Imperial College London

Combining Static Analysis Error Traces with Dynamic Symbolic Execution

Frank Busse • Pritam M. Gharat • Cristian Cadar • Alastair F. Donaldson

3rd International KLEE Workshop on Symbolic Execution
15–16 September 2022, London

ISSTA 2022

Project: https://srg.doc.ic.ac.uk/projects/klee-sa/

Talk: https://youtu.be/J3lwc3bwTRg

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

Frank Busse
Imperial College London
London, United Kingdom
f.busse@imperial.ac.uk

Pritam Gharat
Imperial College London
London, United Kingdom
pritam01gharat@gmail.com

Cristian Cadar
Imperial College London
London, United Kingdom
c.cadar@imperial.ac.uk

Alastair F. Donaldson
Imperial College London
London, United Kingdom
alastair.donaldson@imperial.ac.uk

ABSTRACT

This paper reports on our experience implementing a technique for sifting through static analysis reports using dynamic symbolic execution. Our insight is that if a static analysis tool produces a partial trace through the program under analysis, annotated with conditions that the analyser believes are important for the bug to trigger, then a dynamic symbolic execution tool may be able to exploit the trace by (a) guiding the search heuristically so that paths that follow the trace most closely are prioritised for exploration, and (b) pruning the search using the conditions associated with each step of the trace. This may allow the bug to be quickly confirmed using dynamic symbolic execution, if it turns out to be a true positive, yielding an input that triggers the bug.

To experiment with this approach, we have implemented the idea in a tool chain that allows the popular open-source static analysis tools Clang Static Analyzer (CSA) and Infer to be combined with the popular open-source dynamic symbolic execution engine KLEE. Our findings highlight two interesting negative results. First, while fault injection experiments show the promise of our technique, they also reveal that the traces provided by static analysis tools are not that useful in guiding search. Second, we have systematically applied CSA and Infer to a large corpus of real-world applications that are suitable for analysis with KLEE, and find that the static analysers are rarely able to find non-trivial true positive bugs for this set of applications.

We believe our case study can inform static analysis and dynamic symbolic execution tool developers as to where improvements may be necessary, and serve as a call to arms for researchers interested in combining symbolic execution and static analysis to identify more suitable benchmark suites for evaluation of research ideas.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534384

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Software testing, symbolic execution, static analysis, KLEE, Clang Static Analyzer, Infer

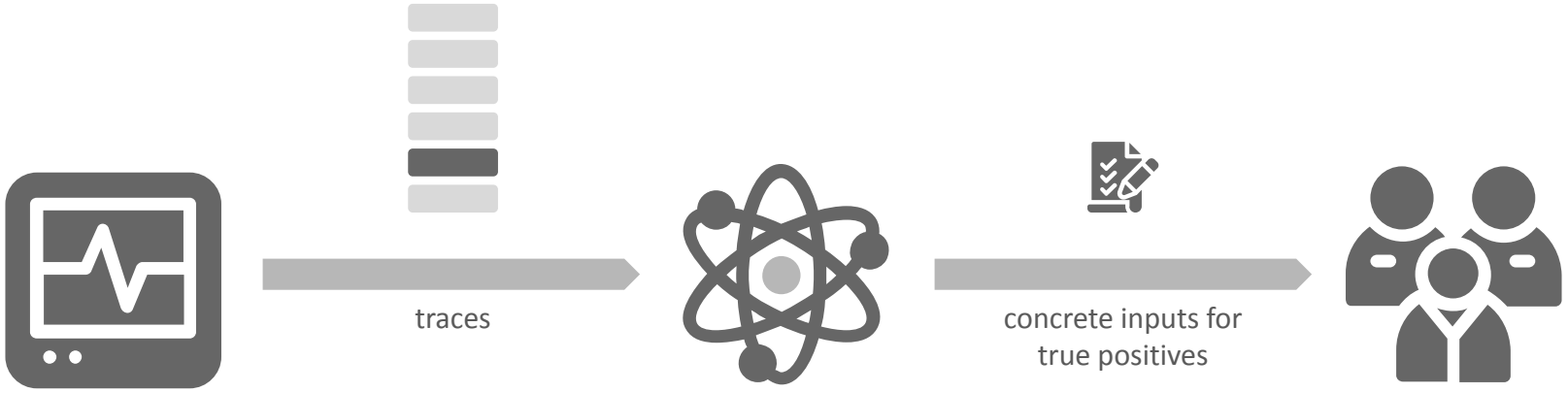
ACM Reference Format:
Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F. Donaldson. 2022. Combining Static Analysis Error Traces with Dynamic Symbolic Execution (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3533767.3534384

1 INTRODUCTION

Static analysis is a popular method for assisting developers in building correct and secure software. Despite the wide availability of static analysis tools, e.g. open source tools such as the Clang Static Analyzer [14], Frama-C [23] and Infer [10], and commercial offerings such as CodeSonar [29], Coverity Scan [16] and Fortify [22], many projects still disregard these tools due to incorrect bug reports, known as *false positives*. The more time developers waste investigating reports that turn out to be false positives, the more likely they are to abandon using a static analysis tool in the future.

