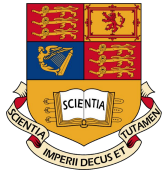
SOFTWARE RELIABILITY GROUP



Imperial College London

Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?

Timotej Kapus Martin Nowack Cristian Cadar
Imperial College London

Motivation

- “Why are you using bitvectors when integer solvers are so much faster?”
- We get asked this a lot
- Most recently at Dagstuhl on constraints solving
- Hand-wavy answer about precision and overflow

Overflow

- Bitvectors overflow
- Integers do not
- Does it matter?

```
a: integer, b: integer
(> a 0)
(> b 0)
(< (+ a b) 0)
```

UNSAT

```
a: bitvector, b: bitvector
(> a 0)
(> b 0)
(< (+ a b) 0)
```

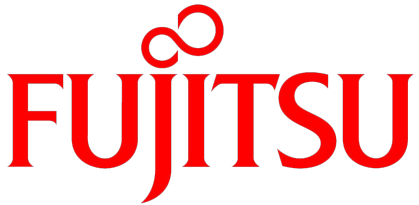
SAT

(a,b = MAX_INT)

Symbolic Execution

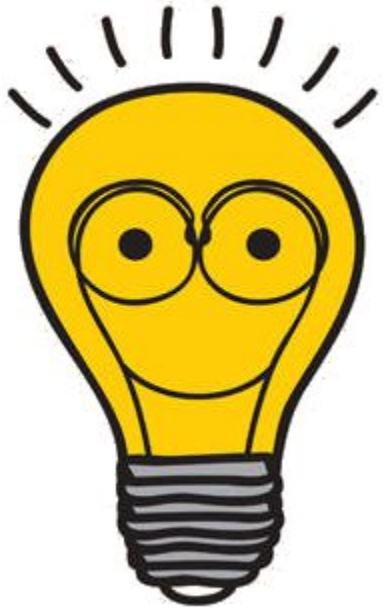
- Program analysis technique
- Active research area
- Used in industry
  - IntelliTest, SAGE
  - KLOVER

Angr



Why symbolic execution?

- *No false-positives!*
  - Every bug found has a concrete input triggering it
- Can interact with the environment
  - I/O, unmodeled libraries



Why (not) symbolic execution?

- Scalability, scalability, scalability
  - Constraint solving is hard
  - Path explosion

