Pending Constraints in Symbolic Execution for Better Exploration and Seeding

Timotej Kapus • Frank Busse • Cristian Cadar

3rd International KLEE Workshop on Symbolic Execution
15–16 September 2022, London
Pending Constraints in Symbolic Execution for Better Exploration and Seeding

Timotej Kapus
Imperial College London
United Kingdom
t.kapus@imperial.ac.uk

Frank Busse
Imperial College London
United Kingdom
c.busse@imperial.ac.uk

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT
Symbolic execution is a well established technique for software testing and analysis. However, scalability continues to be a challenge, both in terms of constraint solving cost and path explosion. In this work, we present a novel approach for symbolic execution, which can enhance its scalability by aggressively prioritising execution paths that are already known to be feasible, and deferring all other paths. We evaluate our technique on nine applications, including Gluaco, make and tupmanp and show it can achieve higher coverage for both seedled and non-seedled exploration.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.

KEYWORDS
Symbolic execution, KLEE

ACM Reference Format:

1 INTRODUCTION
Symbolic execution is a dynamic program analysis technique that has established itself as an effective approach for many software engineering problems such as test case generation [4, 12], bug finding [6, 13], equivalence checking [10, 11], vulnerability analysis [8, 27] and debugging [14, 29].

Even with well-engineered tools like KLEE [4], symbolic execution still faces important scalability challenges. These fall into two broad categories: constraint solving and path explosion. As symbolic execution proceeds, the complexity of constraints and the number of paths typically increase, often making it difficult to make meaningful progress.

In this work, we propose a novel mechanism that aggressively explores paths whose feasibility is known via caching or seeding.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE ’20, September 21–25, 2020, Virtual Event, Australia.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6396-4/20/09...$15.00
https://doi.org/10.1145/3388814.344589

Our approach tackles both scalability challenges of symbolic execution. On the one hand, it enables more efficient use of solved constraints, thus reducing the burden on the solver. On the other hand, it provides a meta-search heuristic that gives a way to guide the exploration towards interesting parts of the program.

Before presenting our idea, we briefly summarise symbolic execution. We focus here on the EUT-style of dynamic symbolic execution [5], embodied in tools such as KLEE [4], which unlike concolic execution tools [12, 24], store partially explored paths in memory. Symbolic execution works by running the program on some symbolic inputs, which means they can initially take any value, as they are unconstrained. During execution, if a branch condition depends on a symbolic value, symbolic execution queries an SMT solver for the feasibility of each of the two branches (under the current path condition which is initially empty). If both the then and else branches are feasible, it forks the execution exploring both paths and adding the respective branch conditions to each path condition (PC). After every fork, symbolic execution uses a search heuristic to decide what path to explore next. Each path explored in symbolic execution is encoded by a state which keeps all the information necessary to resume execution of the associated path (PC, program counter, stack contents, etc.).

The core of our idea revolves around inverting the fetching process. Instead of doing an expensive feasibility check first and then fetching the execution, we fetch the execution first. The branch condition is then added as a pending constraint, which means its feasibility has not been checked yet. We refer to states (or paths) with pending path constraints as pending states.

The responsibility for feasibility checking of pending path constraints is passed to the search heuristic. This gives the search heuristic the capability to decide when and for which states it wants to pay the price of constraint solving. For example, it could schedule pending states immediately, thus restoring the original algorithm, or could take into account the (estimated) cost of constraint solving in its decisions.

In our approach, we take advantage of an important characteristic of symbolic execution runs: the feasibility of some paths/states can be quickly determined without using a constraint solver. There are two common cases. First, modern symbolic execution systems like KLEE make intensive use of caching and many queries can be solved without involving the constraints solver [1, 4, 26]. Second, symbolic execution is often bootstrapped with a set of seeds from which to start exploration: these can come from regression test suites [38, 19] or greybox fuzzers in hybrid greybox/whitebox fuzzing systems [9, 21, 25]. By aggressively following paths for which feasibility can be quickly determined without using a constraint solver, our approach can minimise the constraint solving.
Symbolic Execution

state represents program path
Symbolic Execution

... and hits symbolic branch
Symbolic Execution

SMT solver checks feasibility of both branches
Symbolic Execution

if both branches are feasible
the state is forked to explore
both branches
Symbolic Execution

