Multi-Solver Support in Symbolic Execution

Hristina Palikareva, Cristian Cadar

Imperial College
London

SMT Workshop 2014, Vienna, 17 July 2014

Dynamic Symbolic Execution

Automated program analysis technique that employs an SMT solver to systematically explore paths through a program

- heuristics to prioritise interesting paths
- generating, for each explored path, a test input exercising it

GENERATION OF HIGH-COVERAGE TEST SUITES

BUG FINDING

Uncovered deep, corner-case bugs in complex real-world software

Active Area of Research

OPEN-SOURCE CREST, KLEE, SYMBOLIC JPF

INDUSTRY MICROSOFT (SAGE, PEX)
NASA (SYMBOLIC JPF, KLEE)
IBM (APOLLO)
FUJITSU (SYMBOLIC JPF, KLEE/ KLOVER)

KLEE

Symbolic execution tool based on LLVM compiler framework

- Mixed concrete/symbolic interpreter for LLVM bit code
- Targets mainly C programs
- Employs STP as default solver
- Available as open source from:

http://klee.llvm.org

Plan for the Talk

Toy example

Outline the main characteristics of the SMT queries

- illustrated with data obtained by running `KLEE`

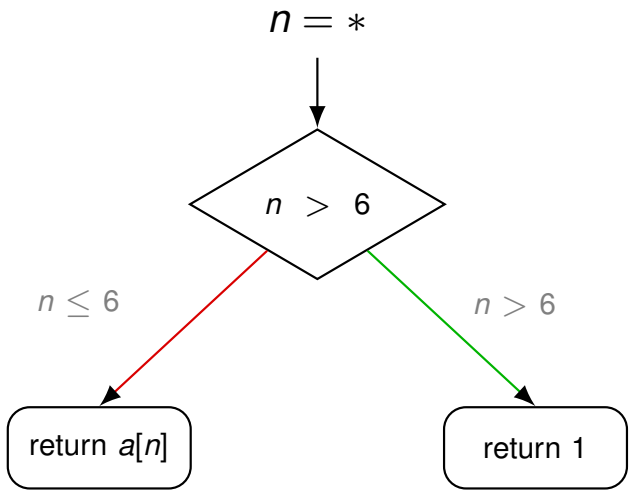
Introduce an extension of `KLEE` that uses `Boolector`, `STP` and `Z3` via `metaSMT`

- compare the solvers' performance

Discuss options for designing a parallel portfolio solver

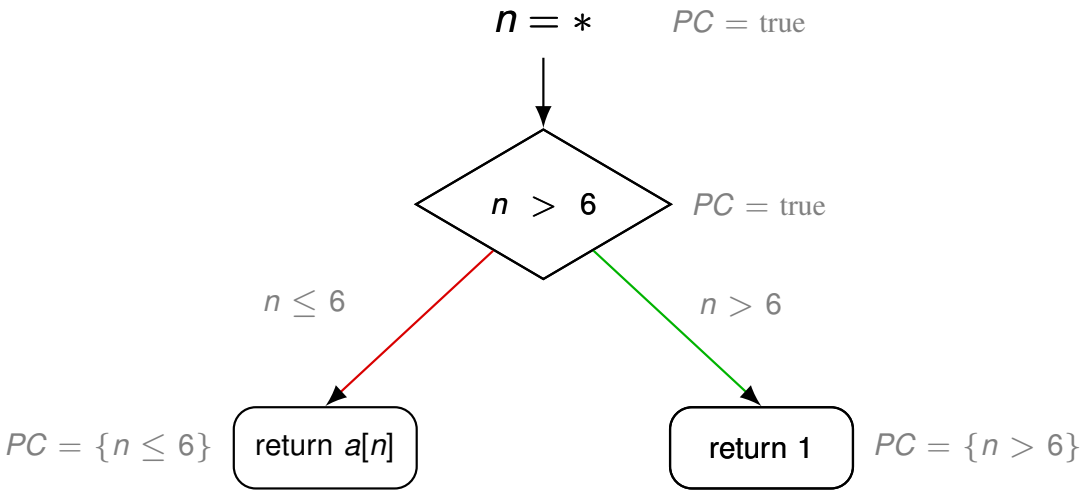
Dynamic Symbolic Execution: Toy Example

```
int main() {
  int a[7] =
      {2,3,5,7,11,13,17};
  int n = symbolic();
  if (n > 6) {
    return 1;
  }
  return a[n];
}
```



