Tracer-X: Dynamic Symbolic Execution with Interpolation

Joxan Jaffar, Rasool Maghareh, Sangharatna Godboley, Xuan-Linh Ha

National University of Singapore

`{joxan,rasool,sanghara,haxl}@comp.nus.edu.sg`

Abstract

Dynamic Symbolic Execution (DSE) is an important method for testing of programs. An important system on DSE is KLEE [1] which inputs a C/C++ program annotated with symbolic variables, compiles it into LLVM, and then emulates the execution paths of LLVM using a specified backtracking strategy. The major challenge in symbolic execution is **path explosion**. The method of **abstraction learning** has been used to address this. The key step here is the computation of an **interpolant** to represent the learnt abstraction. In this paper, we present a new interpolation algorithm and implement it on top of the KLEE system. The main objective is to address the path explosion problem. We show that despite the overhead of computing interpolants, the **pruning** of the symbolic execution tree that interpolants provide often significant overall benefits. We performed a comprehensive experimental evaluation against KLEE, as well as against two well-known systems that are based on Static Symbolic Execution (SSE), CBMC and LLBMC. Our primary conclusion is that we can perform **complete search** to a new level. A secondary conclusion is that when our search is incomplete, we are competitive or better than the baseline compared systems in terms of code coverage and bug finding.

Proposed Framework

Our primary objective is to address **the path explosion problem in DSE**. We wish to perform path-by-path exploration of DSE to enjoy its benefits, but we include a **pruning mechanism** so that path generation can be eliminated if the path generated so far is guaranteed not to violate the stated safety conditions. The main contribution of our tool is the design and implementation of **a new interpolation algorithm**, and integration into the KLEE system.

