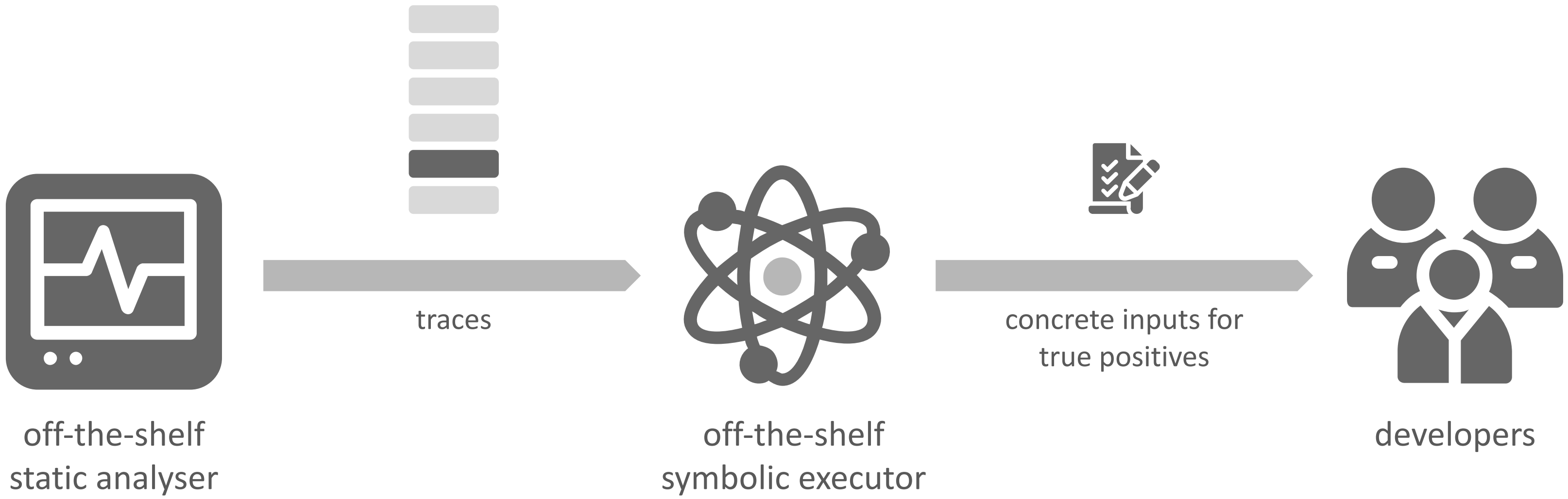
Imperial College London

Combining Static Analysis Error Traces with Dynamic Symbolic Execution (Experience Paper)

Frank Busse • Pritam M. Gharat • Cristian Cadar • Alastair F. Donaldson
Software Reliability Group • Multicore Programming Group


traces
concrete inputs for
true positives
off-the-shelf
static analyser
off-the-shelf
symbolic executor
developers

Disclaimer - Experience Paper

What we did:

- combined traces for **two bug classes** from **two static analysis engines** with **one symbolic execution engine** in a particular way

What we didn't:

- compare **static analysis vs. (dynamic) symbolic execution**
- **generalise results** to any combination of static analysis with symbolic execution
- **benchmark static analysers**

Example

```
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

Example

```
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

Example

```
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

```
> example foo
> example Hi!
Aborted (core dumped)
> example Ii!
Aborted (core dumped)
```

Example

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

```
> example foo
> example Hi!
Aborted (core dumped)
> example Ii!
Aborted (core dumped)
```

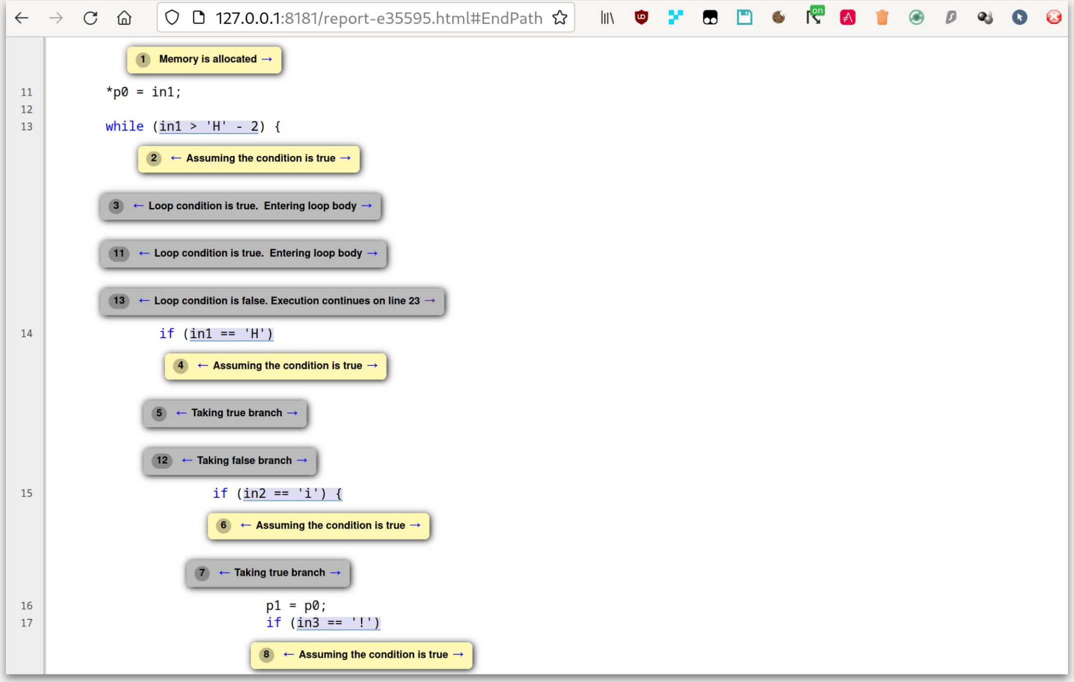
Example
Static Analysis Traces