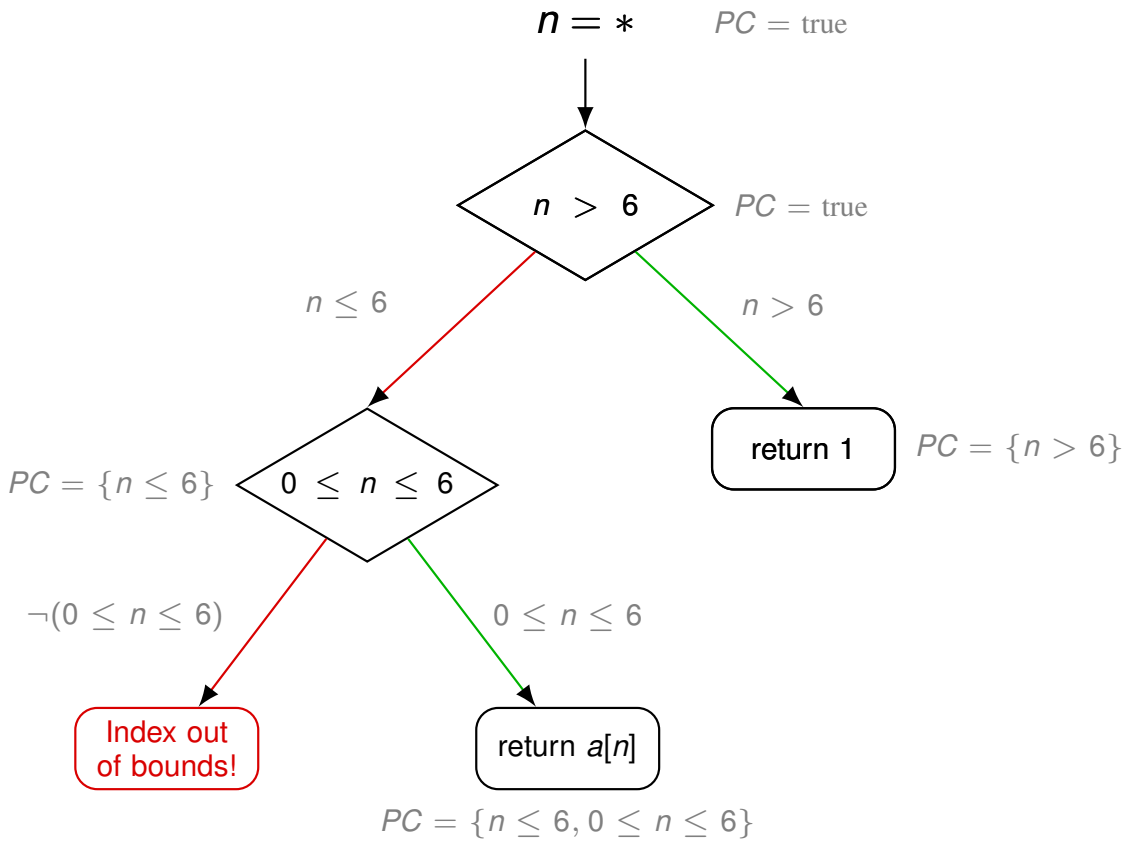
Dynamic Symbolic Execution: Toy Example

```
int main() {
  int a[7] =
      {2,3,5,7,11,13,17};
  int n = symbolic();
  if (n > 6) {
    return 1;
  }
  return a[n];
}
```



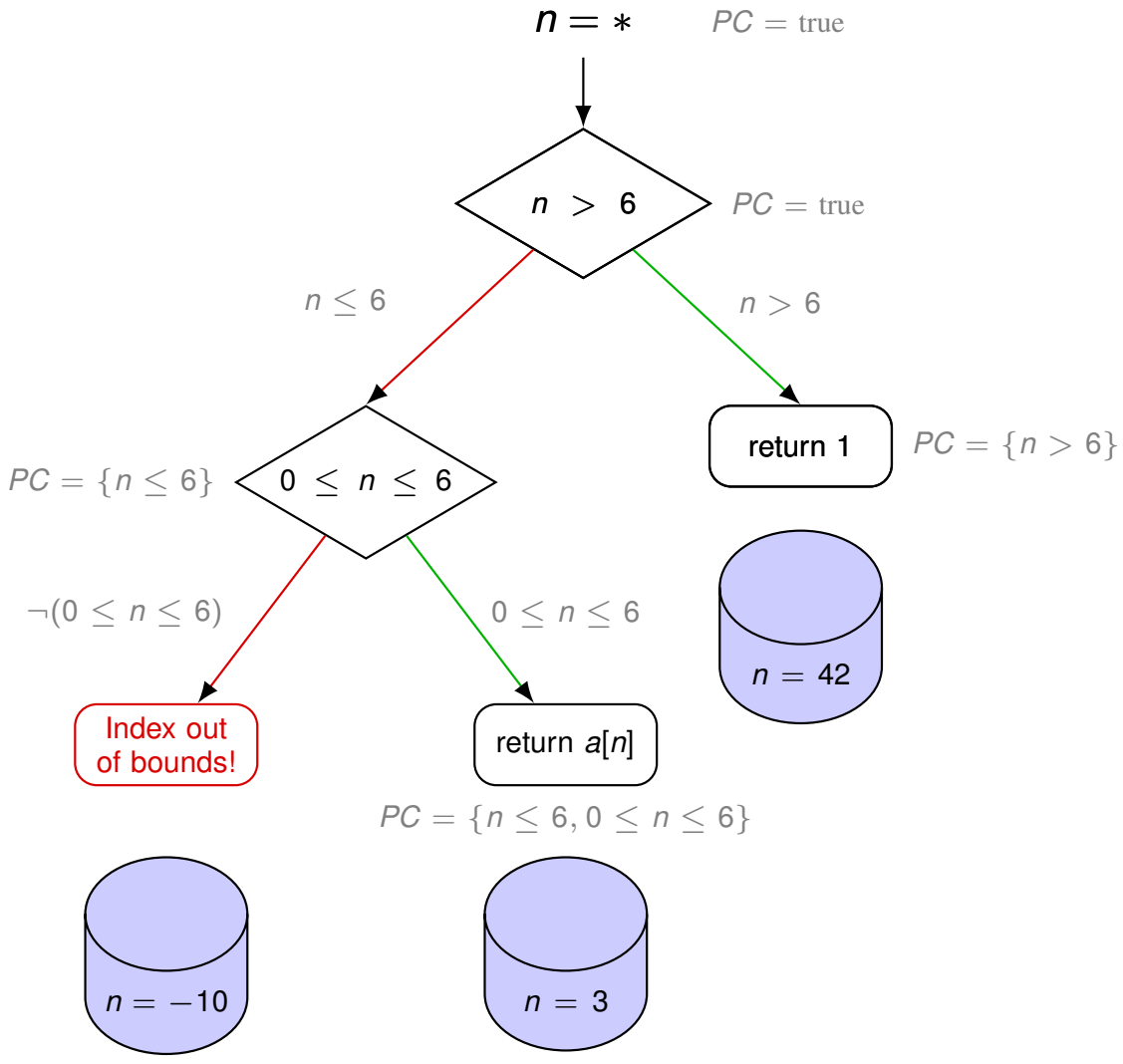
Dynamic Symbolic Execution: Toy Example

```
int main() {
  int a[7] =
      {2,3,5,7,11,13,17};
  int n = symbolic();
  if (n > 6) {
    return 1;
  }
  return a[n];
}
```