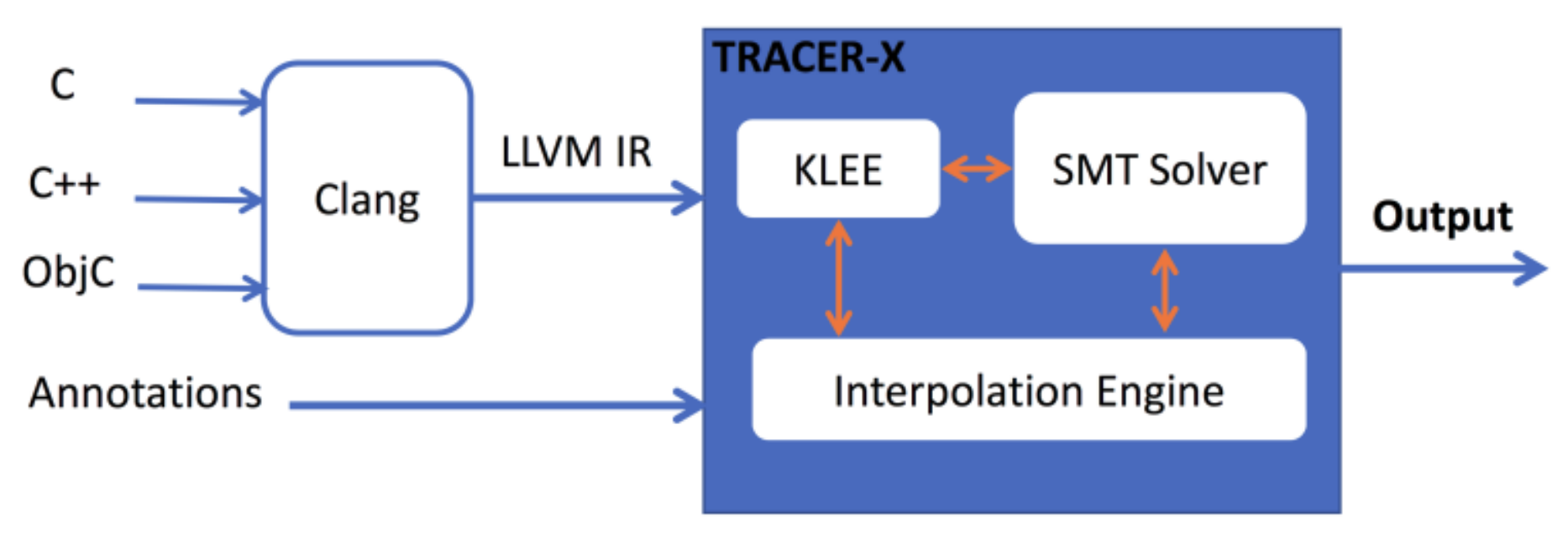

Figure 1: The Overall Architecture of Tracer-X

DSE vs. Pruning

We employ the method of **abstraction learning** [2]. The core feature of this method is the use of **interpolation**, which serves to generalize the context of a node in the symbolic execution tree with an approximation of the weakest precondition of the node. This method has been implemented in **the TRACER system** [3] which was the first system to demonstrate DSE with pruning. While TRACER was able to perform bounded verification and testing on many examples, it could not accommodate industrial programs which often dynamically manipulate the heap memory.

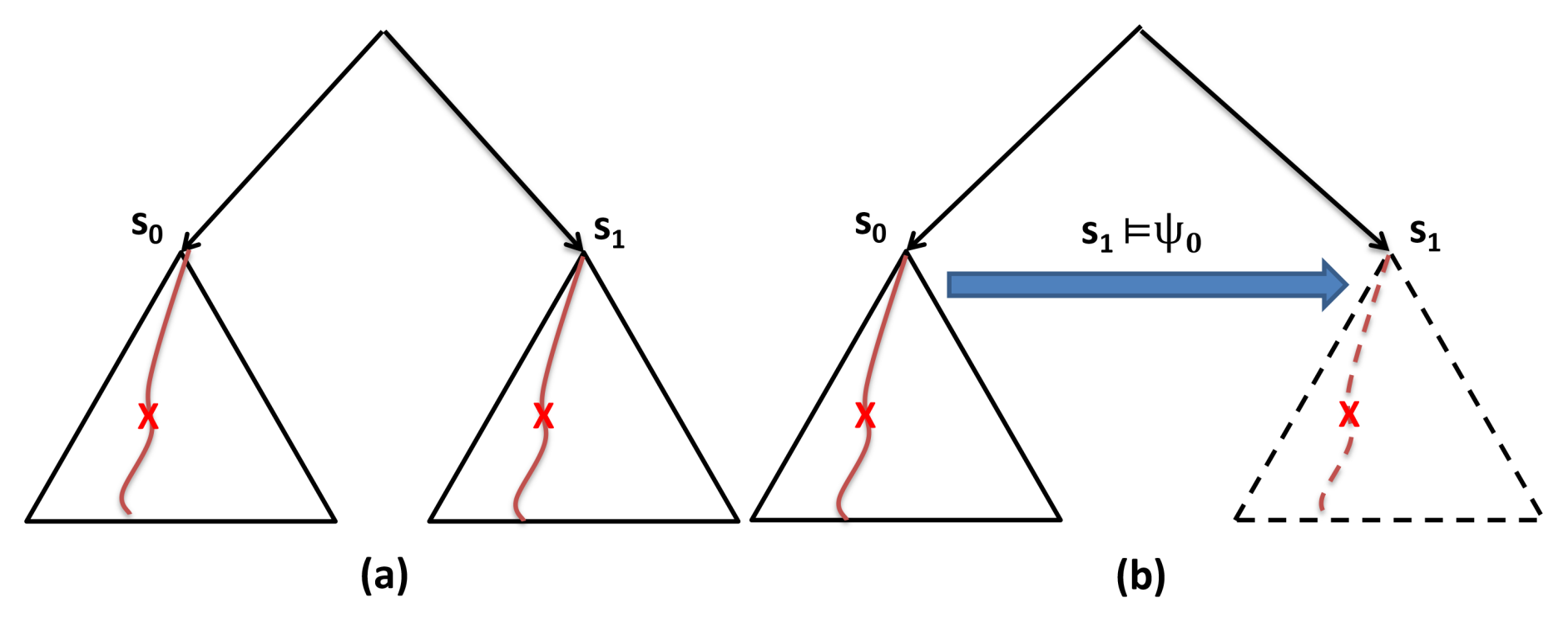

Figure 2: Exploration of Symbolic Execution Tree in Non-pruning DSE vs. Pruning DSE

