FAULT LOCALIZATION

Matthias Kettl
22.07.2020
Bachelor‘s Thesis

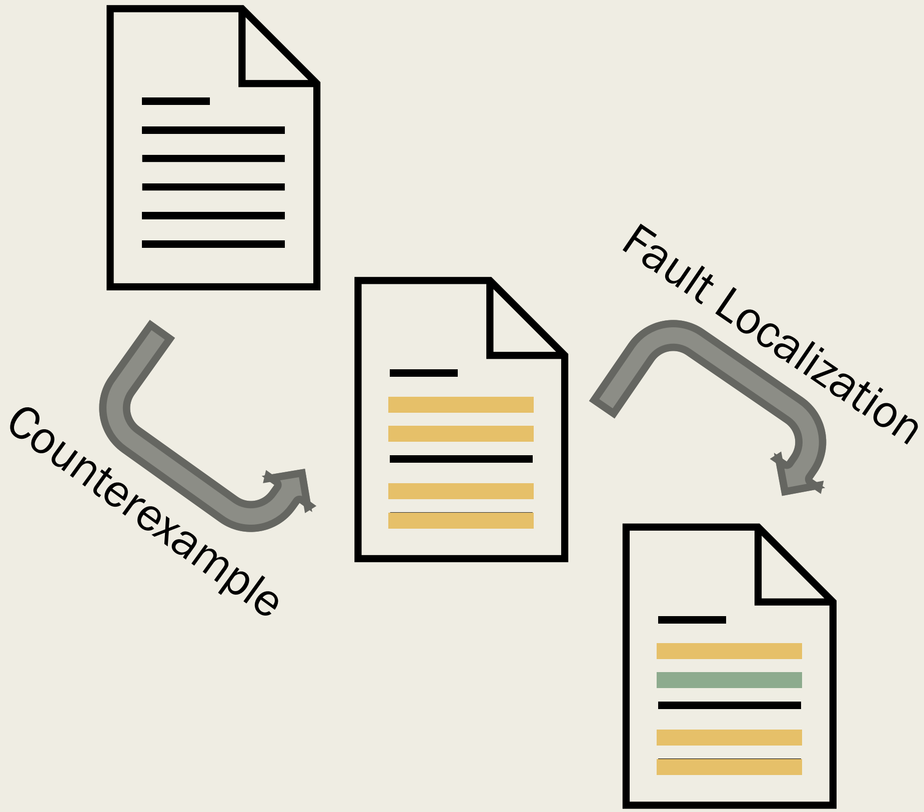
Agenda

- Motivation
- Fault Localization
- Evaluation
- Future Work

1 MOTIVATION

1 Motivation

- verifier determine whether an error location is reachable
- the verifier produces a counterexample
- the root cause of the error is still not obvious
- fault localization reduces the locations to look at



