Triereme: Speeding up hybrid fuzzing through efficient query scheduling

Elia Geretto\textsuperscript{1}  Julius Hohnerlein\textsuperscript{1}  Cristiano Giuffrida
Herbert Bos  Erik van der Kouwe  Klaus v. Gleissenthall

Vrije Universiteit Amsterdam

Annual Computer Security Applications Conference, 2023

\textsuperscript{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
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

```c
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

```c
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 \(a = \{1, 2, 3\}\):

\[
1 \geq n
\]
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

```c
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 \( a = \{1, 2, 3\} \):

\[
\begin{align*}
1 & \geq n \\
1 & < n \land a[0] > a[1]
\end{align*}
\]
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

```c
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

```c
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

```c
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`
SMT solvers have an assertion stack that allows to push and pop states

Easily exploitable with unmodified query schedules (common prefix)

Query schedule:

1 \geq n
1 < n \land a[0] > a[1]
1 < n \land a[0] \leq a[1]
\land 2 \geq n
...

Interpretation:

\neg A
A \land \neg B
A \land B \land \neg C
...

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:
\[
1 < n \land a[0] \leq a[1] \\
\land 2 < n \land a[1] \leq a[2] \\
\land 3 < n
\]

After:
\[
1 < n \land 2 < n \land 3 < n
\]
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
Triereme - Example

Path constraints:

1 \geq n
a[0] > a[1]
1 < n \land 2 \geq n
a[0] \leq a[1] \land a[1] > a[2]
1 < n \land 2 < n \land 3 < n

Trie:
Triereme - Example

Path constraints:

1 \geq n
\text{a}[0] > \text{a}[1]
1 < n \quad \&\& \quad 2 \geq n
\text{a}[0] \leq \text{a}[1] \quad \&\& \quad \text{a}[1] > \text{a}[2]
1 < n \quad \&\& \quad 2 < n \quad \&\& \quad 3 < n

Trie:
Path constraints:

1 \geq n

\text{a}[0] > \text{a}[1]

1 < n \text{ and } 2 \geq n

\text{a}[0] \leq \text{a}[1] \text{ and } \text{a}[1] > \text{a}[2]

1 < n \text{ and } 2 < n \text{ and } 3 < n

Trie:
Triereme - Example

Path constraints:

\[ 1 \geq n \]
\[ a[0] > a[1] \]
\[ 1 < n \land 2 \geq n \]
\[ a[0] \leq a[1] \land a[1] > a[2] \]
\[ 1 < n \land 2 < n \land 3 < n \]

Trie:
Triereme - Example

Path constraints:

1 \geq n
a[0] > a[1]
1 < n \&\& 2 \geq n
1 < n \&\& 2 < n \&\& 3 < n

Trie:
Triereme - Example

Path constraints:

1 >= n
a[0] > a[1]
1 < n && 2 >= n
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
Evaluation - Example

![Graph showing coverage over time for freetype2_ftfuzzer, AFL++, Baseline, and Trie.](image)

- **x-axis**: Time (hours)
- **y-axis**: Cov. (# edges)

Lines and markers represent:
- **Blue line**: AFL++
- **Orange line**: Baseline
- **Green line**: Trie

Coverage increases over time for all methods, with Trie generally showing the highest coverage, followed by Baseline and then AFL++.
Evaluation - Negative selection

- harfbuzz_hb-sha...
- woff2-2016-05-0...
- re2-2014-12-09

Cov. (# edges)

Time (hours)

AFL++
Baseline
Trie
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 15.2× speedup on a complete concolic execution
- statistically significant coverage progression improvements in 79% of tested benchmarks