Metamorphic Testing for Verification and Fault Localization in Industrial Control Systems

VeriDevOps@Åbo

Åbo Akademi University

VeriDevOps

Motivation

- **Objective**
  - **Ensure software systems are resilient against cyberattacks**
  - **Identifying and remediating vulnerabilities in software systems to mitigate the risk of possible exploitation**
  - Integrating the verification and vulnerability localization in the DevOps lifecycle to maintain cyber resilience

Åbo Akademi University

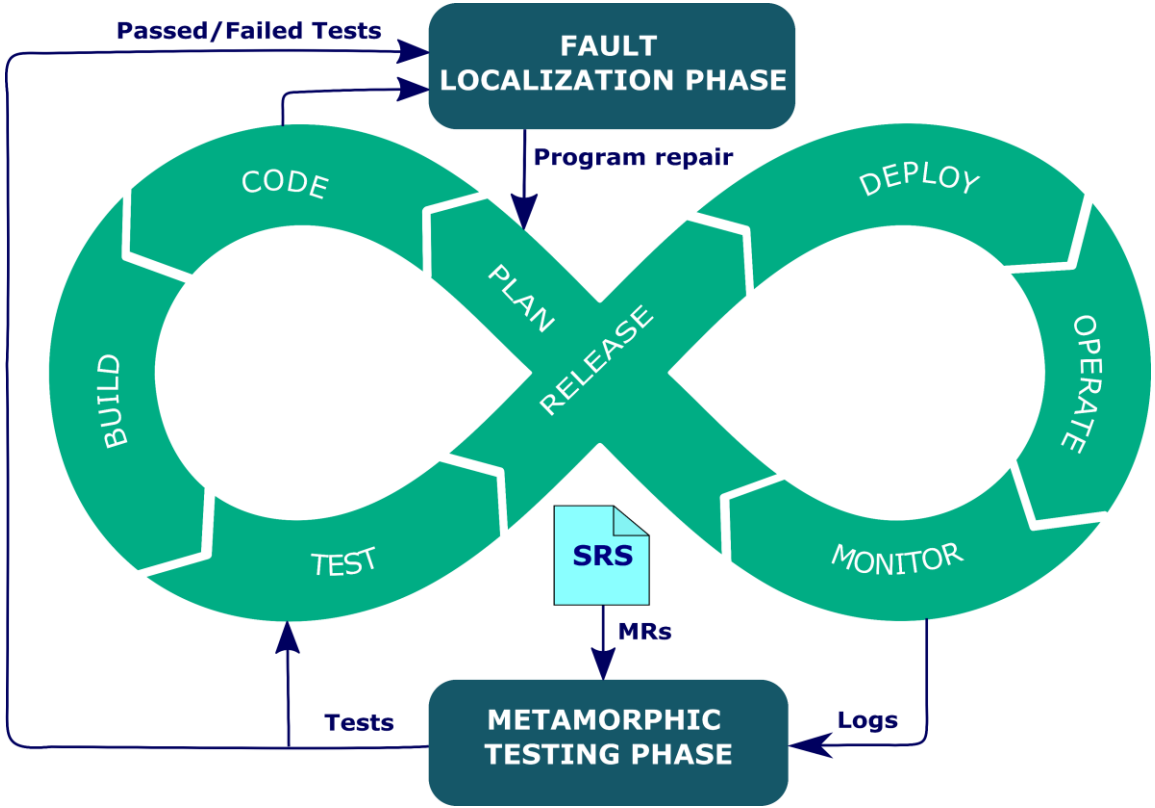
VeriDevOps

Methodology

- Efficient verification of software systems that lacks an explicit test oracle → **Metamorphic testing phase**

- To identifying vulnerabilities and perform root cause analysis in software systems → **Fault localization phase**

Overview


Passed/Failed Tests
FAULT LOCALIZATION PHASE
Program repair
CODE
PLAN
DEPLOY
BUILD
RELEASE
OPERATE
TEST
MONITOR
SRS
MRs
Tests
METAMORPHIC TESTING PHASE
Logs