Dynamic Symbolic Execution: Toy Example

```
int main() {
  int a[7] =
      {2,3,5,7,11,13,17};
  int n = symbolic();
  if (n > 6) {
    return 1;
  }
  return a[n];
}
```



Challenges in Symbolic Execution

Path Explosion

Number of paths exponential in number of symbolic branches

- Possibly infinite!

Constraint Solving

- Often the main performance bottleneck!
  - 12 `GNU Coreutils`, each ran for 1h using `KLEE`

Solver (% of time)	
total	STP
97.1	90.5

Characteristics of the SMT Queries

Characteristics of the SMT Queries

1. Array Operations

- Programs often take as input arrays (e.g., strings)
- Concrete arrays become part of the symbolic constraints
  - when indexed by symbolic input
- Symbolic pointers and pointer arithmetic modelled using arrays

2. Bit-Level Accurate Constraints

Motivation: bugs are often triggered by corner cases related to:

- Bitwise operations, arithmetic overflows, pointer casting, . . .

Characteristics of the SMT Queries

1. Array Operations

- Programs often take as input arrays (e.g., strings)
- Concrete arrays become part of the symbolic constraints
  - when indexed by symbolic input
- Symbolic pointers and pointer arithmetic modelled using arrays

2. Bit-Level Accurate Constraints

Motivation: bugs are often triggered by corner cases related to:

- Bitwise operations, arithmetic overflows, pointer casting, . . .

`KLEE` is precise!

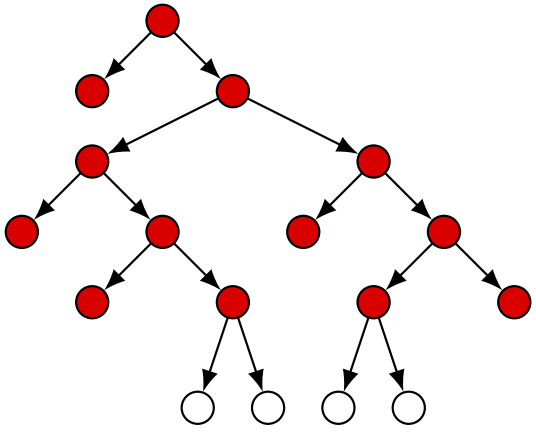
- Treats memory as untyped bytes
  - models each memory block as an array of 8-bit BVs
- Encodes program executions using the SMT theory `QF_ABV`

Characteristics of the SMT Queries

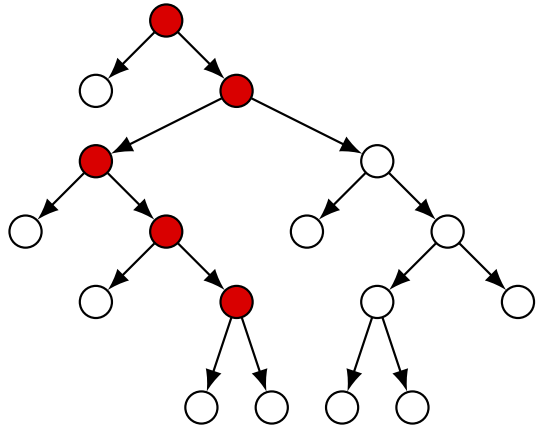
3. Large Number of Queries

Query at every symbolic branch and dangerous symbolic operation

BMC



Symbolic Execution

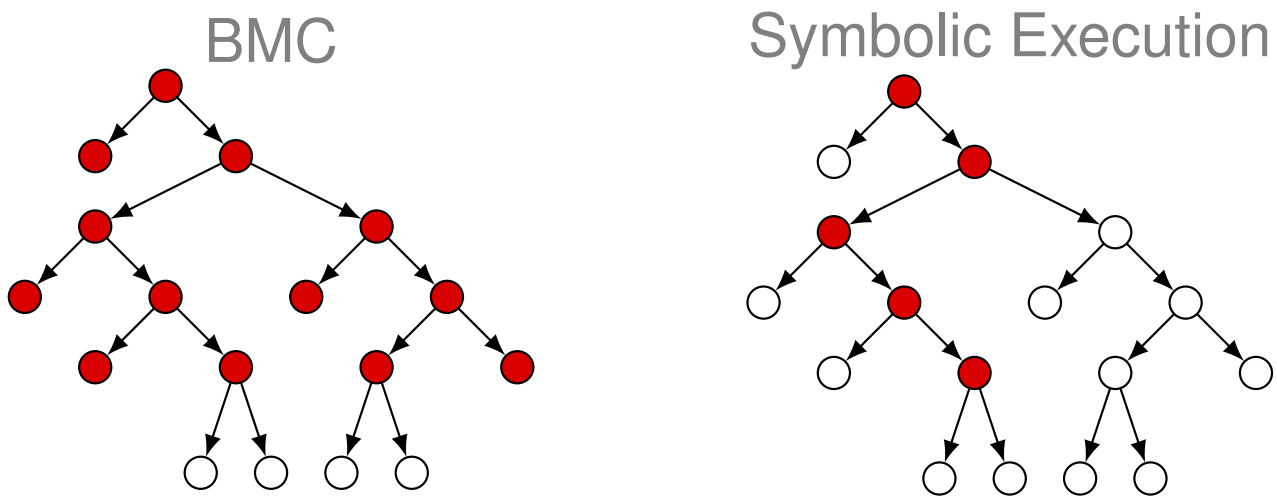


Queries *typically* much simpler, but significantly more of them!