We report our experience designing and evaluating a technique that aims to automate the process of confirming potential bugs reported by static analysis. If successful, such a technique could make static analysers more useful in practice by reducing the amount of time that would need to be spent triaging reports of potential bugs. Given a bug report from a static analysis tool, our idea is to use *dynamic symbolic execution* (DSE) [9] to try to *automatically* generate an input that triggers the reported bug.

Suppose a static analyser reports a possible bug at a given program location. The analyser typically yields a *trace* providing (possibly incomplete) details of a path through the program that, if followed, might trigger the bug. Our idea is to then apply a DSE tool to the program, additionally providing the DSE tool with information related to the trace. Rather than attempting to explore *all* paths of the program in the hope of finding *some* bug, the DSE tool exploits the trace to explore a massively-pruned subset of paths that agree with the trace, with the aim of confirming the *specific*


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

https://clang-analyzer.llvm.org/

C/C++/Objective-C

Clang Static Analyzer

traces

traces

Infer

C/C++/Objective-C

https://fbinfer.com/

Instrumentation

bitcode

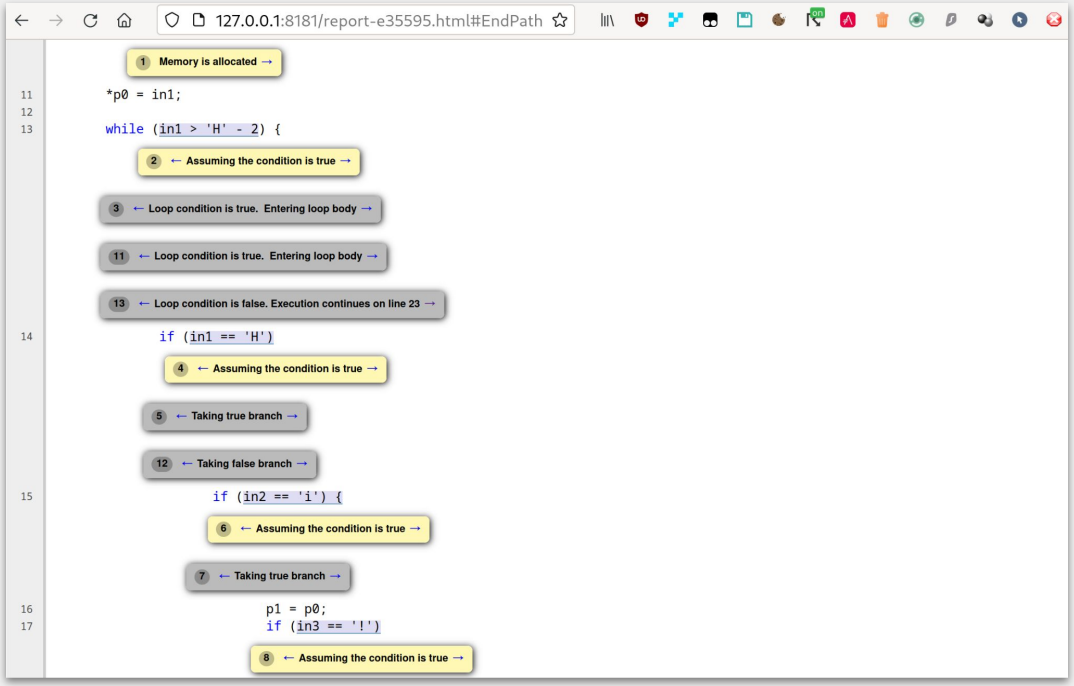
https://klee.github.io/

LLVM IR

(targeted) KLEE

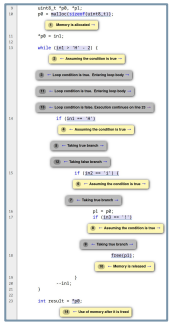
Example
Static Analysis Traces

```
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));  (1)
11      *p0 = in1;
12
13      while (in1 > 'H' - 2) {  (2) (7) (8)
14          if (in1 == 'H')  (3)
15              if (in2 == 'i') {  (4)
16                  p1 = p0;
17                  if (in3 == '!')  (5)
18                      free(p1);  (6)
19              }
20          --in1;
21      }
22
23      int result = *p0;  (9)
24      free(p0);
25      return result;
26  }
```



