Bachelor Thesis

Test-based Fault Localization in the Context of Formal Verification: Implementation and Evaluation of the Tarantula Algorithm in CPAchecker

Schindar Ali

Software and Computational Systems Lab
LMU Munich

02.09.2020

Agenda

1. Motivation
2. Background
3. Implementation
4. Evaluation
5. Future Work
6. Conclusion

Motivation

1.1 Motivation

1. Debugging software is an expensive and mostly manual process.

2. Of all debugging activities, locating the fault is the most challenging one.

1.1 Motivation

1. Debugging software is an expensive and mostly manual process.

2. Of all debugging activities, locating the fault is the most challenging one.

1.1 Motivation

There are two important concepts of checking whether the software contains bugs

Testing	Formal Verification
Where we make sure our code - as written - actually works the way it’s supposed to work	Checking whether the software design satisfies some requirements (properties)

1.1 Motivation

There are two important concepts of checking whether the software contains bugs

Testing	Formal Verification
Where we make sure our code - as written - actually works the way it's supposed to work	Checking whether the software design satisfies some requirements (properties)

1.2 Motivation - Goal

We want to check whether Test-Based Tarantula works better with Abstract reachability graph (ARG) than with test suites

Background

2.1 Background - Tarantula

<u>Insight</u>

– Program elements that are executed by failed test cases/Counterexample are more likely to be faulty than those that are executed by **passed** test cases/Safe paths.

<u>Solution</u>

– Make ranking for the program by giving probability for each code line based on **suspiciousness**.

$$\boxed{\textit{Suspicious(s)} = \frac{\frac{fail(s)}{totalfail}}{\frac{fail(s)}{totalfail} + \frac{pass(s)}{totalpass}}}$$

$\left\{ \text{We need at leat one fail(s) and one pass(s) to prevent divided by 0} \right\}$

2.1.2 Background -Tarantula Example Process of using Tarantula

$$\textit{Suspicious(s)} = \frac{\frac{fail(s)}{totalfail}}{\frac{fail(s)}{totalfail}+\frac{pass(s)}{totalpass}}$$

(1/1)/((1/1) +(5/5)) = 0.5

(1/1)/((1/1)+(1/5)) =0.83

`mid() {` `int x,y,z,m;`	Test Cases: 3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspiciousness	rank
`1: read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●	0.5	7
`2: m = z;`	●	●	●	●	●	●	0.5	7
`3: if (y<z)`	●	●	●	●	●	●	0.5	7
`4: if (x<y)`	●	●			●	●	0.63	3
`5: m = y;`		●					0.0	13
`6: else if (x<z)`	●				●	●	0.71	2
`7: m = y; // *** bug ***`	●					●	0.83	1
`8: else`			●	●			0.0	13
`9: if (x>y)`			●	●			0.0	13
`10: m = y;`			●				0.0	13
`11: else if (x>z)`				●			0.0	13
`12: m = x;`							0.0	13
`13: print("Middle number is:",m);`	●	●	●	●	●	●	0.5	7
`}` Pass/Fail Status	P	P	P	P	P	F		

2.3 Background - DStar and Ochiai

In our Evaluation we compared Tarantula against:

1. DStar Metric

$$\textbf{Suspicious(s)} = \frac{Failed(s)^{\delta}}{Passed(s)*(TotalFailed - Failed(s))}$$

We used ($\boldsymbol{\delta}$ = 2), the most efficient value

2. Ochiai Metric

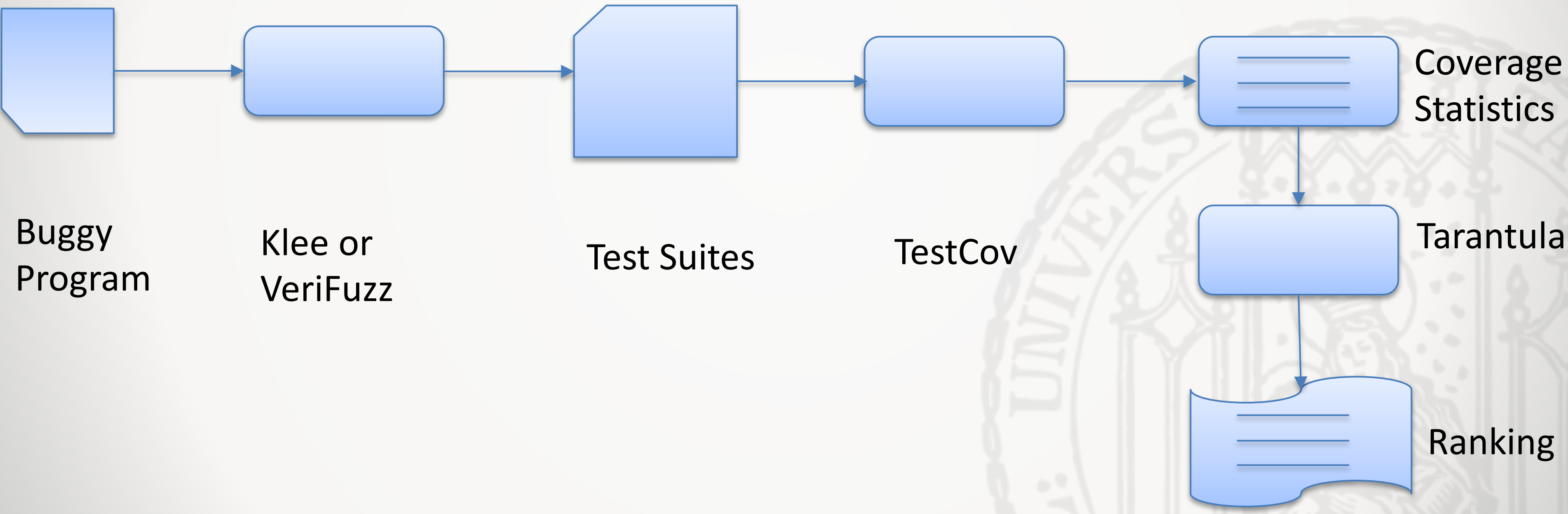
$$\textbf{Suspicious(s)} = \frac{Failed(s)}{\sqrt{TotalFailed*(Failed(s)+Passed(s))}}$$

Needs at least only one failed(s)