Characteristics of the SMT Queries

3. Large Number of Queries

Query at every symbolic branch and dangerous symbolic operation

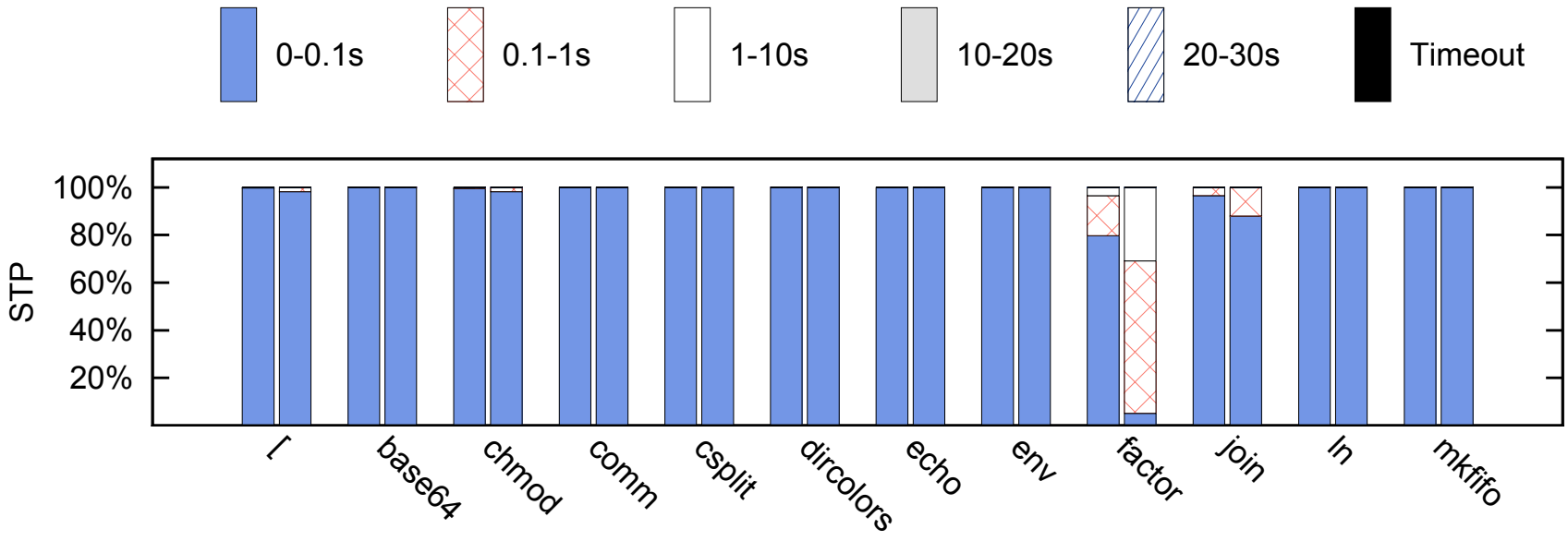


Queries *typically* much simpler, but significantly more of them!

SMT solver needs to:

- Solve efficiently myriads of relatively simple queries (conjunctions)
  - `KLEE` uses default per-query timeout of 30s
- Optimise performance for sequences of queries

Distribution of Query Types



- Left bar: % of queries solved by `STP` in each time interval
- Right bar: % of time spent executing queries of each type

High Query Rates

Application	Queries/sec
[	55.1
base64	73.8
chmod	36.4
comm	189.0
csplit	49.7
dircolors	49.3
echo	34.8
env	109.1
factor	5.3
join	36.6
ln	103.8
mkfifo	62.3
Average	67.1

Characteristics of the SMT Queries

4. Frequent Need for Concrete Solutions

Concrete solutions for satisfiable SMT queries required to:

- Generate test cases
- Interact with outside environment
  - e.g., before calling an uninstrumented function, all symbolic bytes that the function may access need to be concretized
- Simplify constraints
  - e.g., double pointer dereferences
- Apply optimizations
  - e.g., `KLEE` caches solutions for all SAT queries

Satisfiable assignments required for the majority of queries!

KLEE: Counterexample Cache

Maps constraint sets (PCs) to:

- Counterexample if SAT
- Special sentinel if UNSAT

Exploits subset/superset relations among constraint sets to determine satisfiability of subsequent queries:

- If set is UNSAT, any of its supersets is UNSAT too
- If set is SAT, any of its subsets is SAT too

KLEE: Counterexample Cache

$$\{x > 3, y > 2, x + y = 10\} \longmapsto \{x = 4, y = 6\}$$

More Observations

1. Adding constraints often does not invalidate solution:

$$\{x > 3, y > 2, x + y = 10, x < y\} \longmapsto \{x = 4, y = 6\}$$

- Specific to symbolic execution
- Reason: upon symbolic branch, we add constraint to the current PC
- $\Rightarrow$ solution will hold for either `then` or `else` branch
- Cheap to check: substitute solution in constraints, spare solver call
- The cache tries all of its stored subsets in turn until cache hit