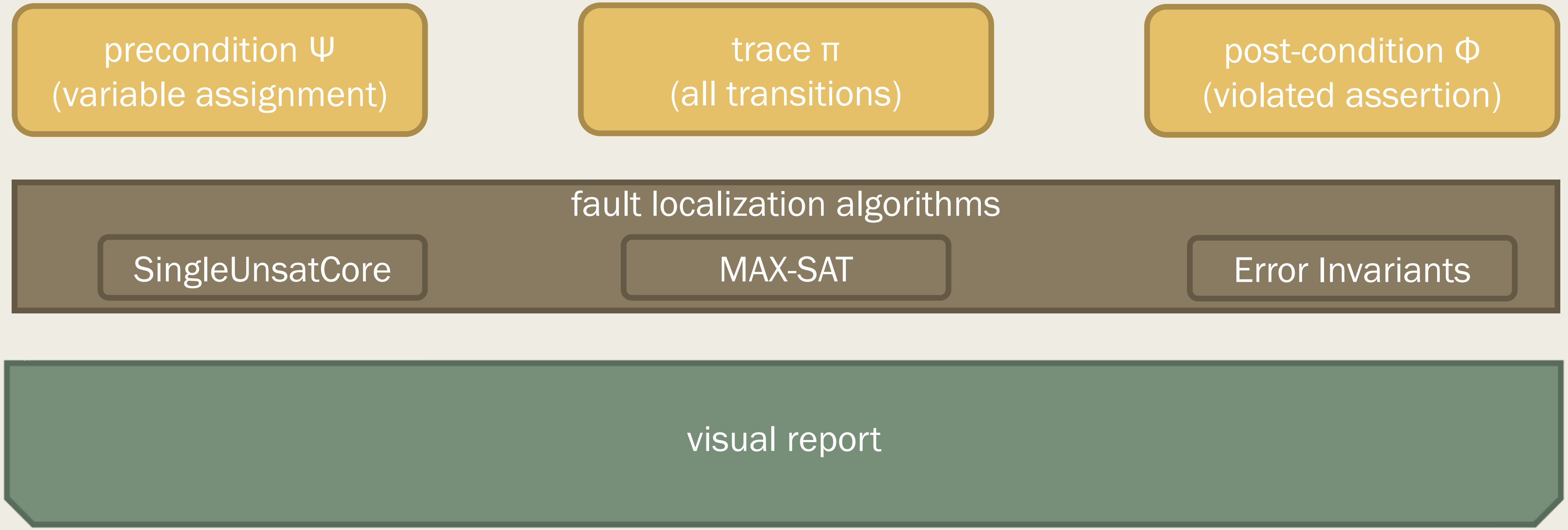
1 Motivation

```
1  int test(int nondet){
2    int x = nondet; // failing input
3    int result = 1;
4    for (int i = 2; i <= x/2 + 1; i++)
5       if (x % i == 0) result = 0;
6    if(result == 0 && isPrime(x))
7          goto ERROR;
8    EXIT: return 0;
9    ERROR: return 1;
10 }
```

- counterexample contains every edge
- fixes are possible on lines 4 and 5 (marked red)
- fault localization identifies these positions

2 ALGORITHMS

2 Algorithms Overview



2 Algorithms Trace Formula

precondition Ψ
(variable assignment)

trace π
(all transitions)

post-condition Φ
(violated assertion)

- counterexample produces a list of `CFAEdges` representing the path to a reachable error location
- the **precondition Ψ** represents the initial variable assignment
- the **post-condition Φ** equals the negation of the Boolean representation of the last *assume edge*
- the **trace π** contains all remaining edges
- the trace formula TF(Ψ**,** π, Φ) equals the conjunct of all three parts

2 Algorithms Trace Formula

precondition Ψ
(variable assignment)

trace π
(all transitions)

post-condition Φ
(violated assertion)

trace formula

$$\mathtt{TF}(\psi, \pi, \phi) = \psi \wedge \bigwedge_{\mathtt{i}=0}^{\mathtt{n}} (\pi[\mathtt{i}]) \wedge \phi$$

selector formula

$$\mathtt{TF}_{\Lambda}(\psi, \pi, \phi) = \psi \wedge \bigwedge_{\mathtt{i}=0}^{\mathtt{n}} (\lambda_{\mathtt{i}} \Rightarrow \pi[\mathtt{i}]) \wedge \phi$$

2 Algorithms MAX-SAT

MAX-SAT

- finds every possible MIN-UNSAT core
- guarantees to mark a line where a fix is possible
- however, the fix might not be suitable

- MIN-UNSAT core: minimal and unsatisfiable set of clauses
- in our case: a minimal subset S of selectors such that:

$$\mathtt{TF}_\Lambda \wedge \bigwedge_{\lambda \in \mathtt{S}} \lambda$$

is unsatisfiable

Based on *"Error Localization using Maximum Satisfiability"* by Manu Jose and Rupak Majumdar

