Symbolic Execution
Challenges and Opportunities

SOFTWARE RELIABILITY GROUP

Imperial College London

Symbolic execution systematically explores paths in a program by using a constraint solver to reason about the feasibility of each path. Symbolic execution has gathered significant attention in the last decade, with applications in a wide variety of areas, including software engineering, systems, and security.

The main open challenges for symbolic execution are: scalability (path explosion, constraint solving) and inadequate feature support (e.g., floating point).

KLEE klee.github.io

- ★ Open-source symbolic execution engine for C and C++ based on the LLVM compiler infrastructure
- ★ Support for different solvers (STP, Z3, MetaSMT)
- ★ Used in both academia & industry
- ★ Over 3500 citations of the paper that introduced KLEE; over 250 papers directly building on or using KLEE: klee.github.io/publications
- ★ Active community with over 80 GitHub contributors, over 2000 stars and over 400 mailing list subscribers
- ★ Download it at klee.github.io, follow @kleesymex, or try it in your browser at klee.doc.ic.ac.uk

Automated Chopped Symbolic Execution

Nowack, Ruiz, Zaki, Cadar. Ongoing work.

Chopped symbolic execution [Trabish et al., ICSE 2018] lets us avoid symbolically executing code portions irrelevant to a given task. Can we *automate* it by identifying beneficial code portions to skip, without the need for an expert input?

```c
int main() {
   char buf[32];
   char *data = read_string();
   unsigned int magic = read_number();

   parse(data); // should be chopped

   if (magic == 0xdeadbeef) {
      // buffer overflow
      memcpy(buf, data, 100);
   }
   return 0;
}
```

```c
void parse(char *data) {
   // symbolic execution will suffer
   // from path explosion
   for (int i = 0; i < 100; i++) {
      switch (data[i]) {
      case ’A’:
         handleA();
      case ’B’:
         handleB();
      ...
      }
   }
}
```

We iterate over the revision history of public Git repositories and demonstrate benefits for patch testing. Our heuristics greedily attempt to skip code that the patch does not obviously depend on, then may backtrack.

Confirming Static Analysis Bug Reports

Busse, Gharat, Cadar, Donaldson. Combining Static Analysis Error Traces with Dynamic Symbolic Execution (Experience Paper). ISSTA 2022.

Static analysis engines are fast but they over-approximate and create many false positives. Symbolic execution on the other hand is precise but often gets lost in the sheer number of program paths. In this project we tried to combine both approaches and confirm static analysis error traces from two off-the-shelf static analysis engines (CSA and Infer) with KLEE.

To follow the paths described by an error trace we use two techniques: 1) *program instrumentation* to add constraints that have to hold according to the trace, and 2) a new *targeted search heuristic*.

However, our experiments show that most of these constraints are already implied by the path to the target and that even the intermediate steps do not aid targeting. That means, the targeted search heuristic performs equally well when given just the bug location instead of the full trace with conditions and intermediate steps.