KLEE: Counterexample Cache

$$\{x > 3, y > 2, x + y = 10\} \longmapsto \{x = 4, y = 6\}$$

More Observations

1. Adding constraints often does not invalidate solution:

$$\{x > 3, y > 2, x + y = 10, x < y\} \longmapsto \{x = 4, y = 6\}$$

   - Specific to symbolic execution
   - Reason: upon symbolic branch, we add constraint to the current PC
   - $\Rightarrow$ solution will hold for either `then` or `else` branch
   - Cheap to check: substitute solution in constraints, spare solver call
   - The cache tries all of its stored subsets in turn until cache hit

2. Cache hit rate depends on counterexamples stored in cache
   - If assignment in cache was $\{x = 7, y = 3\}$, no cache hit

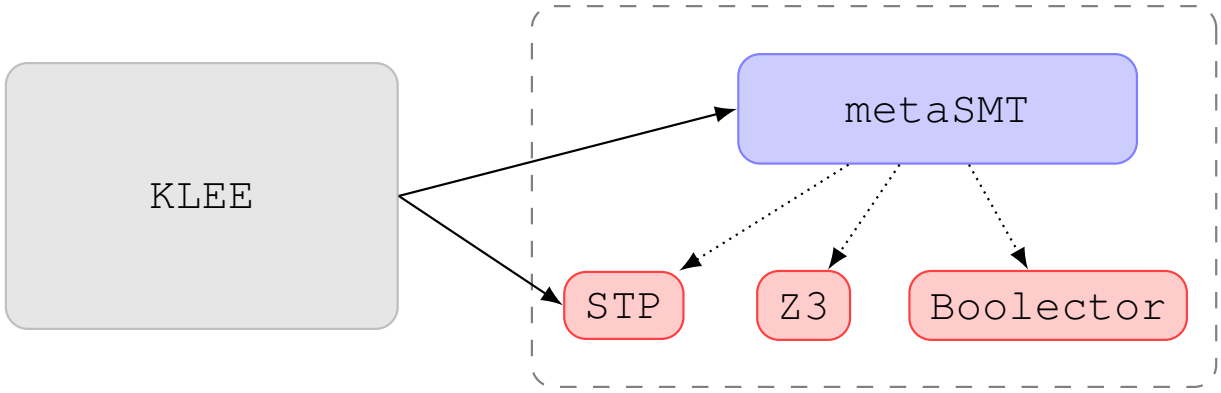
KLEE: Speedup With Counterexample Cache

Application	Queries/sec		Speedup
	No caching	Caching	
[	55.1	7.9	0.2
base64	73.8	42.2	0.6
chmod	36.4	12.6	0.4
comm	189.0	305.0	1.6
csplit	49.7	63.5	1.3
dircolors	49.3	4,251.7	86.2
echo	34.8	4.5	0.1
env	109.1	26.3	0.2
factor	5.3	22.6	4.2
join	36.6	3,401.2	92.9
ln	103.8	24.5	0.2
mkfifo	62.3	7.2	0.2
Average	67.1	680.8	10.2

Caching overall helps, but sometimes hurts performance!

Need better, more adaptive caching algorithms

Multi-Solver Support: KLEE with metaSMT



Critical to interact with solvers using their native APIs

- High query rate
- Average size of a KLEE query in SMTLIB – 100s of Kb
- Sending SMTLIB text through pipes too much parsing overhead

metaSMT

- Unified API for transparently using a number of SMT solvers
  - which is efficiently translated at compile time, through template meta-programming, into native APIs of solvers (< 3% overhead)