```c
4   int main (int argc, char *argv[]) {
5       uint8_t in1 = argv[1][0];
6       uint8_t in2 = argv[1][1];
7       uint8_t in3 = argv[1][2];
8
9       uint8_t *p0, *p1;
10      p0 = malloc(sizeof(uint8_t));
11      *p0 = in1;
12
13      while (in1 > 'H' - 2) {
14          if (in1 == 'H')
15              if (in2 == 'i') {
16                  p1 = p0;
17                  if (in3 == '!')
18                      free(p1);
19              }
20          --in1;
21      }
22
23      int result = *p0;
24      free(p0);
25      return result;
26  }
```

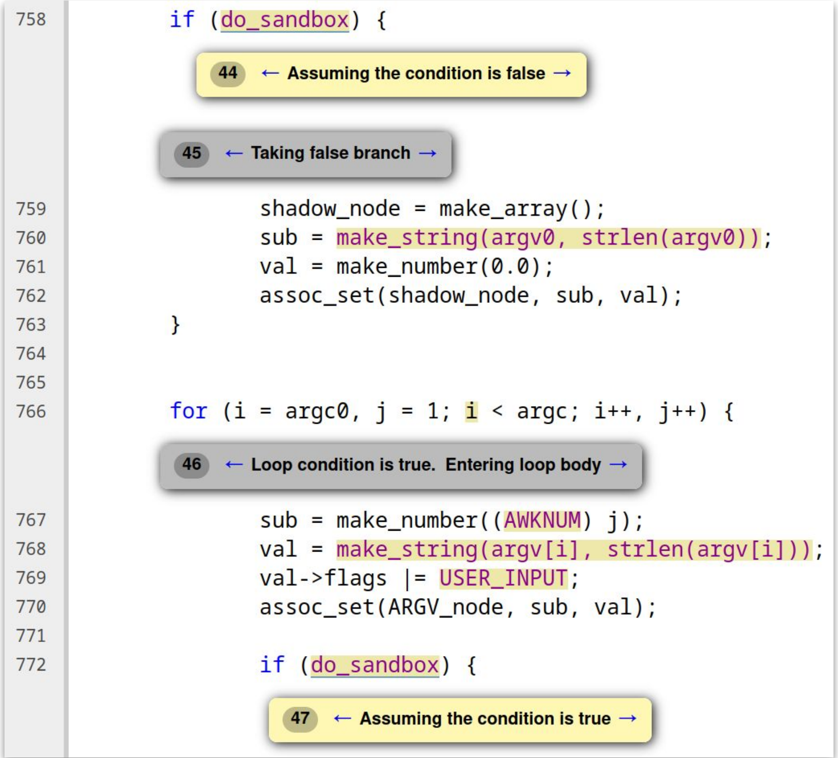
```
> scan-build clang example.c
example.c:23:15: warning: Use of memory
after it is freed [unix.Malloc]
    int result = *p0;
                 ^~~
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
```

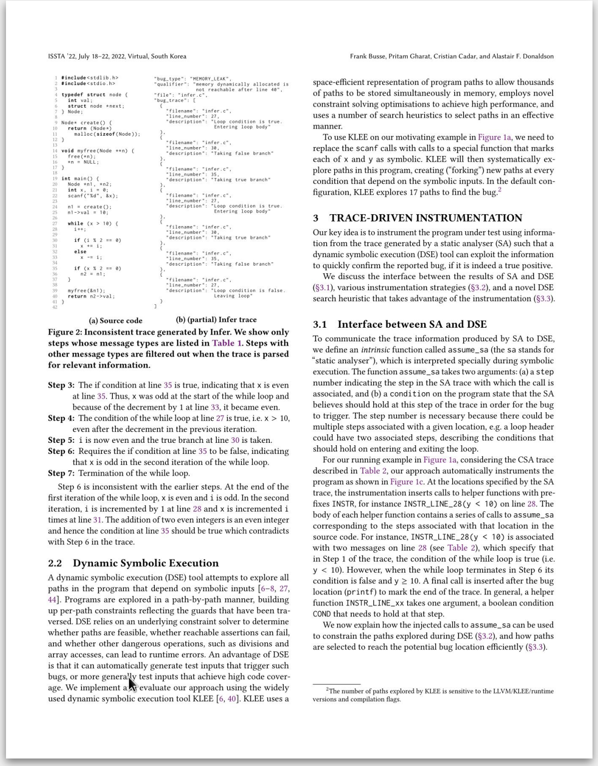
Example
Static Analysis Traces