Symbolic execution

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```

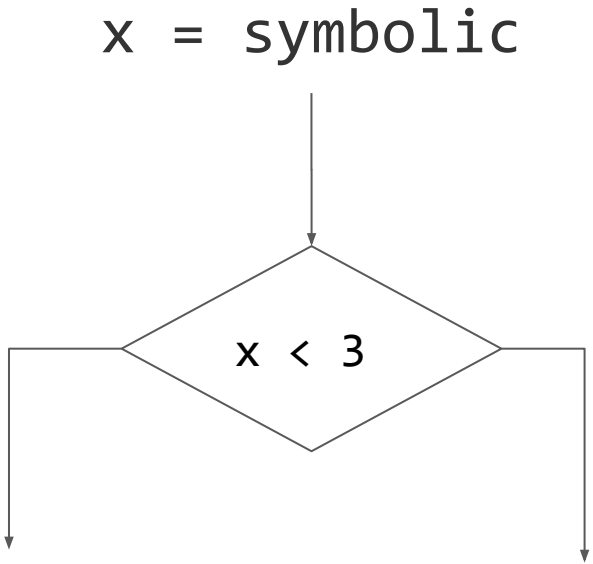
Symbolic execution

x = symbolic

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```

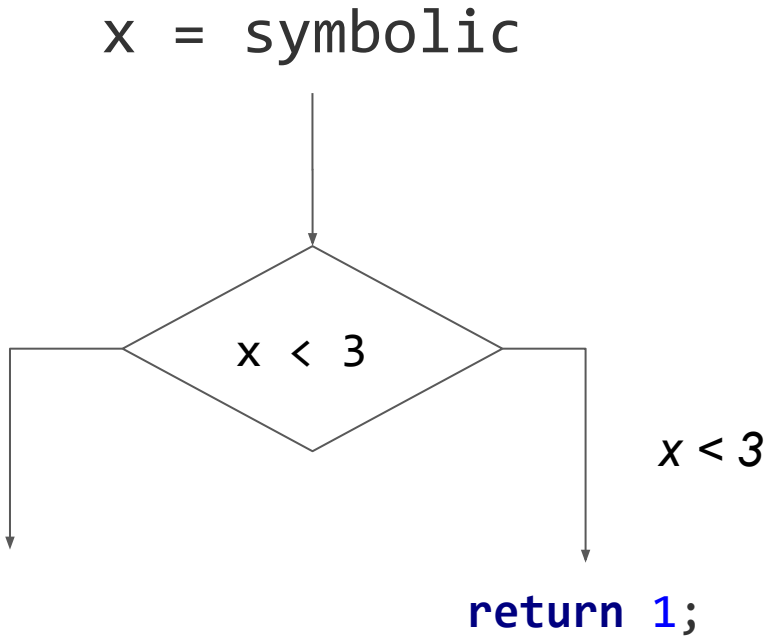
Symbolic execution

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```



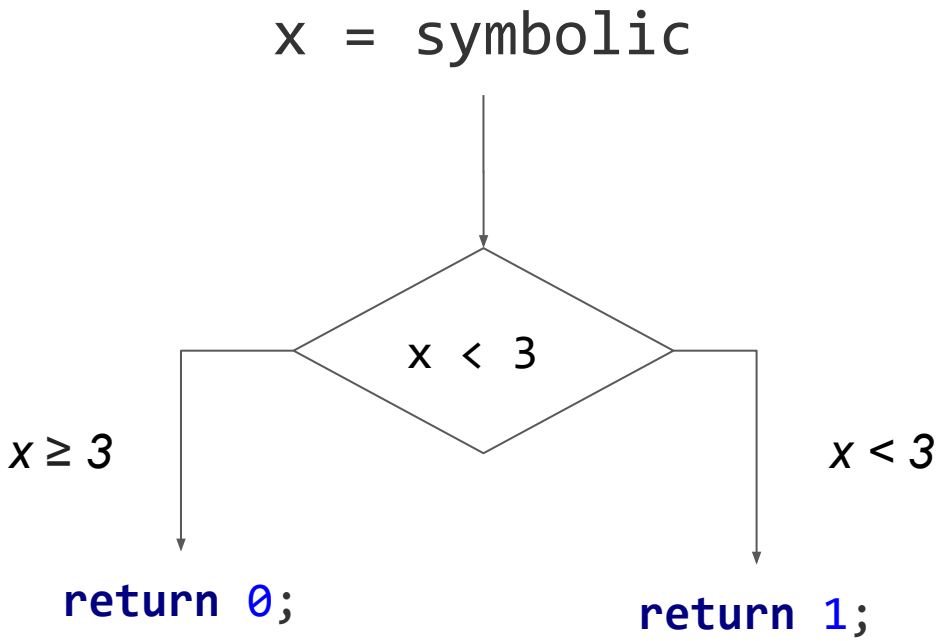
Symbolic execution

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```



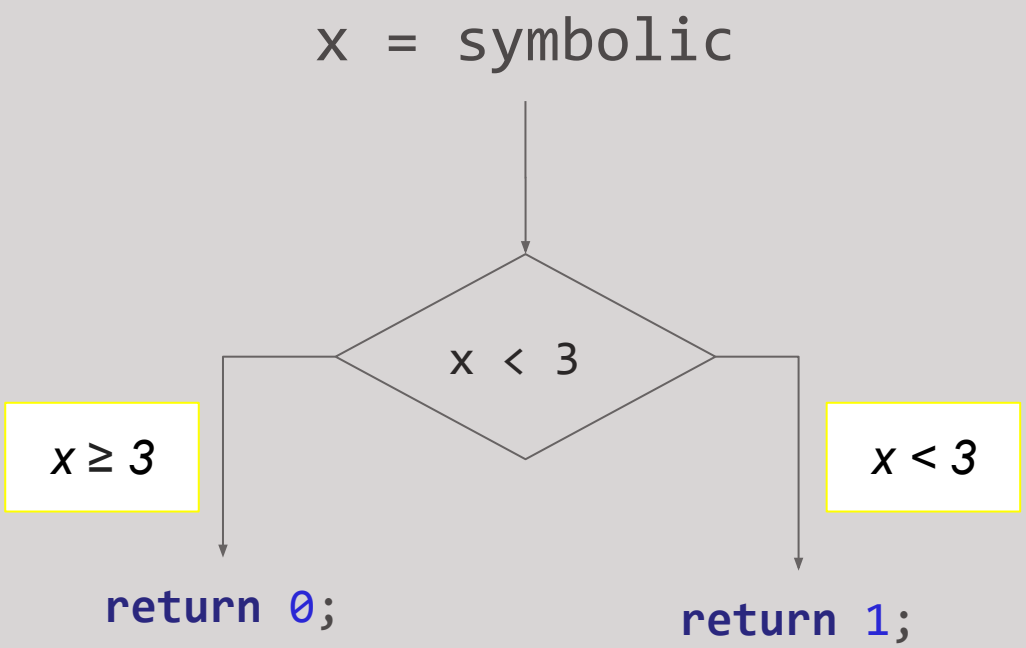
Symbolic execution

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```



This talk

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```



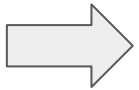
This talk

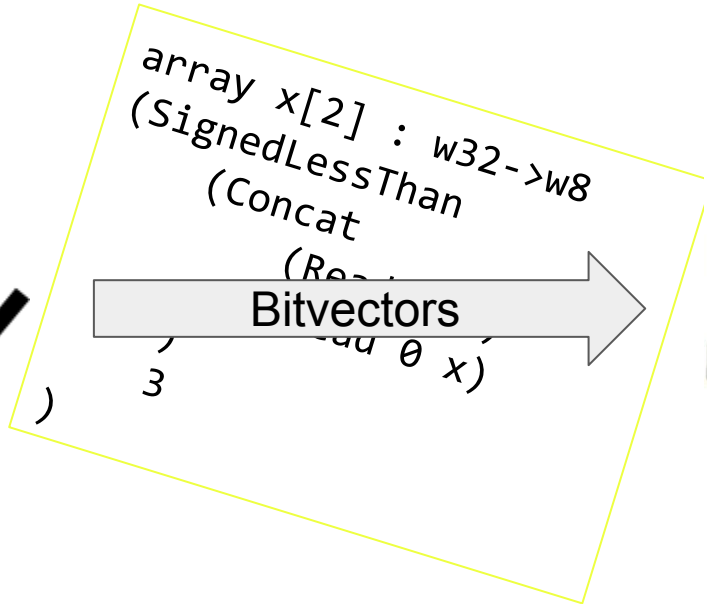
We are going to discuss the difference between encoding constraints in theory of integers vs theory of bitvectors for symbolic execution.

Background: KLEE

- Robust and optimized symbolic executor
- Gathers constraints in theory of bitvectors
- Off-the-shelf solver: Z3

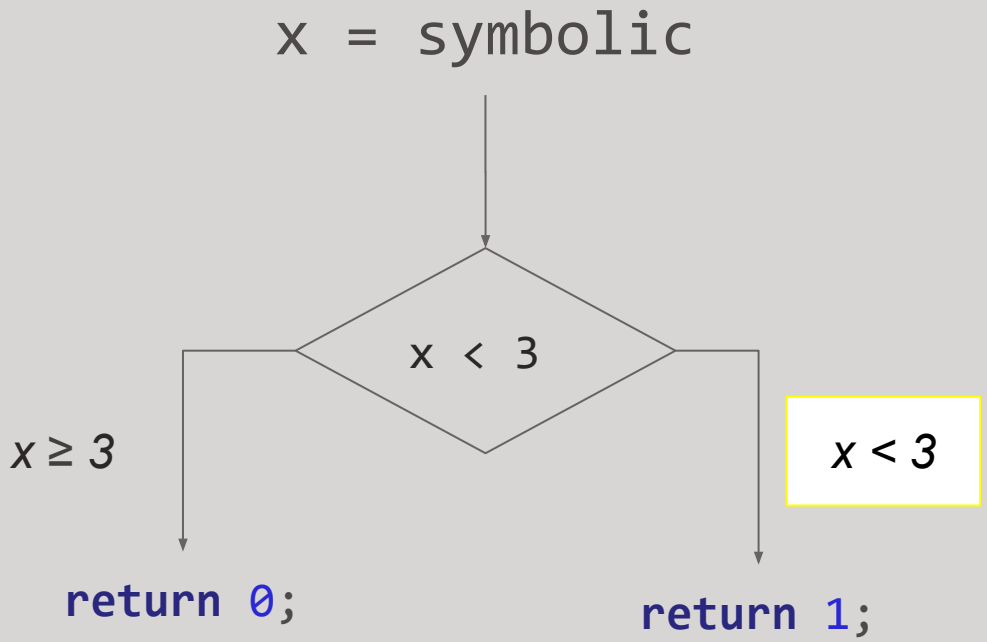
```
int lessThanThree(short x){
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```





Constraints in theory of bitvectors (and arrays)

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```



Constraints in theory of bitvectors (and arrays)

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```

x = symbolic

```
array x[2] : w32->w8
(SignedLessThan
    (Concat
        (Read 1 x)
        (Read 0 x)
    )
    3
)
```

x < 3

