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*

bug reported by the static analyser. Our hypothesis is that—if the bug turns out to be a true positive—the DSE tool may be able to confirm the bug, producing an associated triggering test case, more efficiently than if it were run on the program in a default, undirected fashion. If the DSE tool is unable to trigger the bug then we still do not know whether the bug report is a true or a false positive, but the DSE tool might be able to produce an input that partly matches the bug report by following the trace as closely as possible, which might help to inform further manual analysis.

We present a practical implementation of our ideas for the popular open-source static analysis tools Clang Static Analyzer (CSA) [14] and Infer [10] and the popular open-source dynamic symbolic execution engine KLEE [6]. We have implemented several strategies for using the static analysis error trace as guidance during symbolic execution, and propose a novel search heuristic that prioritises exploration of paths that follow the trace.

Evaluating these ideas has led to two interesting negative results. The first negative result relates to our investigation of the potential for our technique to help in confirming real-world bugs detected by CSA and Infer. Unfortunately, a large and systematic survey of available C/C++ applications to which KLEE can be readily applied reveals that these analysers either do not find any bugs, report almost exclusively false positives, or only find kinds of bugs that KLEE has not been designed to detect (such as resource leaks or redundant writes to variables). This negative result prevents us from evaluating our technique on real-world bugs, but our survey constitutes an important empirical contribution: developers of static analysis tools can use our findings as a starting point for refining their techniques, and our experience can serve as a call to arms for researchers interested in combining symbolic execution and static analysis to identify more suitable realistic benchmarks.

In lieu of suitable real-world examples, we present a rigorous evaluation of our technique using a set of 55 synthetic bugs injected into benchmarks from the *GNU Coreutils* suite [26]—a de facto standard for evaluating DSE tools. While our results show the promise of our approach—with KLEE able to find this set of bugs 4.13 times faster (from a total of 2076.80 minutes down to only 503.24 minutes) when using static analysis guidance than when running in its default mode—it also highlights an interesting negative result. Most of the time, using solely the bug location as guidance is as effective as using the entire trace. This suggests that trace information is not very useful, and improving trace quality could benefit techniques like the one we are proposing. One hypothesis is that since static analysis tools are not evaluated based on their traces, relatively little attention has been paid to their quality.

In summary, our main contributions are:

(1) A technique that aims to convert a potential bug reported by a static analyser into a concrete test input that triggers the bug, via a form of dynamic symbolic execution restricted to explore only those paths that agree (or mostly agree) with the trace generated by static analysis.
(2) An implementation of this technique using two popular open-source static analysis tools, the *Clang Static Analyzer* (CSA) [14] and *Infer* [10], and an extension to the *KLEE* [6] open-source dynamic symbolic execution engine.
(3) Two negative results that could act as a call for arms for researchers working on static analysis or the combination of static analysis and dynamic symbolic execution: the fact that these static analysers cannot detect non-trivial bugs on benchmarks that can be analysed via symbolic execution; and the fact that, when applied to a collection of synthetic bugs, the full error traces produced by these static analysis tools are not much more useful than merely the location of the bug, with respect to accelerating symbolic execution.
(4) A complete artefact [32, 33] containing our implementation and benchmarks for reproducibility.

2 BACKGROUND

We provide necessary background on static analysis, dynamic symbolic execution, and the Clang Static Analyzer, Infer and KLEE tools used in our case study. Figure 1a is a contrived example featuring a bug that we use to illustrate these techniques: a use-after-free bug on line 48 is triggered when all of the following conditions hold: $y < 10$, $x > 10$, x is odd. In this case, n2 is aliased to n1 and is freed at line 47. The bug is that n2 is dereferenced on the next line.

2.1 Static Analysers and Traces

In order to scale, static analyses typically over-approximate parts of the information computed about the program under analysis. Over-approximation may lead to a static analyser reporting *false positives*—reported program defects that are not possible in practice.

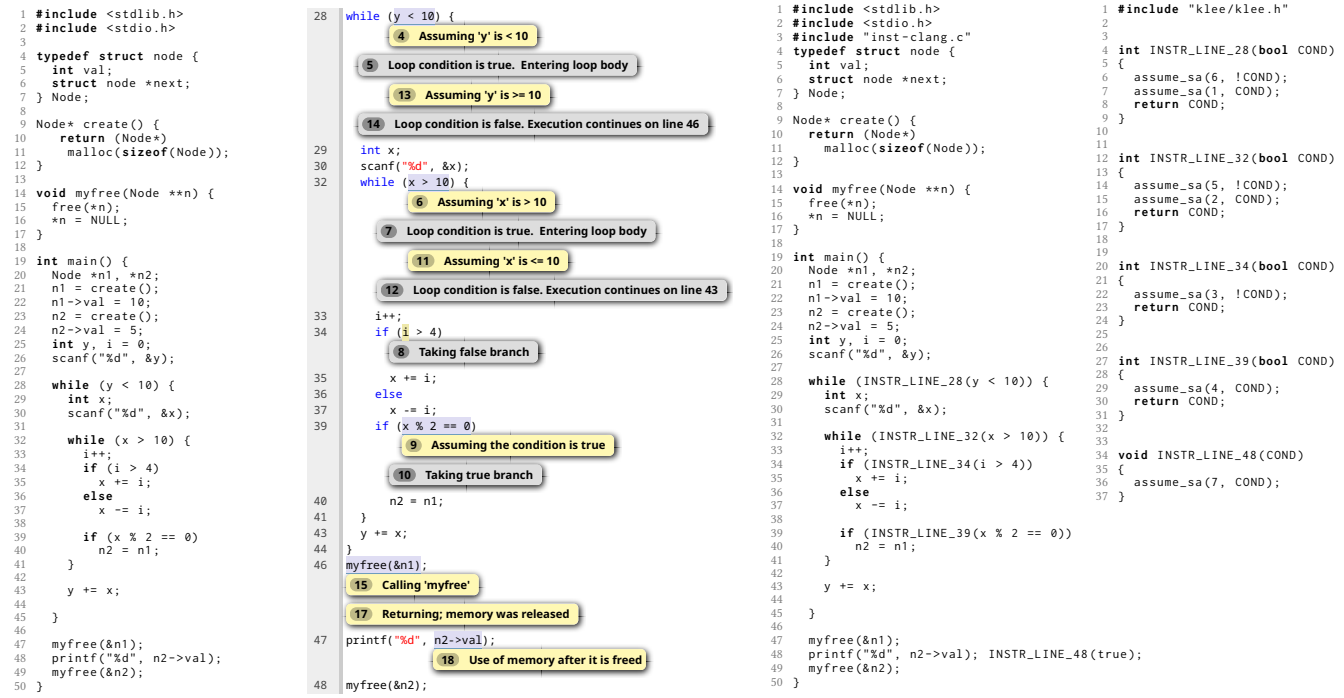
We focus on static analysers that report potential bugs in the form of a *trace*. A trace is a finite sequence of steps $\sigma_1, \sigma_2, \ldots, \sigma_n$ leading to the reported bug. Each step σ_i is a tuple ⟨*source-location, message*⟩. Here, *source-location* typically captures the file, line and column associated with the step. The file name is important as the trace could hop over multiple files. Information computed by the static analyser on conditions that must hold to reach the bug in the source code are captured in *message*. Table 1 provides the list of messages generated by a typical static analyser for C programs which can be used to reconstruct the path to the bug.