Metamorphic testing

- Why are certain programs considered non-testable ?
  - Lack of an explicit test oracle
  - Complexity
  - Vast input space

Metamorphic testing

- An approach to create follow-up tests from existing tests
- Can be used to uncover underlying errors using metamorphic relations
- Metamorphic relations can be defined based on system properties

Metamorphic testing steps

1) Define the **metamorphic relations** for testing the system

2) Extract/generate **seed input**

3) Create **morphed input** by applying the **morphing transformation**

4) Check the **metamorphic relation** between **seed output** and **morphed output**

Fault localization

- Spectrum-based fault localization
- Program slicing

Spectrum-based fault localization

- Locates likely faulty program elements using program spectra

  - Collects run-time measurements using program spectra such as BHS, BCS
  - Compares two sets of execution traces
  - Passed and Failed

Program slicing

- Focus on analyzing a **slice** (relevant part) that may contain a fault
- **Metamorphic slicing :** A slice extracted using **execution slicing** and **dynamic slicing** of a metamorphic test group

Case study

Input	Output
• **Number of markers** • **Marker coordinates : (x, y)** • **Position & Size** • **Hoist position** • Marker's relative position to head block's center • Position of Trolley, Gantry and Hoist relative to camera mounting	• **Valid_Markers** (BOOL) • **True Marker coordinates** • **True Marker index** • **Scores (For top marker and low marker pair)**

Metamorphic testing steps

- Define the **metamorphic relations** for testing the system
  - **Property :**

"If the system is able to classify a set of positional markers detected by the camera module as true markers in the absence of reflections (noise), the system should be able to classify correctly the same positional markers in the presence of reflections"

  - **Input MR :** $\boldsymbol{seed \rightarrow seed + noise}$
  - **Output MR:** $\boldsymbol{O(seed) \equiv O(seed + noise)}$