```
urn 1;
```

Constraints in theory of bitvectors (and arrays)

Byte view of memory

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```

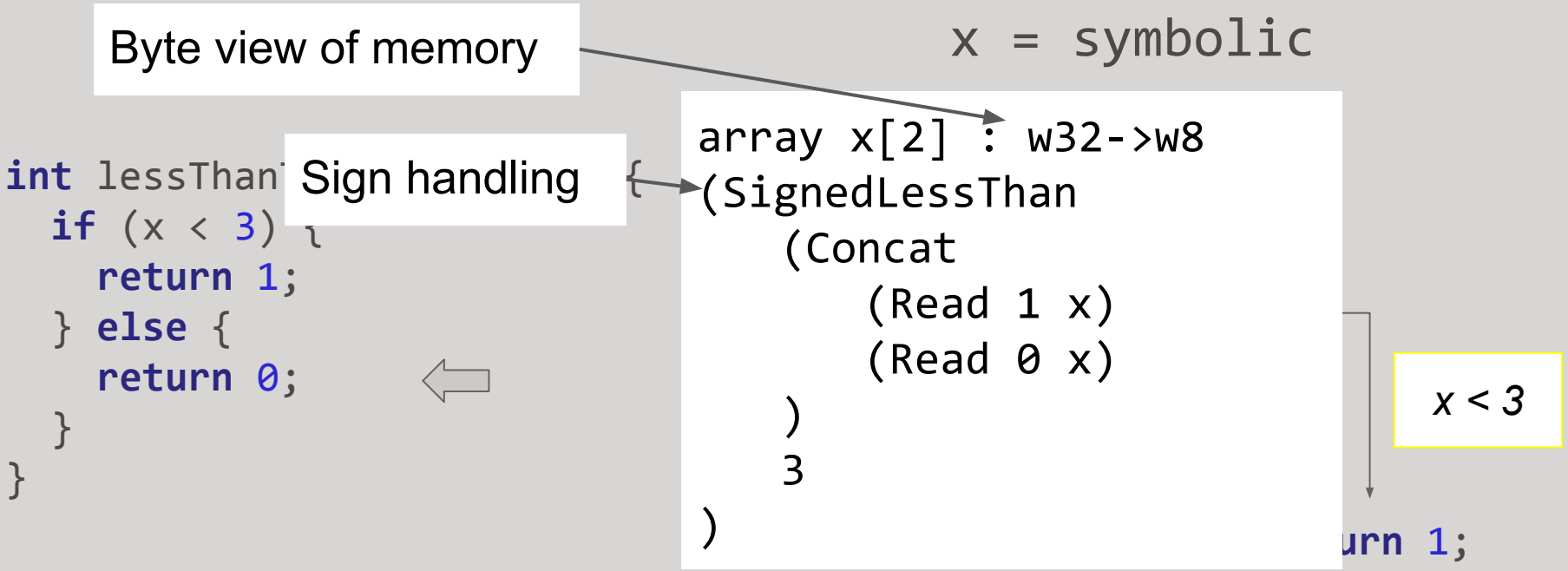
x = symbolic

```
array x[2] : w32->w8
(SignedLessThan
    (Concat
        (Read 1 x)
        (Read 0 x)
    )
    3
)
```

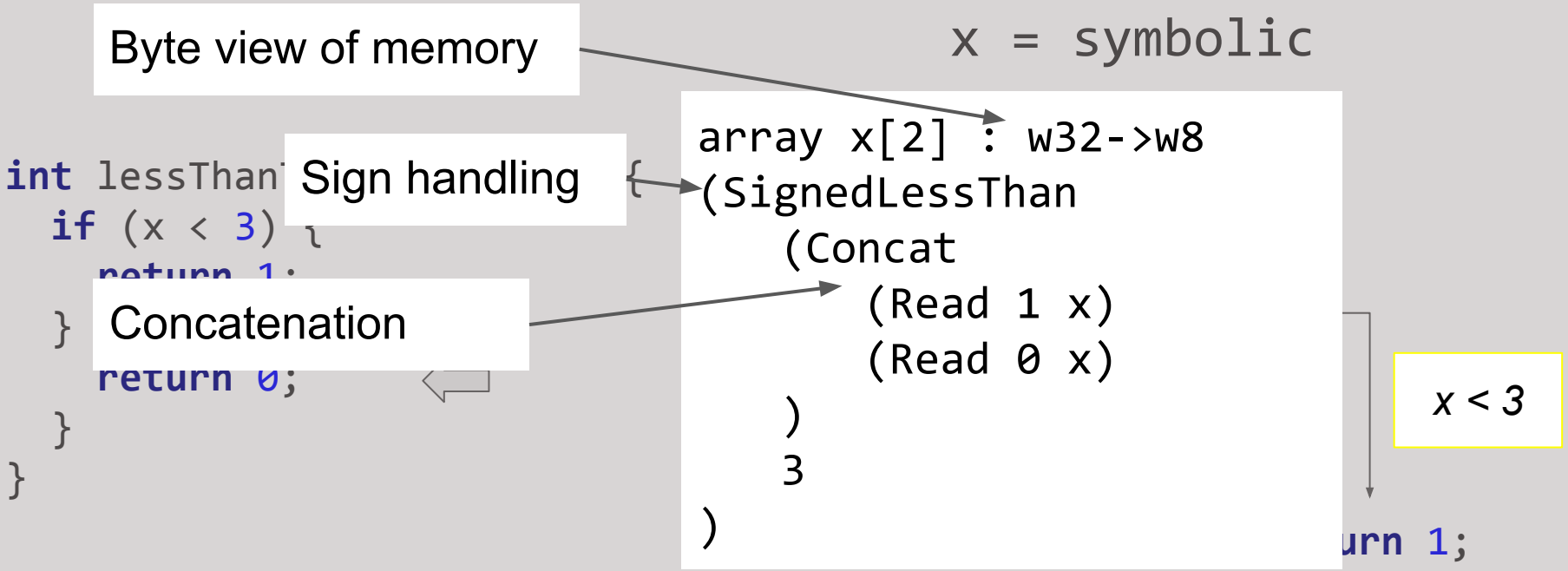
x < 3

```
urn 1;
```

Constraints in theory of bitvectors (and arrays)



Constraints in theory of bitvectors (and arrays)



(target) Constraints in theory of integers (and arrays)

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```

x = symbolic

```
array x[1] : i->i
(< (Read 0 x) 32768)
(> (Read 0 x) -32769)

(<
   (Read 0 x)
   3
)
```

x < 3

```
urn 1;
```

(target) Constraints in theory of integers (and arrays)

"Integer" view of memory