Table 1: Trace message types generated by a typical static analyser that can be used to recover the path to the bug.

Type	Message
Branch	Take true/false branch
Switch	Control should jump to case ⟨case-info⟩
Constraining a variable	Assume ⟨var⟩ is equal to ⟨constant⟩/⟨var2⟩

Clang Static Analyzer (CSA) [14] is a lightweight source code analysis tool that finds bugs in C, C++, and Objective-C programs. CSA performs interprocedural analysis, but is restricted to a single translation unit.[1] Any call to a function outside the translation unit is over-approximated. The analyser can identify defects such as division by zero, null pointer dereferences, usage of uninitialised values, and dead code. Since the analysis is restricted to a single file, the source-location component of a trace step omits the file name.

[1]We were recently made aware of *CodeChecker*, a new extension of CSA with cross-translation unit analysis [15]; we have not yet integrated it with our approach.



(a) Program with use-after-free bug on line 48.

(b) (partial) CSA generated trace

(c) CSA-driven instrumented code

Figure 1: Bug report generated by CSA for the motivating example in Figure 1a and the instrumented source code.

Figure 1b provides the HTML view of a partial trace generated by CSA when applied to the running example of Figure 1a. Table 2 shows the trace generated by CSA for this example, ignoring information-only messages (e.g. message 4 of Figure 1b), so that Step 1 of Table 2 corresponds to message 5 of Figure 1b.

Step 1: The condition of the outer while loop on line 28 is true.
Step 2: The condition of the inner while loop on line 32 is true.
Step 3: The if condition on line 34 is false indicating that `i` ≤ 4.
Step 4: The if condition on line 39 is true indicating that `x` is even at line 39. As a result, `n2` is now a copy of `n1` and points to the same object as `n1`.
Steps 5–6: The conditions of the two while loops on lines 32 and 28 are false. Thus, the trace involves only a single iteration of both while loops.
Step 7: Access of the `val` field of `n2` at line 48. The object pointed by `n1` is freed at line 47, thereby freeing the object pointed by `n2` because `n1` and `n2` are aliases.

Infer [10] is an automated analyser for C, C++, Objective-C and Java programs. In the context of C, C++, and Objective-C, Infer checks e.g. for null-pointer dereferences, memory leaks or coding conventions. For scalability, Infer uses a compositional interprocedural analysis, which is sound with respect to the underlying model of separation logic [11]. Unlike CSA, the interprocedural analysis is not restricted to a single translation unit, thus a trace can span multiple source files.

When run on the example in Figure 1a, Infer does not find the use-after-free bug with the default options, nor the extended ones that we use in our experiments. However, with options `-no-filtering -pulse -pulse-model-return-nonnull=malloc`, it finds the use-after-free bug, together with a memory leak (caused by reassigning `n2` in line 40) and two null-pointer dereferences (triggered if the allocations fail).

Table 2: Trace generated by CSA (ignoring steps that do not affect the control flow) for the example in Figure 1a.

Step no.	Line no.	Message
1	28	Loop condition is true
2	32	Loop condition is true
3	34	Taking false branch
4	39	Taking true branch
5	32	Loop condition is false
6	28	Loop condition is false
7	48	Use-after-free error

Infeasible traces. Static analysers sometimes generate infeasible traces and we observed this for both CSA and Infer—here we provide an example for Infer to illustrate the problem. Figure 2a shows a simplified version of Figure 1a and Figure 2b an associated trace generated by Infer. Infer claims to have found a memory leak at the exit of the main function, which is a false positive as `n1` is always freed and `n2` is either uninitialised or an alias of `n1`. Moreover, the trace itself leads to an infeasible path condition in Step 6:

Step 1: The condition of the while loop at line 27 is true, i.e. `x` > 10.
Step 2: The if condition at line 30 is false, indicating that `i` is odd.

```c
#include<stdlib.h>
#include<stdio.h>

typedef struct node {
  int val;
  struct node *next;
} Node;

Node* create() {
  return (Node*)
    malloc(sizeof(Node));
}

void myfree(Node **n) {
  free(*n);
  *n = NULL;
}

int main() {
  Node *n1, *n2;
  int x, i = 0;
  scanf("%d", &x);

  n1 = create();
  n1->val = 10;

  while (x > 10) {
    i++;

    if (i % 2 == 0)
      x += i;
    else
      x -= i;

    if (x % 2 == 0)
      n2 = n1;
  }

  myfree(&n1);
  return n2->val;
}
```

(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.

3.2 Constraining Search via `assume_sa`

An `assume_sa (step, condition)` call means that the SA believes that the error of interest can be reached when `condition` holds at `step` of the trace. There is potential for pruning the space of paths to be explored by DSE by restricting attention to only those paths where these conditions really do hold. However, if the SA is *incorrect* then the search may be overly-constrained, risking the bug being missed if it is indeed a true positive.

We present three strategies for constraining the search using `assume_sa` conditions: *Ignore*, *Require*, and *Try*.

Ignore Calls to `assume_sa` are ignored. This provides a useful baseline against which to compare other strategies. The coverage-guided search performed by DSE can be sensitive to the exact syntactic structure of the input program, so that even syntactic instrumentation can affect DSE performance [5]. The *Ignore* strategy allows us to compare DSE performance against other strategies with respect to syntactically-identical programs.

Require DSE exploration requires all the encountered `conditions` to hold, and adds them to the path condition. Any path for which an encountered condition is infeasible is terminated.

Try This strategy is more liberal than *Require*. It prunes the search based on feasible conditions by adding them to the path condition, and acts like a no-op when conditions are infeasible.

On our running example of Figure 1a, using KLEE with the guidance provided by CSA leads to the following results for each strategy: 11 paths with the *Ignore* strategy (no guidance) to find the bug, and 1 path by using the *Require* and *Try* strategies.