2 Algorithms Example MAX-SAT

```
int test(int nondet){
  int x = nondet;
  if (x > 0)
    if (x < 5)
      if (x > 1)
          if(x != 6 && x != -1)
              goto ERROR;
  EXIT: return 0;
  ERROR: return 1;
}
```

```
Ψ:      (nondet = 2) ∧
π:      (S0 => x = nondet) ∧
        (S1 => x > 0) ∧
        (S2 => x < 5) ∧
        (S3 => x > 1) ∧
Φ:      x = 6 v x = -1
```

2 Algorithms Example MAX-SAT

```
Ψ:    (nondet = 2) ∧
π:    (S0 => x = nondet) ∧
      (S1 => x > 0) ∧
      (S2 => x < 5) ∧
      (S3 => x > 1) ∧
Φ:    x = 6 v x = -1
```

H = {}

S = {S0, S1, S2, S3}

Step 0: create a copy of all selectors (S)

Step 1: already tested each selector once?

Step 2: remove a selector λ from S

Step 3: check if S is a super- or subset of any set in H

Step 4: check if $TF_\Lambda \wedge \bigwedge_{\lambda \in S} \lambda$ is UNSAT

$S \cup \{\lambda\}$

$H \cup \{S\}$

2 Algorithms Example MAX-SAT

```
Ψ:      (nondet = 2) ∧
π:      (S0 => x = nondet) ∧
        (S1 => x > 0) ∧
        (S2 => x < 5) ∧
        (S3 => x > 1) ∧
Φ:      x = 6 v x = -1
```

H = {{S0}, {S1, S2}, {S2, S3}, {S1, S2, S3}}

S = {S0, S1, S2, S3}

Step 0: create a copy of all selectors (S)

Step 1: already tested each selector once?

Step 2: remove a selector λ from S

Step 3: check if S is a super- or subset of any set in H

Step 4: check if $TF_{\Lambda} \wedge \bigwedge_{\lambda \in S} \lambda$ is UNSAT

$S \cup \{\lambda\}$

$H \cup \{S\}$

2 Algorithms Example MAX-SAT

```
int test(int nondet){
    int x = nondet; // failing input
    int result = 1;
    for (int i = 2; i <= x/2 + 1; i++)
      if (x % i == 0) result = 0;
    if(result == 0 && isPrime(x))
          goto ERROR;
    EXIT: return 0;
    ERROR: return 1;
}
```

2 Algorithms Error Invariants

Error invariants (ERRINV)

- abstracts the counterexample
- concise explanation of the faulty behavior
- summarizes parts of the program with inductive interpolants (error invariants)

- Craig interpolant *I*:
  - $A \Rightarrow I$ *is valid*
  - $I \wedge B$ *is UNSAT*
  - *free variables in I are free in A and B*
- we can split the trace formula on all positions $0 \leq i < n$ to obtain A and B

2 Algorithms Example ERRINV

```
1  int test(){
2    int x = 5;  //Ψ
3    x = x + 5;
4    x = x + 1;
5    x = x - 1;
6      if(x == 10)  //Φ
7        goto ERROR;
8    EXIT: return 0;
9    ERROR: return 1;
10 }
```

Interpolants:

x = 5

x = 10

x = 11

x = 10

Borders:

I_1:[2;2]

I_2:[3;3]

I_3:[4;4]

I_4:[3;5]

Trace:

I_1: x = 5

L_3: x = x + 5

I_4: x = 10

2 Algorithms Example ERRINV

```
1   int test(int nondet){
2     int x = nondet; // failing input
3     int result = 1;
4     for (int i = 2; i <= x/2 + 1; i++)
5       if (x % i == 0) result = 0;
6     if(result == 0 && isPrime(x))
7         goto ERROR;
8     EXIT: return 0;
9     ERROR: return 1;
10  }
```

Abstract Trace:

I_1: `nondet = 2`

L_4: `i = 2`

I_2: `x = 2`

L_5: `result = 0`

I_3: `result = 0 && isPrime(x)`

2 Algorithms Visual Report

- all algorithms output a **set of faults**
- the data structure for fault localization uses this **set** to create the visual report
- every fault maintains a list of additional information
- all found faults are ranked
- the report displays the information in an interactive HTML page

1.	100	**Details:**

Error suspected on line(s): **18, 19 and 30**

Found **3** possible bug-fixes:

Potential Fix 1: Try to change the assigned value of "test" in "test = test * i;" to another value.

