range invariants, limiting to load,store and function return values.
- Possible Invariants: calc_daynr and calc_weeday always return positive ints.
- When ran on failing inputs, some invs fail, giving a number of candidate root causes.
- With this method, our example contained 95 failed invariants.

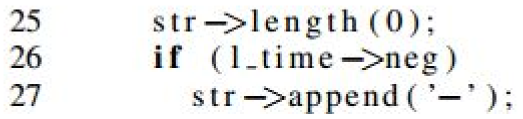
Filtering False Positives

1. **Dynamic Backwards Slicing**

- Find and discard invariants that don’t affect the faulting instruction
    - **Giri -** a custom backwards-slicing compiler; works with LLVM to instrument code to record:
        - basic block exits
        - memory accesses and their addresses
        - function calls and returns.
    - Use this information to create a “backwards slice” representing statements that affect the fault symptom
    - Remove all invariant locations that aren’t a part of the slice from the candidate fault location list

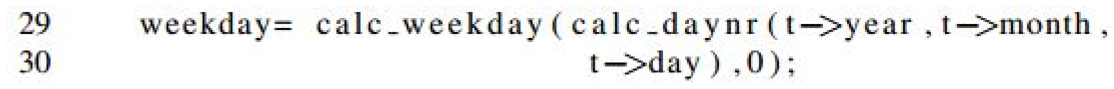
Step 2: Dynamic Program Slicing

- Remove failed invariants that do not contribute to the crash.
- Failed invariants relating to str cannot not be the root cause, and are filtered out.



```
str->length(0);
if (l_time->neg)
  str->append('-');
```

- About 62% of our failed invariants are filtered out, leaving 36 left.
- These function return variables do contribute to weekday.
- These invariants will hold.



```
weekday= calc_weekday(calc_daynr(t->year,t->month,
                      t->day),0);
```

Filtering False Positives

2. **Dependent Chains of Failures**

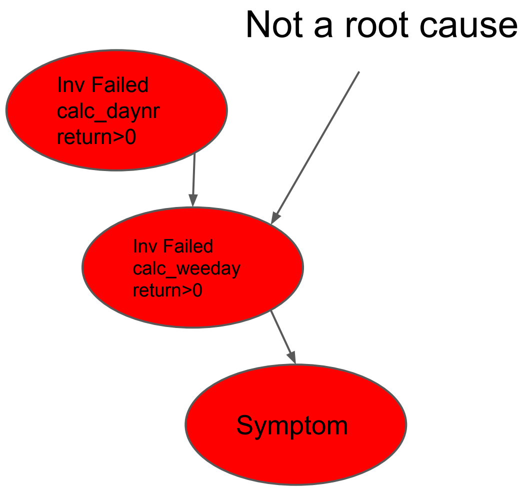
- Use Dynamic Dependence Graph to filter out dependent chains of failing invariants
  - DDG = Graph(N, E) where N = set of instructions executed in a given run of the program, E = $n_i \rightarrow n_j$ if instruction n_j is dependent on instruction n_i
  - Can use depth-first-search

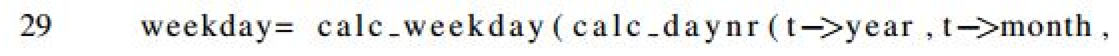
3. **Multiple Failure-inducing Inputs**

- Using input construction algorithms, construct more “bad” inputs
- Use previous filtering techniques to generate a list of relevant failing invariants
- Take intersection of these failing invariants for all generated inputs

Step 3: Dependence Filtering

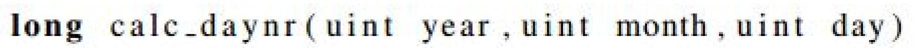
- Errors propagate through programs.
- Eliminate one of our two invariants
- Depends on a failed invariant with no other intervening instructions.
- This step removes about 55% of remaining invariants, leaving around 16.