When KLEE uses the inconsistent trace generated by Infer for the example in Figure 2a, it explores 2 paths with the *Ignore* strategy to find the bug, fails to find the bug with the *Require* strategy, due to the Infer trace featuring mutually inconsistent conditions, and explores 2 paths with the *Try* strategy to find the bug, because the inconsistency is ignored. §5 provides an in-depth comparison of the effectiveness of these strategies in practice.

3.3 Guiding Search to Follow the Trace

The *Require* and *Try* strategies presented above enable DSE to filter out paths that do not satisfy the conditions given by the SA trace. However, this still allows DSE to explore many irrelevant paths: the locations in the adjacent steps of a trace are often at a considerable distance from one another, so that DSE may have to explore multiple paths to reach the next step, and may reach locations from which the next step is actually unreachable. Moreover, some DSE search heuristics tend to create a shallow but wide exploration tree [6], meaning that many, possibly redundant paths are explored and DSE may fail to reach the target due to state explosion.

We propose a new search heuristic, which we call the *Targeted* search heuristic. The *Targeted* heuristic on one hand terminates all paths for which the last step of the trace is unreachable and, on the other hand, guides the exploration along the trace. To follow the trace, our heuristic first prioritises further exploration of states that have already reached the largest number of consecutive steps of the SA trace. If there are multiple such states, it prioritises those whose current program point is closest to the program point associated with the next step in the SA trace. The idea of this heuristic is to push forward exploration of those states that have already followed a substantial prefix of the SA trace in the hope that it may be possible to continue to follow the trace, improving the chances of confirming the possible bug to which the trace corresponds.