```
int lessThanThree(short x) {
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```

x = symbolic

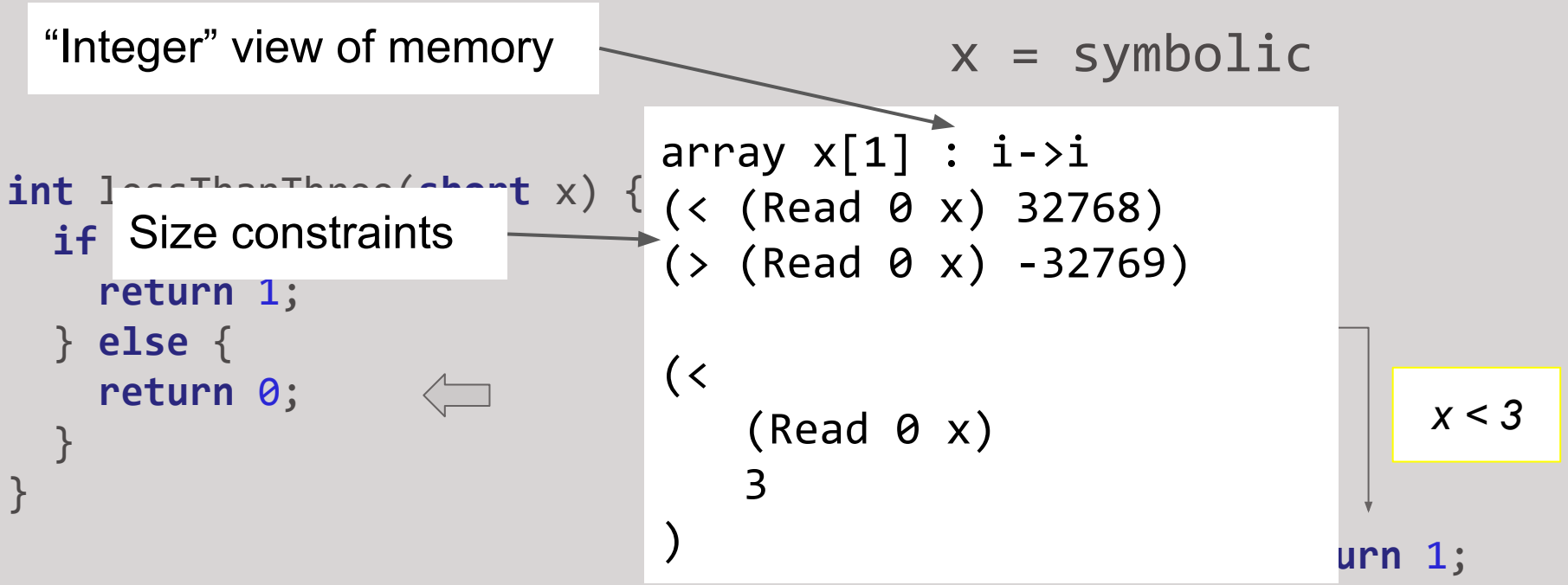
```
array x[1] : i->i
(< (Read 0 x) 32768)
(> (Read 0 x) -32769)

(<
    (Read 0 x)
    3
)
```

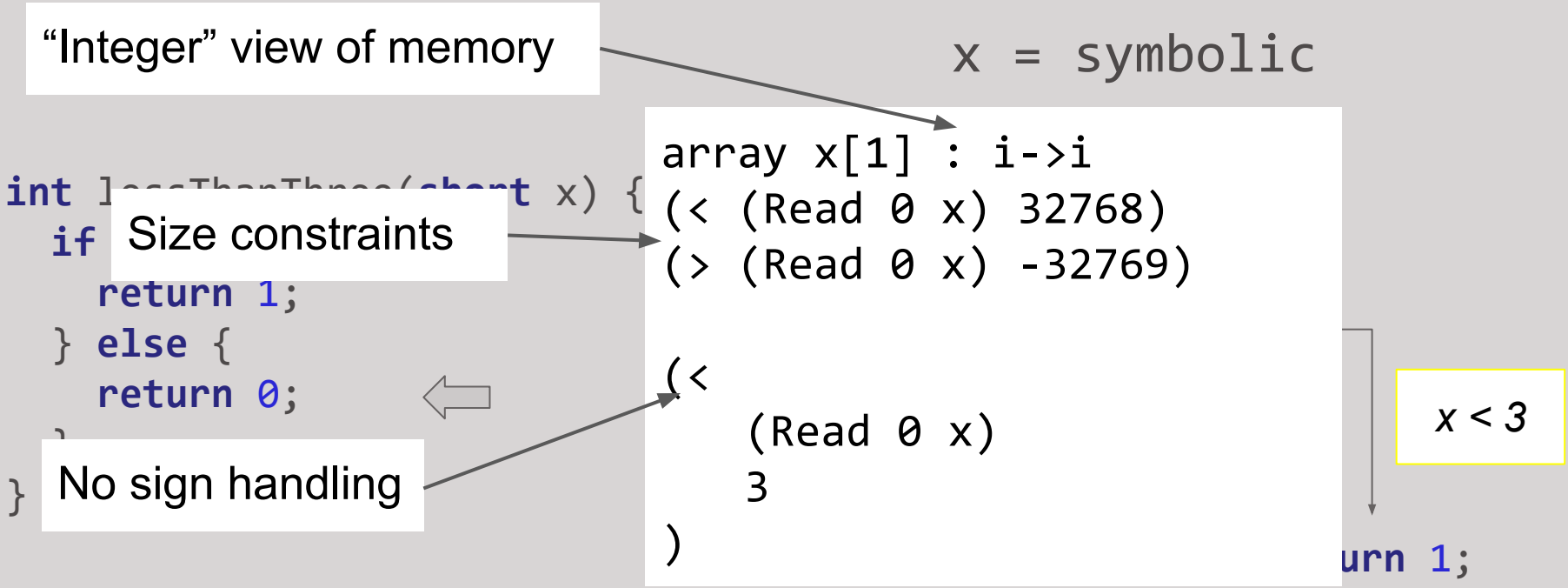
x < 3

```
urn 1;
```

(target) Constraints in theory of integers (and arrays)

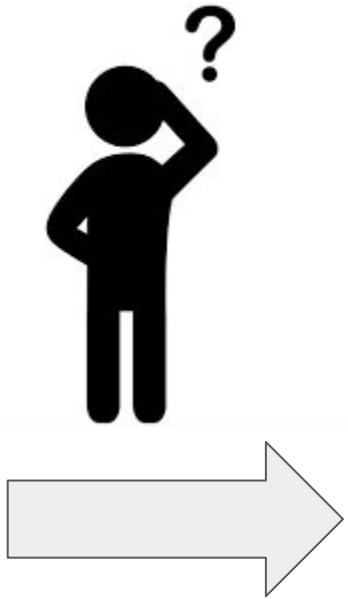


(target) Constraints in theory of integers (and arrays)