```c
4   int main (int argc, char *argv[]) {
5       uint8_t in1 = argv[1][0];
6       uint8_t in2 = argv[1][1];
7       uint8_t in3 = argv[1][2];
8
9       uint8_t *p0, *p1;
10      p0 = malloc(sizeof(uint8_t));
11      *p0 = in1;
12
13      while (in1 > 'H' - 2) {
14          if (in1 == 'H')
15              if (in2 == 'i') {
16                  p1 = p0;
17                  if (in3 == '!')
18                      free(p1);
19              }
20          --in1;
21      }
22
23      int result = *p0;
24      free(p0);
25      return result;
26  }
```



Infeasible Traces






```
1  #include<stdlib.h>
2  #include<stdio.h>
3
4  typedef struct node {
5    int val;
6    struct node *next;
7  } Node;
8
9  Node* create() {
10   return (Node*)
11     malloc(sizeof(Node));
12 }
13
14 void myfree(Node **n) {
15   free(*n);
16   *n = NULL;
17 }
18
19 int main() {
20   Node *n1, *n2;
21   int x, i = 0;
22   scanf("%d", &x);
23
24   n1 = create();
25   n1->val = 10;
26
27   while (x > 10) {
28     i++;
29
30     if (i % 2 == 0)
31       x += i;
32     else
33       x -= i;
34
35     if (x % 2 == 0)
36       n2 = n1;
37   }
38
39   myfree(&n1);
40   return n2->val;
41 }
42
```

(a) Source code

```
"bug_type": "MEMORY_LEAK",
"qualifier": "memory dynamically allocated is
              not reachable after line 40",
"file": "infer.c",
"bug_trace": [
  {
    "filename": "infer.c",
    "line_number": 27,
    "description": "Loop condition is true.
                    Entering loop body"
  },
  {
    "filename": "infer.c",
    "line_number": 30,
    "description": "Taking false branch"
  },
  {
    "filename": "infer.c",
    "line_number": 35,
    "description": "Taking true branch"
  },
  {
    "filename": "infer.c",
    "line_number": 27,
    "description": "Loop condition is true.
                    Entering loop body"
  },
  {
    "filename": "infer.c",
    "line_number": 30,
    "description": "Taking true branch"
  },
  {
    "filename": "infer.c",
    "line_number": 35,
    "description": "Taking false branch"
  },
  {
    "filename": "infer.c",
    "line_number": 27,
    "description": "Loop condition is false.
                    Leaving loop"
  }
]
```

(b) (partial) Infer trace

Figure 2: Inconsistent trace generated by Infer. We show only steps whose message types are listed in Table 1. Steps with other message types are filtered out when the trace is parsed for relevant information.

Step 3: The if condition at line 35 is true, indicating that x is even at line 35. Thus, x was odd at the start of the while loop and because of the decrement by 1 at line 33, it became even.
Step 4: The condition of the while loop at line 27 is true, i.e. $x > 10$, even after the decrement in the previous iteration.
Step 5: i is now even and the true branch at line 30 is taken.
Step 6: Requires the if condition at line 35 to be false, indicating that x is odd in the second iteration of the while loop.
Step 7: Termination of the while loop.

Step 6 is inconsistent with the earlier steps. At the end of the first iteration of the while loop, x is even and i is odd. In the second iteration, i is incremented by 1 at line 28 and x is incremented i times at line 31. The addition of two even integers is an even integer and hence the condition at line 35 should be true which contradicts with Step 6 in the trace.

2.2 Dynamic Symbolic Execution

A dynamic symbolic execution (DSE) tool attempts to explore all paths in the program that depend on symbolic inputs [6–8, 27, 44]. Programs are explored in a path-by-path manner, building up per-path constraints reflecting the guards that have been traversed. DSE relies on an underlying constraint solver to determine whether paths are feasible, whether reachable assertions can fail, and whether other dangerous operations, such as divisions and array accesses, can lead to runtime errors. An advantage of DSE is that it can automatically generate test inputs that trigger such bugs, or more generally test inputs that achieve high code coverage. We implement and evaluate our approach using the widely used dynamic symbolic execution tool KLEE [6, 40]. KLEE uses a space-efficient representation of program paths to allow thousands of paths to be stored simultaneously in memory, employs novel constraint solving optimisations to achieve high performance, and uses a number of search heuristics to select paths in an effective manner.

To use KLEE on our motivating example in Figure 1a, we need to replace the `scanf` calls with calls to a special function that marks each of x and y as symbolic. KLEE will then systematically explore paths in this program, creating ("forking") new paths at every condition that depend on the symbolic inputs. In the default configuration, KLEE explores 17 paths to find the bug.[2]

3 TRACE-DRIVEN INSTRUMENTATION

Our key idea is to instrument the program under test using information from the trace generated by a static analyser (SA) such that a dynamic symbolic execution (DSE) tool can exploit the information to quickly confirm the reported bug, if it is indeed a true positive.

We discuss the interface between the results of SA and DSE (§3.1), various instrumentation strategies (§3.2), and a novel DSE search heuristic that takes advantage of the instrumentation (§3.3).

3.1 Interface between SA and DSE

To communicate the trace information produced by SA to DSE, we define an *intrinsic* function called `assume_sa` (the `sa` stands for "static analyser"), which is interpreted specially during symbolic execution. The function `assume_sa` takes two arguments: (a) a `step` number indicating the step in the SA trace with which the call is associated, and (b) a `condition` on the program state that the SA believes should hold at this step of the trace in order for the bug to trigger. The step number is necessary because there could be multiple steps associated with a given location, e.g. a loop header could have two associated steps, describing the conditions that should hold on entering and exiting the loop.

For our running example in Figure 1a, considering the CSA trace described in Table 2, our approach automatically instruments the program as shown in Figure 1c. At the locations specified by the SA trace, the instrumentation inserts calls to helper functions with prefixes `INSTR`, for instance `INSTR_LINE_28(y < 10)` on line 28. The body of each helper function contains a series of calls to `assume_sa` corresponding to the steps associated with that location in the source code. For instance, `INSTR_LINE_28(y < 10)` is associated with two messages on line 28 (see Table 2), which specify that in Step 1 of the trace, the condition of the while loop is true (i.e. $y < 10$). However, when the while loop terminates in Step 6 its condition is false and $y \geq 10$. A final call is inserted after the bug location (`printf`) to mark the end of the trace. In general, a helper function `INSTR_LINE_xx` takes one argument, a boolean condition `COND` that needs to hold at that step.

We now explain how the injected calls to `assume_sa` can be used to constrain the paths explored during DSE (§3.2), and how paths are selected to reach the potential bug location efficiently (§3.3).

[2] The number of paths explored by KLEE is sensitive to the LLVM/KLEE/runtime versions and compilation flags.

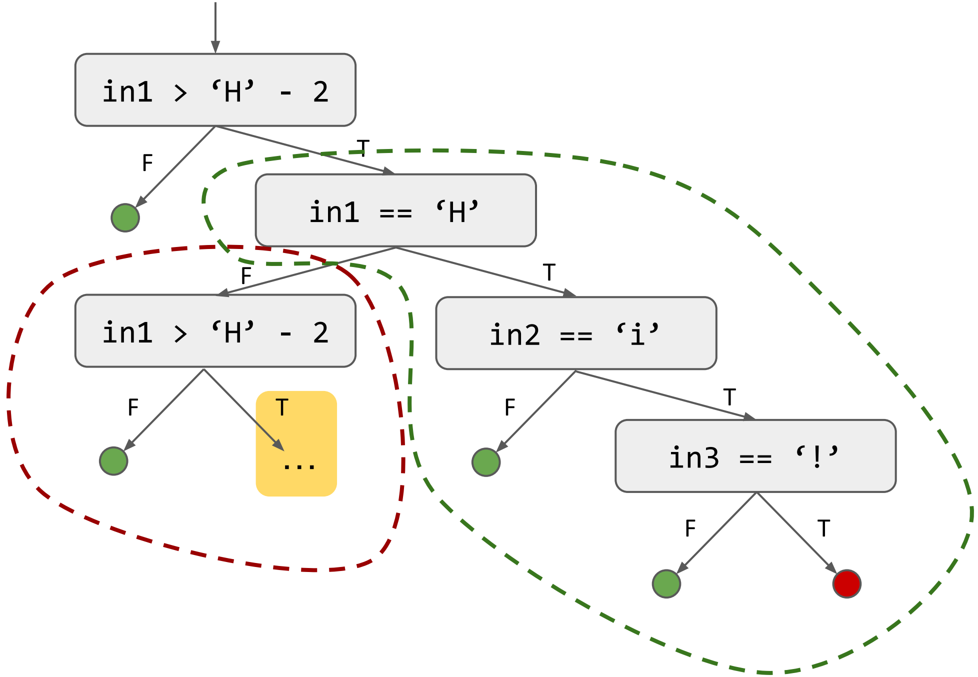
Example
Dynamic Symbolic Execution