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
1  Function update(currentState, newStates, terminatedStates):
2      updateCurrent (currentState)
3      foreach state : newStates do
4          insert (state)
5      foreach state : terminatedStates do
6          remove (state)
7  Function insert(state):
8      state.distance ← computeDistance(state)
9      if state.distance = ∞ then
10         while state.distance = ∞ ∧ state.lastStep < maxStep do
11             state.lastStep ← state.lastStep + 1
12             state.distance ← computeDistance(state)
13     state.pathID ←
         instructionPathIDs[state.pc][state.lastStep]
14     if state.distance = ∞ then
15         terminate(state)
16     else
17         states[state.lastStep][state.pathID].add(state)
18         activePathIDs[state.lastStep].add(state.pathID)
19 Function select()→ State :
20     nextPathID ←
         activePathIDs[maxActiveStep].selectRoundRobin()
21     candidates ←
         states[maxActiveStep][nextPathID].selectByDistance()
22     return candidates.pickRandomly()
```

The next step in a trace might be reachable via different paths in the call graph, simply because it is located in a function that is called from different program points. Prioritising only a single call path with the shortest distance might prevent DSE from reaching this step as necessary path constraints to reach the correct branch might only be fulfilled on other call paths. We refine our search heuristic to circumvent this by introducing *path identifiers*. Path identifiers enumerate all unique (sub-)paths in the interprocedural control flow graph to a step and allow us to partition the state space such that states with shortest distances can be selected for each individual call path. For example, suppose a trace step is located in some function A, and that function B makes three calls to A: two calls in different branches of a switch statement and one call at the end of the function body. Each of the two switch statement branches that call A would get a unique path identifier, and the remaining code-paths to the function return with the third call of A would get a (single) unique path identifier. It is important to notice that this approach identifies a small number of call paths and not a possibly significant number of program paths.

As shown in Algorithm 1, a search heuristic implements two functions: an `update` function to accommodate the progress of the DSE engine, and a `select` function to pick a new state for exploration. More concretely, `update` inserts newly forked states into and removes terminated states from internal data structures. Additionally, data structures have to be updated when the last selected (current) state gets closer to or reaches its next step, or progresses into a path with a different identifier. We leave out the description for the less interesting `remove` and `updateCurrent` functions and focus on `insert`. On insertion, the distance of a state to its next step gets computed. If the next step is unreachable (line 9), adjacent steps are evaluated until a reachable step is found (lines 10-12). If no such step can be found, the state is terminated (line 15). Otherwise, the state's path identifier is set, based on the current program counter (instruction) and the next targeted step (line 13), before it is inserted into the partitioned state set (line 17) and the set of active path identifiers is updated. Early experiments indicated that skipping unreachable steps and allowing more states to advance reduces the time needed to cover the final target.

When selecting a state for further exploration, only the ones with the highest current step number are considered. Such states might progress to the next step on different call paths as described above, hence one of such paths is selected in round-robin manner (line 20). After selecting a candidate set of states with shortest distance to the next step on the previously selected path (line 21), a single one is chosen randomly for further exploration (line 22).

The distance computation itself is similar to coverage-guided heuristics in other DSE engines with the subtle but important difference that we use maximum distances through functions. Using maximum distances to approximate shortest distances may seem counter-intuitive at first and contradicts other implementations [6, 24] but is based on a phenomenon we observed during development: Many functions in real-world code use early returns, for instance for error handling, and have very short shortest paths. Now, when a state enters a function and does not take the short error path, its distance value gets re-computed and would increase when the calculation would be solely based on minimum distances. States that have not called this function yet might have a shorter distance now, causing the search heuristic to pick those states and pulling them into the function as well. This "herd-effect" can simply be prevented by using maximum distances with monotonically decreasing distance values for non-looping code.

4 INVESTIGATING REAL-WORLD BUGS

In order to evaluate our technique, we sought a set of C/C++ applications suitable for analysis with KLEE, and also containing bugs that could be found by either CSA or Infer. We approached this by collecting all applications used in the evaluation sections of the 12 most recent KLEE-related papers listed at http://klee.github.io/publications/ at the time of writing. We excluded applications that were only usable in the context of an extension of KLEE being presented in a specific paper (e.g. the Linux kernel), as well as tiny benchmark programs. The applications considered are listed in Table 3. It is likely that many of these applications were not just tested with symbolic execution engines but also with static analysis tools in the past, so that they might have contained bugs that could have been found via static analysis but that have since been fixed. Therefore, we chose older releases of these applications from around June 2015—when Infer was released as open source. We ran both CSA and Infer on these applications, and *manually* investigated the bug reports from these tools to determine whether or not they were false positives.

All experiments were performed using CSA v.11.0.1 [14] and Infer v.1.0.0 [10], using the default options of both tools. We also tried running Infer with an extended set of options[3] to increase its ability to find real bugs. However, with these options we often got a huge number of reports (e.g. in the thousands). Analysis of a sample of these reports suggested a high false positive rate, making manual analysis of even a sizeable subset of the reports infeasible. We thus reverted to using Infer's standard options.

Table 3 shows the number of reports generated for each application by both static analysers for the memory-related bug classes that KLEE also supports. We investigated up to 20 reports per application and manually categorised them into true and false positives. Almost all reports turned out to be false positives. The few true positives are either in library functions that are not reachable via the main application, depend on failing system calls or are caused by missing error handling code for memory-allocating functions. Bugs that depend on failing system calls are typically unreachable for KLEE as it only models few such failures and does not model out-of-memory scenarios. Furthermore, these bugs are often trivial to confirm, as they locally depend on the environment behaviour rather than the application input.

As this did not yield any usable bugs, we turned our focus to CoREBench [3], a collection of 70 complex regression errors that were systematically extracted from the repositories and bug reports of four open-source software projects: GNU Make, GNU Grep, GNU Findutils, and GNU Coreutils. For each error, it provides information about the commit that introduced the error, the commit that fixed it, and a validating test case. We analysed all error-introducing commits with CSA and Infer, and evaluated whether KLEE is able to detect the respective bug by using its test case as concrete input. KLEE found 17 bugs from a list of 70 regression errors. The relatively low detection rate is due to the fact that CoREBench mostly contains functional errors, such as correct output colouring, which KLEE cannot detect without extra oracles.