Implementation

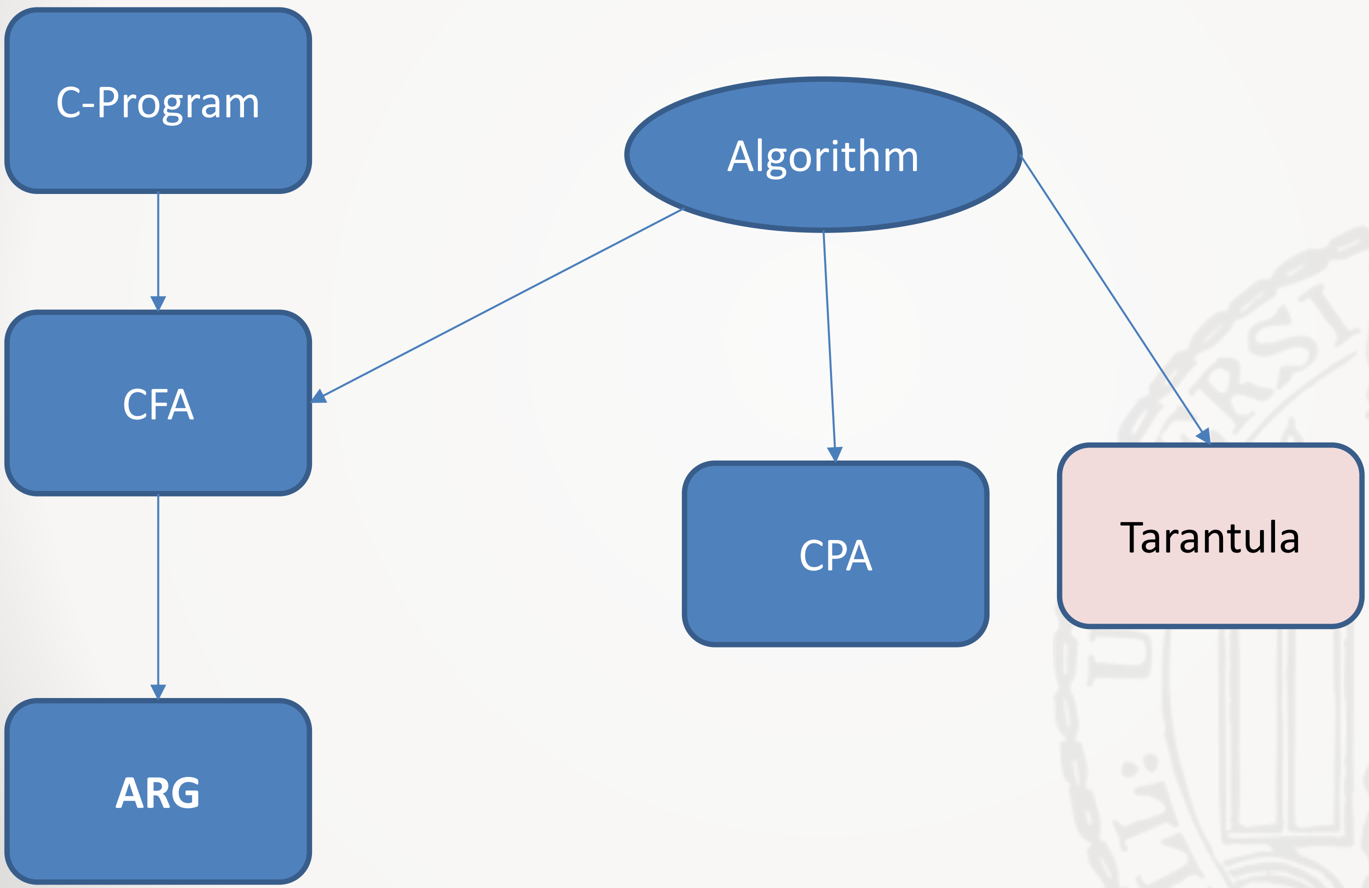
2.4.1 Implementation - Test-based Tarantula

How did we run Tarantula on test suites?



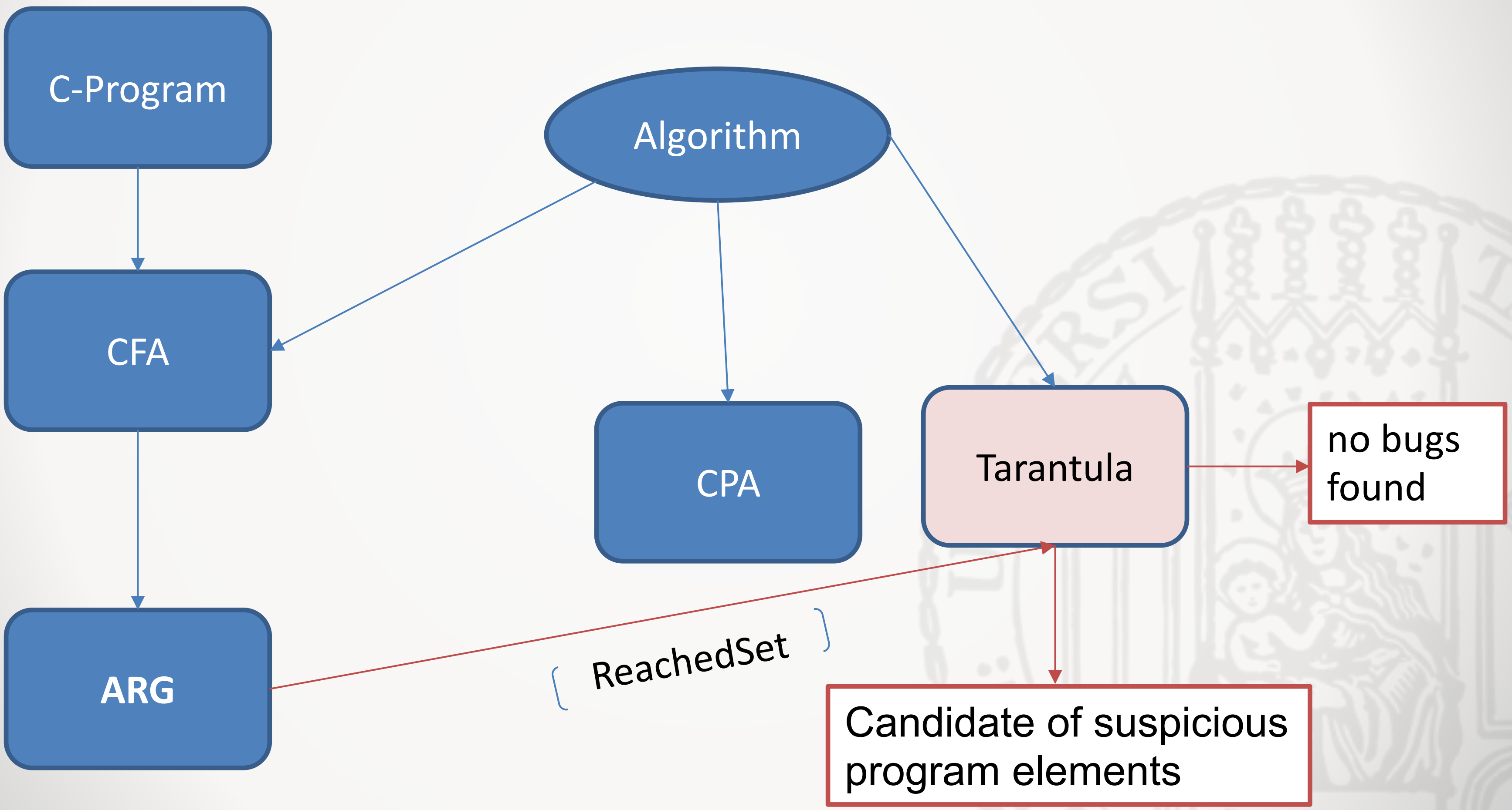
2.4.2 Implementation - Formal-based Tarantula

How did we run Tarantula in CPAchecker?