Not a root cause
Inv Failed calc_daynr return>0
Inv Failed calc_weeday return>0
Symptom



```
29      weekday= calc_weekday(calc_daynr(t->year,t->month,
```

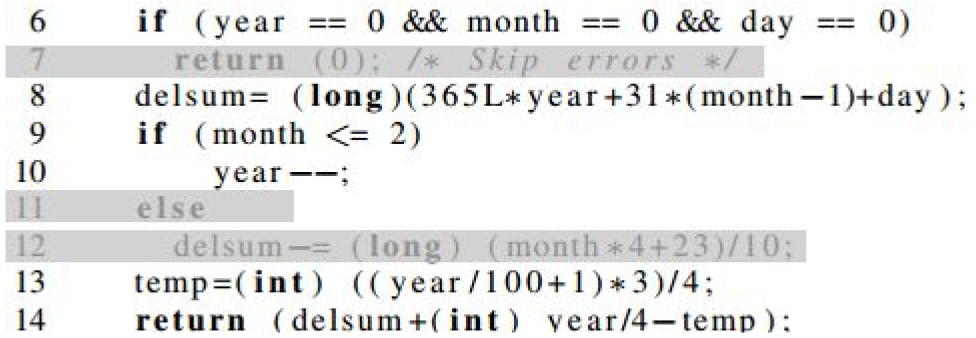
Step 4: Multiple Inputs

- Last 3 steps were performed using a single input.
- Remove invs that arise from chance or that are input dependent.
- 

  ```
  long calc_daynr(uint year,uint month,uint day)
  ```
- For one run, the invariant as month>0 is found
- This will fail on failing input where *t->[year=0,month=-1,day=30]
- Using a new input where *t->[year=0,month=1,day=30], this inv will not fail.
- Taking intersection allows us to filter these false positives.
- This step reduces candidates by 25%, leaving 12 invariants left.

Step 5: Feedback to Programmer

- True location of the crash maybe not be within the invariants.
- Inv on return value of calc_daynr failing to be positive indicates the location of the root cause in this case.
- The program will present the parts of the program that are used to calculate the return value of calc_daynr (lines 6, 8, 9, 10, 13 and 14)
- Inv hints to temp being very large
- temp is calculated using year (13)
- year is decremented, causing overflow.
- Root cause found.



```
 6      if (year == 0 && month == 0 && day == 0)
 7        return (0); /* Skip errors */
 8      delsum= (long)(365L*year+31*(month-1)+day);
 9      if (month <= 2)
10          year--;
11      else
12        delsum-= (long) (month*4+23)/10;
13      temp=(int) ((year/100+1)*3)/4;
14      return (delsum+(int) year/4-temp);
```

Evaluation - Applications

- Tested on software using C/C++ (LLVM compiler)
  - Squid HTTP proxy server
  - Apache HTTP web server
  - MySQL database server
- Table shows different characteristics of each bug (lines of code executed etc.)

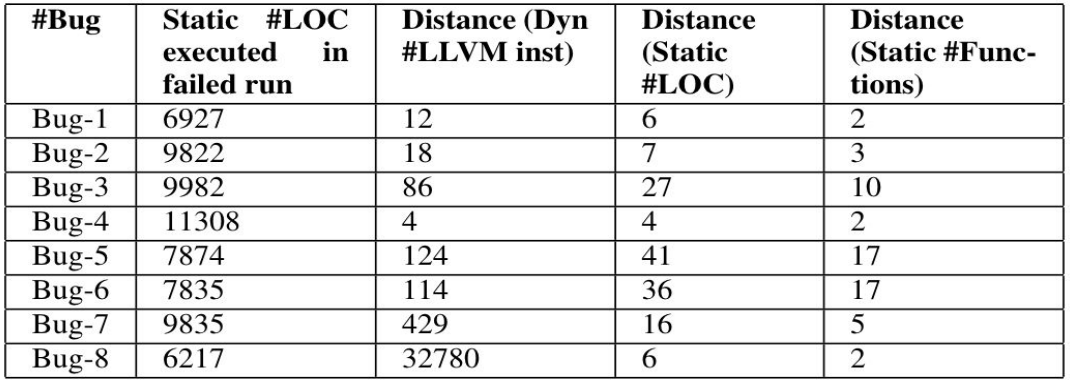


Table 2. Bug Characteristics which may Impact Difficulty of Diagnosis.