The challenge

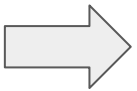
```
array x[2] : w32->w8
(SignedLessThan
    (Concat
        (Read 1 x)
        (Read 0 x)
    )
    3
)
```



```
array x[1] : i->i
(< (Read 0 x) 32768)
(> (Read 0 x) -32769)

(<
    (Read 0 x)
    3
)
```

```
int lessThanThree(short x){
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```

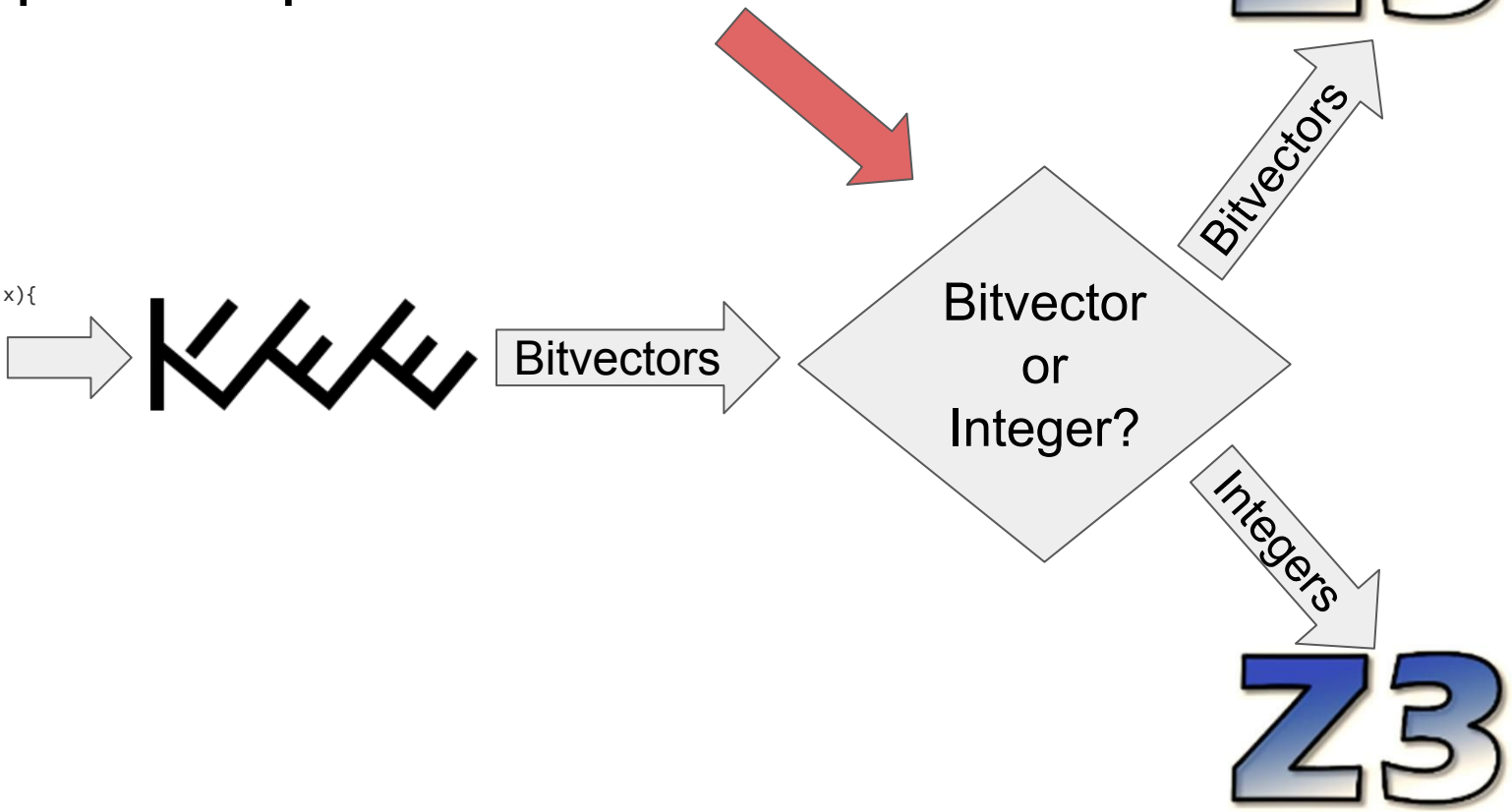


```
array x[2] : w32->w8
(SignedLessThan
    (Concat
        (Re
3
)
```

Bitvectors

Our approach: partial solver

```
int lessThanThree(short x){
  if (x < 3) {
    return 1;
  } else {
    return 0;
  }
}
```



Our approach: partial solver

- Arrays of bytes to arrays of integers
  - Type information
- Pattern match concat and reads
- Sign: Treat everything as signed
  - Casts between signed and unsigned
- Ignore:
  - Bitwise operations
  - Type mismatches
  - Casts
- Best effort: delegate to bitvector solver

Evaluation

- Synthetic:
  - Sort benchmark
  - Test-Comp: ECA-set
- Coreutils

Sort benchmark

- Crosscheck two sort implementations
- Mostly inequality queries
- Measure time it takes KLEE to fully explore all the paths
- No loss of precision

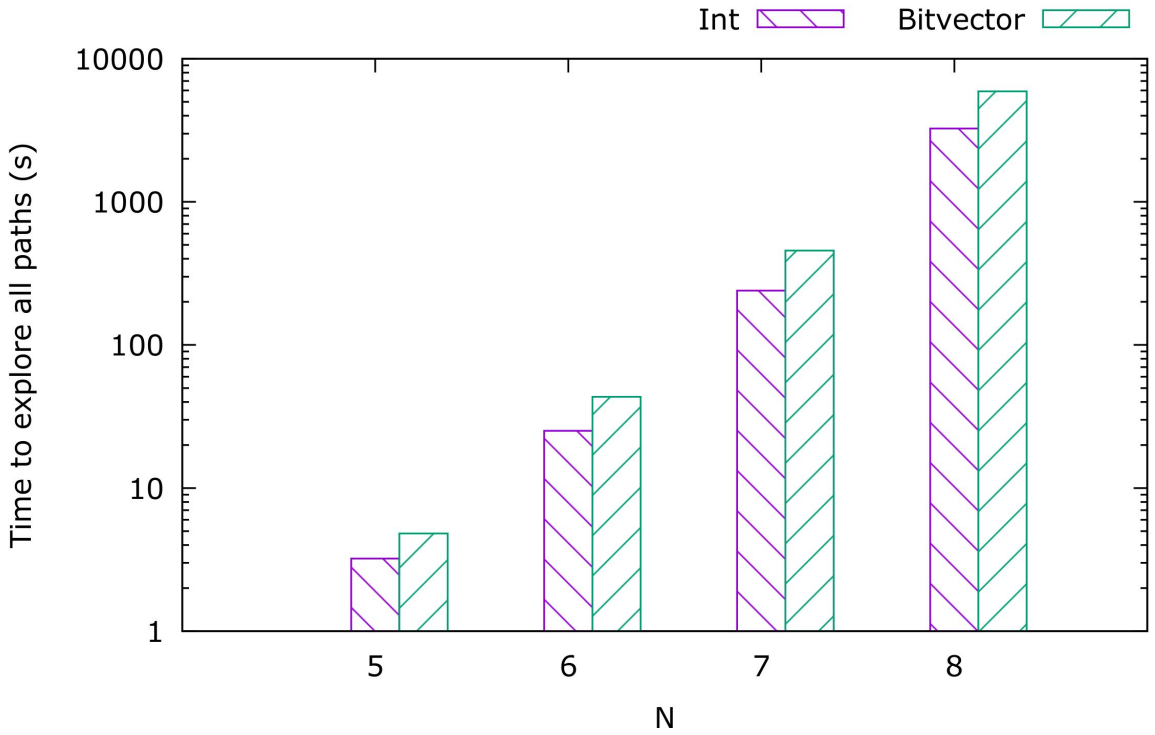
```
int buf1[N] = sym();
int buf2[N] = sym();
for (int i = 0; i < N; i++)
  assume(buf1[i] == buf2[i]);