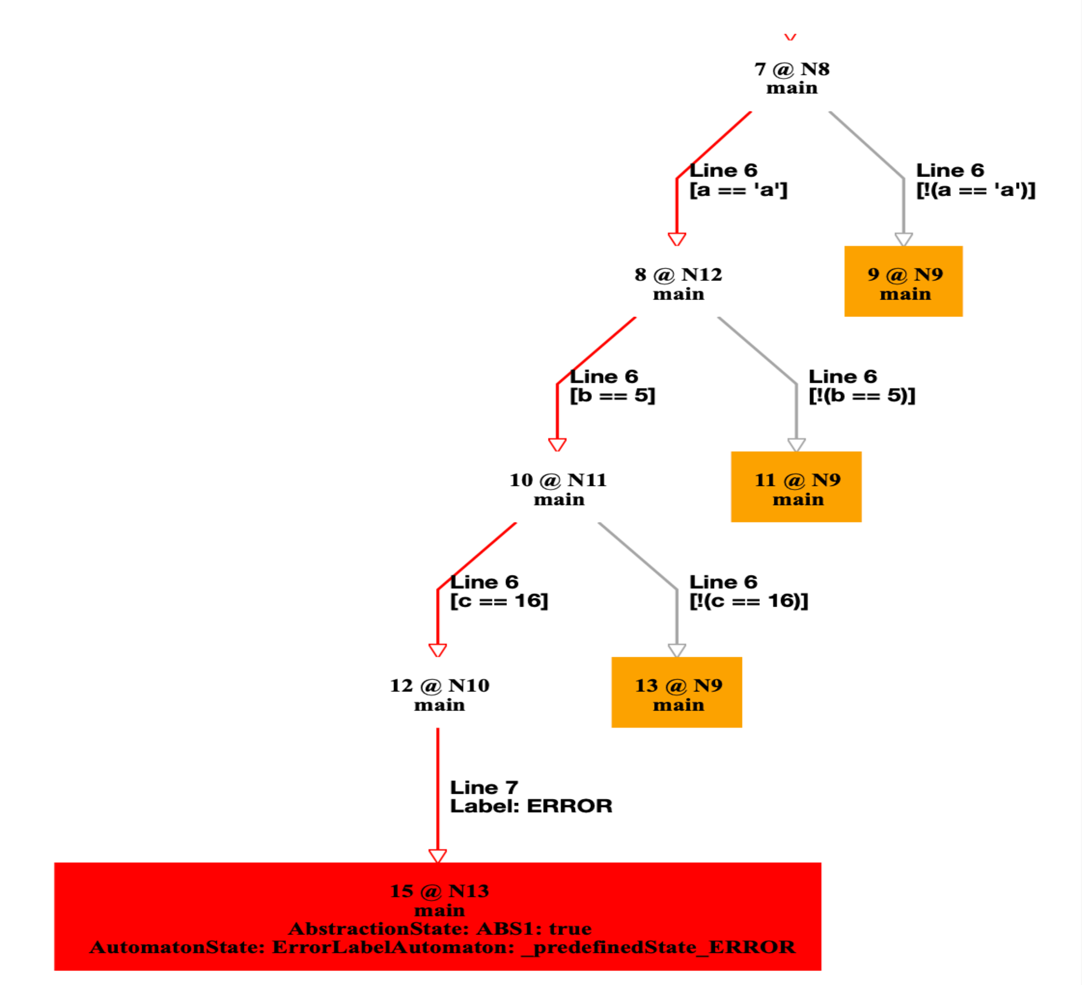
2.4.3 Implementation - Formal-based Tarantula

How did we run Tarantula in CPAchecker?



2.4.4.1 Tarantula on ARG - Example

1. Generating ARG



Example of ARG using Predicate Abstraction without meging the paths together

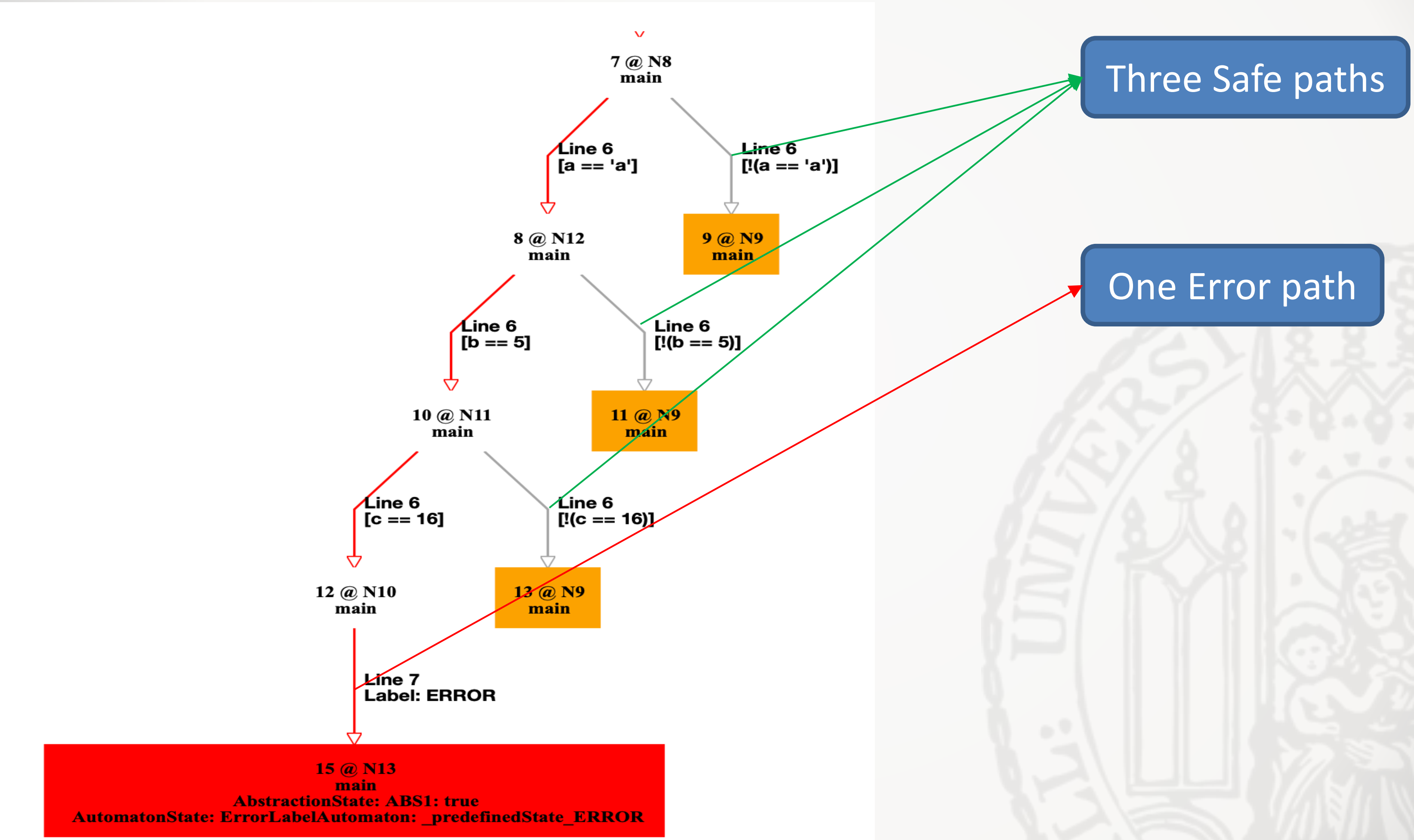
```
int main() {
char a = __VERIFIER_nondet_char();
char b = __VERIFIER_nondet_char();
char c = __VERIFIER_nondet_char();

if (a == 'a' && b == 5 && c == 16) {
ERROR:__VERIFIER_error();

            }
}
```