#Bug	Static #LOC executed in failed run	Distance (Dyn #LLVM inst)	Distance (Static #LOC)	Distance (Static #Functions)
Bug-1	6927	12	6	2
Bug-2	9822	18	7	3
Bug-3	9982	86	27	10
Bug-4	11308	4	4	2
Bug-5	7874	124	41	17
Bug-6	7835	114	36	17
Bug-7	9835	429	16	5
Bug-8	6217	32780	6	2

Number of Root causes & False positives

- Number of Root Causes
  - Dynamic Backward Slicing removed 80% of false positives (Bugs 6, 1, 8)
  - Dependence Filtering removed 58% of false positives (Bug 1: best case 70%)
  - Multiple faulty inputs removed little to none

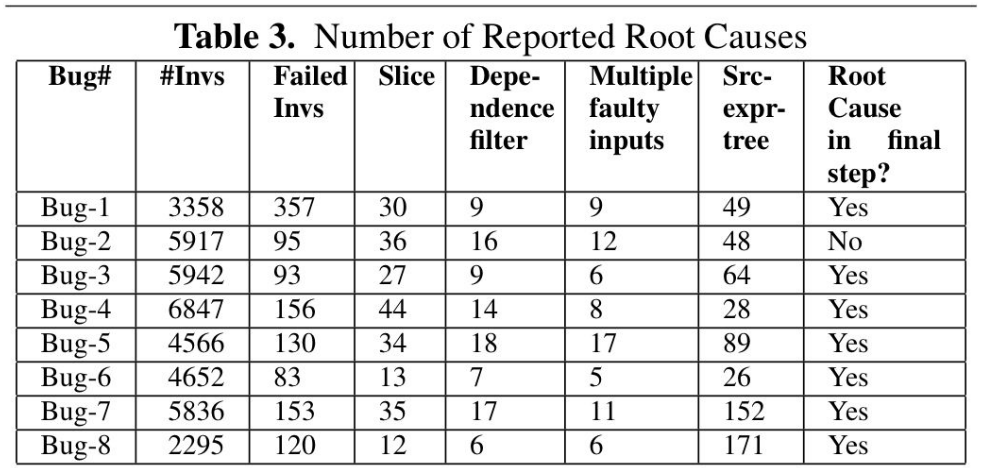


Table 3. Number of Reported Root Causes

Bug#	**#Invs**	**Failed Invs**	**Slice**	**Dependence filter**	**Multiple faulty inputs**	**Src-expr-tree**	**Root Cause in final step?**
Bug-1	3358	357	30	9	9	49	Yes
Bug-2	5917	95	36	16	12	48	No
Bug-3	5942	93	27	9	6	64	Yes
Bug-4	6847	156	44	14	8	28	Yes
Bug-5	4566	130	34	18	17	89	Yes
Bug-6	4652	83	13	7	5	26	Yes
Bug-7	5836	153	35	17	11	152	Yes
Bug-8	2295	120	12	6	6	171	Yes

Finding True Root Causes

- Root cause in final step for 7 out of 8 bugs
- Training sets can produce different diagnosis with different input
- Bugs 7 & 8: significant reduction
    - large Src-expr trees
    - executed 6000 and 9000 static source lines

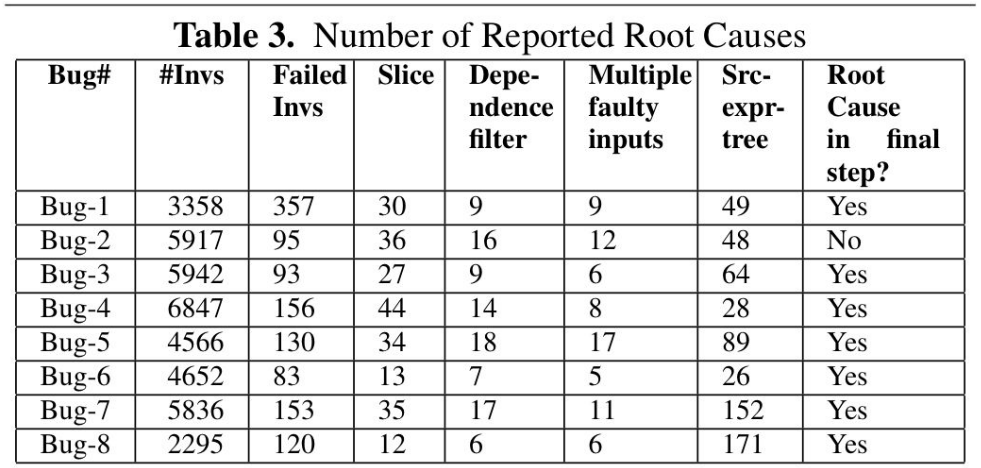


Table 3. Number of Reported Root Causes

Bug#	#Invs	Failed Invs	Slice	Depe-ndence filter	Multiple faulty inputs	Src-expr-tree	Root Cause in final step?
Bug-1	3358	357	30	9	9	49	Yes
Bug-2	5917	95	36	16	12	48	No
Bug-3	5942	93	27	9	6	64	Yes
Bug-4	6847	156	44	14	8	28	Yes
Bug-5	4566	130	34	18	17	89	Yes
Bug-6	4652	83	13	7	5	26	Yes
Bug-7	5836	153	35	17	11	152	Yes
Bug-8	2295	120	12	6	6	171	Yes