```c
4   int main (int argc, char *argv[]) {
5       uint8_t in1 = argv[1][0];
6       uint8_t in2 = argv[1][1];
7       uint8_t in3 = argv[1][2];
8
9       uint8_t *p0, *p1;
10      p0 = malloc(sizeof(uint8_t));
11      *p0 = in1;
12
13      while (in1 > 'H' - 2) {
14          if (in1 == 'H')
15              if (in2 == 'i') {
16                  p1 = p0;
17                  if (in3 == '!')
18                      free(p1);
19              }
20          --in1;
21      }
22
23      int result = *p0;
24      free(p0);
25      return result;
26  }
```

```
> klee [...] example.bc --sym-arg 3
KLEE: Using STP solver backend
KLEE: ERROR: example.c:23: memory error: out of bound
pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 23361
KLEE: done: completed paths = 554
KLEE: done: generated tests = 4
```

Example
Dynamic Symbolic Execution

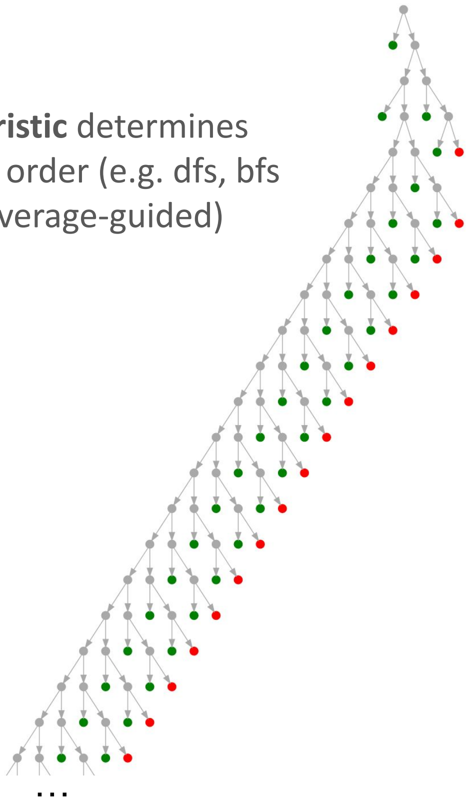
```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