KLEE with metaSMT: Solver Comparison

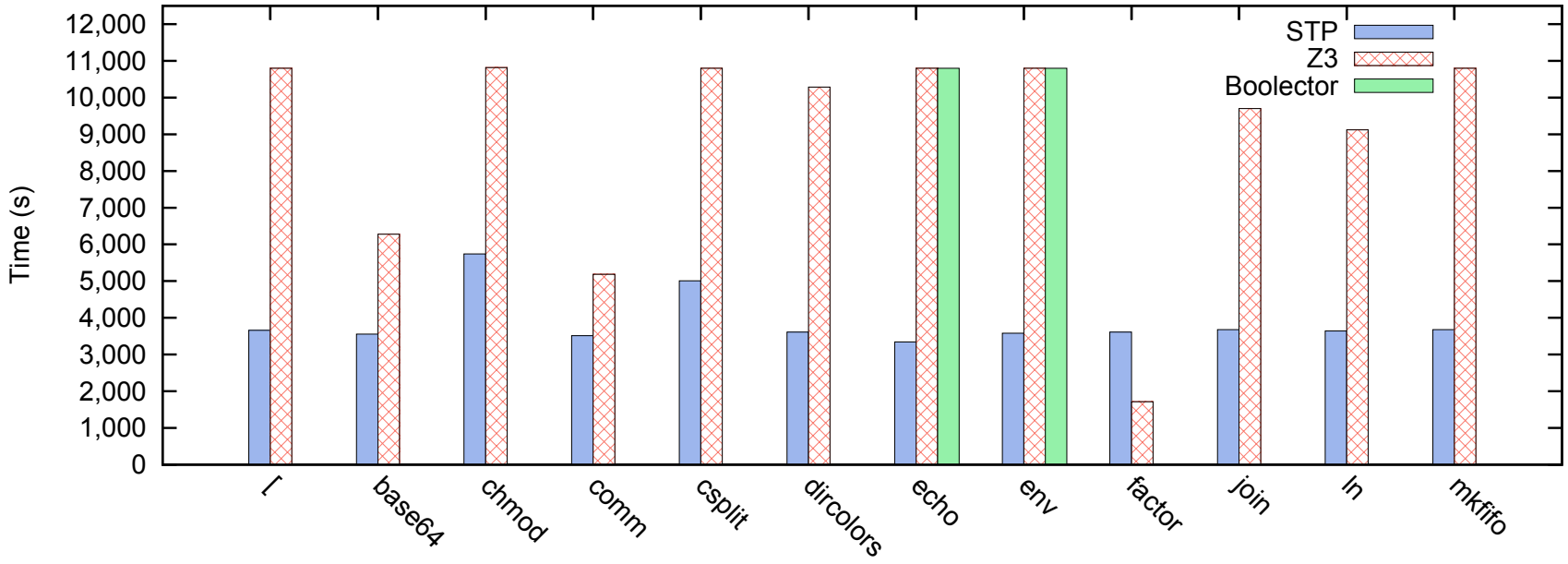
Benchmarks

12 applications from GNU Coreutils 6.10 application suite

Methodology

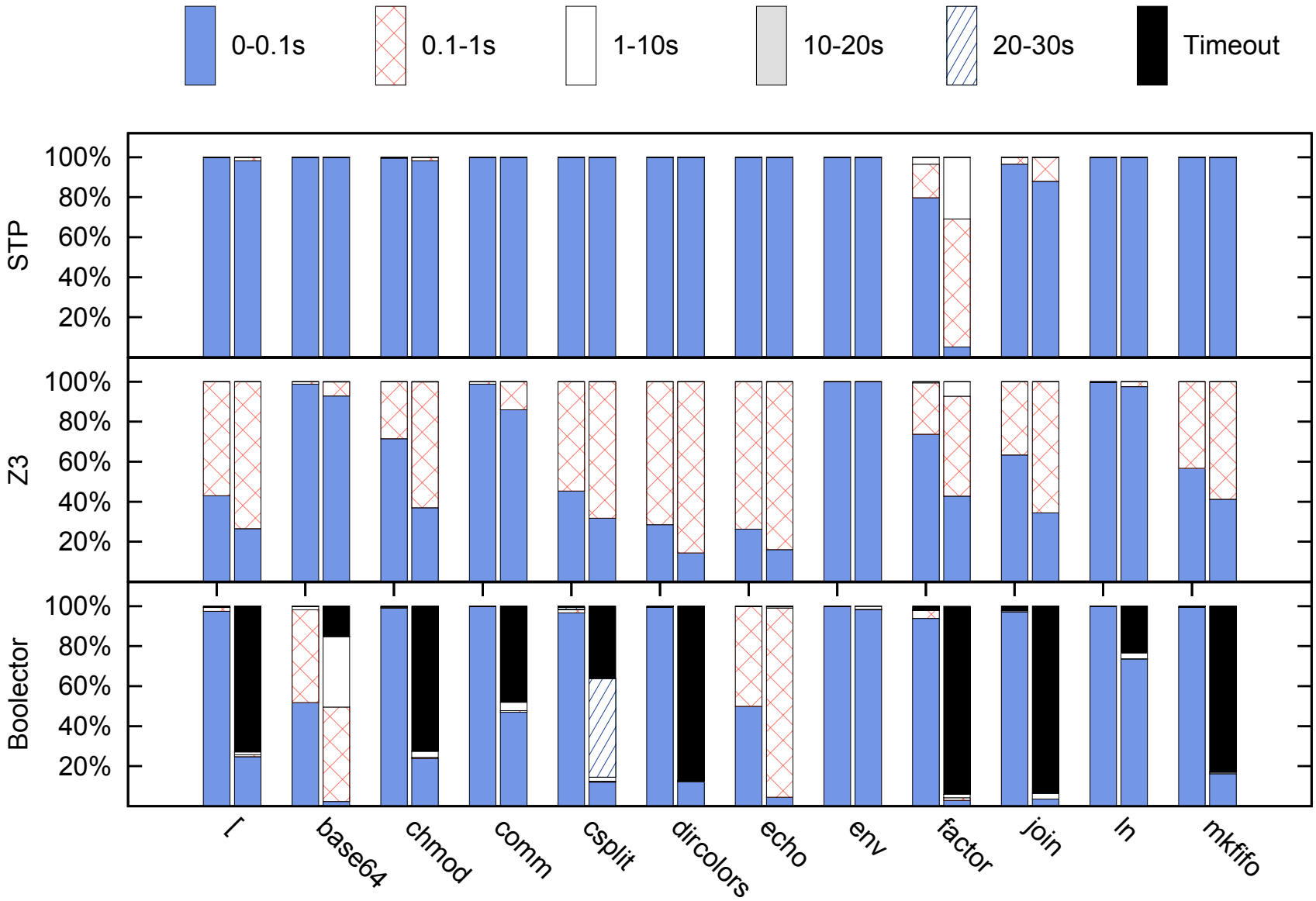
1. Run each benchmark for 1h using KLEE's default solver STP
2. Record number of executed instructions
3. Rerun each benchmark for the same number of instructions
   - with Boolector, STP, Z3 via metaSMT