Unfortunately, none of the bugs detected by KLEE were reported by CSA or Infer. In turn, CSA reports 142 potential bugs across all the versions of the four projects, while Infer reports 171 bugs—none of these are known bugs and are not reported by CoREBench or KLEE. We manually checked a few of these and found that they are false positives. To investigate the full set of reports, we instrumented the source code (*Try* strategy) using the trace generated by the bug reports of CSA and Infer and ran KLEE on the instrumented code with the *Targeted* searcher described in §3.3. For 18 of the CSA reports and 8 of the Infer reports, KLEE terminated without finding the bug within 30 minutes. This indicates that either the trace is incorrect, the report is a false positive or KLEE could not find the bug due to its configuration (e.g. insufficient symbolic input). For the remaining 124 CSA reports and 163 Infer reports, KLEE either

[3] `-pulse -no-filtering -no-default-checkers -bufferoverrun -headers -biabduction -j 1`

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

timed out or ran out of memory without finding the bug. While these results do not allow us to draw a definite conclusion, the fact that no bugs were confirmed as true positives, and our manual assessment that several reports were indeed false positives, suggests that CSA and Infer are either not effective at finding real-world bugs on these benchmarks, or they create incorrect or useless traces to real bugs.

Recently, Joshy et al. [31] applied two commercial static analysers to BugBench [36] and CoREBench. From the generated reports, their proposed LCA-patching approach was able to confirm 48 true positives. None of them were listed as bugs by either BugBench or CoREBench. Consequently, we ran CSA and Infer on the same set of applications,[4] released between 2002 and 2010, but none of the bug locations for the true positives were reported. However, by using non-default flags for Infer, four reports were generated with bug locations at the same line as the true positives. We have not yet had time to investigate these cases further, but plan to do so in future work. This again illustrates the challenge of using static analysis, as the likely true positives are hidden among 7434 reports and only shown when the SA is configured to be less precise. Also, with such a huge number of reports, running our proposed approach would take on the order of days or weeks using a regular machine.

Finally, we considered reproducing previous bugs found by CSA and Infer, but could not find a list of historical bugs found by CSA, while documented bugs found by Infer are mostly in Java code. A set of 15 null dereference bugs in a C application, OpenSSL, are documented as having been found previously by Infer (documented at [21]), but are not detected by the latest version of Infer.

Despite a systematic investigation and a large amount of manual effort, we were unable to find a suitable set of real-world bugs on which to evaluate our technique. To our surprise, CSA and Infer are limited in their ability to find genuine bugs in the corpus of applications that tend to be used for DSE evaluations, and the kinds of bugs they can find are often not the types of bugs that KLEE is capable of analysing.

We hope that developers of static analysis tools may find this negative result useful: perhaps there is scope for refining their tools so that they exhibit a higher ratio of true to false positives in the domain of these applications, though of course this needs to be traded against the requirements of the applications that the core users of these tools are interested in. At the same time, our results could also be of interest to DSE researchers, who could develop better techniques for guiding execution along a trace, thus reducing the number of inconclusive cases, or extend their tools to support more bug types. Finally, our experience may also serve as a call to arms for researchers interested in combining dynamic symbolic execution and static analysis to work together on a set of suitable benchmarks that exhibit non-trivial bugs that are nevertheless in scope for detection with both kinds of techniques.

5 EVALUATION ON INJECTED BUGS

Having established (§4) that it is not straightforward to find real-world C/C++ benchmarks suitable for analysing the combination of current state-of-the-art static analysis and dynamic symbolic execution tools (irrespective of the technique proposed in this paper), we turn our attention to systematically evaluating our idea based on injecting faults into real-world applications. Injected faults are less appealing than real-world bug reports, but do allow us to conduct a large-scale evaluation of our technique, and are not subject to the limited abilities of CSA and Infer when it comes to finding defects in the kinds of code bases to which KLEE can be readily applied.

Our fault-injection experiments were performed in Docker containers running Ubuntu 18.04 on a set of homogeneous machines with Intel Core i7-4790 CPUs at 3.6 GHz and 16 GiB of RAM. We used CSA v.11.0.1 [14], Infer v.1.0.0 [10], and a fork of KLEE branched from Git revision `04f5031c`, configured to use LLVM 11.0.1 [34] and Z3 4.8.8 [18] as constraint solver. As timeouts we used 2 h for KLEE and 1 min for solver invocations.

To increase Infer's bug detection rate, we used the extended options discussed in §4 (see Footnote 3).

5.1 Benchmark Selection

As benchmarks for our fault-injection experiments, we choose the tools from *GNU Coreutils* 8.31 [26]. These tools contain commonly used command-line utilities on UNIX-based systems, such as `ls`, `mkdir` and `echo`. Since being introduced in the original KLEE paper [6], they have become de facto DSE benchmarks.

From the set of 106 *Coreutils* we excluded all utilities that can interfere with the test setup (e.g. `kill`, `chmod`), contain unsupported LLVM intrinsics (e.g. `sha256sum`), cause an assertion in KLEE's Z3

[4] In fact, we analysed 4 out of 7 applications as we could not create a working build environment for Coreutils and Squid (no AMD64 support) on a recent machine.

front-end (e.g. `ptx`), are very similar to other tools (e.g. `base32` is very similar to `base64`) or can be fully explored using KLEE in less than 2 h on our test platform and thus can be considered as easy targets for KLEE-based bug finding (e.g. `[`).

We restricted the remaining 75 applications to a subset for which KLEE produced reasonably deterministic results. This was important to avoid the possibility of misattributing performance results to the success or otherwise of our technique when they are actually due to nondeterminism. After configuring KLEE to use state- and instruction-based limits [43], we run each application twice. When an application was found to cover different code between runs or cover the same instruction with a time difference of more than 2 min we excluded it from the experiment set. We also excluded applications that cover no new code after 10 min in their main source file (e.g. `echo.c`). This reduced the set to 10 applications for our experiments: `comm`, `csplit`, `cut`, `env`, `join`, `ln`, `nl`, `od`, `split`, and `uniq`.

5.2 Methodology for Injecting Bugs

To allow us to evaluate the effectiveness of the instrumentation strategies and search heuristic of §3 in allowing DSE to quickly confirm an SA report, we required a method for injecting bugs that are true positives by construction, of varying degrees of complexity.

We argue that the bug injection strategy we present is a reasonable means of introducing bugs that manifest only when a particular series of events occurs, giving a static analyser a chance to produce a report that highlights this series of events in its associated trace. Many other bug injection strategies are possible; by making our approach available as an artefact we provide an environment in which other researchers could experiment with different strategies. We do not claim that these bugs are representative of real-world bugs, and emphasise again that we would have preferred to evaluate the method primarily on real-world bugs, but cannot do so due to the negative results of the thorough survey described in §4.

Type of Injected Bugs. We inject two types of bugs: null-pointer dereferences and use-after-free bugs. Both types of bugs are introduced at the source-level. A null-pointer dereference bug may consist of multiple events where an event may involve:

(1) Assigning a pointer to NULL as first event (e.g. `p = NULL`).
(2) Creating a copy of a pointer (e.g. `p = q`). Zero or more such assignments may be present.
(3) Dereferencing a pointer, as the last event.

The events may span multiple procedures and control flow constructs. We vary the number of events from 1 to 4. Use-after-free bugs are similar, involving:

(1) Allocating an object on the heap.
(2) Creating zero or more copies of the pointer that holds the address of the allocated object.
(3) Freeing the dynamically-allocated memory as the second-to-last event.
(4) Dereferencing a pointer which points to this freed memory as the last event.