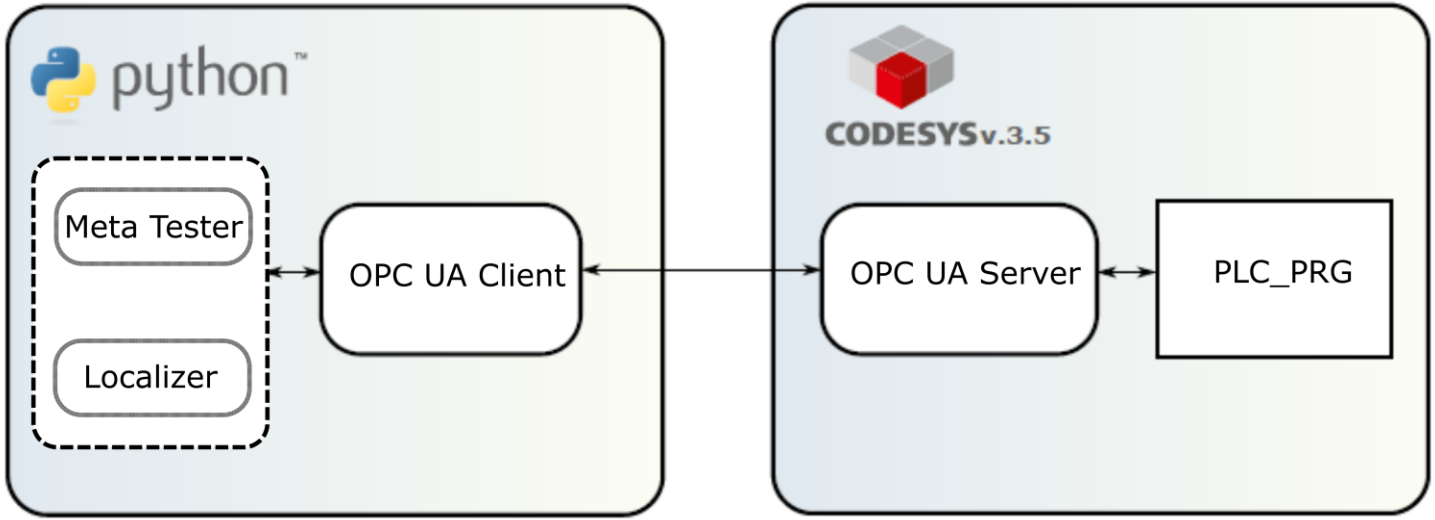
Two phase metamorphic testing

- **Exploration**: Random generation of input patterns to create morphed input

- **Exploitation**: Targeted testing using failure inducing patterns discovered in exploration phase

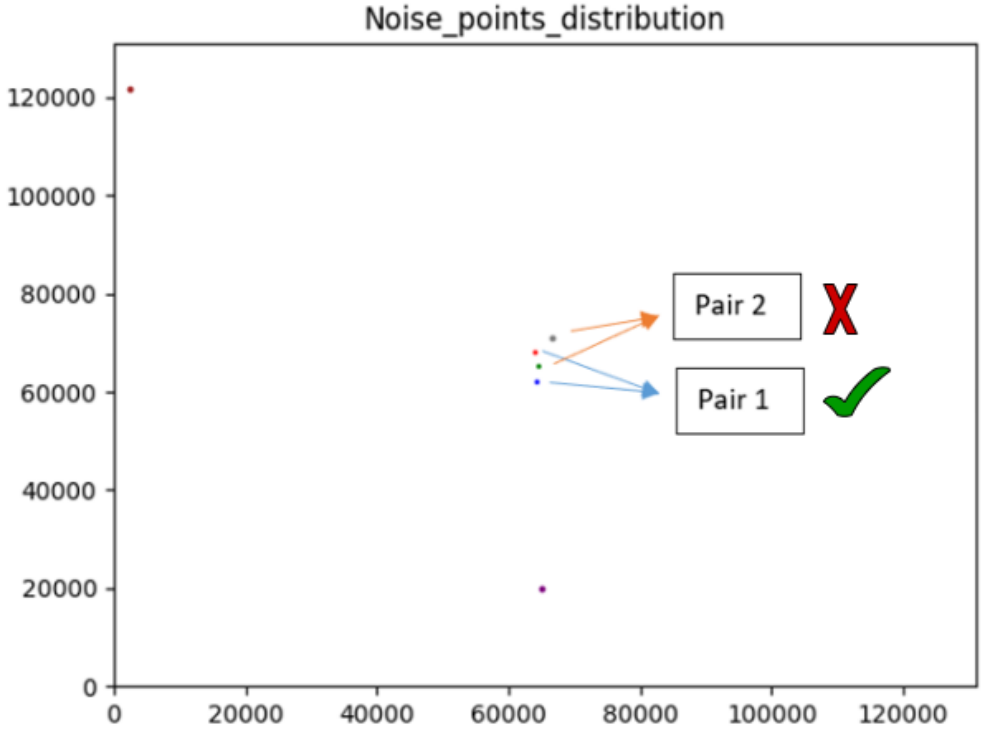
Test execution

- Execute the tests and assign the test verdict based on MR check
- Collect passed and failed tests