Potential Fix 2: Try to change the assigned value of "number" in "number = __VERIFIER_nondet_int();" to another value.

Potential Fix 3: Try to declare the variable in "int copyForCheck = number;" differently.

1 hint is available:

Hint 1: The program fails for the initial variable assignment main::test = 1_32, main::i = 2_32, (4_32 = main::number)

The score is obtained by:

1. Score calculated by edge type(s). (**100.0%**)
2. The set has a size of 3. (**100.0%**)
3. Overall occurrence of elements in this set. (**100.0%**)
4. Minimal distance to error location: 9 line(s) (**100.0%**)
5. This fault is 5 execution step(s) away from the error location. (**100.0%**)

Relevant lines:

```
18: number = __VERIFIER_nondet_int();
19: int copyForCheck = number;
30: test = test * i;
```

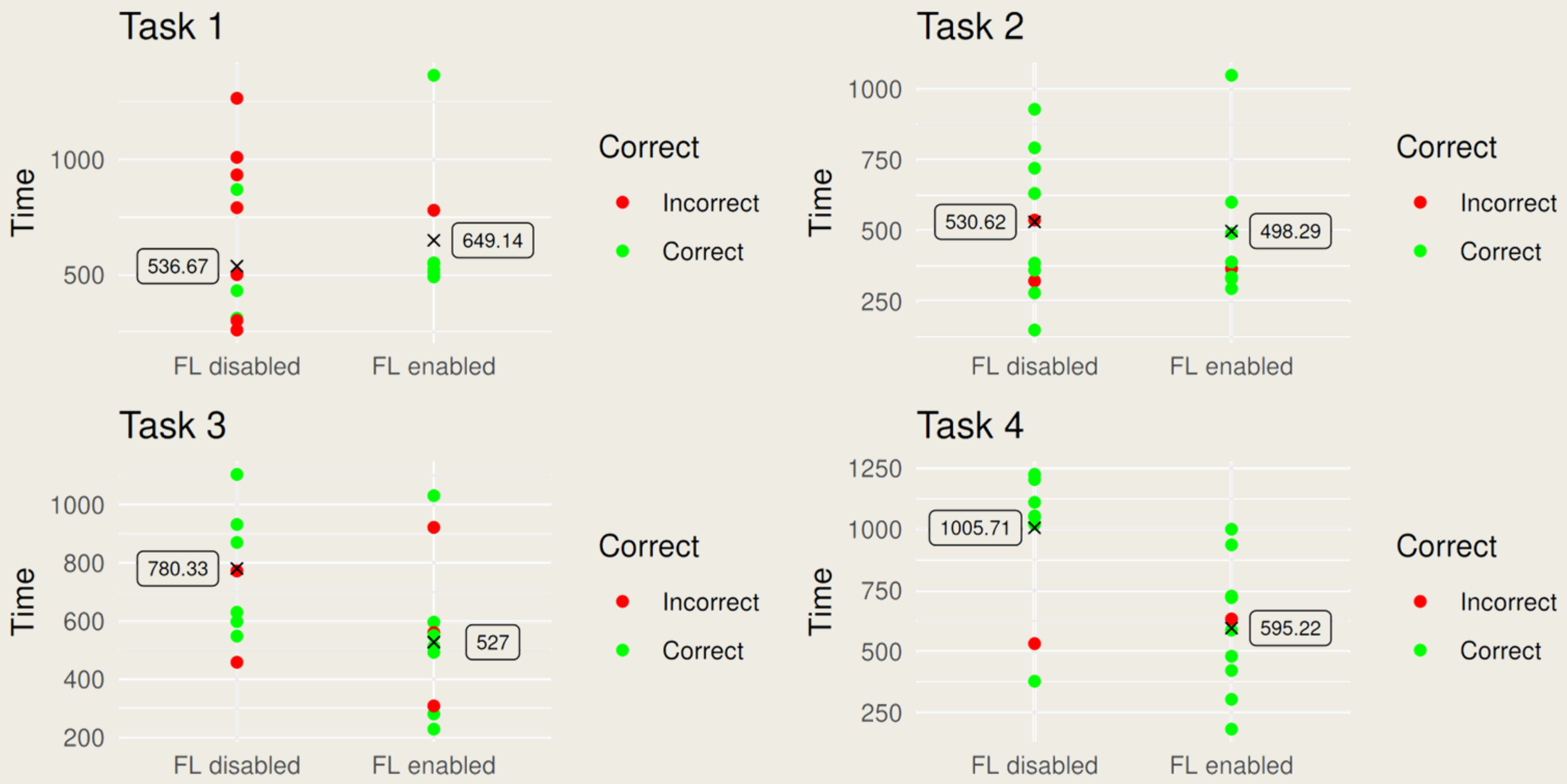
3 EVALUATION

3 Evaluation Survey: Setting

- 18 participants (students, scientists, senior and junior developers)
- participants fixed bugs in 4 programs
- fault localization (FL) report with ERRINV