Weakest Precondition

The weakest precondtion at point $\langle N\rangle$ of the following program, W_N, is $ite(b[N], \phi[x[i]/1], \phi[x[i]/-1])$. Similarly, $W_{N-1}\ldots W_1$ contain variables $b[i]$. Now, proving safety is tantamount to proving $\Psi \models W_1$, an arbitrary boolean formula. Thus in general this **proof is intractable**.

```
bool b[N];
assume(Ψ);
⟨1⟩ if (b[1]) x[1] = 1; else x[1] = -1;
...
⟨N⟩ if (b[N]) x[N] = 1; else x[N] = -1;
assert(φ);
```

A Practical Algorithm for Path-Based Weakest Precondition

The general idea: Compute the weakest precondition of a symbolic state within the symbolic execution tree (SET) by considering only its feasible paths. The full interpolant of a terminal node would be:

- **Infeasible state:** Given a state s that is infeasible, the interpolant Ψ would be $false$.
- **Safely terminated state:** If s terminates normally, the interpolant Ψ is simply $true$.

The full interpolant of a non-terminal node s, having already computed full interpolants from its successor nodes would be:

1. **The child node is an only child**: This is an easy case: Ψ is simply the *weakest precondition* of transition t (between node and its child) w.r.t the child interpolant Ψ'.
2. **The child node is one of two, but one infeasible**: Thus ahead of the parent node is just one feasible branch, say on b. Then Ψ is $\Psi' \wedge b$.
3. **There is another child node which is feasible**: Let the other child node have (full) interpolant Ψ''. Then Ψ is $ite(b, \Psi', \Psi'')$.

Note that the expression in (3) is in fact the weakest precondition, and in general, a *disjunction*. Propagating disjunctions is potentially explosive, in terms of the size (or "entropy") of an interpolant. Here we describe a method to produce a useful *approximation* of (WP). That is, to find the most general *conjunction* Φ which implies (WP). Assume $\Phi \equiv \Phi' \wedge \Phi''$:

- Given: $\Pi \wedge b \models \Psi'$ (Π is path condition)
- **(1)** Find $\overline{\Pi}$ such that $\overline{\Pi} \wedge b \models \Psi'$ (abduction of Ψ' wrt b)
- **(2)** Find $\overline{\Psi}$ such that $\overline{\Pi} \models \overline{\Psi}$ (guard agnostic constraints of Ψ')
- **(3)** Find $\overline{\overline{\Pi}}$ such that $\overline{\overline{\Pi}} \wedge b \models \Psi' - \overline{\Psi}$ (abduction of Π wrt $\Psi' - \overline{\Psi}$)
- Φ' is $\overline{\overline{\Pi}} \wedge \overline{\Psi}$.

We bring back our running example. At any level i in the traversal, we compute just *one* interpolant:

$$-N + i \leq x[1] + x[2] + \cdots + x[i] \leq N - i$$

which subsumes all states at this level which are encountered later. In other words, we have "perfect" subsumption, and our search tree size is linear in N.

Experimental Results

In our primary experiment, we compare against KLEE (Fig. 3 and 4). In a secondary experiment, we consider the related area of Static Symbolic Execution (SSE) in Fig. 5 and 6. SSE is a competitor to DSE because they both address many common analysis problems. Our main experimental result is that our algorithm leads in improved **path coverage**. The programs are from different benchmarks Coreutils-6.11 and SV-Comp (timeout 3600 sec).