Selecting Locations for Injecting Bugs. When injecting bugs, we have two goals: (1) inject bugs that are true positives and (2) inject bugs that are hard for KLEE to find without any guidance (because the approach we propose, for using SA information to help a DSE tool to quickly find a bug, is not necessary if the DSE tool can already readily find the bug). To find interesting paths along which to inject our multi-event bugs, we run KLEE on the program in which we wish to inject bugs for a duration of two hours. This gives us a series of feasible paths explored by KLEE together with associated inputs that cause them to be followed, and information about how long it took KLEE to reach each program statement for the first time. We randomly selected a number of program statements at which to inject bugs, restricting to statements that KLEE did eventually manage to cover, but did not manage to cover during the first 10 minutes of its analysis.

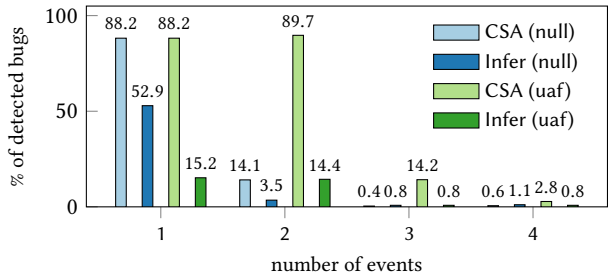


Figure 3: Percentage of multi-event null-dereference (null) and use-after-free (uaf) bugs detected by CSA and Infer.

In total, we injected 297 1-event, 632 2-event, 478 3-event, and 357 4-event bugs across all ten benchmark applications. The number of distinct 1-event bugs is lower than for other counts because a single statement may be reachable by multiple execution paths, catering for numerous multi-event bugs.

Figure 3 provides the percentage of multi-event bugs detected by CSA and Infer. We can make several observations regarding the effectiveness of the two SA analysers considered. First, CSA performs significantly better than Infer on these bugs. A possible explanation for this is that Infer's analysis is less sophisticated and precise for C code compared to code in other languages such as Java. Second, one can see that bugs with more events are harder to find. Looking at null-pointer dereference bugs, we can see that more then half of the 1-event bugs are found by both analysers (with CSA at more than 88%) but both analysers find under 1% of 3- and 4-event bugs. The performance for use-after-free bugs has the same trend, although CSA performs very well on both 1- and 2-event bugs. These results are not surprising, given that bugs with more events are more complex and hard to find. Although not necessarily representative of user errors, the bugs we inject are eminently detectable: they involve analysing simple sequences of assignments to global variables. We were surprised that CSA and Infer could not detect more of the multi-step bugs, and believe that our corpus of injected bugs could prove useful as a resource for testing the precision and soundness of these static analysers.

For the found bugs, Infer tends to generate shorter traces, typically in the range of 1–5 steps, whereas CSA generates longer traces with median lengths between 10–20 (maximum length 55). We note that Infer's use-after-free traces seem to follow a different format and contain different overlapping *parts*, for instance an *invalidation part* and a *use-after-lifetime part*. Our prototype does not fully support such traces as, on the one hand, it is not entirely clear how

to recover a sequential trace from such reports and, on the other hand, the use-after-free traces generated in our evaluation did not seem to require this functionality as all steps pointed to the same line. Although we would expect a sequential trace to the bug from a user's perspective, we acknowledge that this alternative format could contribute to the shorter traces that we observed and that the effectiveness of our technique on Infer's use-after-free traces could be increased by improving support for these traces.

5.3 Using DSE to Confirm Injected Bugs

For each benchmark, we used our methodology to obtain a set of up to 40 injected null-pointer dereference bugs and up to 40 use-after-free bugs for every path explored by KLEE. Recall that we gathered information on how long KLEE took to reach each statement associated with an injected bug path for the first time. Since KLEE explores numerous paths, we restrict the number of injections by allowing only those bugs for which KLEE takes ≥ 10 minutes to reach the last event in the bug. This avoids bugs that are already reasonably trivial for KLEE to find.

In spite of the above restriction, we inject a large set of bugs and we wish to identify a subset of diverse bugs to use for our evaluation of the techniques of §3, such that at least one, but ideally both, of CSA and Infer could find each bug, and such that the time taken by KLEE to cover the final event in each bug is reasonably high.

For every *Coreutils* application, bug type (null-pointer dereference or use-after-free), and an event count S ($1 \leq S \leq 4$) we approached this as follows. If there existed at least one injected bug that both CSA and Infer could detect, we selected the bug among this set for which the time taken by KLEE to reach the final event in the bug was maximal. Otherwise, if there existed at least one injected bug that *one* of CSA or Infer could detect, we selected from this set of bugs, again selecting the bug for which the associated time taken by KLEE to reach the last event was maximal.

For each application we then manually examined the total set of selected injected bugs, trying to select a diverse set in terms of the source code locations that they covered. We wished to select a total of $4 \times 2 \times 10 = 80$ bugs, due to there being 4 different event counts, 2 types of bugs, and 10 applications. However, CSA and Infer were not able to detect many 3- and 4-event bugs. Also, for a few applications, KLEE does not cover enough new instructions after 10 minutes. As a consequence, we ended up selecting 55 bugs for evaluation.

Results. Recall that we proposed three strategies for constraining the search (§3.2): (a) *Ignore*, (b) *Require* and (c) *Try*, and two search heuristics (§3.3): (a) *Default* and (b) *Targeted*, thereby creating six configurations with *Ignore-Default* as the baseline configuration. Additionally, we add two more modes: *Ignore-TargetLast* is a special case of the *Targeted* heuristic that is configured to only target the very last step in a trace and ignore intermediate ones. And *Portfolio* shows the analysis times that would be achieved if all other configurations were executed in parallel and independently, with the portfolio analysis stopping as soon as any configuration finds the bug. The results for this meta configuration are synthesised from the results we gathered for the other configurations.

The *Targeted* heuristic replaces the coverage-guided search in KLEE's default heuristic with our guided exploration of §3.3. We repeated each experiment five times using different random seeds for KLEE[5] and report average timing results, together with the standard error representing the variability in our data across multiple runs in Figure 4. The number of bugs found varies across runs (due to bugs found close to the timeout), hence the ranges above many bars.

Figure 4 shows a bar plot that compares the eight possible configurations in terms of bugs confirmed and total time taken.[6] Besides the *Portfolio* configuration, *Ignore-Targeted*, *Try-Targeted* and *Ignore-TargetLast* perform the best. The over-restrictive nature of the *Require* strategy hurts the performance and also finds fewer bugs. To our surprise, the *Try-Targeted* strategy with a speedup of 4.13× seems to work no better than the *Ignore-Targeted* strategy which gives a speedup of 4.09×. To better understand this, we looked at the percentage of conditions that are feasible in a trace generated by a static analyser. The percentage is very low for Infer traces (median 0%, maximum of 66.7%) compared to that of CSA traces (median 47.6%, maximum 92.9%). This could explain why *Try-Targeted* and *Ignore-Targeted* strategies are similar in performance for Infer traces. However, the percentage of feasible conditions for CSA traces is higher and yet the *Try-Targeted* strategy does not seem to gain any performance benefit. We consider this to be a useful negative result, as we would have expected the conditions emitted by the SA to be useful in constraining the search. Further investigation showed us that the conditions in the `assume_sa` are often already over-constrained. As a consequence, the path conditions already imply the conditions in the `assume_sa` and hence *Ignore* and *Try* are almost identical in our setup. Our data shows that 99.98% of feasible conditions were already implied by the respective path constraints.