- evaluation of *correctness* and *needed time*
- evaluation of the grading of the features and the feedback

Group	Task 1	Task 2	Task 3	Task 4
1	no FL	no FL	FL	FL
2	FL	FL	no FL	no FL

3 Evaluation Survey: Effectiveness



3 Evaluation Survey: Feedback and Grading

- estimated benefit: **Ø 6.4/10**
- clearness of the report: **Ø 7.9/10**
- most liked features: **current values, line numbers**
- critique: redundant information, marking of unnecessary lines, unreadable formulas
- summary:
  - *the chosen visual representation fits the needs of the users and is intuitive*
  - *fault localization improves the needed time and the correctness of fixes*
  - *the final organization of the gathered data can be improved*

3 Evaluation Critical Reflection

Problems of both techniques:

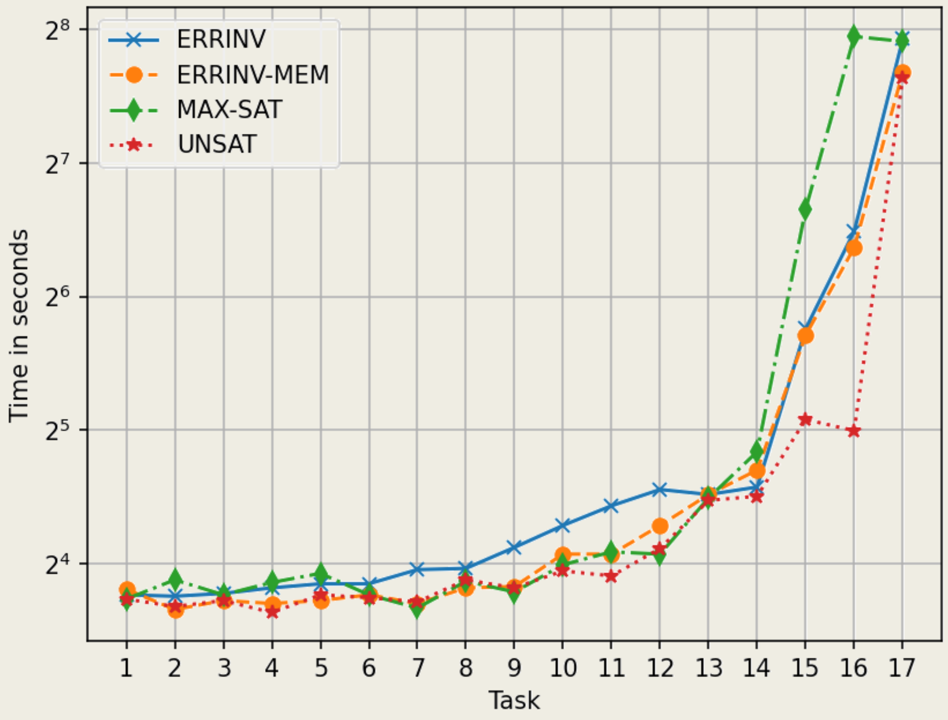
- both tend to mark iteration variables
- both tend to mark test variables
- both tend to mark function call edges
- cannot identify missing method calls
- bad runtime