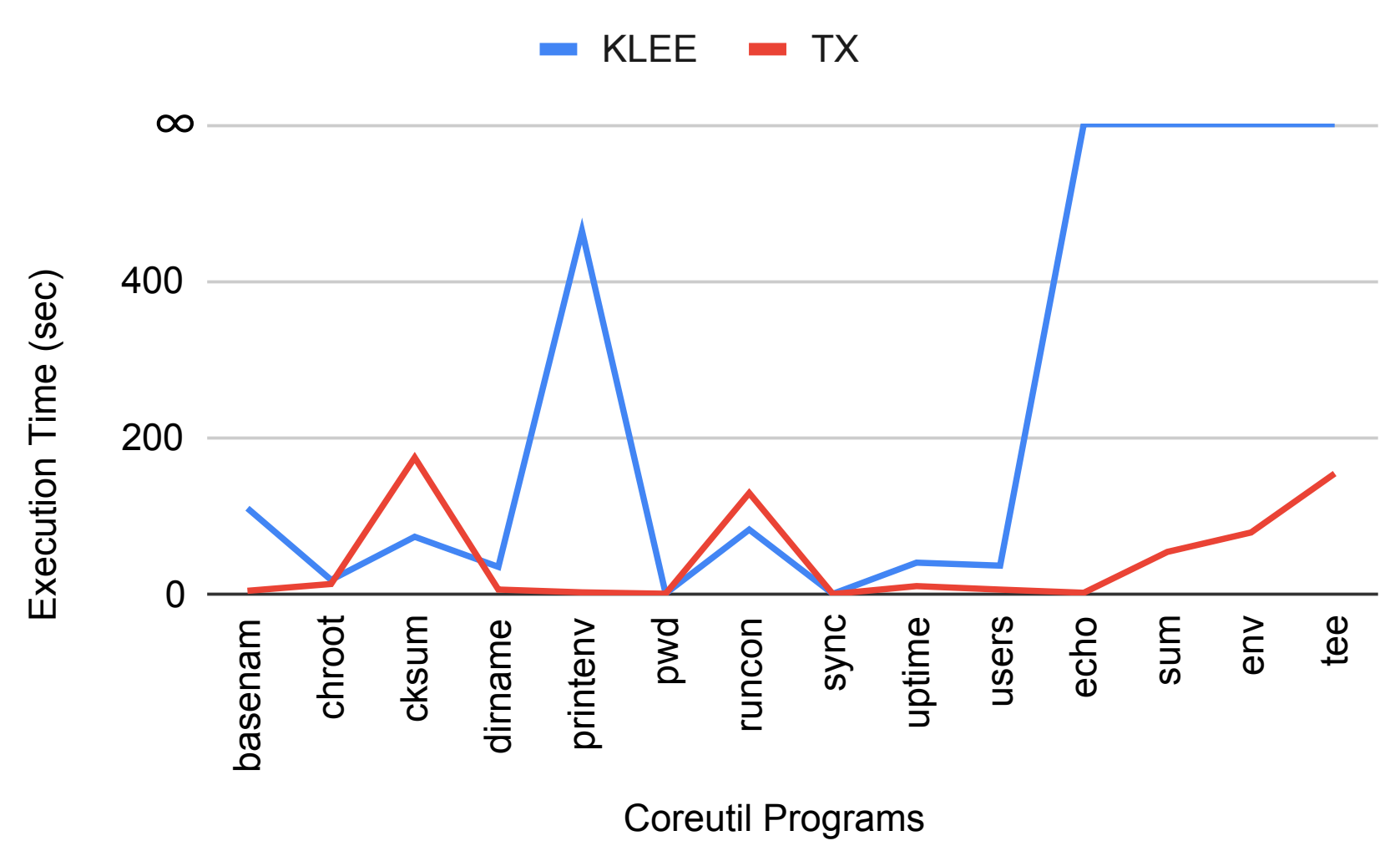

Figure 3: Comparing KLEE and Tracer-X with DFS on Coreutils (Tracer-X finishes)

We show that our algorithm can terminate (given a time/memory limit) on many example programs, while the baseline adversaries cannot, or are significantly slower. A secondary result is that when our algorithm does not terminate, then it is usually the case that our baseline adversaries also do not. Here we then measure **code coverage** and **bug finding**, and we show that our algorithm still has an edge.