Even more surprising for us is the result of the *Ignore-TargetLast* strategy that performs similar to *Ignore-Targeted*. Not just the event conditions are rarely useful, also the intermediate steps seem to be redundant in many cases. We further ascribe this observation to two properties of our search heuristic: (1) paths that cannot reach the final target are terminated such that the DSE engine does not get lost in irrelevant code paths, and (2) the proposed algorithm (§3.3) compartmentalises the call-graph and guides states on different paths to the target in round-robin manner instead of getting stuck as a conventional shortest-distance approach would do.

As expected, the investigation of the *Portfolio* strategy shows that the *Targeted* heuristic across all instrumentation strategies contributes most runs to the *Portfolio* result (*Ignore*: 12-21, *Require*: 11-13, *Try*: 9-16). The *Default* heuristic is favoured by fortune in a few cases (*Ignore*: 5-7, *Require*: 1-3, *Try*: 2-5) whereas the *Ignore-TargetLast* configuration (0 runs) is at best a close second across all runs. The relatively high number of contributed runs for *Require* (*Require-Default*: 1-3 and *Require-Targeted*: 11-13) suggests that the *Require* strategy occasionally pays off by dramatically pruning the search space without eliminating the bug. However, the outstanding

[5] To clarify, random seeds initialise KLEE's internal random number generator and do not serve as concrete input.

[6] Only the time taken by KLEE is reported. CSA/Infer and instrumentation times are not included because they are insignificant in comparison to the time taken by KLEE, and because our aim is to confirm bug reports after SA has been performed.

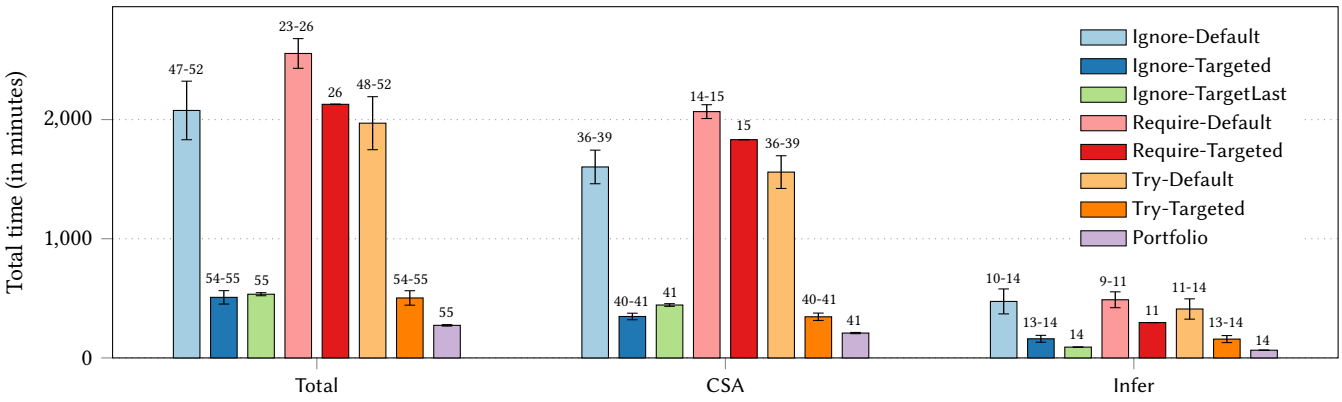


Figure 4: Total analysis time for KLEE for the injected bugs across all instrumentation strategies and search heuristics, as well as the *Ignore-TargetLast* special case and a *Portfolio* strategy. Numbers of bugs detected are shown above each error bar.

number of timeouts (14-19) and early terminations (13-15) shows that this strategy fails completely in some cases resulting in an overall bad performance. Early terminations may occur due to pruning the search space so much that it becomes very small and does not contain the bug, whereas timeouts occur when the search space is either too large and the bug could not be found or the bug-containing path was pruned away.

6 RELATED WORK

Prior work has proposed several techniques for pruning false positive reports from the results of SA, including techniques based on DSE. Our work distinguished itself through its focus on using out-of-the-box open-source SA tools; being oblivious to the type of SA report; and taking advantage of the full trace information.

We first discuss some prior approaches that use DSE to confirm SA reports. DyTa [25] uses Pex [46] to verify code contract violations detected by a static analyser. The location and violation condition associated with the SA report are used to prune paths that are statically determined not to be able to reach the bug. SANTE [12] combines slicing and DSE to validate SA reports. However, the DSE stage is unguided, using the default DFS search heuristic. Both DyTa and SANTE are presented in short papers which have no evaluation beyond an illustrative example.

Joshy et al. [31] introduce LCA-based syntactic patching to create executable code fragments from static warnings generated by commercial static analysers. These code fragments are subsequently tested with different approaches including DSE. Compared to our approach, the trace information (e.g. iteration count) is mostly discarded and reduced to the source lines reported in the traces.

Li et al. [35] look at the specific problem of validating memory leak reports produced by a SA using concolic execution. The approach uses the multiple steps included in the SA report, but the conditions associated with each step are ignored. While the approach has similarities to ours, the problem of memory leak validation has some particularities, which is perhaps why the approach was not applied to other types of SA bug reports.

Gao et al. [24] look at the problem of validating buffer overflow reports. It also uses multiple trace steps, but the conditions associated with each step are ignored. The main idea is to statically generate all the possible paths that follow the trace and then, during DSE, terminate any path that does not follow one of them. The evaluation shows that only a few such paths are created per bug report, while we would have expected a huge number. Furthermore, the evaluation misses essential information needed to understand and reproduce the experiments, e.g. related to the fault injection methodology, benchmark running times, and KLEE configuration.

We have carefully examined all the papers above to see if we can successfully use their benchmarks in our work. With the exception of the recent work by Joshy et al. [31], which uses commercial SA tools (see §4), we did not find any usable benchmarks, reinforcing our findings regarding the mismatch between the bugs found by SA and DSE, especially with respect to open-source tools.

Brown et al. [4], Feist et al. [20] and Babić et al. [1] combine custom static checkers or a dedicated SA with DSE to efficiently find vulnerabilities. Writing custom static checkers or modifying existing SA tools can indeed lead to better synergy with DSE. However, an explicit goal of our work is to use popular out-of-the-box SA tools and understand how the information they provide can be used by a DSE tool. While we hope static analysers will improve to provide better information to help DSE confirm the generated reports, demanding such changes is likely unrealistic.

Work on directed symbolic execution [19, 37, 38, 45, 47] aims to construct cases that reach a particular program statement. Therefore, they could also be used to generate test cases that reach the location of an SA report. These techniques often use search heuristics which are similar to the one used by our approach [37, 38], except that our heuristic is designed to use multiple steps rather than a single target location. However, our evaluation shows that using solely the target location seems to be as effective as using the full trace information, so until this changes, directed symbolic execution can be competitive for validating SA reports.

Research on reproducing field failures sometimes involves processing traces, similar to SA reports, and these techniques often use DSE to reproduce an input that follows the trace [30, 48]. However, field traces do not include event information and thus our condition guidance strategies are not relevant.

In addition to DSE, other techniques have been used to validate SA reports, such as SMT-based refutation [42], deductive verification [39], bounded model checking [41] and random testing [17].

Beyond the problem of validating false positives, SA and DSE have been combined effectively for several problems, including bug finding and verification [2, 13, 28, 49].

7 DISCUSSION AND CONCLUSIONS

We have presented our experience investigating a novel method for integrating traces from two off-the-shelf static analysers into dynamic symbolic execution, with the aim of leveraging DSE to confirm true positive bug reports. Our investigation has led to two interesting negative results:

C/C++ benchmarks suitable for analysis with DSE tools are not handled well by CSA and Infer. As described in §4, we undertook a thorough survey of the C/C++ benchmarks that have been used over recent years in works that evaluate the KLEE DSE tool. Our experience applying CSA and Infer to these benchmarks is that they find very few real bugs, despite the fact that numerous known bugs are present. We hope this serves as a useful call to arms for the SA community: SA tools will inevitably be somewhat imprecise, but our results point to a good set of challenge benchmarks that could be used to guide the tuning of such tools.

The traces generated by CSA and Infer are not useful for accelerating DSE. Our evaluation in §5 shows that, at least with respect to the techniques we have presented, the traces that CSA and Infer produce are not useful for accelerating DSE. They provide no, or only marginal benefit to the speed with which bugs can be confirmed compared with simply providing DSE with the bug location. This may not serve as a call to arms to the SA community, who are presumably mainly interested in whether the traces their tools generate can be understood by humans. However, the idea of having a static analyser generate an alternative, tool-friendly trace for consumption by another program analysis tool, is a promising direction for future research.