Solver Comparison: No Caches



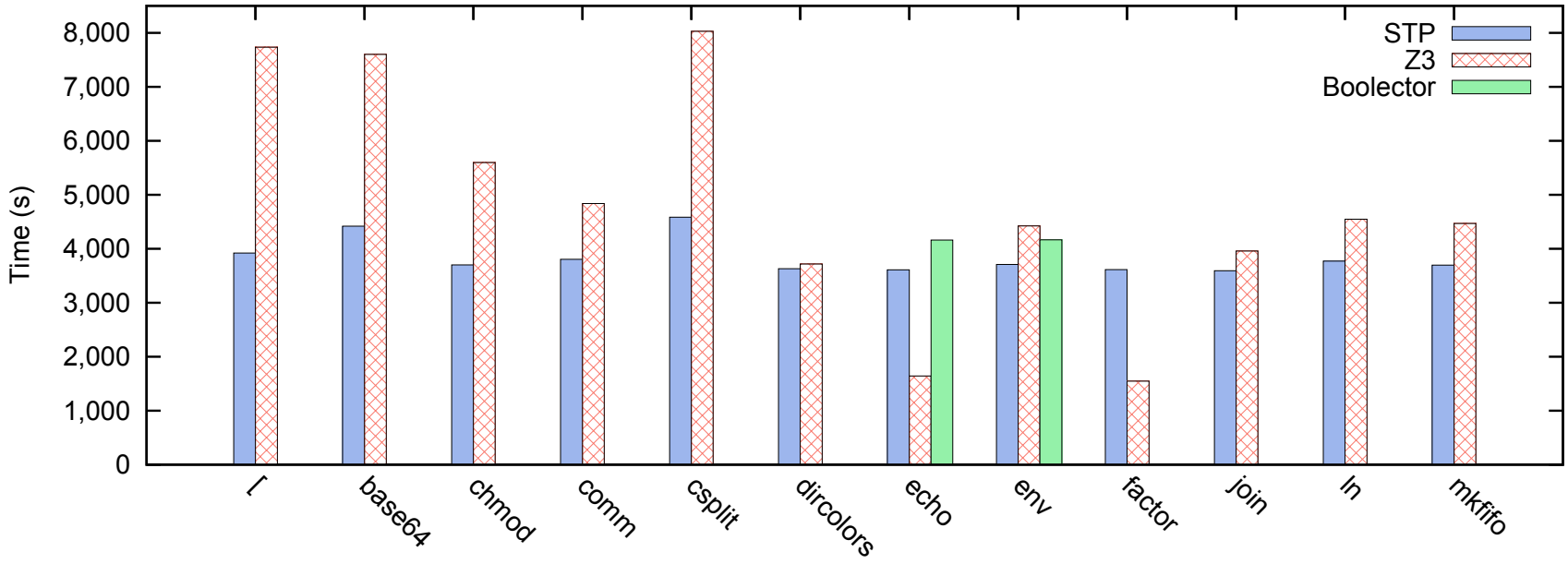
- Query timeout: 30s
- Overall `KLEE` timeout: 3h
- `STP` and `Z3` have no query timeouts
  - upon timeout, `KLEE` terminates the current execution path
- Overall winner: `STP`
- `Z3` beats `STP` on `factor`
- Disclaimer: SMT solvers used with their default configurations

Distribution of Query Types



- Left bar: % of queries solved by solver in time interval
- Right bar: % of time spent executing queries of each type

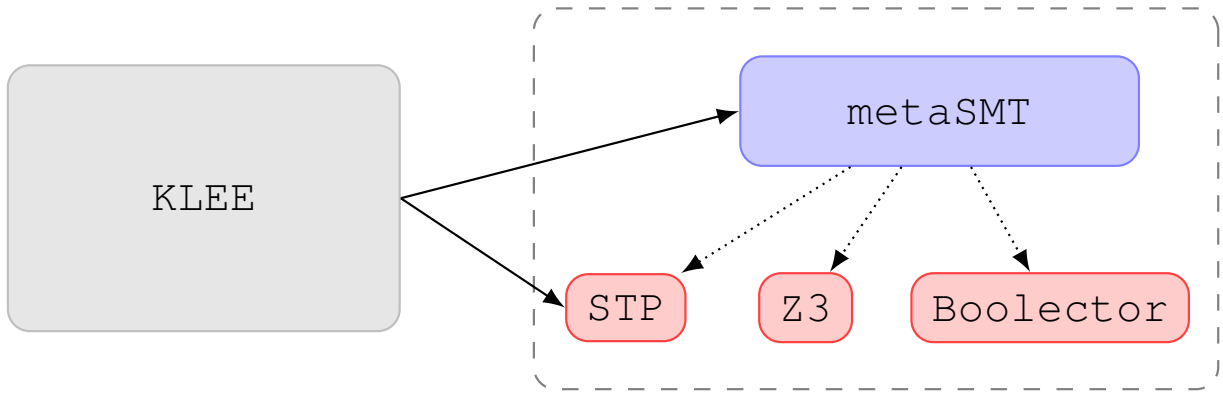
Solver Comparison: Caches Enabled



Effects of Caching

- Different solutions can lead to different cache hit rates
  - `env`: all solvers same # of queries, `Z3` 136 less cache hits
- Even when hit rates are the same, the order of solutions in the cache affects performance
  - `echo`: all solvers same # overall queries and queries reaching solver
  - `Z3`: spends less time in caching module, hence wins
  - for 764 queries that reach the solver, `STP` faster than `Z3`, 14s vs. 74s

Portfolio Solver in Symbolic Execution



Portfolio Solver: Level of Granularity

Coarsest-Grained Option

- Run multiple variants of `KLEE`, each equipped with different solver
- Give user the results associated with the best-performing variant

\+ No overhead, runs are independent, easy to deploy

  - Fix time budget, select the run that optimises a certain metric
  - Fix objective, abort when first variant achieves objective

Portfolio Solver: Level of Granularity

Coarsest-Grained Option

- Run multiple variants of `KLEE`, each equipped with different solver
- Give user the results associated with the best-performing variant

\+ No overhead, runs are independent, easy to deploy

- Fix time budget, select the run that optimises a certain metric
- Fix objective, abort when first variant achieves objective

Portfolio at the Level of Individual Solver Queries

\+ Can outperform all single-solver variants

- Reason: different solvers perform better on different queries

– Performance overhead can be quite high and negate benefits

- Reason: vast majority of queries take very little time to complete
- Time to spawn and monitor threads/processes can be higher than solving time

- Idea: start single solver, spawn more if it goes beyond given time

Portfolio Solver: Caching

Which counterexample values do we keep?