insertion_sort(buf1, N);
bubble_sort(buf2, N);

for (int i = 0; i < N; i++)
  assert(buf1[i] == buf2[i]);
```

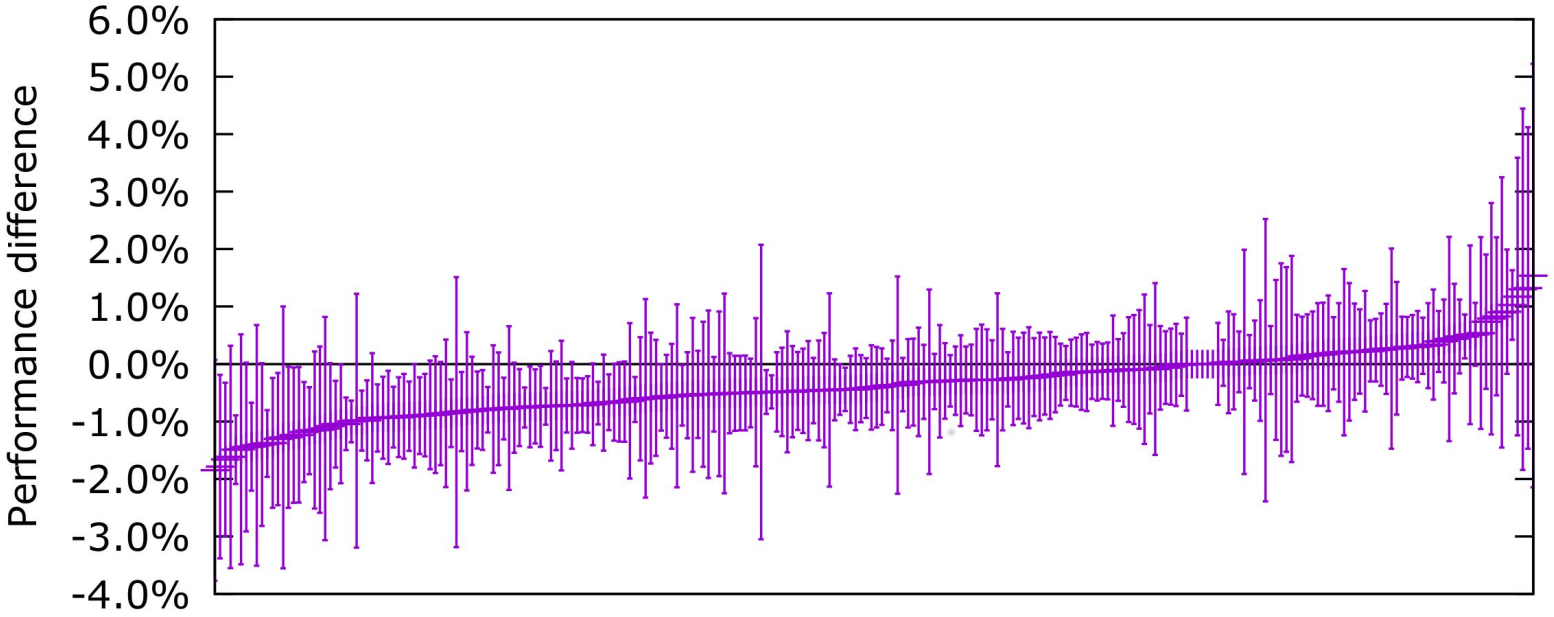
Sort benchmark

49% - 90% faster



Test-Comp: ECA-set

- Mostly equality queries
- Also no loss of precision
- Average speedup: 0.37%

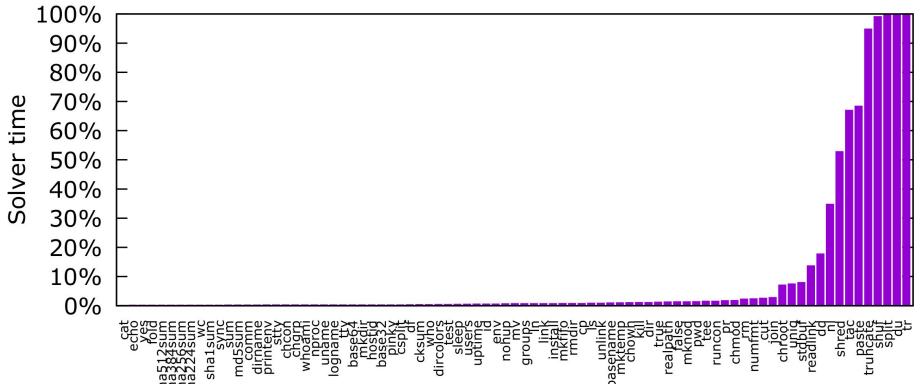


Coreutils

- 105 small/medium-sized UNIX utilities
- De-facto benchmark for KLEE
- 4 research questions