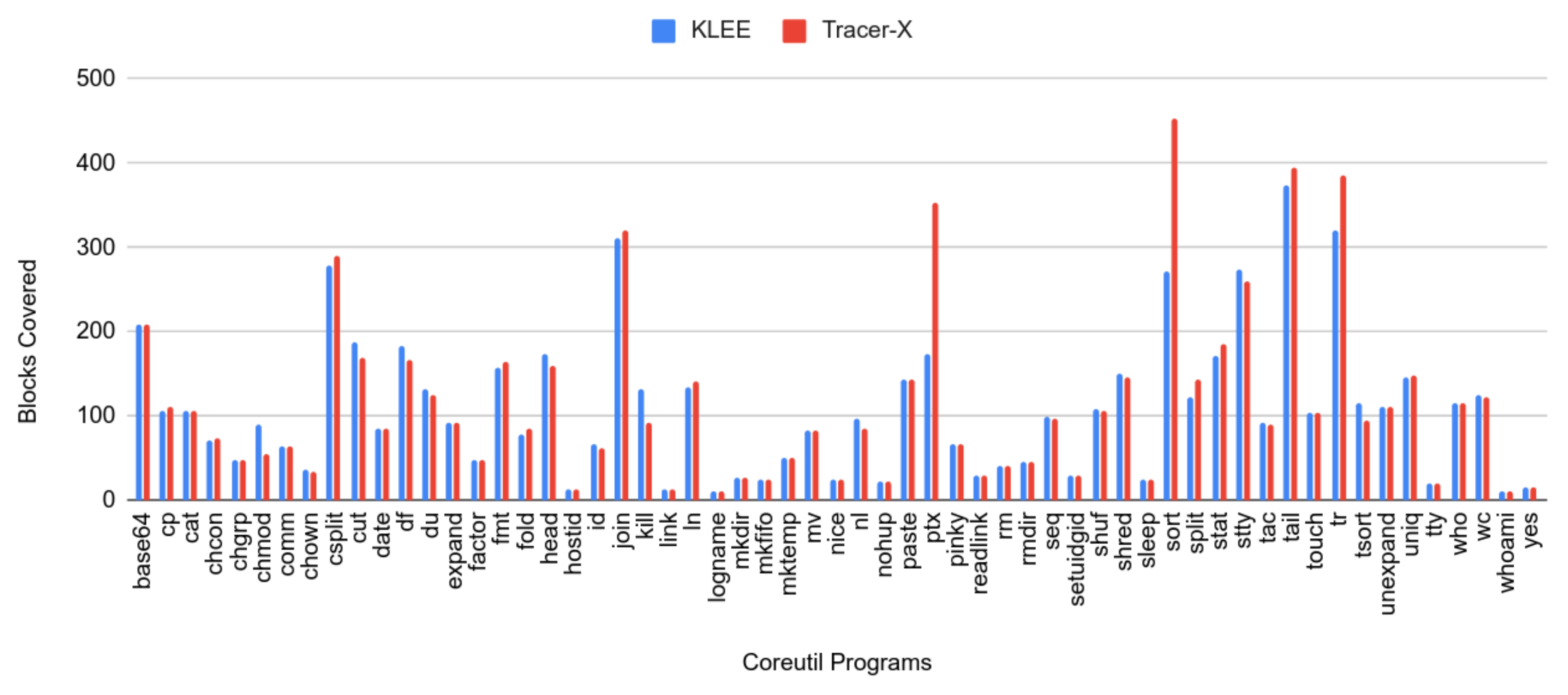

Figure 4: Comparing KLEE and Tracer-X with Random on Coreutils, and both timeout

- **The first group** has 14 smaller size programs where Tracer-X finishes as shown in Fig. 3. Here in this set, execution time is more important than block coverage, since the programs have same coverage. **For 12 programs** Tracer-X is faster as compared to KLEE.
- **The second group** has 60 programs where both the tools don't finish the execution within timeout (Fig. 4). Since both tools timeout, so the block coverage is much more important to observe. Tracer-X covers more or same basic blocks **on 46 programs** (having more block coverage **on 16 programs**) as compared to KLEE.