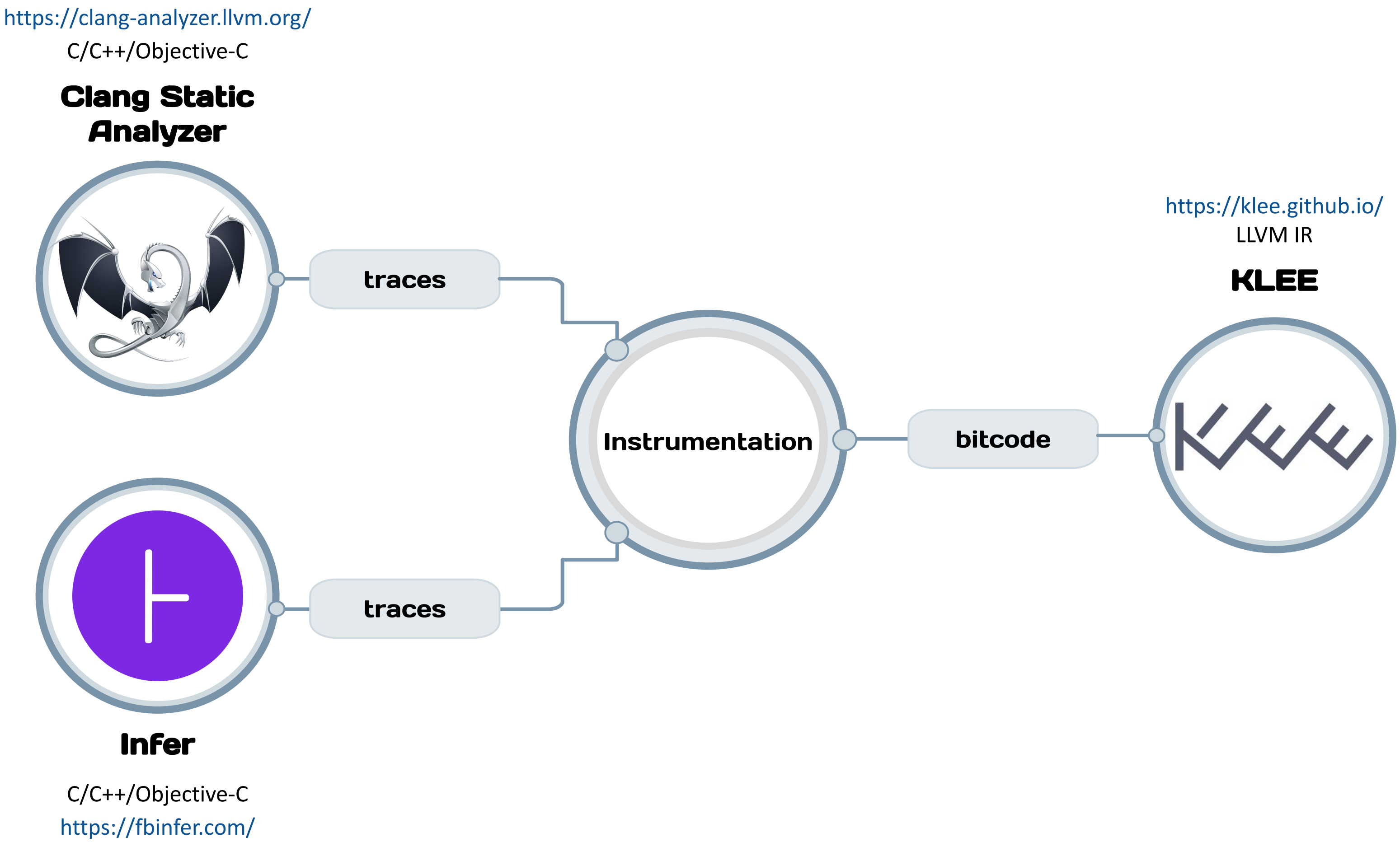


Example
Dynamic Symbolic Execution

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

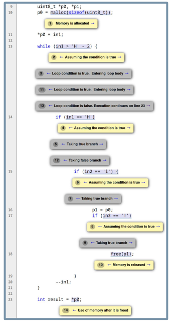
search heuristic determines exploration order (e.g. dfs, bfs random, coverage-guided)




https://clang-analyzer.llvm.org/
C/C++/Objective-C
Clang Static Analyzer
traces
Instrumentation
bitcode
https://klee.github.io/
LLVM IR
KLEE
traces
Infer
C/C++/Objective-C
https://fbinfer.com/

Example Instrumentation

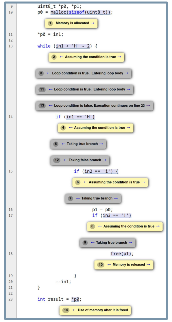
```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```



```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (INSTR_LINE_13(in1 > 'H' - 2)) {
        if (INSTR_LINE_14(in1 == 'H'))
            if (INSTR_LINE_15(in2 == 'i')) {
                p1 = p0;
                if (INSTR_LINE_17(in3 == '!'))
                    free(p1);
            }
        --in1;
    }

    int result = *p0; INSTR_LINE_23();
    free(p0);
    return result;
}
```

Example Instrumentation