Example Instrumentation

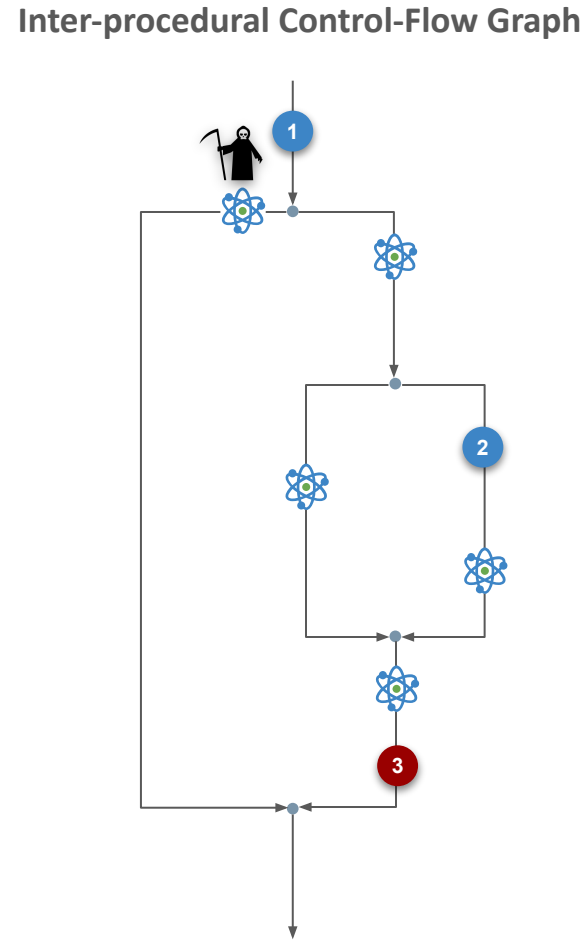
```
4   int main (int argc, char *argv[]) {
5       uint8_t in1 = argv[1][0];
6       uint8_t in2 = argv[1][1];
7       uint8_t in3 = argv[1][2];
8
9       uint8_t *p0, *p1;
10      p0 = malloc(sizeof(uint8_t));  (1)
11      *p0 = in1;
12
13      while (in1 > 'H' - 2) {  (2) (7) (8)
14          (3) if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



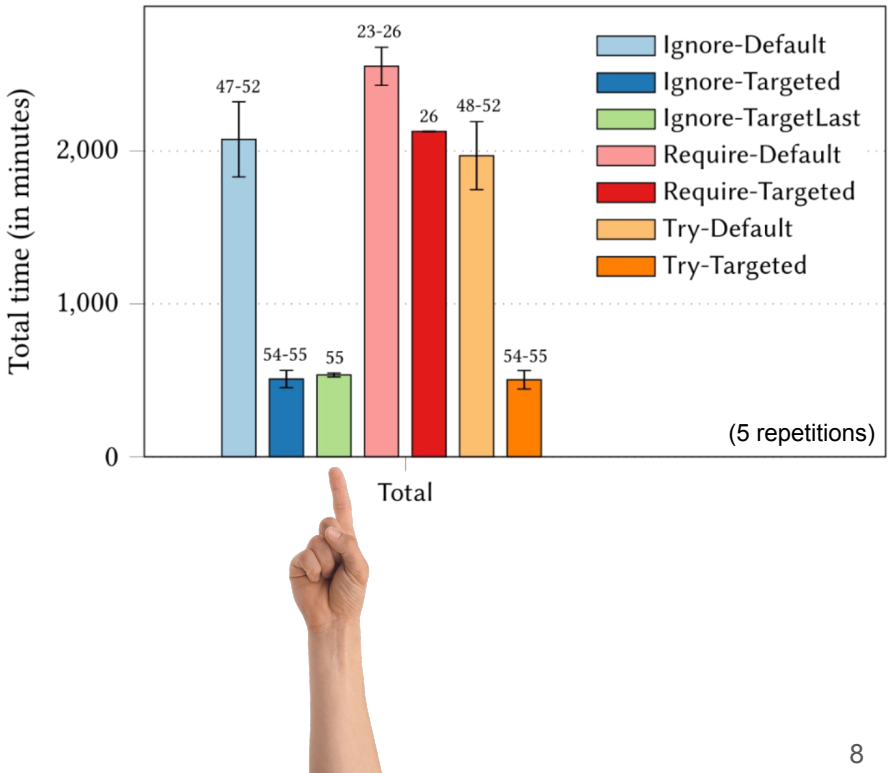
```
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    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**



Results



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