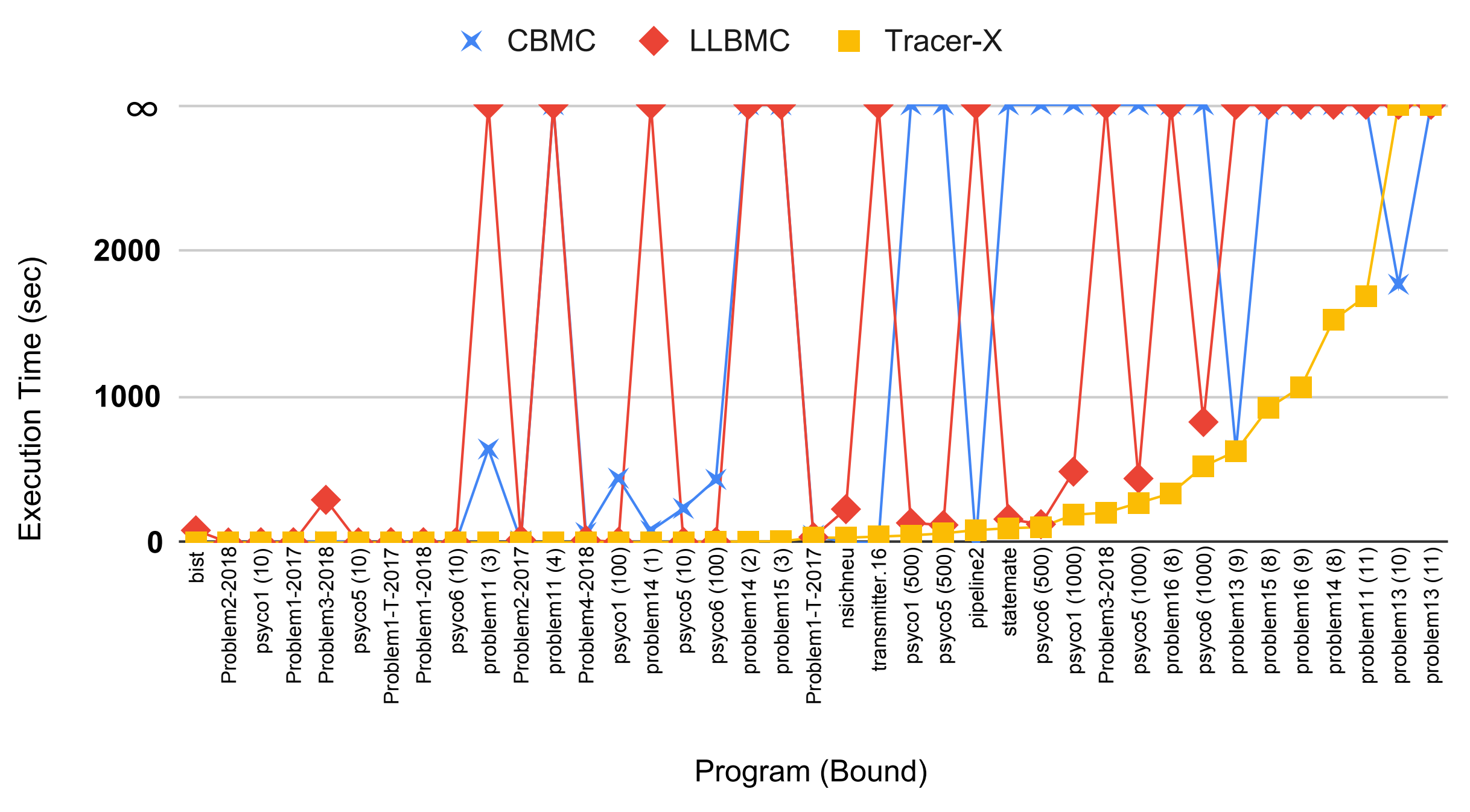

Figure 5: Comparison with LLBMC and CBMC: Analysis time

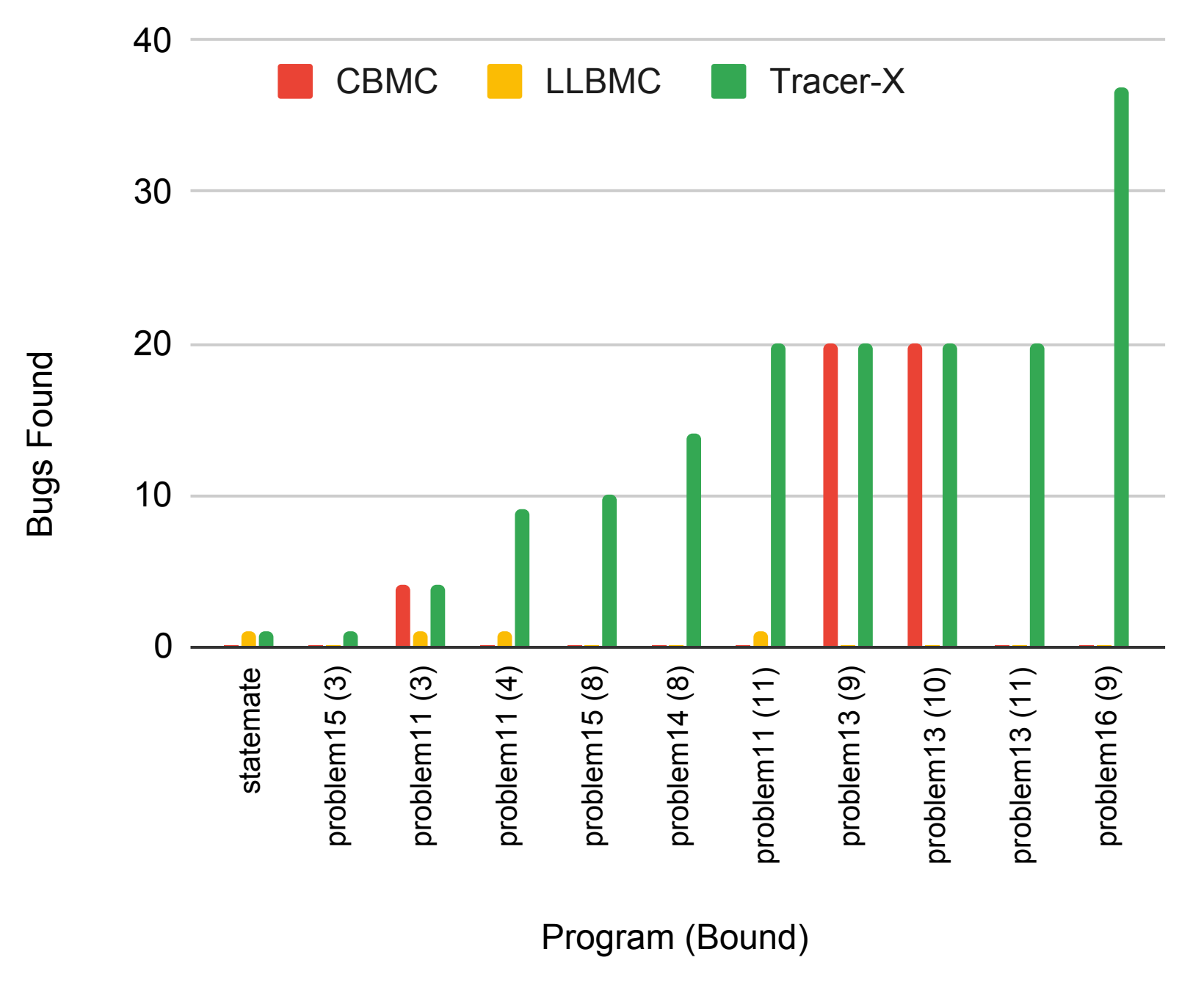

Figure 6: Comparison with LLBMC and CBMC: Number of errors found

- **Complete Search:** Of 20 programs and 39 runs, TRACER-X terminates in 37 while CBMC/LLBMC in 20/23. Where one system terminates, TRACER-X is faster in 24 while slower in 3/1. Aggregating the result for the terminating cases, TRACER-X was about 3.1X faster than CBMC and 2.15X faster than LLBMC.
- **Code Coverage/Bug Finding:** Code coverage is not applicable here (LLBMC does not report this, and CMBC has a different emulation mode). For bug finding, TRACER-X finds 109 more bugs than CBMC/LLBMC.

References

[1] Cristian Cadar et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, 2008.

[2] Joxan Jaffar, Andrew E Santosa, and Răzvan Voicu. An interpolation method for clp traversal. In *CP*, pages 454–469. Springer, 2009.

[3] Joxan Jaffar et al. TRACER: A symbolic execution tool for verification. In *CAV*, 2012.