```
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```



```
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (INSTR_LINE_13(in1 > 'H' - 2)) {
        if (INSTR_LINE_14(in1 == 'H'))
            if (INSTR_LINE_15(in2 == 'i')) {
                p1 = p0;
                if (INSTR_LINE_17(in3 == '!'))
                    free(p1);
            }
        --in1;
    }

    int result = *p0; INSTR_LINE_23();
    free(p0);
    return result;
}
```

Constraint Enforcement

```c
int INSTR_LINE_13(int XY)
{
    assume_sa(8, !XY);
    assume_sa(7, XY);
    assume_sa(2, XY);
    return XY;
}
```

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (INSTR_LINE_13(in1 > 'H' - 2)) {
        if (INSTR_LINE_14(in1 == 'H'))
            if (INSTR_LINE_15(in2 == 'i')) {
                p1 = p0;
                if (INSTR_LINE_17(in3 == '!'))
                    free(p1);
            }
        --in1;
    }

    int result = *p0; INSTR_LINE_23();
    free(p0);
    return result;
}
```

2 7 8

Ignore noop

Try add constraint if feasible

Require constraint has to hold

Example
Constraint Enforcement

```
> klee [...] example.bc --sym-arg 3
KLEE: Using STP solver backend
KLEE: ERROR: example.c:23: memory error: out of bound
pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 23361
KLEE: done: completed paths = 554
KLEE: done: generated tests = 4
```

```
> klee [...] example.bc --sym-arg 3
KLEE: Using STP solver backend
KLEE: ERROR: example_run.c:25: memory error: out of bound
pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 1230
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (INSTR_LINE_13(in1 > 'H' - 2)) {
        if (INSTR_LINE_14(in1 == 'H'))
            if (INSTR_LINE_15(in2 == 'i')) {
                p1 = p0;
                if (INSTR_LINE_17(in3 == '!'))
                    free(p1);
            }
        --in1;
    }

    int result = *p0; INSTR_LINE_23();
    free(p0);
    return result;
}
```

Example
Constraint Enforcement

```
> klee [...] example.bc --sym-arg 3
KLEE: Using STP solver backend
KLEE: ERROR: example_run.c:25: memory error: out of bound
pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 1230
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

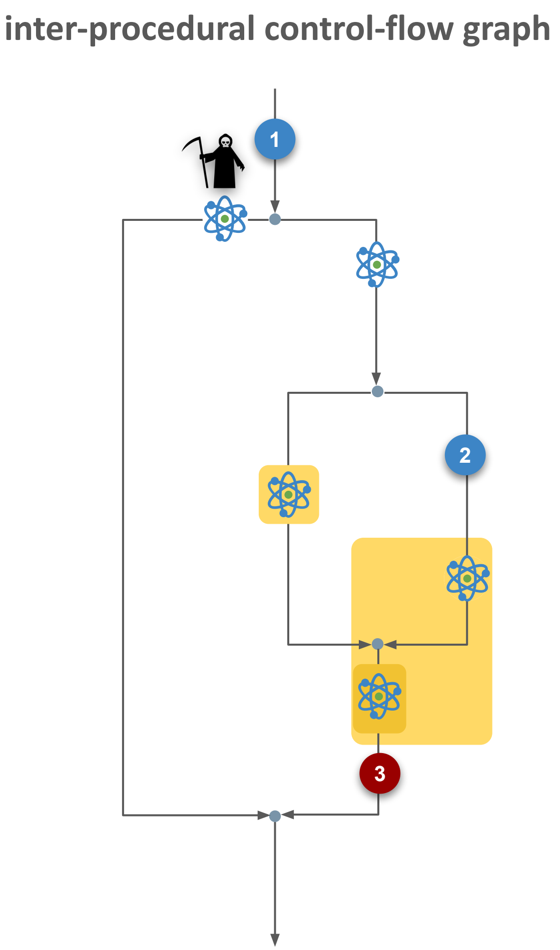
```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (INSTR_LINE_13(in1 > 'H' - 2)) {
        if (INSTR_LINE_14(in1 == 'H'))
            if (INSTR_LINE_15(in2 == 'i')) {
                p1 = p0;
                if (INSTR_LINE_17(in3 == '!'))
                    free(p1);
            }
        --in1;
    }

    int result = *p0; INSTR_LINE_23();
    free(p0);
    return result;
}
```

Targeted Search Heuristic

- drives execution engine **towards instrumented lines**
- **skips** unreachable steps
- **terminates** states that can't reach final step
- **prioritises** states that
  - reached **more steps**
  - are **closer to next step**

Targeted Search Heuristic

- drives execution engine **towards instrumented lines**
- **skips** unreachable steps
- **terminates** states that can't reach final step
- **prioritises** states that
  - reached **more steps**
  - are **closer to next step**



Targeted Search Heuristic

- drives execution engine **towards instrumented lines**
- **skips** unreachable steps
- **terminates** states that can't reach final step
- **prioritises** states that
  - reached **more steps**
  - are **closer to next step**