“Searcher” selects next state for exploration.
KLEE’s “EGT-style” Execution

KLEE keeps all unfinished paths/states in memory.
KLEE’s “EGT-style” Execution

... and randomly selects states for early termination when it runs out of memory.
ABSTRACT
When symbolic execution is used to analyze real-world applications, it often consumes all available memory in a relatively short amount of time, sometimes making it impossible to analyze an application for an extended period. In this paper, we present a technique that can record an ongoing symbolic execution analysis to disk and selectively restore paths of interest later, making it possible to run symbolic execution indefinitely.

To be successful, our approach addresses several essential research challenges related to detecting, leveraging, and restoring on-the-fly execution, instantly long-running executions efficiently, changing search heuristics during re-execution, and providing a global view of the final execution. Our extensive evaluation of 91 Linux applications shows that our approach is practical, enabling these applications to run for days while continuing to explore new execution paths.

CCS CONCEPTS
- Software and its engineering → Software testing and debugging

KEYWORDS
symbolic execution, memization, KLEE

1 INTRODUCTION
For testing real-world software systems, symbolic execution is often proposed as a method for thoroughly enumerating and testing every potential path through an application. While achieving full enumeration is usually impossible due to the fundamental challenge of the state-space explosion problem, even a subset of paths can be used to find bugs or generate a high-quality test suite [5, 7, 11, 14]. And typically, the more paths you explore, the better the outcome.

With the multitude of paths, performing symbolic execution on a modern machine quickly consumes all available memory. For instance, in Figure 1, we use the symbolic execution engine KLEE [1] to run 87 real-world applications from the GNU Coreutils suite with a timeout of 2h, using the default memory limit of 1GB. For more than two-thirds of the runs (63 out of 87), KLEE prematurely terminates a substantial amount of paths as the given memory limit is reached. Each of those paths could have spawned a large number of new paths if exploration was allowed to continue. Even if the memory limit is increased to 10GB, more than half of the benchmarks prematurely terminate at least 80% of the paths they started to explore. And worse, for some applications, the premature killing of paths causes KLEE to run out of paths entirely after a relatively short time. For example, with a limit of 1GB, there are 14 applications where KLEE completely runs out of paths before the 2h timeout. Therefore, for these benchmarks and configurations, no matter how much time one has at their disposal, KLEE won’t be able to explore more than a certain number of paths.

One solution for dealing with this problem is to store the paths being terminated early to disk and then replay them later incrementally. Previous work has proposed memized symbolic execution [20], where executed paths are recorded to disk as a trie, and then paths of interest are brought back to memory on replay, ensuring the recorded constraint solving results to speed up the re-execution. The approach was shown to be applicable to direct caches, re-generation and coverage improvement. But it was applied to rather small Java applications (~500 LOC) and short runs (on the order of minutes), and has important limitations that the same search heuristic needs to be used during re-execution.

In this paper, we build upon this idea to design a technique capable of running symbolic execution on large programs indefinitely, while continuing to explore new paths through the program using any search heuristic. We show that to scale up...
KLEE's “EGT-style” Execution

wasted solving time
Symbolic Execution with Pending Constraints

States are always forked!
Symbolic Execution with Pending Constraints

Feasibility checked with fast but incomplete “solver”.
Symbolic Execution with Pending Constraints

- path is feasible: proceed as in normal symbolic execution
- path is ??: do not alter path condition but add constraint as “pending constraint”

Feasibility checked with fast but incomplete “solver”.
State Selection

- global state set split into **feasible** and **pending** states
- searchers select from feasible states
- if none left, pending states are **revived**
  - pending constraint is finally checked
  - infeasible states are removed and feasible states are selected
Fast incomplete “solver”

Explore paths that are known to be feasible!
Fast incomplete “solver”

Explore paths that are known to be feasible!

