Triereme: Speeding up hybrid fuzzing through efficient query scheduling

Elia Geretto[1] Julius Hohnerlein[1] Cristiano Giuffrida
Herbert Bos Erik van der Kouwe Klaus v. Gleissenthall

Vrije Universiteit Amsterdam

Annual Computer Security Applications Conference, 2023

[1]Both authors contributed equally to the paper.

Teaser

- Triereme is a speed optimization for hybrid fuzzers
- Key insight: it combines two seemingly incompatible optimizations to reduce time spent in SMT solver
- It achieves:
  - up to 15.2× speed increase on a complete concolic execution
  - statistically significant coverage progression improvements in 79% of tested benchmarks

Motivation

- Fuzzing is a trial and error process
- Requires delicate balance between speed and cleverness
- Hybrid fuzzing combines a fuzzer (fast) and a concolic engine (clever)

- Problem: hybrid fuzzers are (often) worse than state-of-the-art fuzzers because concolic engines are:
  - Slow: produce useful results late
  - CPU intensive: interfere with fuzzers
- Goal: speed up concolic engines to improve their usefulness

VRIJE UNIVERSITEIT AMSTERDAM

Background - Concolic engine

- Produces new test cases through queries to an SMT solver
- Query schedule defined by executing an existing test case
- New test cases explore opposite branches

```
bool is_sorted(int *a,
               size_t n) {
  size_t i;
  for (i = 0; i + 1 < n; i++) {
    if (a[i] > a[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Assuming `a = {1, 2, 3}`:

Background - Concolic engine

- Produces new test cases through queries to an SMT solver
- Query schedule defined by executing an existing test case
- New test cases explore opposite branches

```
bool is_sorted(int *a,
               size_t n) {
  size_t i;
  for (i = 0; i + 1 < n; i++) {
    if (a[i] > a[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Assuming `a = {1, 2, 3}`:

```
1 >= n
```

VRIJE UNIVERSITEIT AMSTERDAM

Background - Concolic engine

- Produces new test cases through queries to an SMT solver
- Query schedule defined by executing an existing test case
- New test cases explore opposite branches

```
bool is_sorted(int *a,
               size_t n) {
  size_t i;
  for (i = 0; i + 1 < n; i++) {
    if (a[i] > a[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Assuming a = {1, 2, 3}:

```
1 >= n
1 < n && a[0] > a[1]
```

VRIJE UNIVERSITEIT AMSTERDAM

Background - Concolic engine

- Produces new test cases through queries to an SMT solver
- Query schedule defined by executing an existing test case
- New test cases explore opposite branches

```
bool is_sorted(int *a,
               size_t n) {
  size_t i;
  for (i = 0; i + 1 < n; i++) {
    if (a[i] > a[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Assuming a = {1, 2, 3}:

```
1 >= n
1 < n && a[0] > a[1]
1 < n && a[0] <= a[1]
  && 2 >= n
```

Background - Concolic engine

- Produces new test cases through queries to an SMT solver
- Query schedule defined by executing an existing test case
- New test cases explore opposite branches

```
bool is_sorted(int *a,
               size_t n) {
  size_t i;
  for (i = 0; i + 1 < n; i++) {
    if (a[i] > a[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Assuming `a = {1, 2, 3}`:

```
1 >= n
1 < n && a[0] > a[1]
1 < n && a[0] <= a[1]
  && 2 >= n
1 < n && a[0] <= a[1]
  && 2 < n && a[1] > a[2]
```

Background - Concolic engine

- Produces new test cases through queries to an SMT solver
- Query schedule defined by executing an existing test case
- New test cases explore opposite branches

```
bool is_sorted(int *a,
               size_t n) {
  size_t i;
  for (i = 0; i + 1 < n; i++) {
    if (a[i] > a[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Assuming `a = {1, 2, 3}`:

```
1 >= n
1 < n && a[0] > a[1]
1 < n && a[0] <= a[1]
  && 2 >= n
1 < n && a[0] <= a[1]
  && 2 < n && a[1] > a[2]
1 < n && a[0] <= a[1]
  && 2 < n && a[1] <= a[2]
  && 3 < n
```

Background - Incremental solving

- SMT solvers have an assertion stack that allows to push and pop states
- Easily exploitable with unmodified query schedules (common prefix)

Query schedule:

```
1 >= n
1 < n && a[0] > a[1]
1 < n && a[0] <= a[1]
  && 2 >= n
...
```