Metamorphic test results

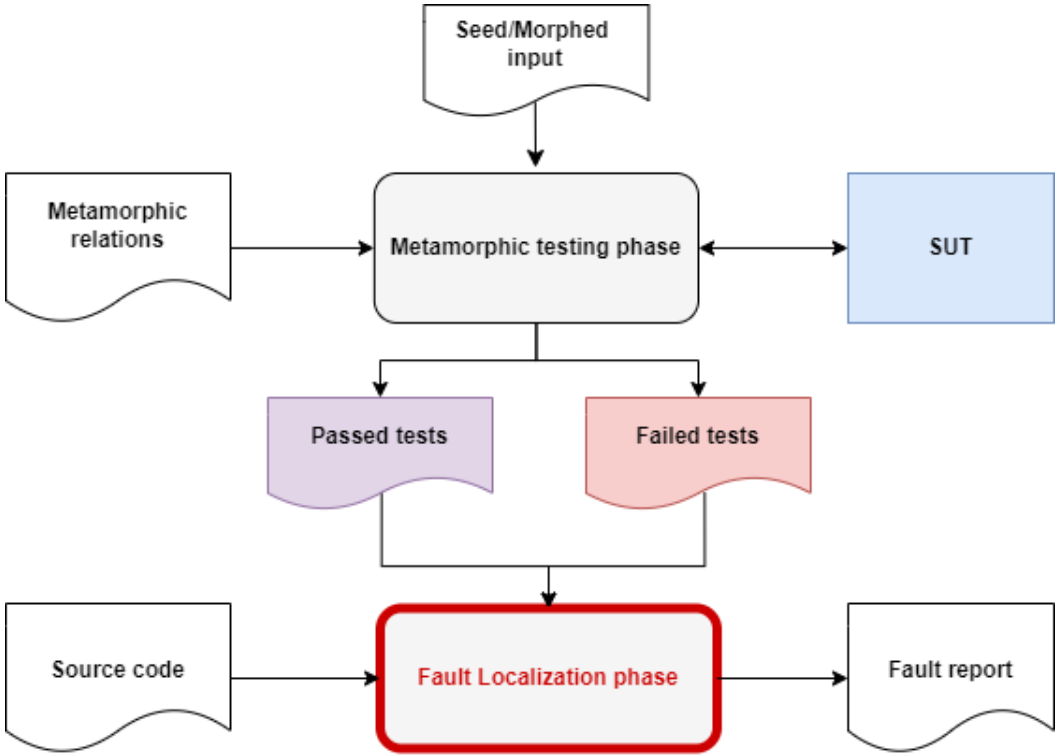
- 3 categories
  - Correct identification
  - Incorrect identification
  - Missed identification



Åbo Akademi University

VeriDevOps

Approach – fault localization



Åbo Akademi University

VeriDevOps

Instrumentation for program spectra

```
1:  x := in_var1;
2:  y := in_var2;
3:  z := in_var3;
4:  FOR i:=1 TO ABS(x) DO
5:    IF x >= 0 THEN
6:      rxy := rxy + y;
7:    ELSE
8:      rxy := -(rxy + y);
9:    END_IF
10: END_FOR
11: FOR j:=1 TO ABS(z) DO
12:   IF z >= 0 THEN;
13:     rxyz := rxyz + rxy;
14:   ELSE
15:     rxyz := rxyz + rxy;
16:   END_IF
17: END_FOR
18: out_product := rxyz;
```

```
1:  x := in_var1;
2:  y := in_var2;
3:  z := in_var3;
4:  FOR i:=1 TO ABS(x) DO
5:    IF (flag)  THEN c[1] := c[1] + 1; END_IF
6:    IF x >= 0 THEN
7:      IF (flag)  THEN c[2] := c[2] + 1; END_IF
8:      rxy := rxy + y;
9:    ELSE
10:     IF (flag)  THEN c[3] := c[3] + 1; END_IF
11:     rxy := -(rxy + y);
12:   END_IF
13: END_FOR
14: FOR j:=1 TO ABS(z) DO
15:   IF (flag)  THEN c[4] := c[4] + 1; END_IF
16:   IF z >= 0 THEN;
17:     IF (flag)  THEN c[5] := c[5] + 1; END_IF
18:     rxyz := rxyz + rxy;
19:   ELSE
20:     IF (flag)  THEN c[6] := c[6] + 1; END_IF
21:     rxyz := rxyz + rxy;
22:   END_IF
23: END_FOR
24: out_product := rxyz;
```

Test execution

- Execute passed/failed tests against instrumented code
- Collect program spectra (Branch Count spectra)
- Calculate suspiciousness scores

Suspiciousness scores

- ef: the number of times a statement is executed (e) in failed tests
- ep: the number of times a statement is executed (e) in passed tests
- nf: the number of times a statement is not executed (e) in failed tests
- np: the number of times a statement is not executed (e) in passed tests