- paths where symbolic variables have concrete assignments that satisfy the path condition
  - *seeds* (test cases, production data, fuzzing, …)
  - *cached assignments*
KLEE's solver chain

INDEPENDENT SOLVER → QUERY CACHE → COUNTEREXAMPLE CACHE → SMT SOLVER

expensive
Fast incomplete “solver”
Seeding

Efficient seeding seeds are placed in cache as assignments

Stop here!
Example

Solve constraints only when necessary to make progress

Explore paths that are known to be feasible

```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

get_sign(x);

Known assignments

∅
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}

Known assignments

∅

No "feasible states" left: pick one!
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments

\(x = -2\)

\(x = 0\)

\(x \neq 0\)

\(x \geq 1\)
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

**Known assignments**

- \( x = -2 \)
- \( x = 0 \)
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

**Known assignments**

- $x = -2$
- $x = 0$

Diagram:

- $x = -2$ results in $r = 0$
- $x = 0$ results in $r = 0$
- $x = 0$ results in $r = 0$
- $x < 1$ follows the flowchart
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}

Known assignments

- x = -2
- x = 7
- x = 0

get_sign(x);

- x ≥ 1
  - r = -1;
  - x ≥ 1
  - r = 1;

- x = 0
  - r = 0;
  - x = 0
  - return r;

- x ≠ 0
  - return r;
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

**Known assignments**

- \( x = -2 \)
- \( x = 7 \)
- \( x = 0 \)

Diagram:

```
get_sign(x);
```

```
r = -1;
```

```
x >= 1
```

```
x < 1
```

```
x == 0
```

```
x != 0
```

```
x = -2
```

```
x = 7
```

```
x = 0
```

```
return r;
```

```
r = 0;
```

```
x != 0
```

```
x = 0
```

```
return r;
```
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

Known assignments:
- $x = -2$
- $x = 7$
- $x = 0$

Diagram:
- $x = -2$: $r = 0$
- $x = 7$: $r = 1$
- $x = 0$: $r = \text{unknown}$
```c
int get_sign(int x) {
    int r = -1;
    if (x >= 1) r = 1;
    if (x == 0) r = 0;
    return r;
}
```

**Known assignments**

- $x = -2$
- $x = 7$
- $x = 0$
Example - Seeding

Reversing md5 hash is hard for SMT solvers

Use

\[ 1471037522 = \text{md5("ase2020")}[0] \]

as seed.
Suppose this exploration tree for md5

md5("ase2020")
Solver queries: 0

Pending

Vanilla
Solver queries: 0

Pending

Vanilla
Solver queries: 0

Pending

Vanilla
Solver queries: 1

Pending

Vanilla
Solver queries: 2

Pending

Vanilla
Solver queries: 3

Pending

Vanilla
Solver queries: 4

Pending

Vanilla
Solver queries: 5

Pending

Vanilla
Solver queries: 6

Pending

Vanilla
Solver queries: 7

Pending

Vanilla
Solver queries: 8

Pending

Vanilla
Why pending constraints?

More efficient use of solver solutions
- explore **more instructions per query**
- spend **less time solving infeasible queries**

Allows deeper search tree exploration

Empowering search heuristics
- control over constraint solving
- **ZESTI**
Why pending constraints?

More efficient use of solver solutions
- explore more instructions per query
- spend less time solving infeasible queries

Allows deeper search tree exploration

Empowering search heuristics
- control over constraint solving
Evaluation

8 real world applications
Hard targets for symbolic execution
2hr runs, 3 searchers, 3 repetitions
24h SQLite study
Instruction coverage - non-seeded

35%, 34% resp. 24% more instructions across benchmarks
Instruction coverage - seeded

25%, 30% resp. 23% more instructions across benchmarks
SQLite3: 24 hour run - non-seeded (random path)
SQLite3: 24 hour run - seeded (random-path)
Pending constraints

- aggressively follow feasible paths and **explore more instructions per query**
- reduce the constraint solving time
- could **improve the coverage** for 8 hard programs

---

Zesti-Reimplementation

Explores sensitive instructions around seed.

Found **2 new bugs**.