Interpretation:

```
!A
A && !B
A && B && !C
...
```

VU VRIJE UNIVERSITEIT AMSTERDAM

Background - Unrelated constraint elimination

- Keeps only branch constraints that transitively share variables with final branch constraint
- Makes queries shorter (faster to solve), but breaks incremental pattern

Before:

```
1 < n && a[0] <= a[1]
  && 2 < n && a[1] <= a[2]
  && 3 < n
```

After:

```
1 < n && 2 < n && 3 < n
```

Triereme

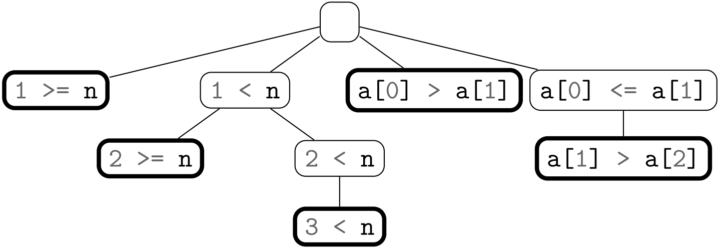
- During execution, collect only branch constraints, solve later
- Construct path constraints using unrelated constraints elimination
- Organize path constraints in a persistent prefix tree (trie)
- Use incremental solving while exploring the trie with a depth first search

Trieme - Example

Path constraints:

```
1 >= n
a[0] > a[1]
1 < n && 2 >= n
a[0] <= a[1] && a[1] > a[2]
1 < n && 2 < n && 3 < n
```

Trie:

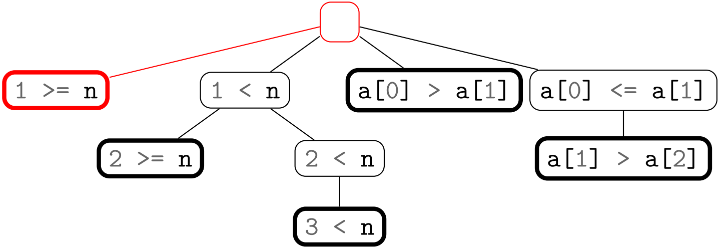


Triereme - Example

Path constraints:

```
1 >= n
a[0] > a[1]
1 < n && 2 >= n
a[0] <= a[1] && a[1] > a[2]
1 < n && 2 < n && 3 < n
```

Trie:

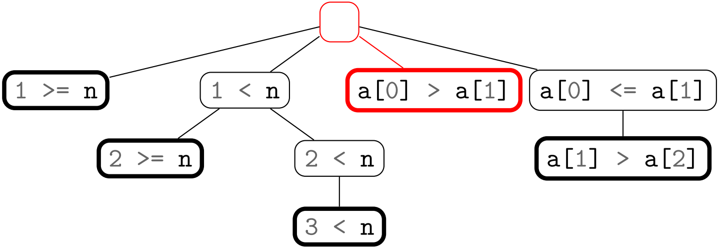


Trireme - Example

Path constraints:

```
1 >= n
a[0] > a[1]
1 < n && 2 >= n
a[0] <= a[1] && a[1] > a[2]
1 < n && 2 < n && 3 < n
```

Trie:

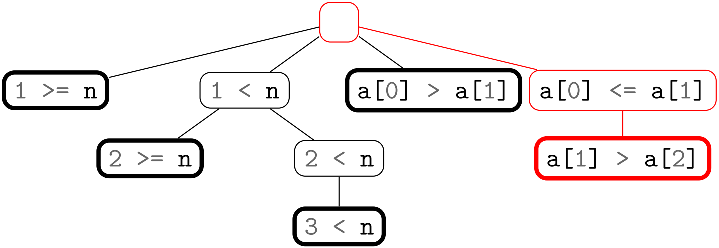


Triereme - Example

Path constraints:

```
1 >= n
a[0] > a[1]
1 < n && 2 >= n
a[0] <= a[1] && a[1] > a[2]
1 < n && 2 < n && 3 < n
```

Trie:

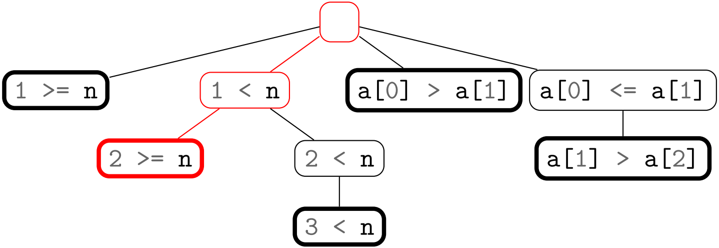


Triereme - Example

Path constraints:

```
1 >= n
a[0] > a[1]
1 < n && 2 >= n
a[0] <= a[1] && a[1] > a[2]
1 < n && 2 < n && 3 < n
```