2.4.4.2 Tarantula on ARG - Example

2. Determine of Safe/fail paths

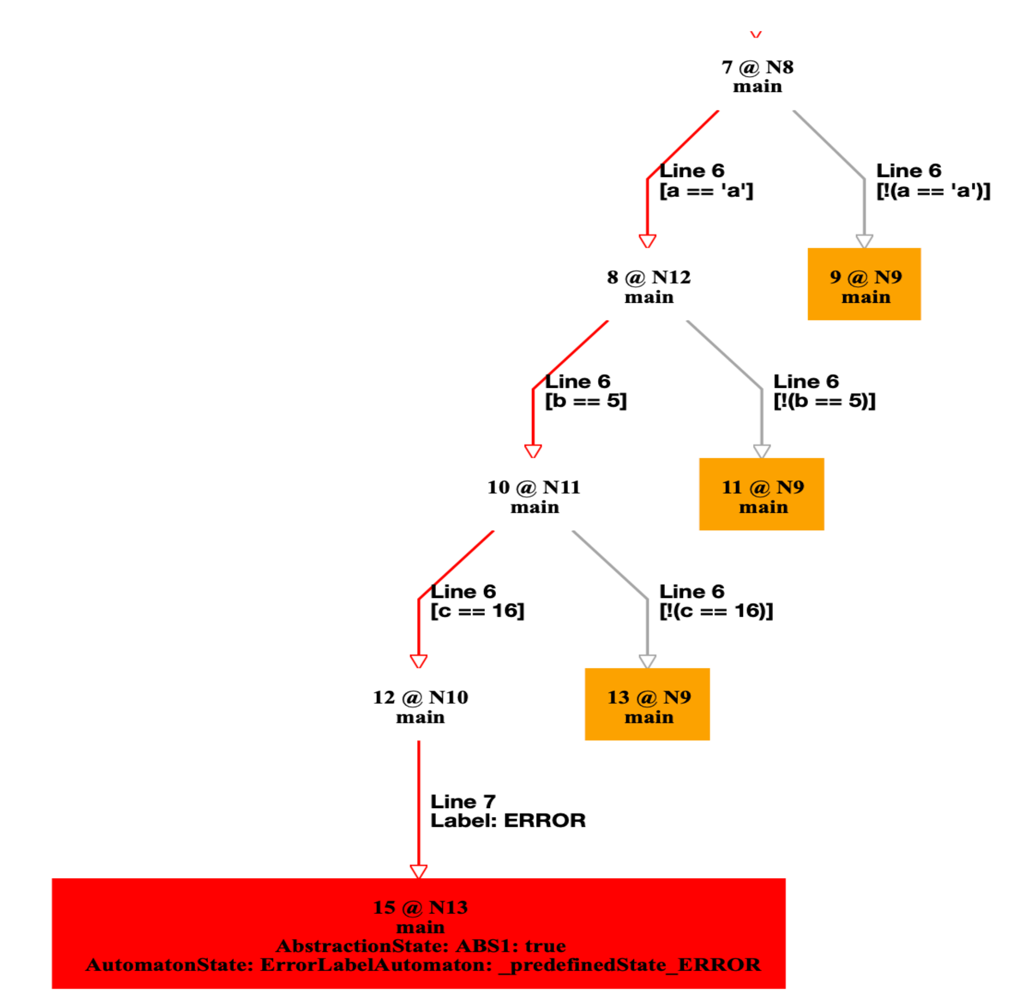


Example of ARG using Predicate Abstraction without meging the paths together

2.4.4.3 Tarantula on ARG - Example

3. Determine of Coverage for each CFAEdge

CFAEdge	Coverage	Suspicious
[(a==`a`)]	((S,2),(E,1))	
[!(a==`a`)]	((S,1),(E,0))	
[(b==5)]	((S,1),(E,1))	
[!(b==5)]	((S,1),(E,0))	
[(c==16)]	((S,0),(E,1))	
[!(c==16)]	((S,1),(E,0))	
ERROR	((S,0),(E,1))	