Algorithm 1: Targeted search heuristic

```
Data: activePathIDs : Step → {PathID}
Data: states : Step × PathID → {State}
Data: instructionPathIDs : Instruction × Step → pathID
Data: maxActiveStep: maximum step number among states
Data: maxStep: maximum step number in program

Function update(currentState, newStates, terminatedStates):
    updateCurrent (currentState)
    foreach state : newStates do
        insert (state)
    foreach state : terminatedStates do
        remove (state)

Function insert(state):
    state.distance ← computeDistance(state)
    if state.distance = ∞ then
        while state.distance = ∞ ∧ state.lastStep < maxStep do
            state.lastStep ← state.lastStep + 1
            state.distance ← computeDistance(state)
    state.pathID ←
      instructionPathIDs[state.pc][state.lastStep]
    if state.distance = ∞ then
        terminate(state)
    else
        states[state.lastStep][state.pathID].add(state)
        activePathIDs[state.lastStep].add(state.pathID)

Function select()→ State :
    nextPathID ←
      activePathIDs[maxActiveStep].selectRoundRobin()
    candidates ←
      states[maxActiveStep][nextPathID].selectByDistance()
    return candidates.pickRandomly()
```

Evaluation

We investigated

- historical SA bug reports
- CoREBench[1]
- 25 applications/suites > 7yrs old

In short

- (almost) no historical reports
- known bugs not found
- true positives trivial and/or bug class not supported by KLEE

Table 3: Examined applications and static analysis reports. We investigated up to 20 reports per application for each analyser but found only true positives caused by trivial allocation errors or failing system calls.

		Relevant reports		False positives		True positives	
Application	Release	CSA	Infer	CSA	Infer	CSA	Infer
APR	1.5.2	8	2	8	2	0	0
flex	2.5.39	13	17	12	7	1	10
awk	4.1.2	124	70	20	20	0	0
bc	1.06	11	0	11	0	0	0
Binutils	2.25.1	0	38	0	14	0	6
combine	0.4.0	1	10	1	10	0	0
Coreutils	8.24	25	5	20	5	0	0
datamash	1.0.6	0	1	0	1	0	0
Diffutils	3.3	6	0	6	0	0	0
Findutils	4.4.2	6	2	6	1	0	1
grep	2.21	20	8	20	8	0	0
Gzip	1.6	1	0	0	0	1	0
Libtasn1	4.5	1	4	1	1	0	3
M4	1.4.17	9	2	8	2	1	0
Make	4.1	3	2	3	2	0	0
oSIP	4.1.0	1	6	1	6	0	0
sed	4.2	6	7	3	7	3	0
Trueprint	5.4	0	7	0	6	0	1
ImageMagick	6.9.4-8	10	11	10	3	0	8
JasPer	1.900.1	9	3	9	1	0	2
libjpeg	9a	17	2	17	2	0	0
LibTIFF	3.9.7	6	12	6	3	0	9
libxml2	2.9.2	33	91	20	20	0	0
tcpdump	4.7.4	0	2	0	0	0	2
Vorbis Tools	1.4.0	1	19	1	1	0	18

[1] https://www.comp.nus.edu.sg/~release/corebench/

Evaluation

We investigated

- historical SA bug reports
- CoREBench[1]
- 25 applications/suites > 7yrs old

In short

- (almost) no historical reports
- known bugs not found
- true positives trivial and/or bug class not supported by KLEE

➟ We had to artificially inject bugs.

relevant reports		true/false positives	
CSA	Infer	CSA	Infer
311	322	6/183	60/122

Infer reports many missing NULL-checks for `malloc()`, `strdup()`, `localtime()`, …

[1] https://www.comp.nus.edu.sg/~release/corebench/

Bug Injection

- **two bug types**
  - null-pointer dereferences
  - use-after-free errors
- **1–4 events** along path
- only in hard to reach instructions (KLEE needs more than 10min to cover instruction)
- 10 applications from Coreutils 8.31

```
int *xtmp;

int *ytmp = (int *)malloc(sizeof(int));
xtmp = ytmp; // 0-3 aliases for 1-4 event bugs

free(ytmp);
printf("%d", *xtmp); // use-after-free
```

2-event bug

Bug Injection

- **two bug types**
  - null-pointer dereferences
  - use-after-free errors
- **1–4 events** along path
- only in hard to reach instructions (KLEE needs more than 10min to cover instruction)
- 10 applications from Coreutils 8.31

program path

```
int _i = 1, *xtmp = &_i, *ytmp = &_i;
xtmp = (int *)malloc(sizeof(int));
ytmp = xtmp; // 0-2 aliases for 1-4 event bugs
free(xtmp); printf("%d", *ytmp); // use-after-free
```

