Symbolic Execution
Challenges and Opportunities

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).

Memoised Symbolic Execution

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. 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.

Automated Chopped Symbolic Execution
Nowack, Ruiz, Zak, 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, thus allowing for an expert input?

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

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.

Deterministic Memory Allocation

Dispatching memory allocation to the system allocator, as in KLEE, leads to nondeterministic behaviour. With KDAloc, we build 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.

KDAloc 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 fork. 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, KDAloc showed more deterministic behaviour than KLEE’s default allocator, without sacrificing performance or increasing memory usage.

Pending Constraints

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.

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

```
// product program for both versions of foo
int product(int x) {
    int res1 = 2 * x;
    int res2 = 2 * (res1);
    return res1 + res2;
}
```