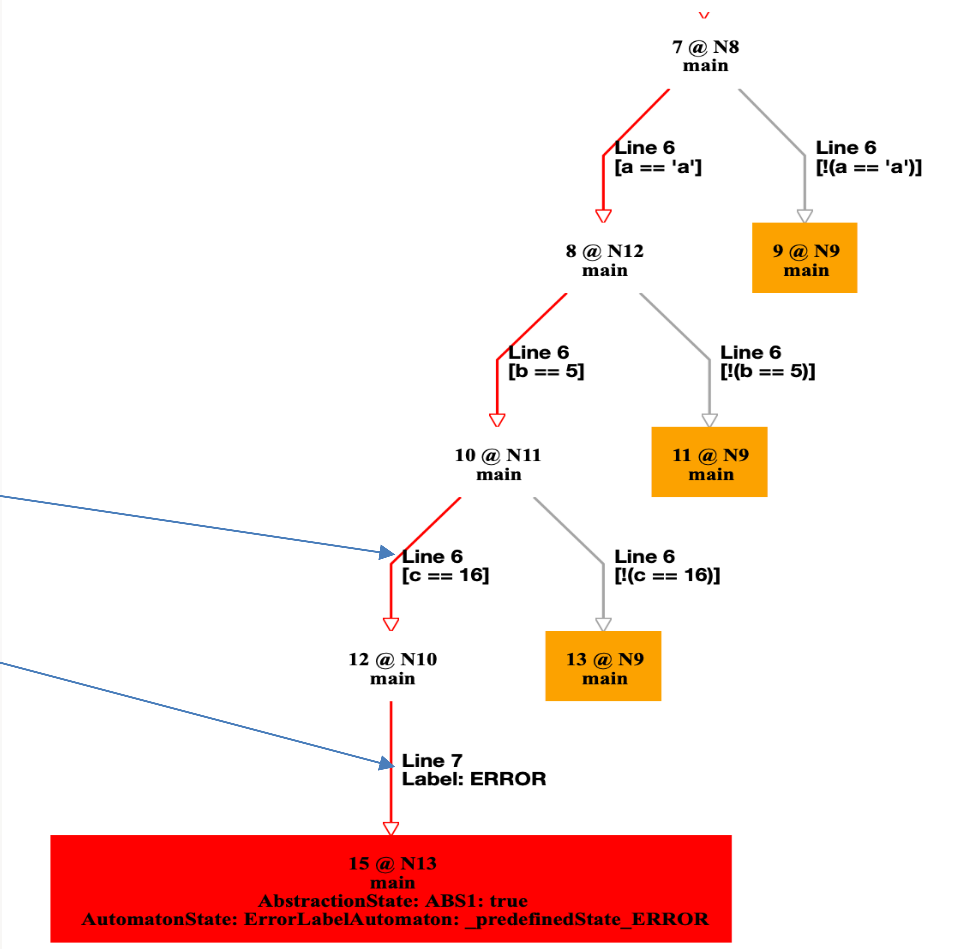
S: means Coverage of Safe paths
E: means Coverage of Error paths

2.4.4.4 Tarantula on ARG - Example

4. Calculate the Suspicious

$$Suspicious(s) = \frac{\frac{failPath(s)}{totalfailPaths}}{\frac{failPath(s)}{totalfailPaths} + \frac{safePaths(s)}{totalsafePaths}}$$

CFAEdge	Coverage	Suspicious
[(a==`a`)]	((S,2),(E,1))	0.75
[!(a==`a`)]	((S,1),(E,0))	0.0
[(b==5)]	((S,1),(E,1))	0.60
[!(b==5)]	((S,1),(E,0))	0.0
[(c==16)]	((S,0),(E,1))	1.0
[!(c==16)]	((S,1),(E,0))	0.0
ERROR	((S,0),(E,1))	1.0



S: means Coverage of Safe paths
E: means Coverage of Error paths

Evaluation

3.2 Evaluation - Setup

1. Sv-Benchmarks and Bekkouche Benchmarks
2. Omega evaluation metric
3. Predicate Abstraction with $merge^{sep}$
4. Symbolic Execution with CEGAR
5. Test Generators with Branch Coverage
6. BenchExec for Time Measurement
7. Time limit: 900 seconds
8. Memory limit: 4869 MB

3.2.1 Evaluation - Setup - Benchmarks

Benchmark-set consists of 35 programs. An overview of error type is as following:

Error Type	Explaitation of the Error
assign	Wrong assignment expression
op	Wrong operator usage e.g. : <=instead of <
init	Wrong value initialization of a variable
branch	Error in branching due to negation of branching condition
assign-for-loop	Wrong assignment inside loop
if-for-loop	Wrong check inside loop
index-for-loop	Use of wrong array index
index-while	Use of wrong array index inside while loop

This type of bug is taken from BugAssist's evaluation

3.2.2 Evaluation - Evaluation Metric

Worst-Case step

$$worst - case - step = |\{codeline; rank(codeline) \leq rank(faulty\ codeline)\ \&\&\ codeline\ \ != \ faulty\ codeline\}|$$

cardinality of a set of code lines, whose rank is less than or equal to the rank of the actual error code line and this set should not contain any faulty code line.

Omega Percentage = Worst-Case step / Total Code-Lines

3.2.2.1 Evaluation - Evaluation Metric - Example

codeLine	suspicious	rank
5	1.0	1
6	1.0	1
1	0.5	2
16	0.5	2
2	0.5	2
11	0.5	2
14	0.5	2
4	0.5	2
12	0.5	2

Worst-Case step = |{5, 6, 1, 2, 11, 14, 4, 12}|=8

Omega percentage = 8/20 = 0.400

The lower the omega result the better the technique

$$worst-case-step = |\{codeline; rank(codeline) \leq rank(faulty\ codeline)\ \&\&\ codeline\ != faulty\ codeline\}|$$

3.2.3 Evaluation - Setup - merge operator

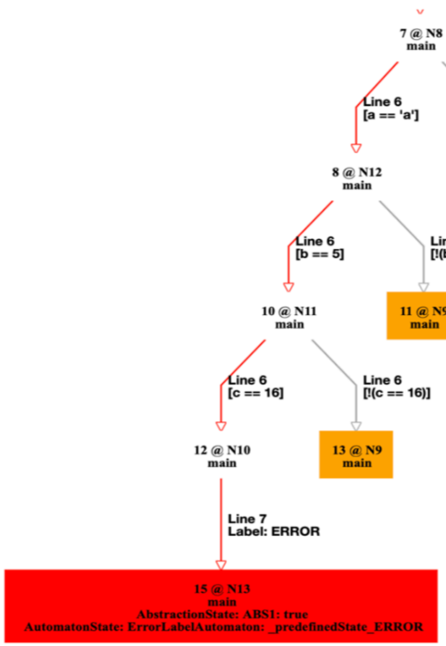


Figure 6.3: $merge_{sep}$

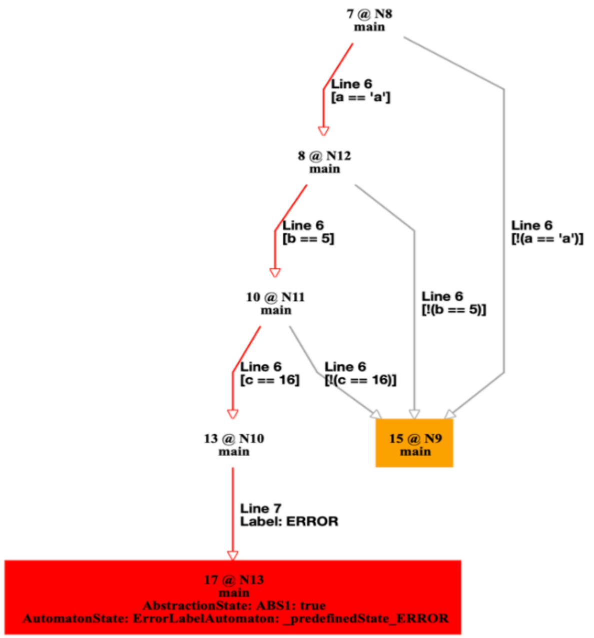


Figure 6.4: Default Merge

Figure 6.5: Comparison between ARGs created by PredicateCPA with $merge_{sep}$ (left) and with default merge (right) operators generated by CPAchecker