Suspiciousness metric	Formula
Ochiai	$\frac{ef}{\sqrt{(ef+nf)\cdot(ef+ep)}}$
Jaccard	$\frac{ef}{(ef+ep+nf)}$
Tarantula	$\frac{\frac{ef}{(ef+nf)}}{\frac{ef}{(ef+nf)}+\frac{ep}{(ep+np)}}$

Suspiciousness elements extraction

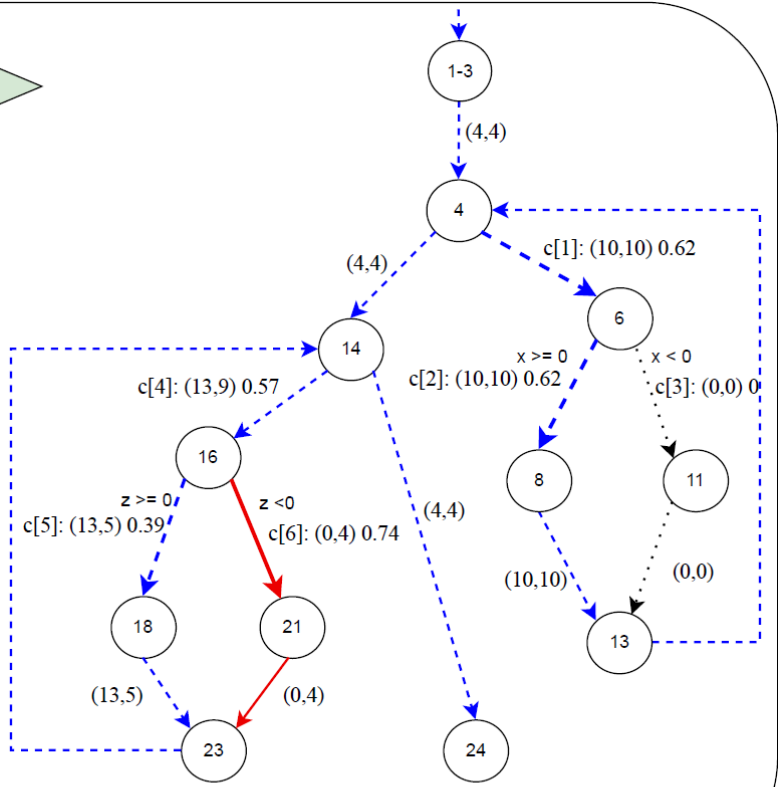
- Uses an average score to identify suspicious elements
  - $s_{avg} = (s_{Ochiai} + s_{Jaccard} + s_{Tarantula}) / 3$
- Suspicious statements are extracted from the **metamorphic slices**
  - $Set\ of\ statements\ in\ the\ execution\ trace\ of\ a\ failed\ metamorphic\ test$
  - $Set\ of\ variables\ with\ highest\ scores\ and\ the\ statements\ in\ which\ those$
  - $variables\ are\ defined\ and\ used$