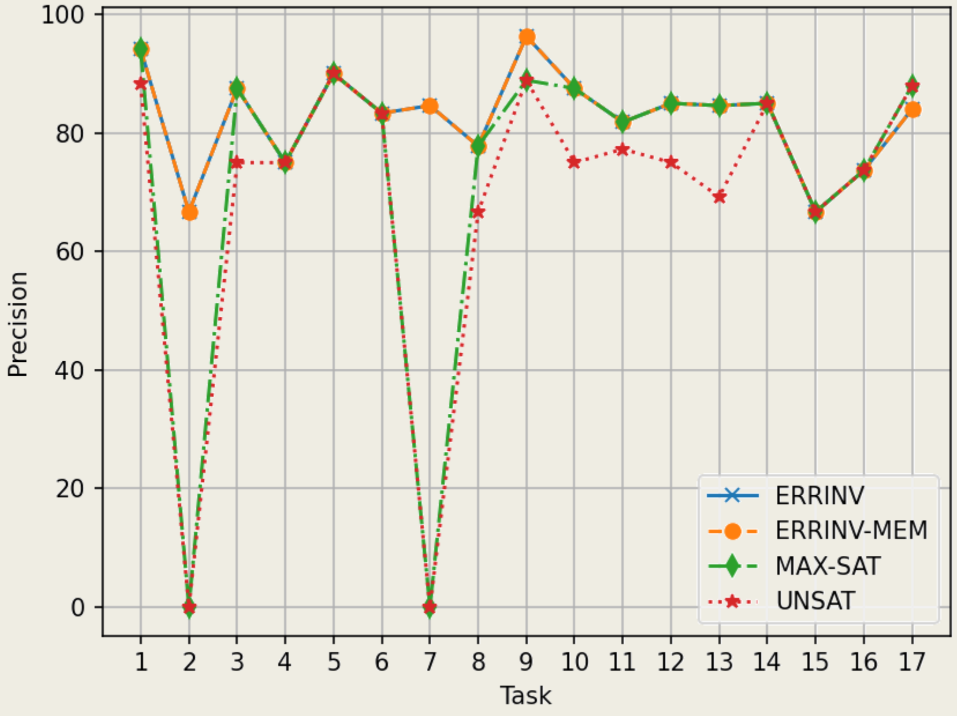
Solutions:

- useful options (ban, ignore, filter uniqueSelectors)
- memoization

```
//Check if a and b are even
int test(a = 3, b = 2){
    int expResult = 0;
    int check = isEven(b);
      if(expResult != check)
        goto ERROR;
    EXIT: return 0;
    ERROR: return 1;
}
```