3-event bug

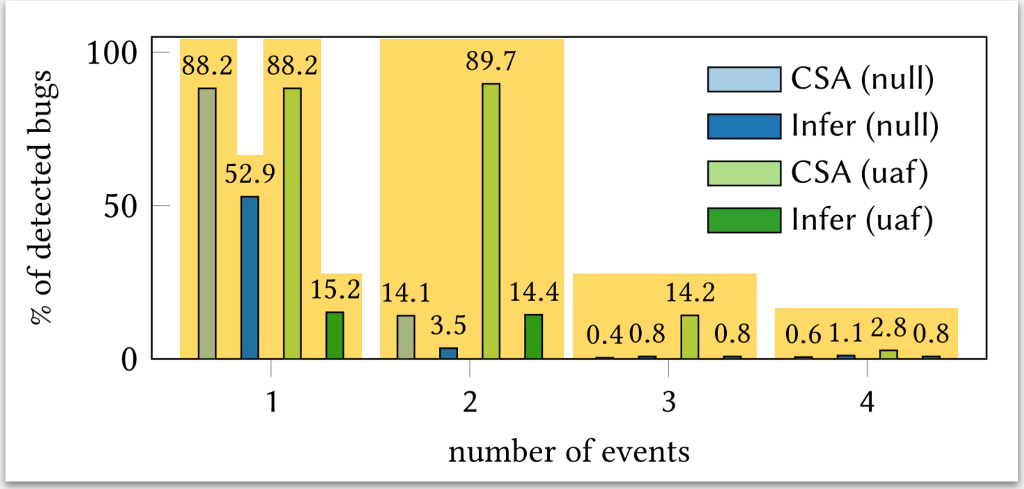
Bug Injection

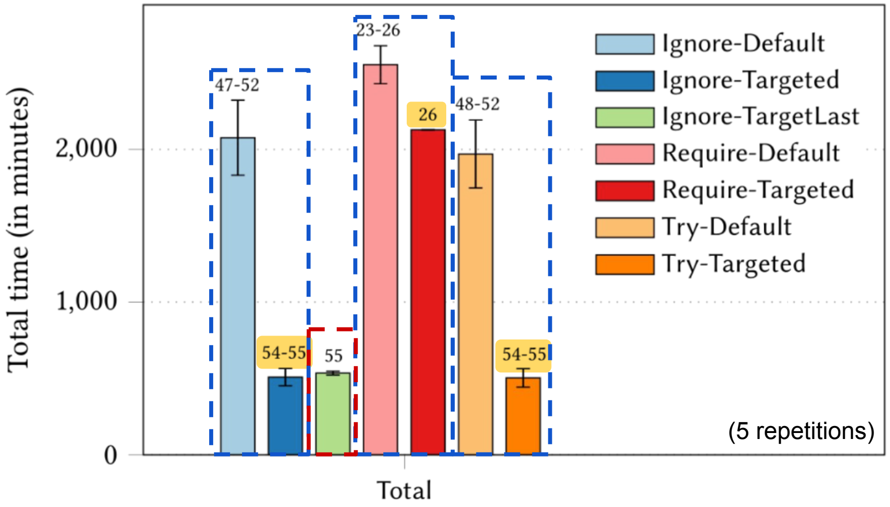
- 297 one-event bugs
- 632 two-event bugs
- 478 three-event bugs
- 357 four-event bugs

55 bugs for further evaluation

Typical trace lengths:

- **CSA** 10–20 steps (max. 55)
- **Infer** 1–5 steps

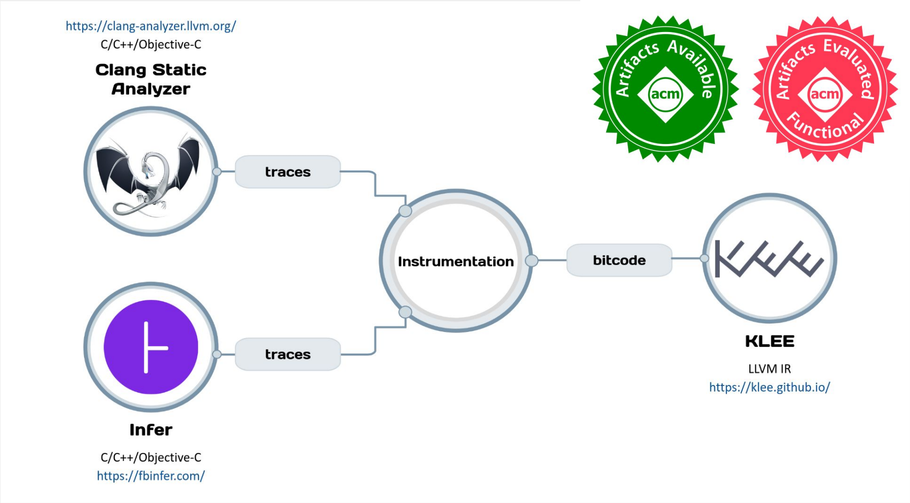




Results Instrumented Code

- **targeted** heuristic finds more bugs in less time
- **require** performs worst
- **try** only slightly better than **ignore**
- intermediate steps rarely beneficial

The static analysis error **traces** in our experiments in general **do not add (m)any benefits** when combined with targeted symbolic execution.




```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

```c
int INSTR_LINE_13(int XY)
{
    assume_sa(8, !XY);
    assume_sa(7, XY);
    assume_sa(2, XY);
    return XY;
}
```

https://klee.github.io/

3rd International KLEE Workshop
15–16 Sep 2022 London
(Hybrid)

Targeted KLEE on Instrumented Code

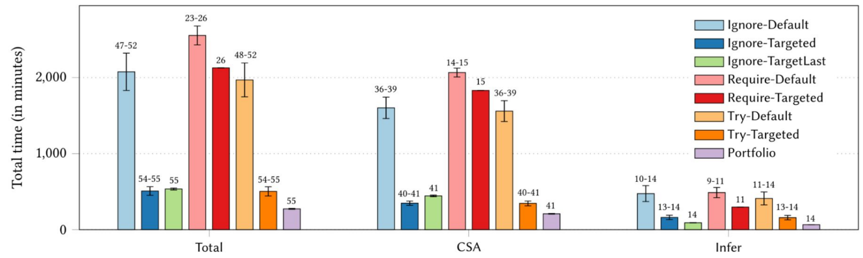


Figure 4: Total analysis time for KLEE for the injected bugs across all instrumentation strategies and search heuristics, as well as the *Ignore-TargetLast* special case and a *Portfolio* strategy. Numbers of bugs detected are shown above each error bar.