Control flow graph

Instrumented ST code

CFG

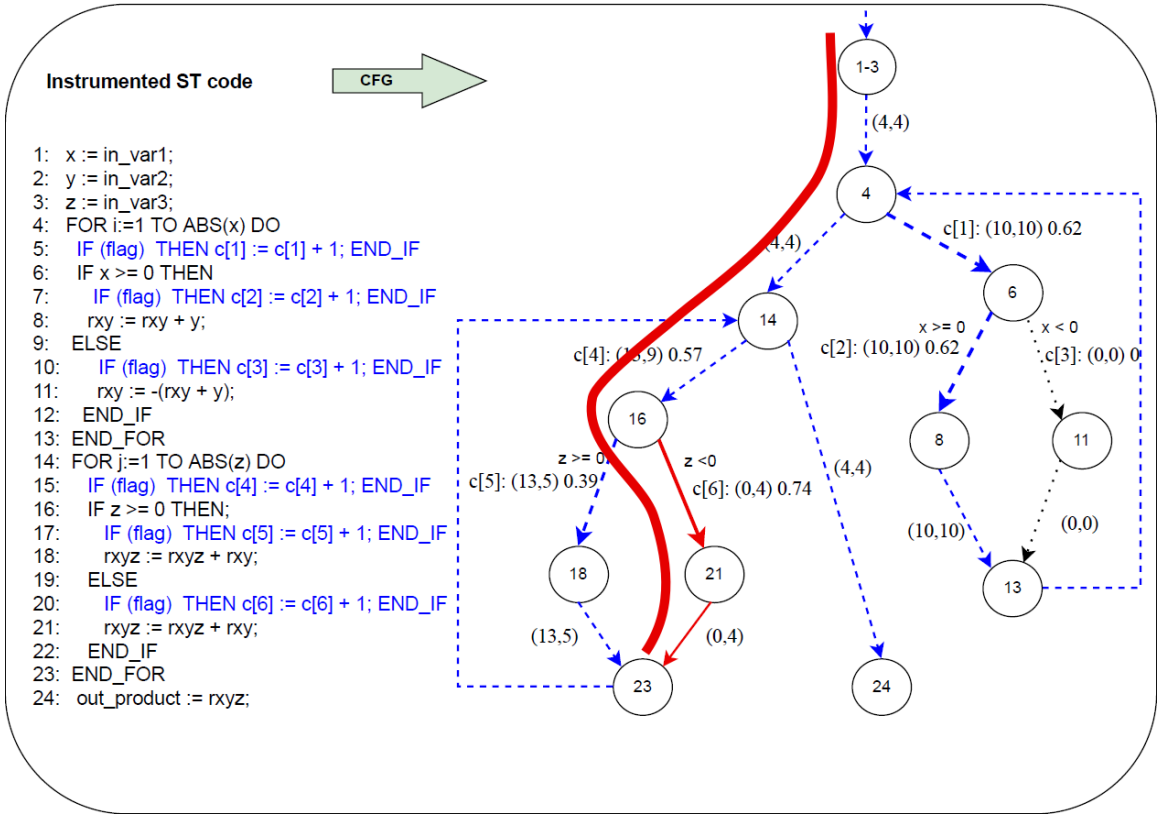
```
1:  x := in_var1;
2:  y := in_var2;
3:  z := in_var3;
4:  FOR i:=1 TO ABS(x) DO
5:    IF (flag)  THEN c[1] := c[1] + 1; END_IF
6:    IF x >= 0 THEN
7:      IF (flag)  THEN c[2] := c[2] + 1; END_IF
8:      rxy := rxy + y;
9:    ELSE
10:     IF (flag)  THEN c[3] := c[3] + 1; END_IF
11:     rxy := -(rxy + y);
12:   END_IF
13: END_FOR
14: FOR j:=1 TO ABS(z) DO
15:   IF (flag)  THEN c[4] := c[4] + 1; END_IF
16:   IF z >= 0 THEN;
17:     IF (flag)  THEN c[5] := c[5] + 1; END_IF
18:     rxyz := rxyz + rxy;
19:   ELSE
20:     IF (flag)  THEN c[6] := c[6] + 1; END_IF
21:     rxyz := rxyz + rxy;
22:   END_IF
23: END_FOR
24:  out_product := rxyz;
```