As well as investigating the future directions associated with these negative results, another area for future work would be to widen the scope of our investigation to consider other analysis tools, and analysers for other programming languages. We restricted to open-source analysers in this work because it is not always straightforward to get licenses for commercial analysers, and publicly reporting on the results that commercial analysers produce is often not allowed. Nevertheless, it would be interesting to see whether commercial static analysers fare better on the benchmarks we have considered compared with the open-source analysers that we tried.

ACKNOWLEDGMENTS

We would like to thank Manuel Carrasco and the anonymous reviewers for their valuable feedback on the paper. This research has received funding from the DSO National Laboratories, Singapore, the EPSRC IRIS Programme Grant (EP/R006865/1), and from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 819141).

DATA AVAILABILITY STATEMENT

All tools and detailed instructions to reproduce our study are openly available from Zenodo https://doi.org/10.5281/zenodo.6539575 and the project page https://srg.doc.ic.ac.uk/projects/klee-sa/.

REFERENCES

[1] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed Dynamic Automated Test Generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)* (Toronto, Canada). https://doi.org/10.1145/2001420.2001423

[2] Bernhard Beckert, Mihai Herda, Michael Kirsten, and Shmuel S. Tyszberowicz. 2020. Integration of Static and Dynamic Analysis Techniques for Checking Non-interference. In *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*. Lecture Notes in Computer Science, Vol. 12345. Springer, 287–312. https://doi.org/10.1007/978-3-030-64354-6_12

[3] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors". In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'14)* (San Jose, CA, USA). https://doi.org/10.1145/2610384.2628058

[4] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)* (Virtual Conference).

[5] Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering, New Ideas Track (ESEC/FSE NI'15)* (Bergamo, Italy). https://doi.org/10.1145/2786805.2803205

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, CA, USA).

[7] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)* (San Francisco, CA, USA). https://doi.org/10.1007/11537328_2

[8] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)* (Alexandria, VA, USA). https://doi.org/10.1145/1455518.1455522

[9] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[10] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proc. of the 3rd International Conference on NASA Formal Methods (NFM'11)* (Pasadena, CA, USA).

[11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[12] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2011. The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C Program Debugging. In *Proc. of the 5st International Conference on Tests and Proofs (TAP'11)* (Zurich, Switzerland).

[13] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2006. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *Proc. of the 28th International Conference on Software Engineering (ICSE'06)* (Shanghai, China). https://doi.org/10.1145/2884781.2884843

[14] Clang Static Analyzer 2022. https://clang-analyzer.llvm.org.

[15] Code Checker 2021. Clang 12 documentation: Cross Translation Unit (CTU) Analysis. https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html.

[16] Coverity Scan 2022. https://scan.coverity.com/.

[17] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining static checking and testing. In *Proc. of the 27th International Conference on Software Engineering (ICSE'05)* (St. Louis, MO, USA). https://doi.org/10.1145/1062455.1062533

[18] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)* (Budapest, Hungary).

[19] Peter Dinges and Gul Agha. 2014. Targeted Test Input Generation Using Symbolic-Concrete Backward Execution. In *Proc. of the 29th IEEE International Conference on Automated Software Engineering (ASE'14)* (Västerås, Sweden). https://doi.org/10.1145/2642937.2642951

[20] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. 2016. Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-after-Free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'16)* (Los Angeles, California, USA). https://doi.org/10.1145/3015135.3015137

[21] Finding inter-procedural bugs at scale with Infer static analyzer 2022. https://engineering.fb.com/2017/09/06/android/finding-inter-procedural-bugs-at-scale-with-infer-static-analyzer/.

[22] Fortify Static Code Analyzer 2022. https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer.

[23] Frama-C 2022. https://frama-c.com.

[24] Fengjuan Gao, Yu Wang, Linzhang Wang, Zijiang Yang, and Xuandong Li. 2020. Automatic Buffer Overflow Warning Validation. *Journal of Computer Science and Technology* 35, 6 (2020), 1406–1427. https://doi.org/10.1007/s11390-020-0525-z

[25] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. 2011. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results. In *Proc. of the 33rd International Conference on Software Engineering (ICSE'11)* (Honolulu, HI, USA). https://doi.org/10.1145/1985793.1985971

[26] GNU Coreutils 2022. https://www.gnu.org/software/coreutils/.

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)* (Chicago, IL, USA).

[28] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proc. of the 22st USENIX Security Symposium (USENIX Security'13)* (Washington, D.C., USA).

[29] GrammaTech Inc. 2022. CodeSonar. https://www.grammatech.com/codesonar-cc.

[30] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)* (Zurich, Switzerland).

[31] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating Static Warnings via Testing Code Fragments. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'21)* (Online). https://doi.org/10.1145/3460319.3464832

[32] KLEE-SA Artefact 2022. https://doi.org/10.5281/zenodo.6539575.

[33] KLEE-SA Project 2022. https://srg.doc.ic.ac.uk/projects/klee-sa/.

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA).

[35] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. 2013. Dynamically Validating Static Memory Leak Warnings. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'13)* (Lugano, Switzerland). https://doi.org/10.1145/2483760.2483778

[36] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools.*

[37] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proc. of the 18th International Static Analysis Symposium (SAS'11)* (Venice, Italy).

[38] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)* (Saint Petersburg, Russia). https://doi.org/10.1145/2491411.2491438

[39] T. T. Nguyen, P. Maleehuan, T. Aoki, T. Tomita, and I. Yamada. 2019. Reducing False Positives of Static Analysis for SEI CERT C Coding Standard. In *IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI'19) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER IP'19).* https://doi.org/10.1109/CESSER-IP.2019.00015

[40] Martin Nowack and Cristian Cadar. 2020. KLEE Symbolic Execution Engine in 2019. *International Journal on Software Tools for Technology Transfer* (2020). https://doi.org/10.1007/s10009-020-00570-3

[41] H. Post, C. Sinz, A. Kaiser, and T. Gorges. 2008. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. In *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)* (L'Aquila, Italy). https://doi.org/10.1109/ASE.2008.29

[42] M. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. Nicole. 2019. SMT-Based Refutation of Spurious Bug Reports in the Clang Static Analyzer. In *Proc. of the 41th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion'19)* (Montreal, Canada). https://doi.org/10.1109/ICSE-Companion.2019.00026

[43] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. 2022. A Deterministic Memory Allocator for Dynamic Symbolic Execution. In *Proc. of the 36th European Conference on Object-Oriented Programming (ECOOP'22)* (Berlin, Germany).

[44] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)* (Lisbon, Portugal). https://doi.org/10.1145/1081706.1081750

[45] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. eXpress: guided path exploration for efficient regression test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)* (Toronto, Canada). https://doi.org/10.1145/2001420.2001422

[46] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)* (Prato, Italy). https://doi.org/10.1007/978-3-540-79124-9_10

[47] Zhihong Xu and Gregg Rothermel. 2009. Directed Test Suite Augmentation. In *Proc. of the 16th Asia-Pacific Software Engineering Conference (ASPEC'09)* (Penang, Malaysia).

[48] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging. In *Proc. of the 5th European Conference on Computer Systems (EuroSys'10)* (Paris, France). https://doi.org/10.1145/1755913.1755946

[49] Bin Zhang, Chao Feng, Bo Wu, and Chaojing Tang. 2016. Detecting integer overflow in Windows binary executables based on symbolic execution. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).* 385–390. https://doi.org/10.1109/SNPD.2016.7515929