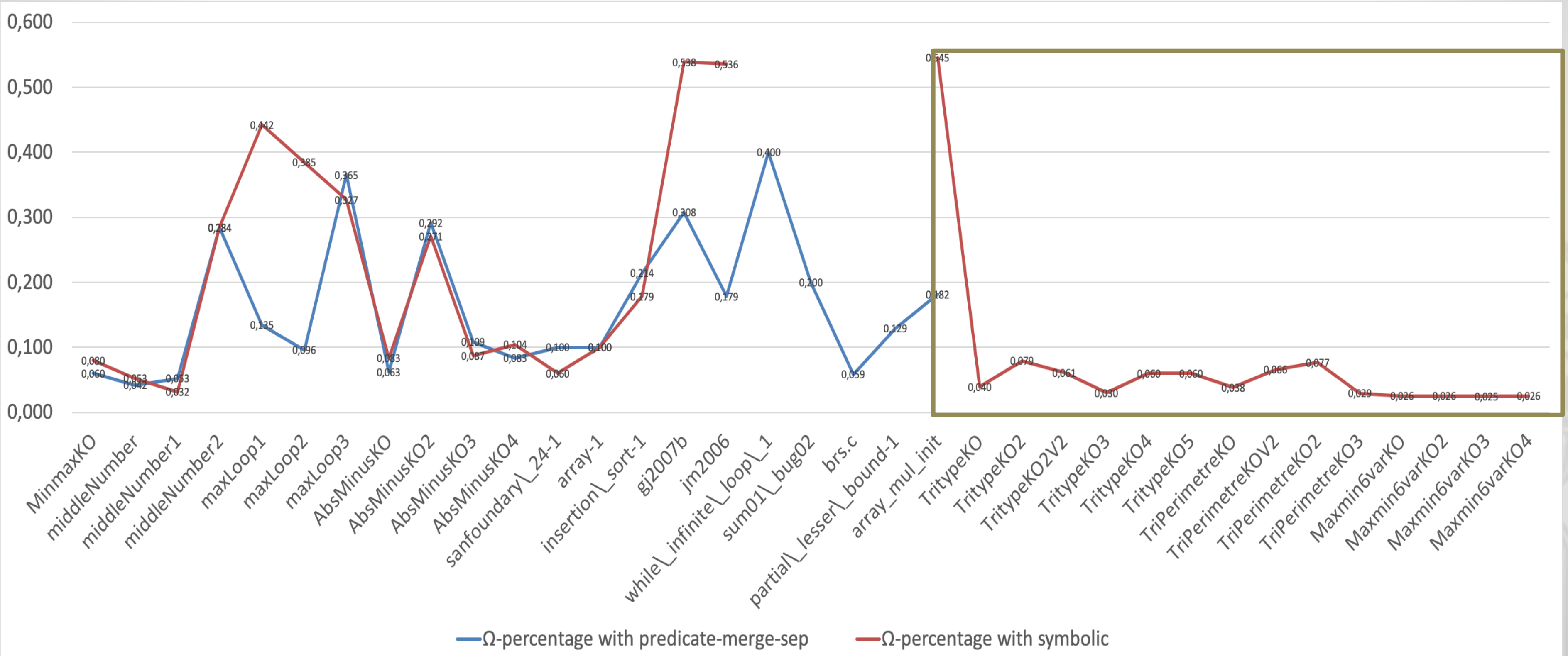
With default merge we get for all merged paths as suspicious value 0.5, therefore we use $merge^{sep}$

3.3.1 Evaluation - Overview

1. Tarantula SymExec vs Predicate Abstraction
2. Tarantula vs DStar and Ochiai
3. Test-Based Tarantula vs Formal-based Tarantula

3.4.1 Evaluation - Discussions - Symbolic vs Predicate

Symbolic Execution is better than predicate analysis with mergeSEP

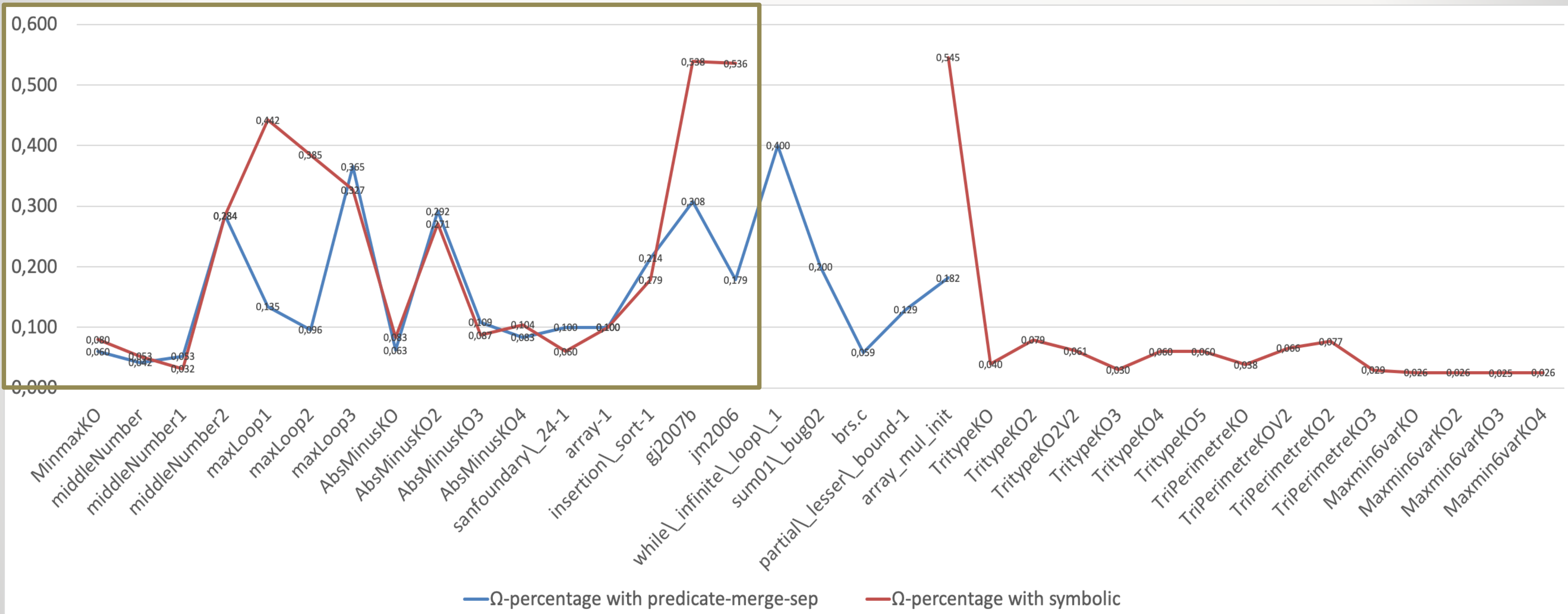


Reasons:

1. ARG of Predicate is merged together so we need to apply mergSEP which is very expensive and slows down the analysis, and even runs the analysis for certain large programs infinitely.