Åbo Akademi University

VeriDevOps

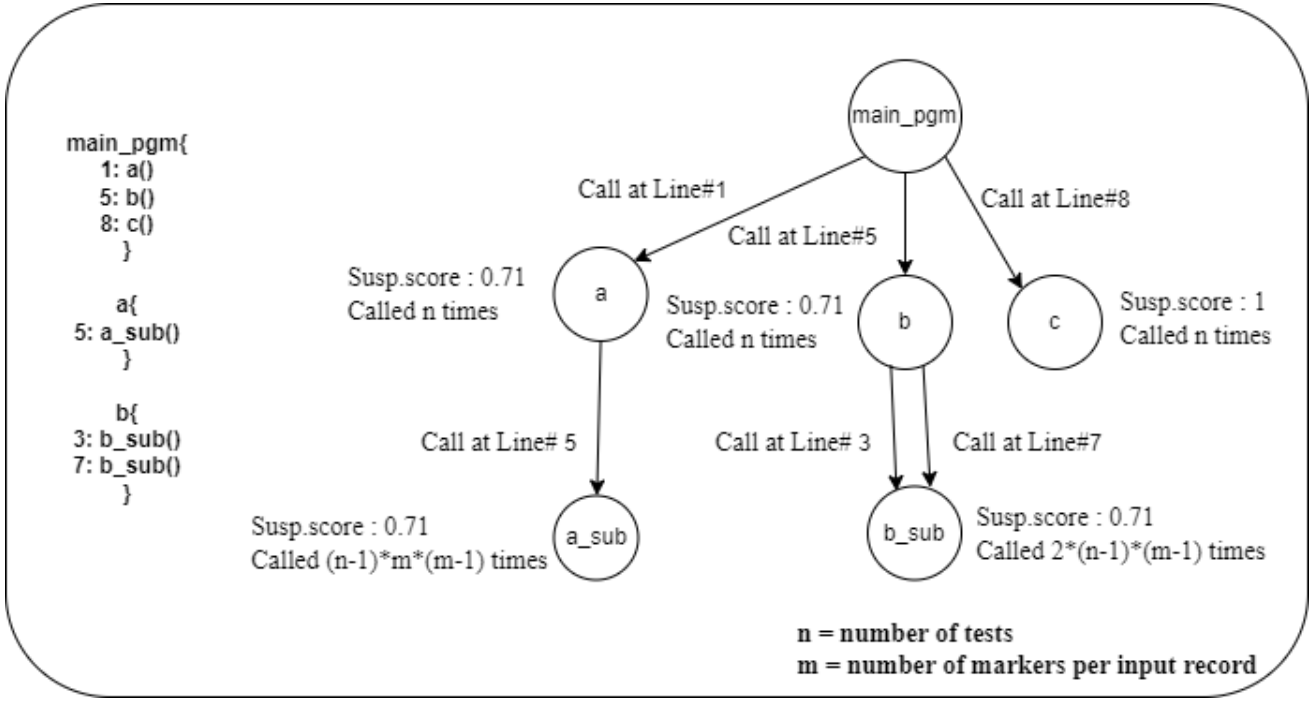
Control flow graph



Data flow analysis (1)

- Definition-usage of the variables with high suspicious scores are extracted
- Analyze definition-use chain of suspicious variables from the metamorphic slices to localize the fault (manually at the moment)