Trie:

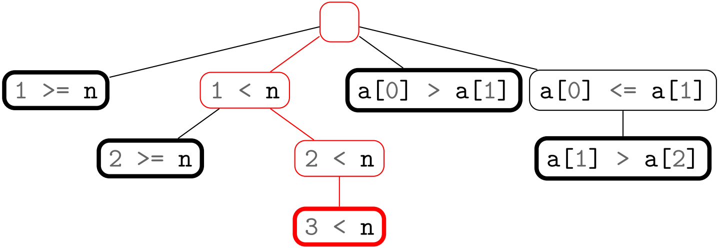


Trireme - Example

Path constraints:

```
1 >= n
a[0] > a[1]
1 < n && 2 >= n
a[0] <= a[1] && a[1] > a[2]
1 < n && 2 < n && 3 < n
```

Trie:

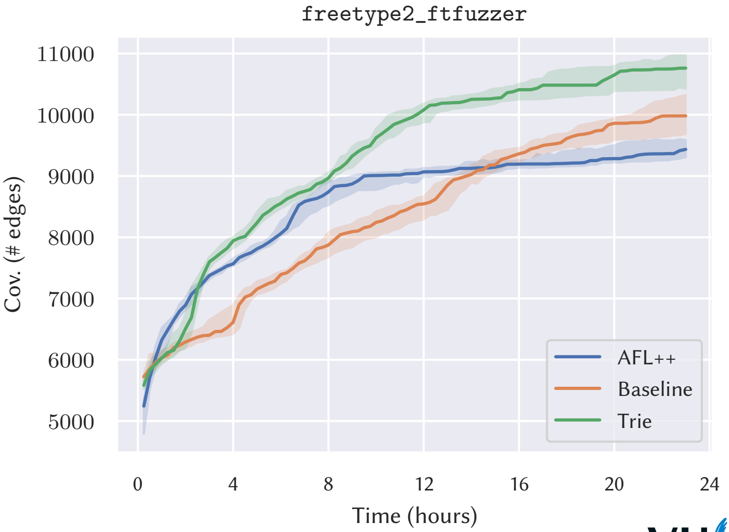


Other optimizations

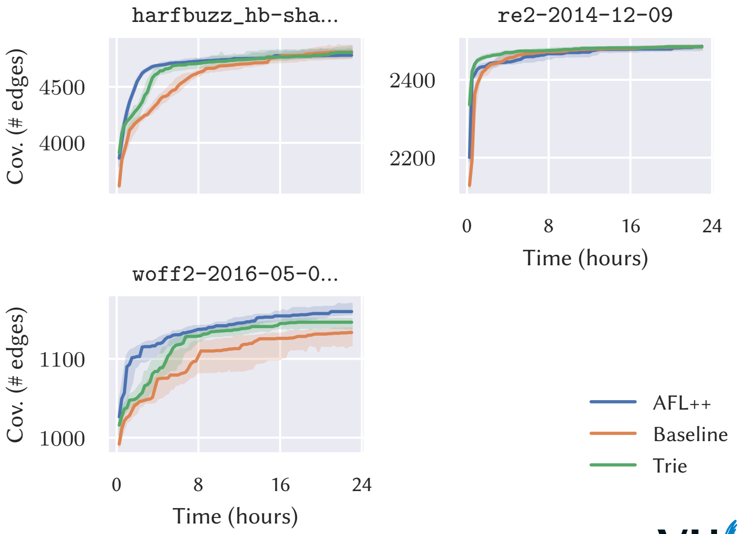
- Optimistic solving (from related work)
  - If a path constraint cannot be solved, repeat the query only with the last (negated) branch constraint.
- Optimistic pruning
  - Optimistic solving on pruned constraints
- Unsatisfiability derivation
  - Unsatisfiable intermediate nodes allow to skip their subtrie
- Other low-level optimizations

VU VRIJE UNIVERSITEIT AMSTERDAM

Evaluation - Example



Evaluation - Negative selection



Conclusion

Triereme improves concolic engine performance by combining incremental solving and unrelated constraint elimination

It reorganizes path constraints in a trie and traverses it with a depth first search

It achieves:

- up to 15.2x speedup on a complete concolic execution
- statistically significant coverage progression improvements in 79% of tested benchmarks