3 Evaluation Benchmarks



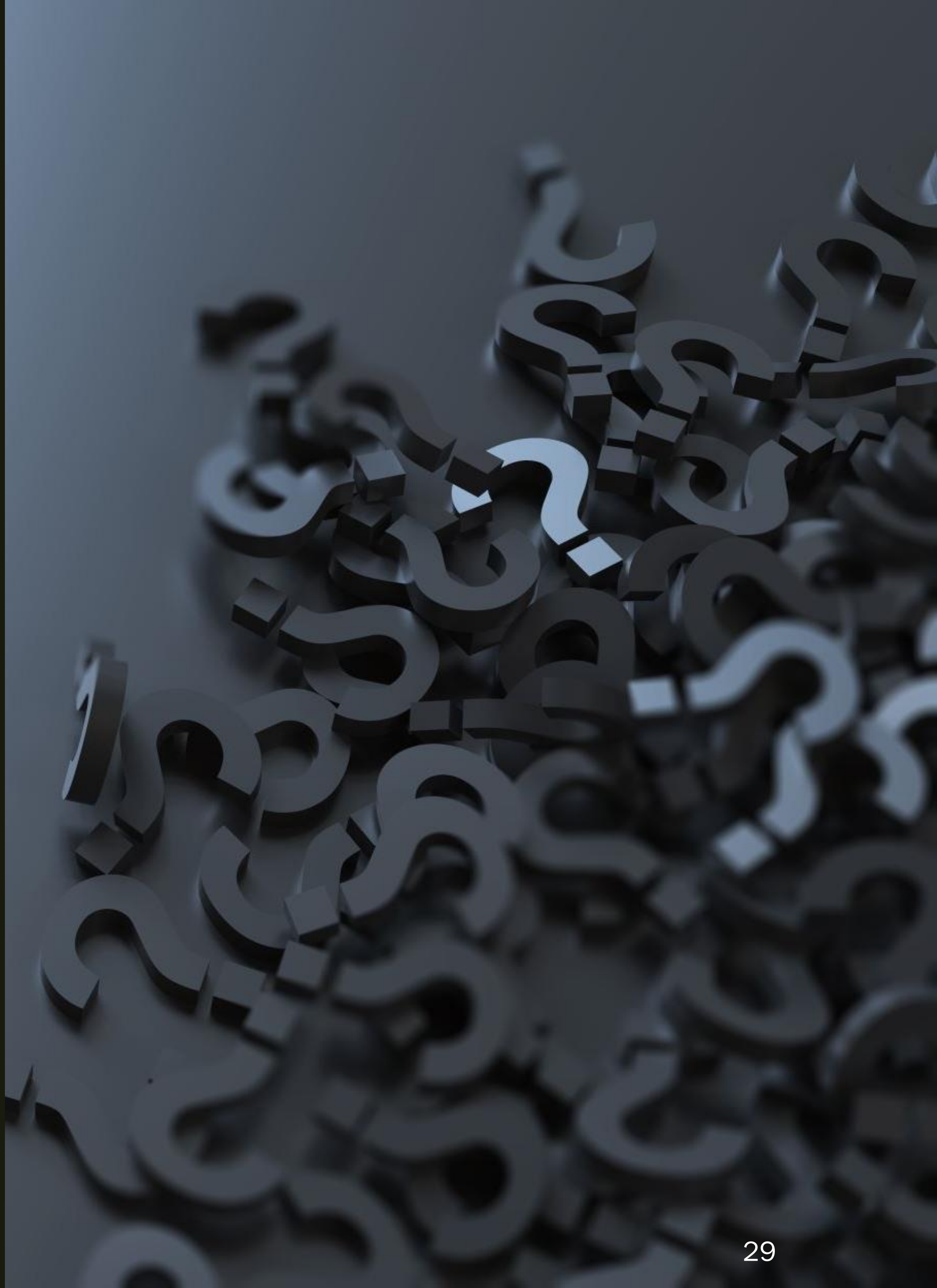


4 FUTURE WORK

4 Future Work

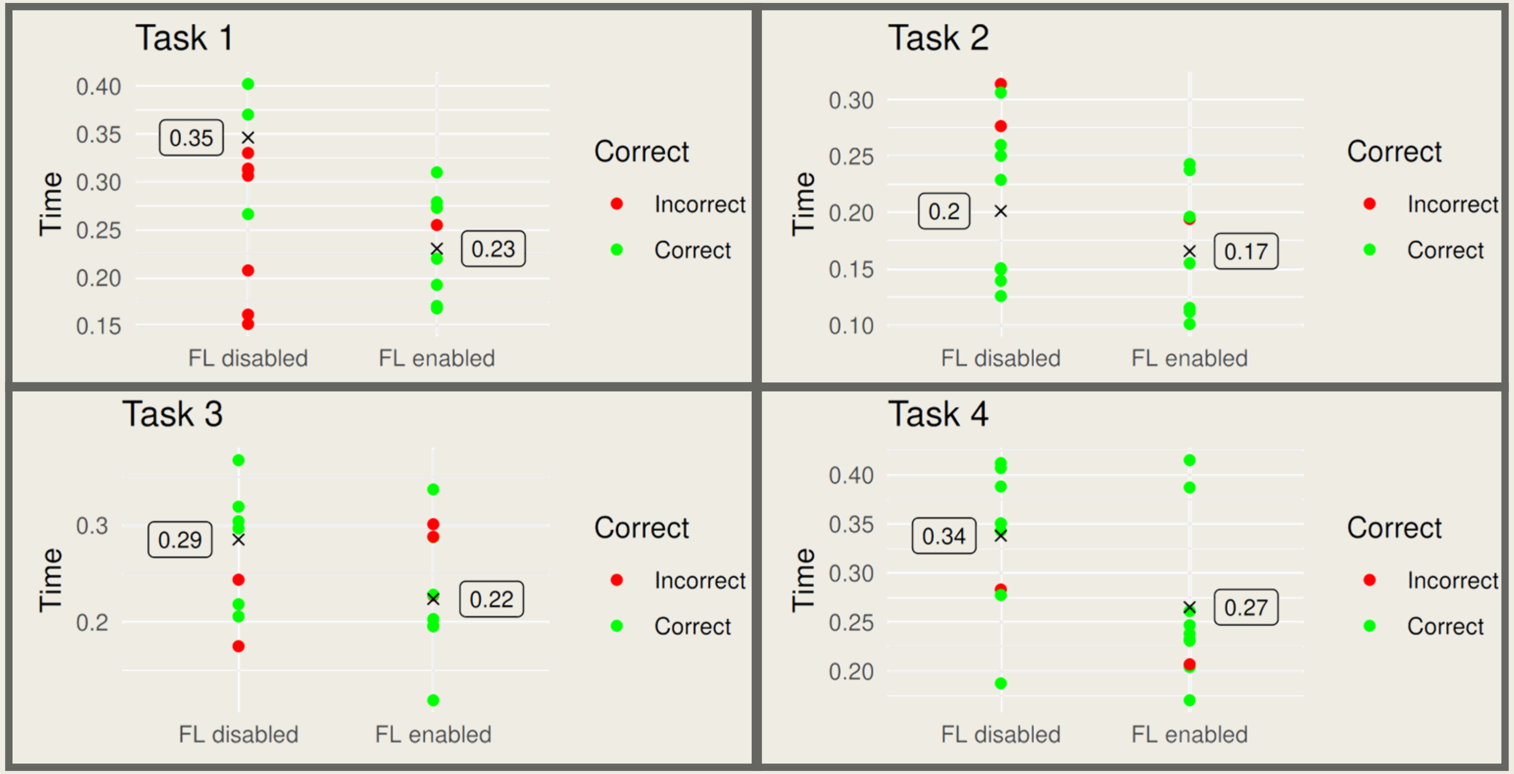
- implement flow-sensitive trace formula for ERRINV
  - *assumes of if statements imply all statements of the if-block*
  - *allows to identify important and unimportant if-blocks*
- implement auto repair mechanism for MAX-SAT
  - *find off-by-one errors*
  - *increase or decrease the value of promising variables and restart the analysis*
- adapt the report
  - *improve visualization of formulas*
  - *remove redundant information*

QUESTIONS?



APPENDIX

Evaluation Survey: Normalized Effectiveness



Evaluation Survey: Normalized Effectiveness