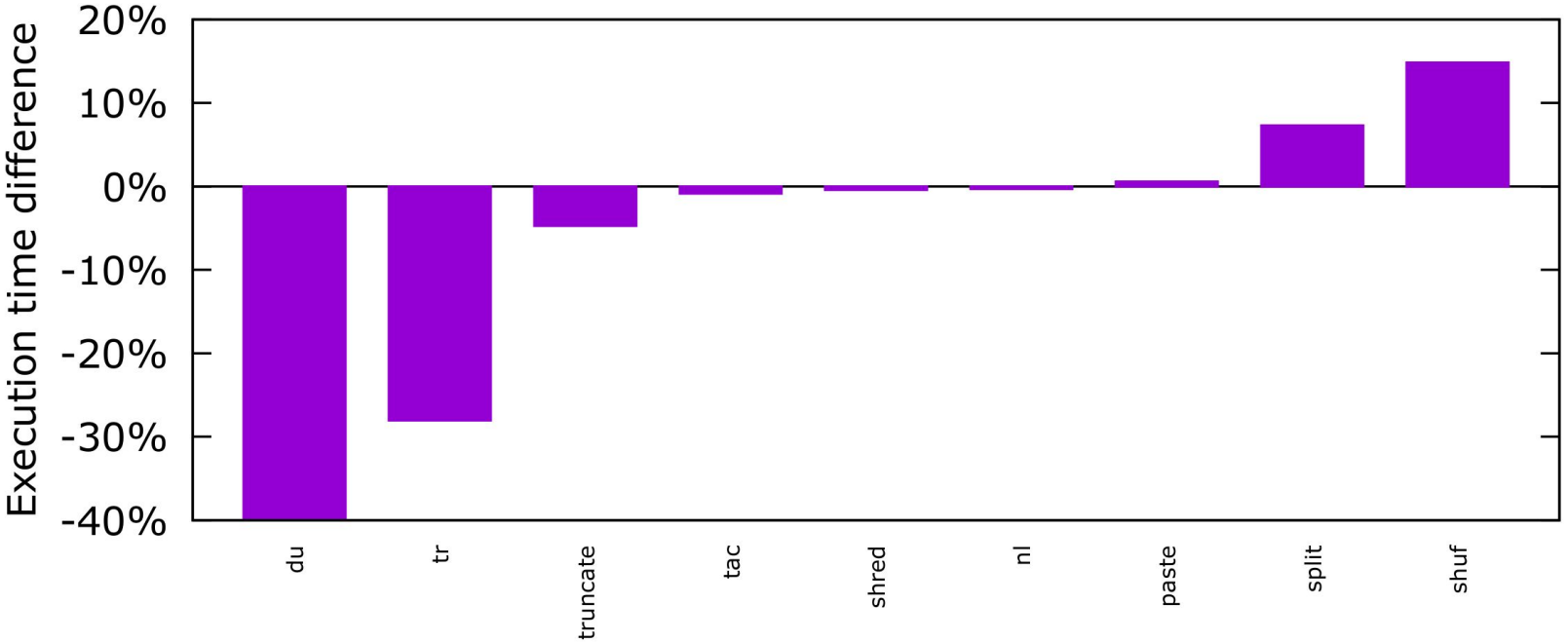
Coreutils: performance experiments

- Deterministic runs: DFS
- 1 hour
- Only Coreutils with more than 30% solver time
  - 9 benchmarks



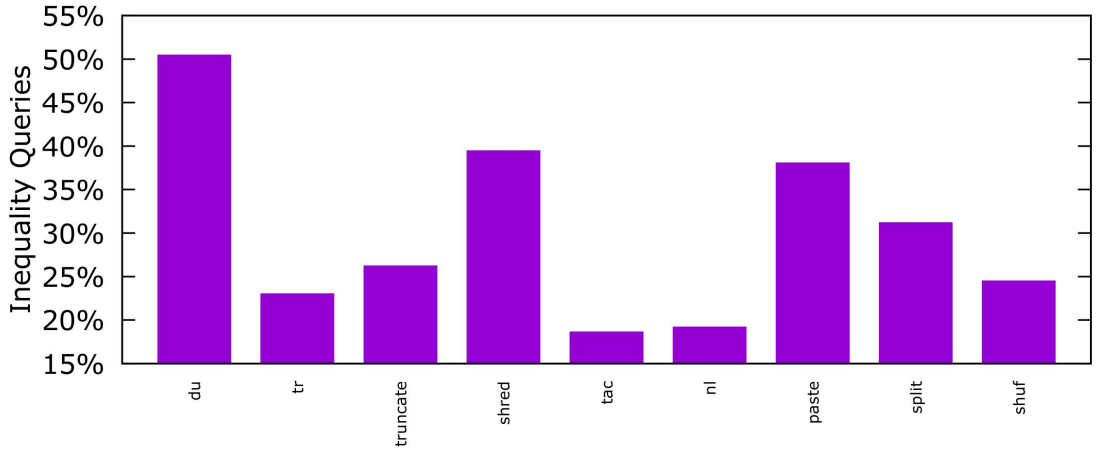
RQ1: What is the performance impact of our partial integer solver on the symbolic execution of Coreutils?

RQ1: Performance impact of integer solver

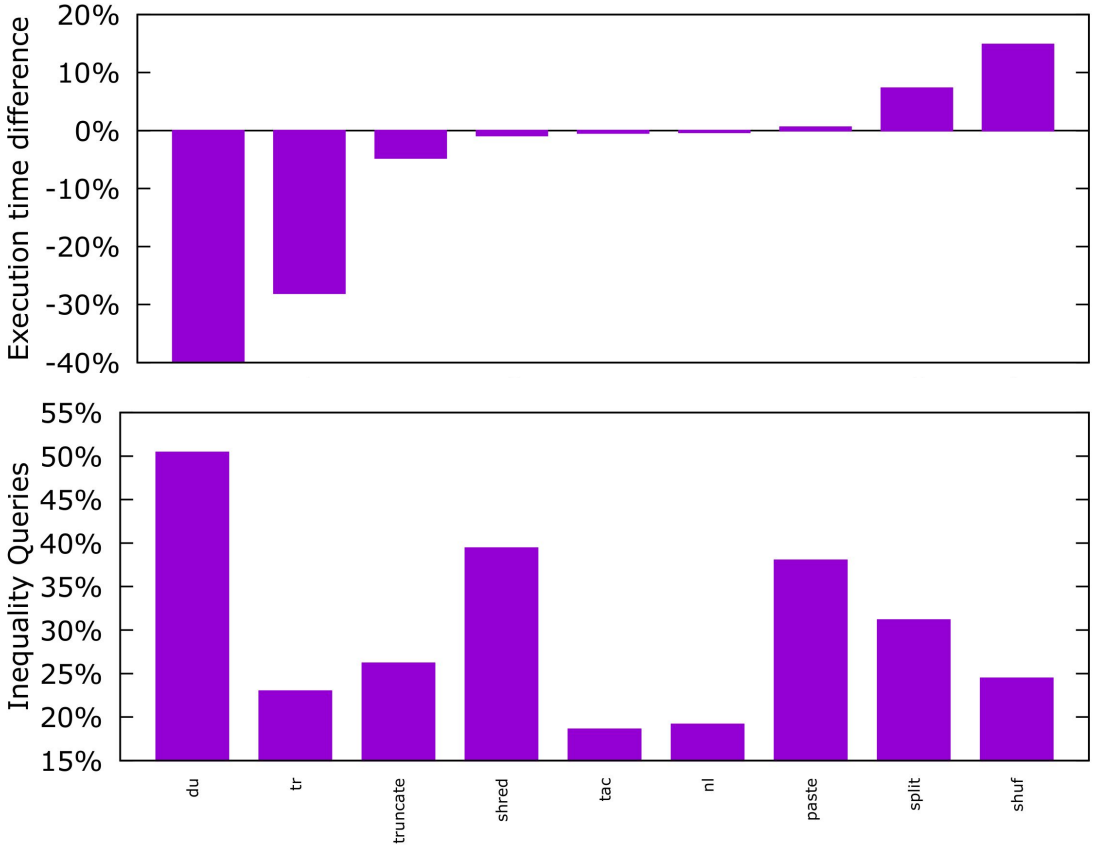