3.4.1 Evaluation - Discussions - Symbolic vs Predicate

Symbolic Execution is better than predicate analysis with mergeSEP

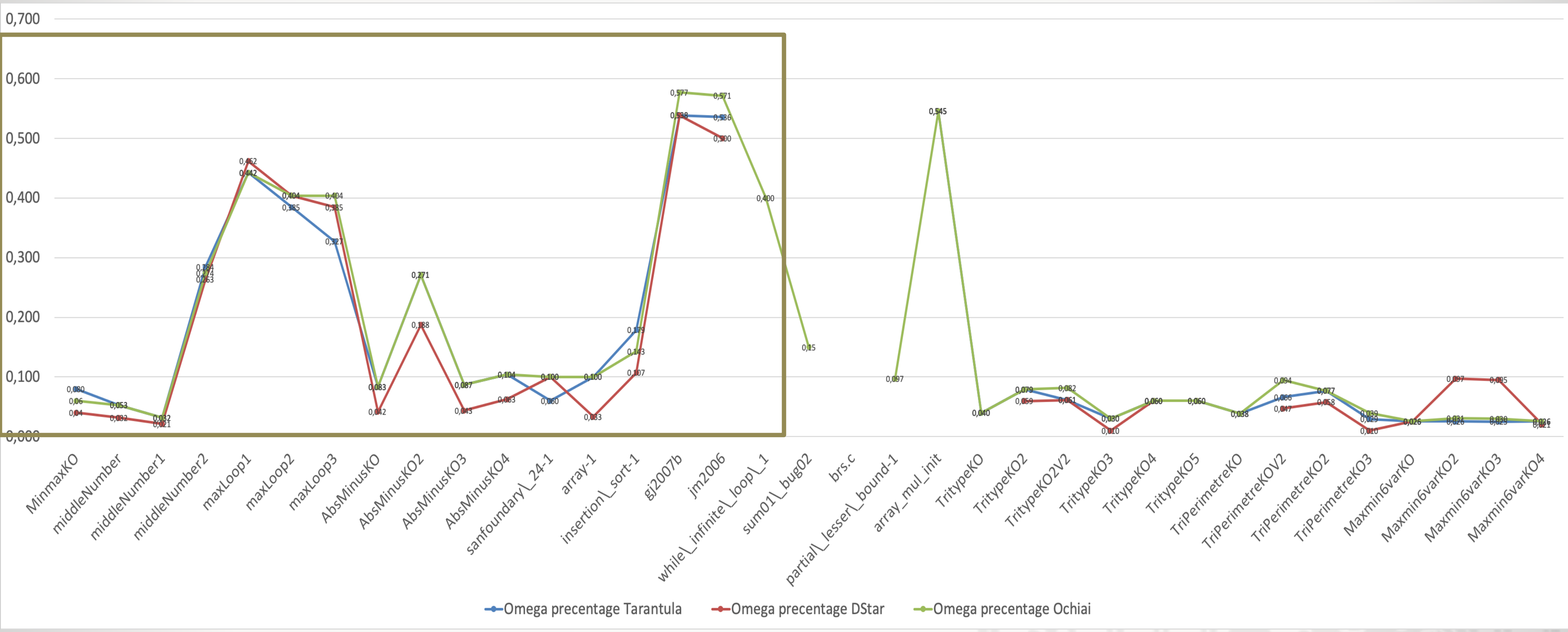


Reasons:

2. ARG Graph from Predicate analysis is constructed in such a way that the bug location is more often on the safe path than on the failed path which lowers the suspicious.

3.4.2 Evaluation - Discussions -Tarantula vs DStar and Ochiai

DStar is better than Ochiai and Tarantula

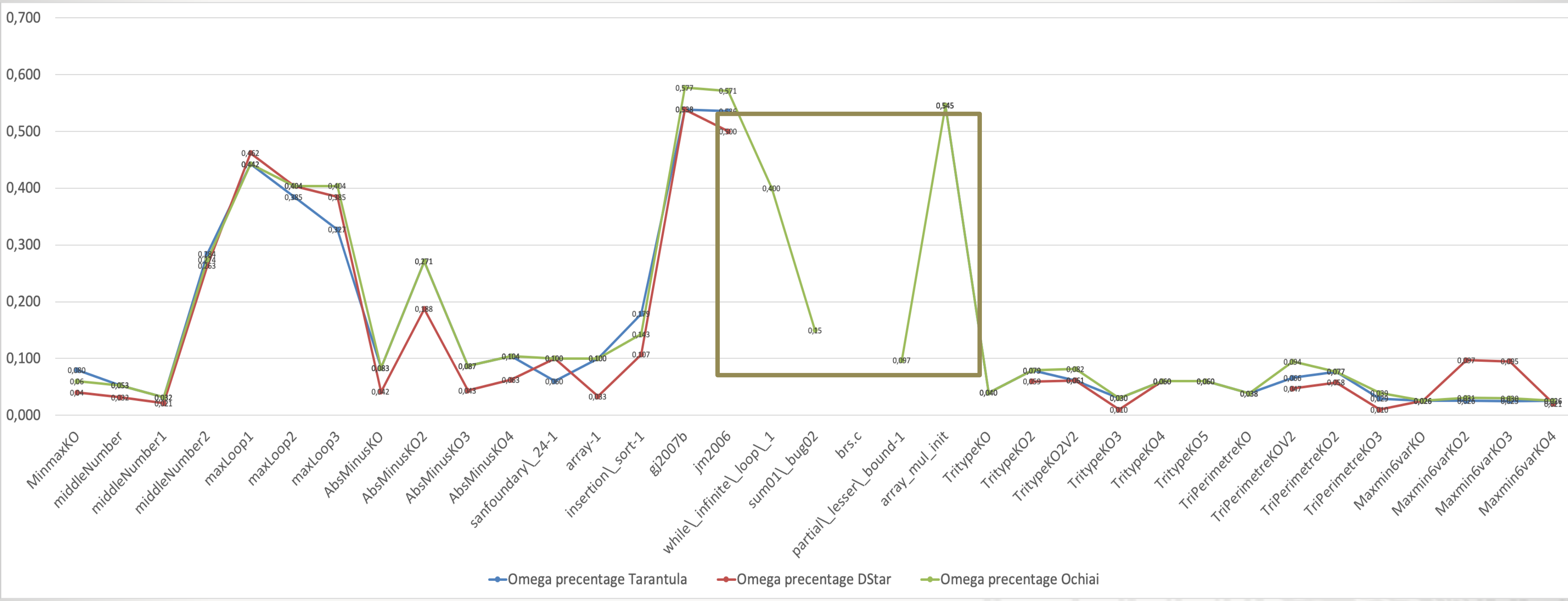


Reasons:

DStar does not take TotalSafePaths into account in its suspicious form and with the help of the delta exponential variable, the suspicion of the fault position was increased

3.4.2 Evaluation - Discussions -Tarantula vs DStar and Ochiai

Ochiai is better than Tarantula

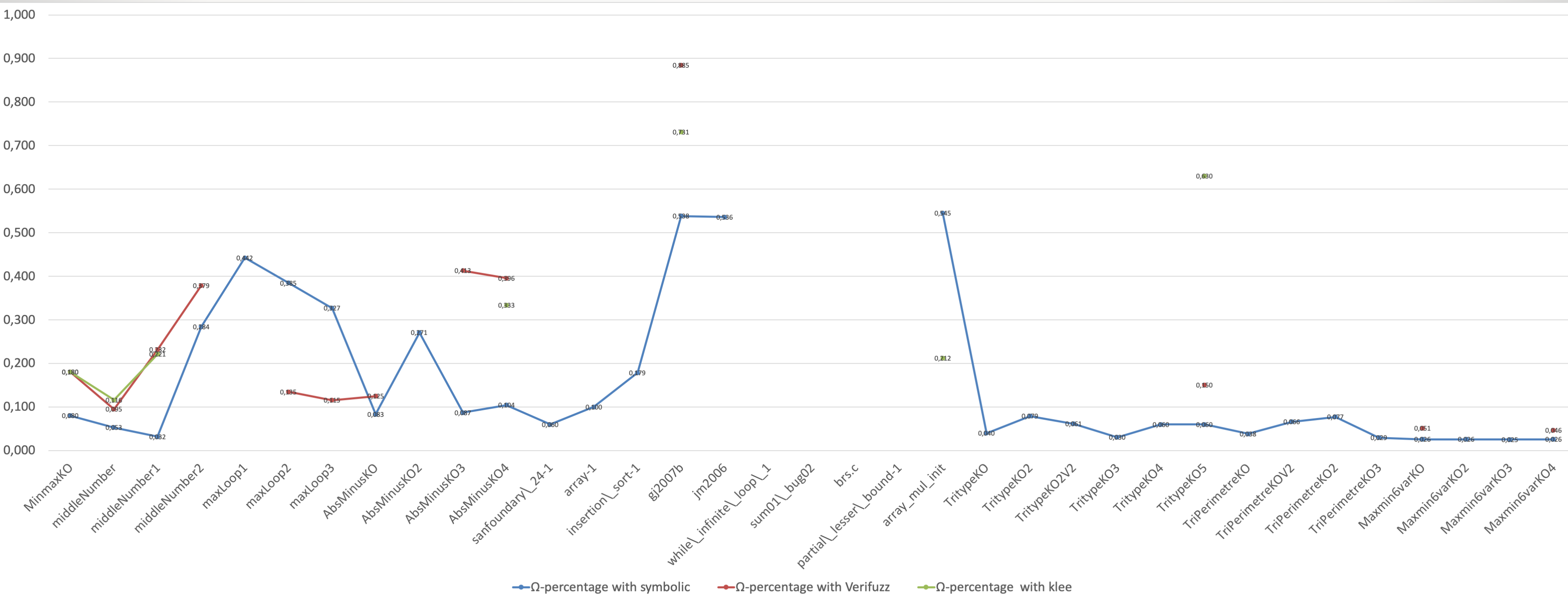


Reasons:

Ochiai's Ω percentage was almost the same as Tarantula's, but Ochiai analyzed more test programs than Tarantula. The reason for this is that Ochiai does not need at least one failure path and at least one safe path in contrast to Tarantula.

3.4.3 Evaluation - Discussions - Formal-based vs Test-based Tarantula

Formal-based is better than test-based Tarantula



Reasons:

Klee and VeriFuzz very often generated bad analyse through the whole program, so the bug sometimes suspected 0.0. Quite often both techniques delivered only counterexamples but no safe cases, so Tarantula can work perfectly well, thus the suspicious is 1.

Future Work

4.1 Future Work

Future work should include:

1. The use of more advanced fault localization analysis on CPAchecker to choose the best fault localization technique or to design a new ranking method and use it as a default feature in CPAchecker.

2. The work on more ranking metrics, such as Barinel and Op2 is still open and can be analysed.

3. Improving CPAchecker to be able to not only analyse C-programs but also java and Java Script programs.

Conclusion

4.2 Conclusion

- DStar and Ochiai are improvements and work better than Tarantula
- Symbolic execution was able to identify potential faults, 88.57% of the chosen benchmarks with a very good percentage of Ω, while predicate-merge-set found 60% of the total benchmarks with very good results from Ω
- Klee was only successful in 17.14% of all benchmarks used
- VeriFuzz was better than Klee but not CPAchecker in 37.14%

⇒ In our experimental Evaluation:
Techniques such as model checking and data flow analysis can find subtle and more bugs in programs as test generators.

References

- J. A. Jones and M. J. Harrold. "**Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique**." In: (2005).

- W. E. Wong, V. Debroy, R. Gao, and Y. Li. "**The DStar Method for Effective Software Fault Localization**." In: IEEE Transactions on Reliability 63.1 (2014),pp. 290–308

- M. A. Ali pour. "**Automated fault localization techniques: a survey.**" In:(2012), pp. 6–7.

- D. Beyer and E.Keremoglu " **CPAchecker: A Tool for Configurable Software Verification**". 2011, pp. 184–190

- D. Beyer and T. Lemberger. "**TESTCOV: Robust Test-Suite Execution and Coverage Measurement**." In: (2019)

References

- M. Jose and R. Majumdar. “**Cause Clue Clauses: Error Localization using Maximum Satisfiability**” In: (2002), pp. 49, 76

Available Sources

The used benchmark-set, evaluation data and python script of test-based tarantula algorithm are available under:

https://gitlab.com/Schindar/fault_localization_tarantula

Thank you for your Attention
Questions?