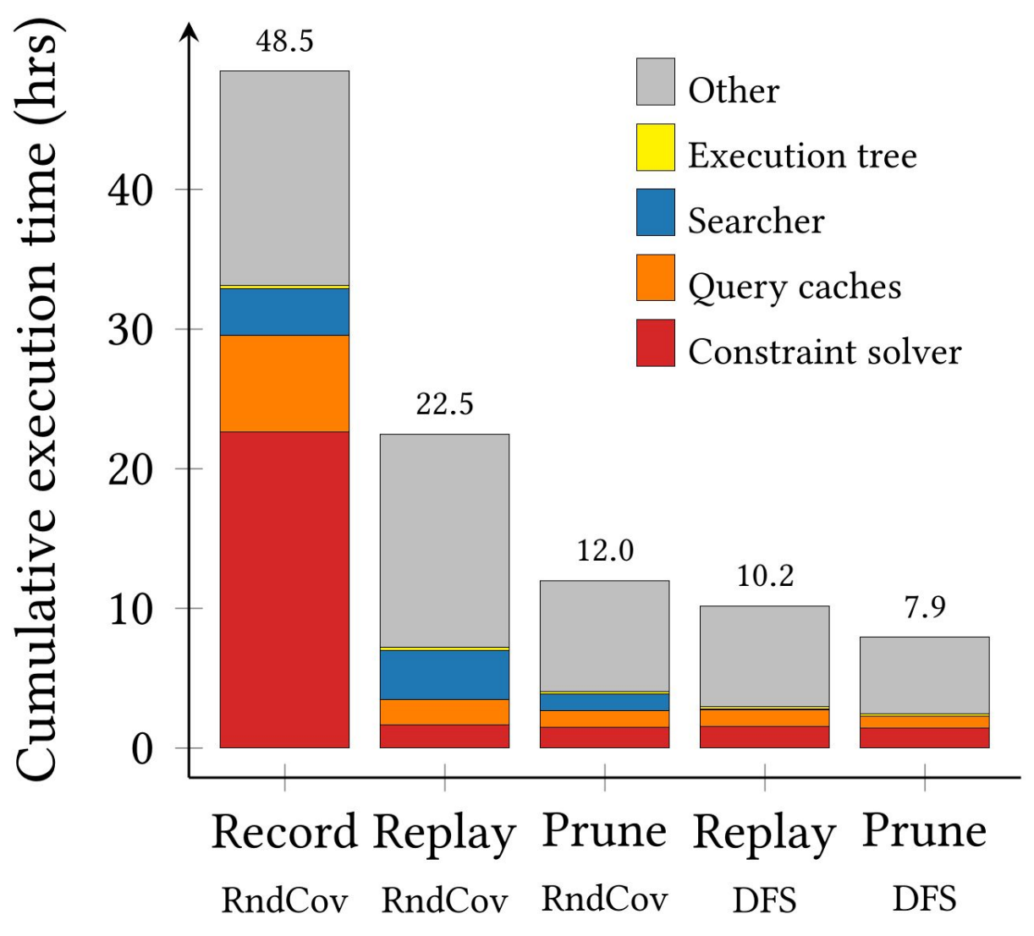
Memoised Symbolic Execution

Busse, Nowack, Cadar. Running Symbolic Execution Forever. ISSTA 2020.

Symbolic execution runtime can be divided into execution time and solving time. KLEE currently has no means to reuse solver results across different runs of the same program or to resume interrupted executions.

Our extension, named *MoKlee,* efficiently stores solver decisions and metadata on disk as execution tree nodes and uses this information in subsequent runs to:

1. Reuse solver results from previous runs
2. Prune already explored paths from subsequent runs
3. Detect diverging paths during re-execution



This allows users of MoKlee to restart testing campaigns immediately without wasting time on re-exploring already tested paths.

Approximating Floating Point via Fixed Point

Hughes, Nowack, Schemmel, Cadar. Ongoing work.

Floating-point numbers are notoriously complex, with subnormal numbers, infinities and NaNs, dual zeroes and a generally complex format. This in turn makes SMT solvers for the theory of floating-point numbers slow and a bit-precise analysis of programs using floating-point numbers hard.

In this project we explore the impact of approximating floating-point numbers with fixed-point numbers, about which SMT solvers can reason with significantly higher efficiency.

By only performing approximations when querying the SMT solver, analysis of concrete floating-point numbers remains bit-precise. Additionally, lowering the SMT theory of floating-point numbers to that of bitvectors using a fixed-point approximation, enables the use of more SMT solvers to drive the symbolic execution of programs using floating-point numbers.

Deterministic Memory Allocation

Schemmel, Büning, Busse, Nowack, Cadar. A Deterministic Memory Allocator for Dynamic Symbolic Execution. ECOOP 2022.

Dispatching memory allocation to the system allocator, as in KLEE, leads to nondeterministic behaviour. With KDAlloc, we built a memory allocator that is cross-run and cross-path deterministic, maximises the probability of finding memory-safety bugs, keeps a low memory and performance overhead, and allows the interaction with the outside environment.

KDAlloc uses `mmap` to allocate a large memory region from which it serves addresses. Allocation metadata is separated from this memory region and attached to the symbolic state instead, directly supporting forking. As the object data is also attached to each symbolic execution state, the memory region is only used as a source of addresses and for external function calls.

In our experiments running KLEE on a variety of real-world applications, KDAlloc showed more deterministic behaviour than KLEE’s default allocator, without sacrificing performance or increasing memory usage.