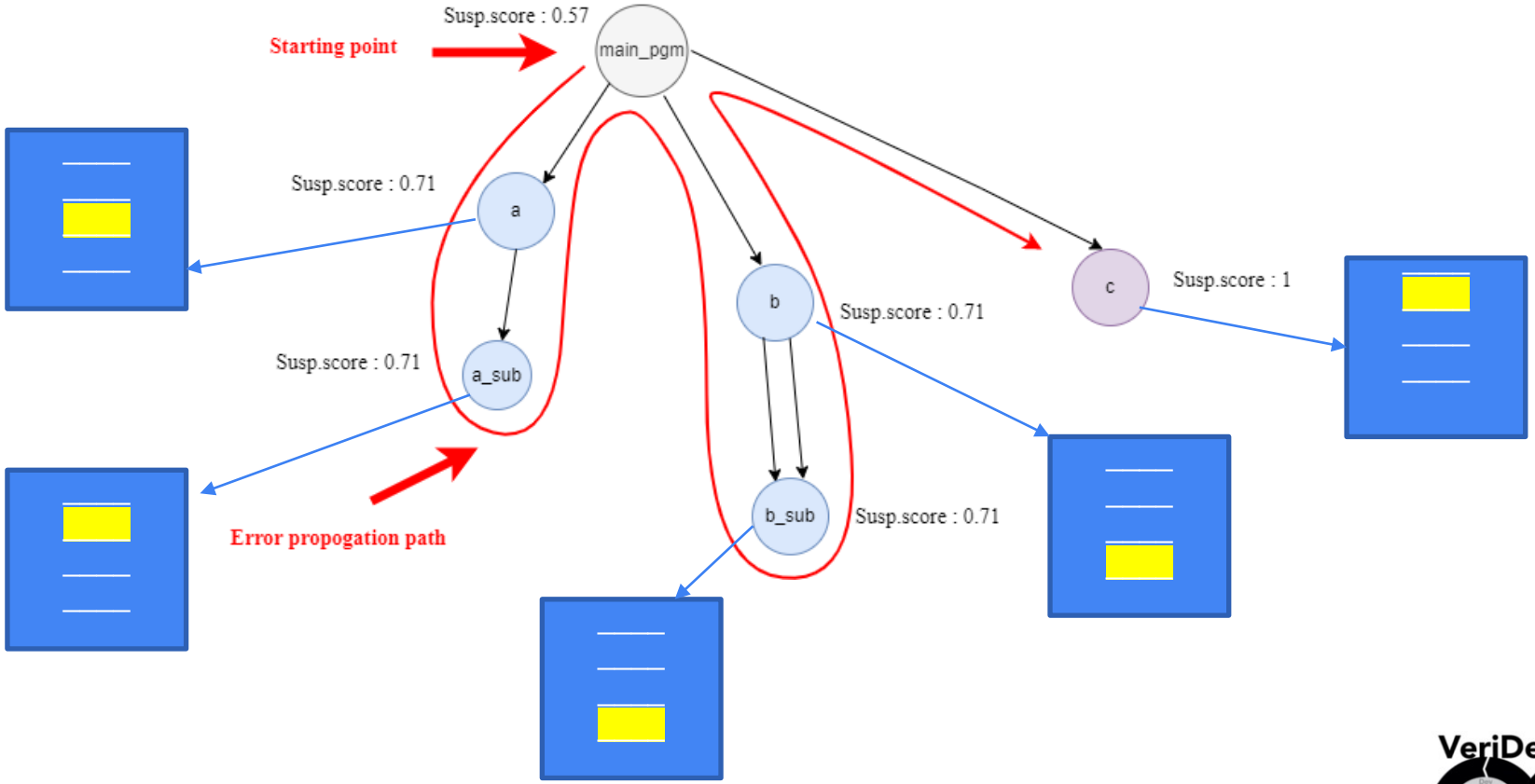
Call graph


main_pgm{
1: a()
5: b()
8: c()
}
a{
5: a_sub()
}
b{
3: b_sub()
7: b_sub()
}
main_pgm
Call at Line#1
Call at Line#5
Call at Line#8
Susp.score : 0.71
Called n times
a
Susp.score : 0.71
Called n times
b
c
Susp.score : 1
Called n times
Call at Line# 5
Call at Line# 3
Call at Line#7
Susp.score : 0.71
Called (n-1)*m*(m-1) times
a_sub
b_sub
Susp.score : 0.71
Called 2*(n-1)*(m-1) times
n = number of tests
m = number of markers per input record

Data flow analysis (2)

- Def-use chain analysis of variables with highest suspiciousness revealed the propagation path and starting point of error
- Camera system: Error was caused by an incorrectly initialized variable
- Upon fixing this and rerunning the metamorphic tests, no failed tests were found

Data flow analysis (3)



Benefits

- Alleviates the test oracle problem and can detect code-level vulnerabilities
- Assists the developers with root cause analysis and program repair

Phase	Reduction in scope of search	Reduction of scope of search in percentage
Code analysis (eLOC)	233/701	33
Code analysis (Branch level basic block)	65/133	48
Data flow analysis (Variable level)	60/170	35

Questions?

Thank you for your attention!