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.

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:

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 then to 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 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 of 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 $σ_1, σ_2, \ldots, σ_n$ 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>
<thead>
<tr>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>Take true/false branch</td>
</tr>
<tr>
<td>Switch</td>
<td>Control should jump to case (case-info)</td>
</tr>
<tr>
<td>Constraining a variable</td>
<td>Assume (var) is equal to (constant)/(var2)</td>
</tr>
</tbody>
</table>

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. 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.

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.

---

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.
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 \( i \leq 4 \).
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 \( n2 \).
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.

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

<table>
<thead>
<tr>
<th>Step no.</th>
<th>Line no.</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>Loop condition is true</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>Loop condition is true</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
<td>Taking false branch</td>
</tr>
<tr>
<td>4</td>
<td>39</td>
<td>Taking true branch</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>Loop condition is false</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
<td>Loop condition is false</td>
</tr>
<tr>
<td>7</td>
<td>48</td>
<td>Use-after-free error</td>
</tr>
</tbody>
</table>

(c) CSA-driven instrumented code

Figure 1: Bug report generated by CSA for the motivating example in Figure 1a and the instrumented source code.
with other message types are filtered out when the trace is parsed for relevant information.

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 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 ≥ 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 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).

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.

\(^2\)The number of paths explored by KLEE is sensitive to the LLVM/KLEE/runtime versions and compilation flags.
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.

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

<table>
<thead>
<tr>
<th>Algorithm 1: Targeted search heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data:</strong> activePathIDs: Step (\rightarrow) {PathID}</td>
</tr>
<tr>
<td><strong>Data:</strong> states: Step (\times) PathID (\rightarrow) {State}</td>
</tr>
<tr>
<td><strong>Data:</strong> instructionPathIDs: Instruction (\times) Step (\rightarrow) pathID</td>
</tr>
<tr>
<td><strong>Data:</strong> maxActiveStep: maximum step number among states</td>
</tr>
<tr>
<td><strong>Data:</strong> maxStep: maximum step number in program</td>
</tr>
<tr>
<td><strong>Function</strong> update(currentState, newState, terminatedStates):</td>
</tr>
<tr>
<td>updateCurrent(currentState)</td>
</tr>
<tr>
<td>foreach state: newStates do</td>
</tr>
<tr>
<td>insert(state)</td>
</tr>
<tr>
<td>foreach state: terminatedStates do</td>
</tr>
<tr>
<td>remove(state)</td>
</tr>
<tr>
<td><strong>Function</strong> insert(state):</td>
</tr>
<tr>
<td>state.distance (\leftarrow) computeDistance(state)</td>
</tr>
<tr>
<td>if state.distance = (\infty) then</td>
</tr>
<tr>
<td>while state.distance = (\infty) &amp; state.lastStep &lt; maxStep do</td>
</tr>
<tr>
<td>state.lastStep (\leftarrow) state.lastStep + 1</td>
</tr>
<tr>
<td>state.distance (\leftarrow) computeDistance(state)</td>
</tr>
<tr>
<td>state.pathID (\leftarrow) instructionPathIDs[state.pc][state.lastStep]</td>
</tr>
<tr>
<td>if state.distance = (\infty) then</td>
</tr>
<tr>
<td>terminate(state)</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>states[state.lastStep][state.pathID].add(state)</td>
</tr>
<tr>
<td>activePathIDs[state.lastStep][state.pathID].add(state)</td>
</tr>
<tr>
<td><strong>Function</strong> select() (\rightarrow) State</td>
</tr>
<tr>
<td>nextPathID (\leftarrow) activePathIDs[maxActiveStep].selectRoundRobin()</td>
</tr>
<tr>
<td>candidates (\leftarrow) states[maxActiveStep][nextPathID].selectByDistance()</td>
</tr>
<tr>
<td>return candidates.pickRandomly()</td>
</tr>
</tbody>
</table>
we chose older releases of these applications from around June when the calculation would be solely based on minimum distances. After selecting a candidate set of states with shortest distance to (line 13), before it is inserted into the partitioned state set (line 17) (line 9), adjacent steps are evaluated until a reachable step is found. In Infer v.1.0.0 [10], using the default options of both tools. We also false positives.

reports from these tools to determine whether or not they were and Infer on these applications, and manually found via static analysis but that have since been fixed. Therefore, likely that many of these applications were not just tested with programs. The applications considered are listed in Table 3. It is a specific paper (e.g. the Linux kernel), as well as tiny benchmark usable in the context of an extension of KLEE being presented in at the time of writing. We excluded applications that were only cent KLEE-related papers listed at http://klee.github.io/publications/ in order to evaluate our technique, we sought a set of C/C 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 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.

\(^3\)\(\text{--pulse --no-filtering --no-default-checkers --bufferoverrun --headers --biabduction} -j\)
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.

<table>
<thead>
<tr>
<th>Application</th>
<th>Relevant reports</th>
<th>False positives</th>
<th>True positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>APR</td>
<td>1.5.2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>flex</td>
<td>2.53.99</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>awk</td>
<td>4.1.23</td>
<td>124</td>
<td>70</td>
</tr>
<tr>
<td>bc</td>
<td>1.06</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Binutils</td>
<td>2.25.1</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>combine</td>
<td>0.4.0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Coreutils</td>
<td>8.24</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>datamash</td>
<td>1.0.6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Diffutils</td>
<td>3.3</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Findutils</td>
<td>4.3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>grep</td>
<td>2.21</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>Gzip</td>
<td>1.6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Libtasn1</td>
<td>4.5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>M4</td>
<td>1.4.17</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Make</td>
<td>4.1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>oSIP</td>
<td>4.1.0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>sed</td>
<td>4.2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Trueprint</td>
<td>5.4</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>ImageMagick</td>
<td>6.9.4.8</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>JasPer</td>
<td>1.900.1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>libjpeg</td>
<td>9a</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>LibTIFF</td>
<td>3.9.7</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>libxml2</td>
<td>2.9.2</td>
<td>33</td>
<td>91</td>
</tr>
<tr>
<td>tcpdump</td>
<td>4.7.4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vorbis Tools</td>
<td>1.4.0</td>
<td>1</td>
<td>19</td>
</tr>
</tbody>
</table>

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, 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. k111, chmod), contain unsupported LLVM intrinsics (e.g. sha256sum), cause an assertion in KLEE’s Z3 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. I).

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 ap-
lications 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 reason-
able 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 in-
cluded 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 con-
structs. 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) in-
ject 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

![Figure 3: Percentage of multi-event null-dereference (null) and use-after-free (uaf) bugs detected by CSA and Infer.](image-url)
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 ≤ $S$ ≤ 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-TargetedLast 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 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. Besides the Portfolio configuration, Ignore-Targeted, Try-Targeted and Ignore-TargetedLast 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 4.13x seems to work no better than the Ignore-Targeted strategy which gives a speedup of 4.09x. 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 99.8% of feasible conditions were already implied by the respective path constraints.

Even more surprising for us is the result of the Ignore-TargetedLast 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-TargetedLast 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 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.

---

3To clarify, random seeds initialise KLEE’s internal random number generator and do not serve as concrete input.

4Only 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.
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).

**REFERENCES**