Comparison with Tarantula and Ochiai

- Performs analysis on statement execution given inputs (Statistical)
  - Statements ranked in decreasing order for similarity
- Performance of correlation
  - Good - Bug 5 (Control-Flow divergence)
  - Bad - Bug 6 (Success & Failure inputs)
    - executed on same root causes
  - performed well on 2 out of 8 bugs

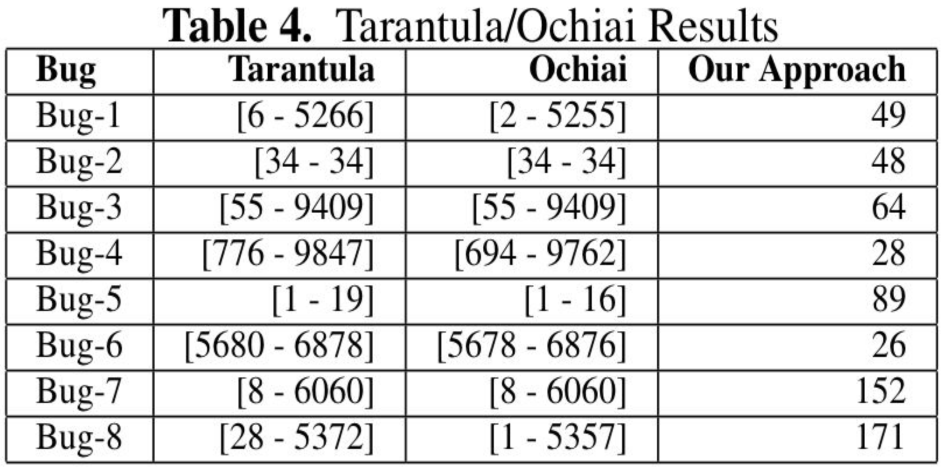


Table 4. Tarantula/Ochiai Results

Bug	Tarantula	Ochiai	Our Approach
Bug-1	[6 - 5266]	[2 - 5255]	49
Bug-2	[34 - 34]	[34 - 34]	48
Bug-3	[55 - 9409]	[55 - 9409]	64
Bug-4	[776 - 9847]	[694 - 9762]	28
Bug-5	[1 - 19]	[1 - 16]	89
Bug-6	[5680 - 6878]	[5678 - 6876]	26
Bug-7	[8 - 6060]	[8 - 6060]	152
Bug-8	[28 - 5372]	[1 - 5357]	171

Limitations

- Only LLVM Compatible
- Missing Code bugs
  - Initializations
  - Concurrency
- Robust Input Generation
  - Multiple fault dropped root cause
- Does not report valuable information
  - invariant failures and values

Discussion

- How useful is this tool if it only returns load instructions, store instructions, and method calls as candidates for fault locations?
    - We believe that developers may have to write more, smaller methods in order for this tool to be effective, since it can only track load, store, and return variables. If the method’s return value violates an invariant, it may be difficult to localize the true root cause if the method contains many lines of code.

Discussion

- As is, the tool only detects range invariants. What other types of invariants should it detect?
    - It may be helpful for the tool to detect invariants which capture relationships between variables (e.g. a>b), even though they are less efficient to generate and keep track of (i.e. the number of invariants is no longer not linear in the size of the program, as with range invariants on single variables).

Discussion

- How could we modify this so that it detects missing code bugs, or concurrency bugs?
    - In order to detect these types of bugs, it may be necessary to include control flow invariants in addition to range invariants. How this can be done is unclear and is described as “beyond the scope” of the authors’ work.

Discussion

- Is it more important to reduce the number of false positive fault location candidates, or to make sure that the true fault location is not thrown out of the candidate list?
    - This is more of an opinion question; there are pros and cons to each. The candidate locations list is useless if it does not contain the true root cause, but could potentially cost the developer a lot of time if they need to dig through many false fault locations to unearth the true cause.

Discussion

- This tool uses the LLVM compiler, implying that it works with C and C++ code. Can it be extended to other languages as well?
    - LLVM is actually used by many languages (Java bytecode, Python, Go, R, Ruby, Scala, etc.), which seems to imply that the tool may be used for bug localization with those languages as well.