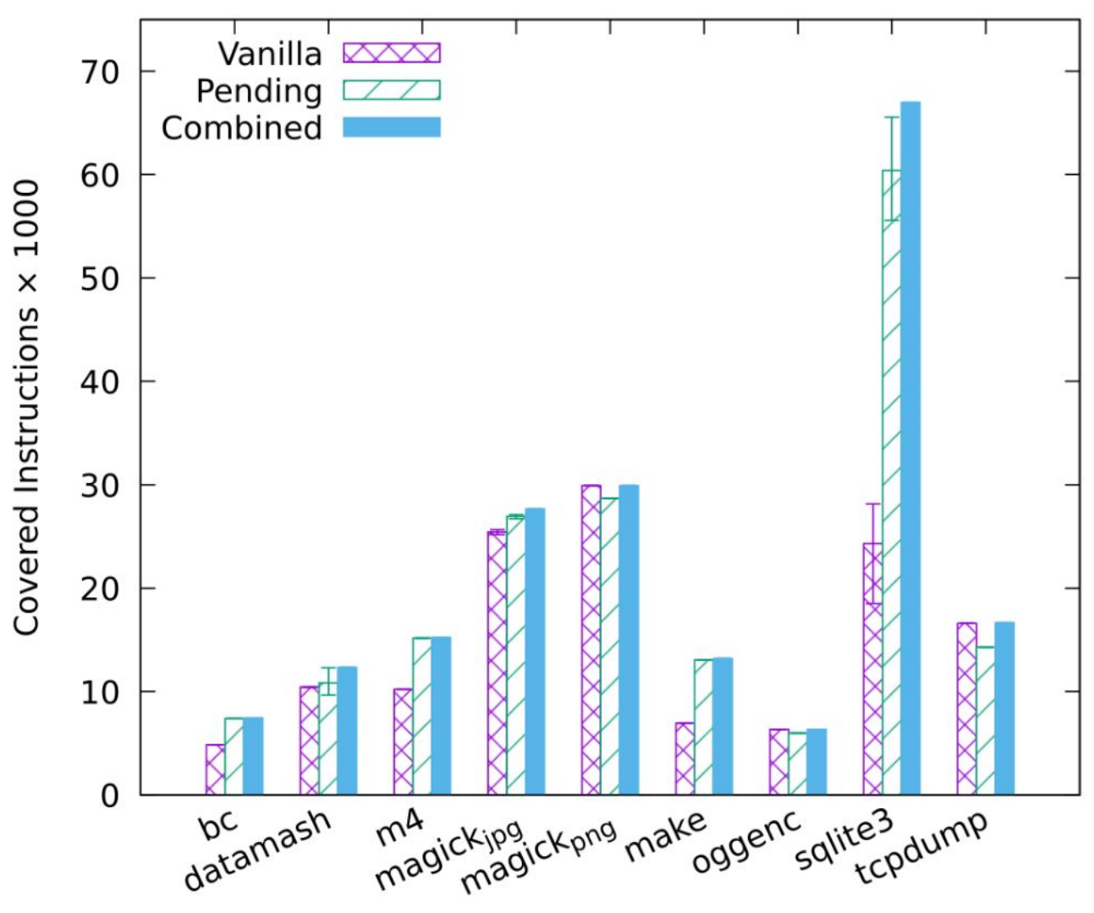
Pending Constraints

Kapus, Busse, Cadar. Pending Constraints in Symbolic Execution for Better Exploration and Seeding. ASE 2020.

This work improves symbolic execution using two observations:

1. Most states are terminated early due to memory pressure and the cost for their feasibility checks is wasted
2. Search heuristics are not cache- and seed-aware

Pending constraints are constraints that cannot immediately be solved by KLEE’s solver caches or using existing seeds. Our approach enhances symbolic execution scalability by aggressively deferring such paths. This reduces the amount of solver calls significantly, leading to more efficient exploration. Our evaluation on eight popular applications shows this approach can achieve higher coverage for both seeded and non-seeded exploration.



Multi-Version Testing with Product Programs

Sharma, Schemmel, Cadar. Ongoing work.

Comparing multiple versions of the same program is useful in various contexts, such as validating refactorings or ensuring that a patch has the expected effect. By automatically constructing a product program that executes multiple versions of one program at once, existing program testing approaches, such as symbolic execution or fuzzing can be used for multi-version testing.

```c
// foo version 1
int foo(int x) {
   int res = 2 * x;
   return res;
}

// foo version 2
int foo(int x) {
   int res = x << 1;
   return res;
}
```

```c
// product program for both versions of foo
void foo(int pp1, int pp2, int* pp_out, int pp1_x, int pp2_x) {
   // runs both versions at once
   int pp1_res = 2 * x;
   int pp2_res = x << 1;
   // (global) assertions can compare values from both versions
   assert(pp1_res == pp2_res);

   pp_out[0] = pp1_res;
   pp_out[1] = pp2_res;
}
```

SRG GROUP