Store in Cache Counterexample of the First Solver

+ Easiest

Store in Cache Counterexamples of Multiple Solvers

- If they perform similarly
- Option to keep all or some of the counterexamples

+ Keeping more counterexamples may increase the hit rate

− But degrade overall performance
  - depending on order of counterexamples in cache

+ SMT and SAT solvers are incremental
  - may better tune their underlying heuristics

Portfolio Solver: Candidate Solvers

Include in Portfolio:

- Different SMT solvers
- Different versions of the same SMT solver
- Different configurations of the same SMT solver
  - plethora of options availabe
  - often impossible to tell in advance which parameter values will perform better on which queries
- Same SMT solver configured with different SAT solvers
  - and/or different SAT configurations

Summary

Presented our experience and results on integrating support for multiple SMT solvers in the symbolic execution engine `KLEE`.

- Outlined the key characteristics of the SMT queries
- Identitified several aspects for designing better SMT solvers
  - Counterexample values and caching
- Discussed options for designing a parallel portfolio solver

Support for using `Boolector`, `STP` and `Z3` via `metaSMT` now fully integrated in the mainstream version of `KLEE`:

`http://klee.llvm.org`

Discussion

New division in SMT-COMP targeted at symbolic execution tools

Why?

Symbolic execution needs are quite different from most other application domains, and the current SMT-COMP format and benchmarks are not exactly representative.

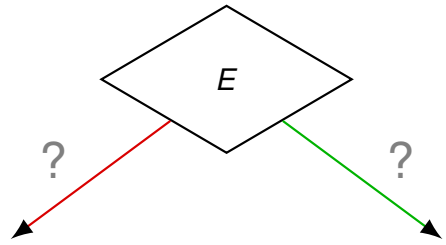
Significant Aspects

- Counterexample values
- Solving relatively simple queries fast
- Solving **sequences** of similar queries fast
- Fast API-based communication

KLEE: Types of Queries

Branch Queries

Issued when KLEE reaches a symbolic branch: (PC, E)



$\mathrm{PC} = \{C_1, C_2, \ldots, C_n\}$, SAT

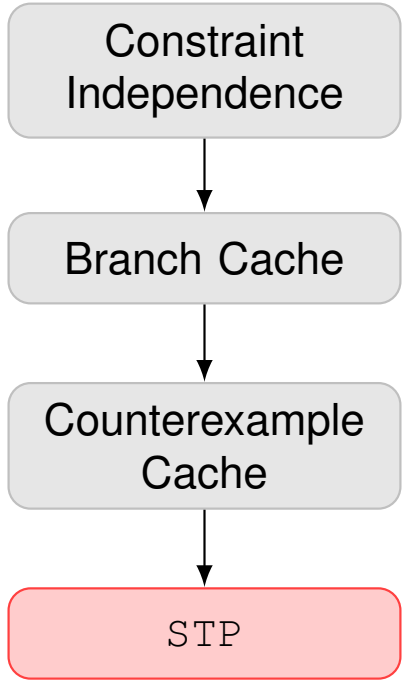
Is $\mathrm{PC} \Rightarrow E$ valid?

- provably true: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \neg E$ is UNSAT
- provably false: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge E$ is UNSAT
- neither

Counterexample Queries

- Used to request solution for current PC (e.g., at end of path)

KLEE: Constraint-Solving Optimisations



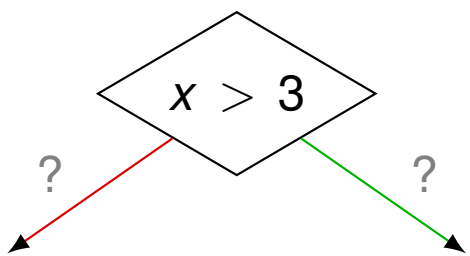
- Structured as a sequence of solver passes
- Can be enabled and disabled via KLEE's command-line options

KLEE: Constraint Independence

Essentially, eliminating redundant constraints

- Branches typically depend on small number of variables
- Given a branch query (PC, E), remove from PC all constraints which are not transitively related to E

$$\mathrm{PC} = \{x < 10,$$
$$z < 20,$$
$$x + y = 10,$$
$$w = 2 \mod z,$$
$$y > 5\}$$

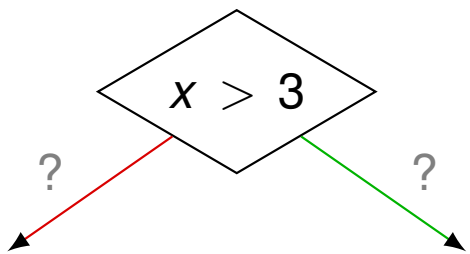


KLEE: Constraint Independence

Essentially, eliminating redundant constraints

- Branches typically depend on small number of variables
- Given a branch query (PC, E), remove from PC all constraints which are not transitively related to E

$$\mathrm{PC} = \{x < 10,$$
$$\cancel{z < 20,}$$
$$x + y = 10,$$
$$\cancel{w = 2 \mod z,}$$
$$y > 5\}$$



Experimental Evaluation: Main Challenge

Configure `KLEE` to behave deterministically across runs

Reason

`KLEE` relies on:

- Timeouts, time-sensitive search heuristics
- Concrete memory addresses (e.g., values returned by `malloc`)
- Satisfying assignments returned by SMT solver

Steps to Address Nondeterminism

- Employed DFS search heuristics
- Turned off address-space layout randomisation
- Implemented deterministic memory allocator
- Used deterministic seeds for random number generation
- Details in the paper