RQ2: Does the proportion of inequality queries explain the performance differences?

RQ2: Proportion of inequality queries



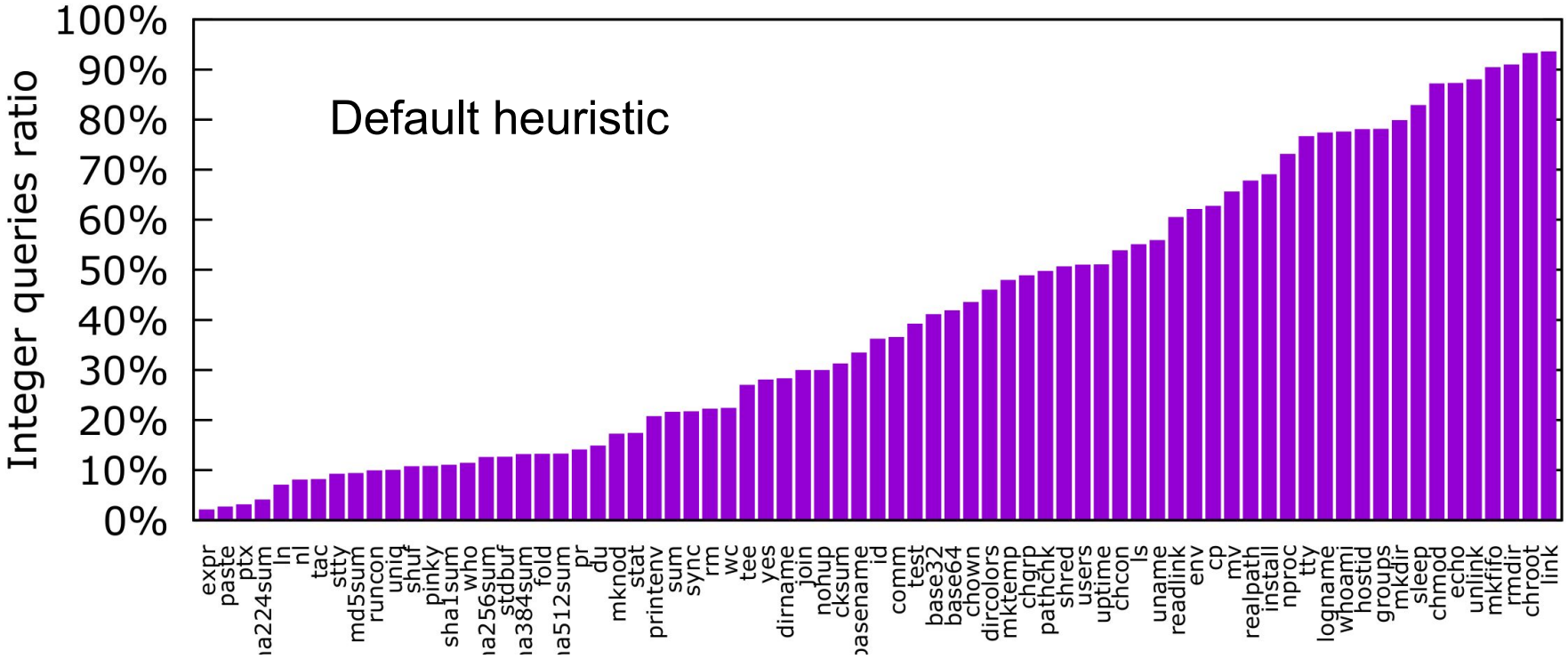
RQ2: Proportion of inequality queries



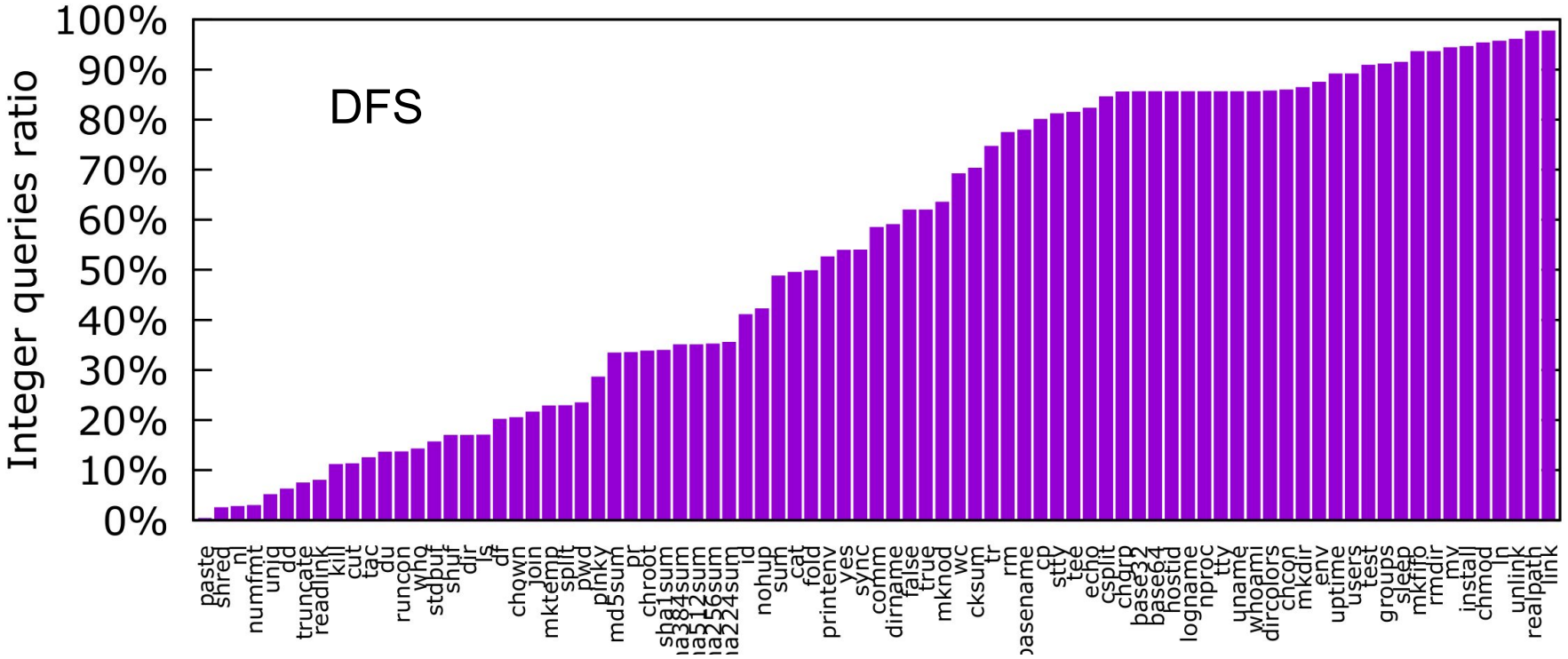
RQ3: What portion of the queries can the partial integer solver handle and what are the reasons for delegating queries to the bitvector solver?

- Explorations runs
  - DFS
  - Default: random path + coverage
- ALL Coreutils

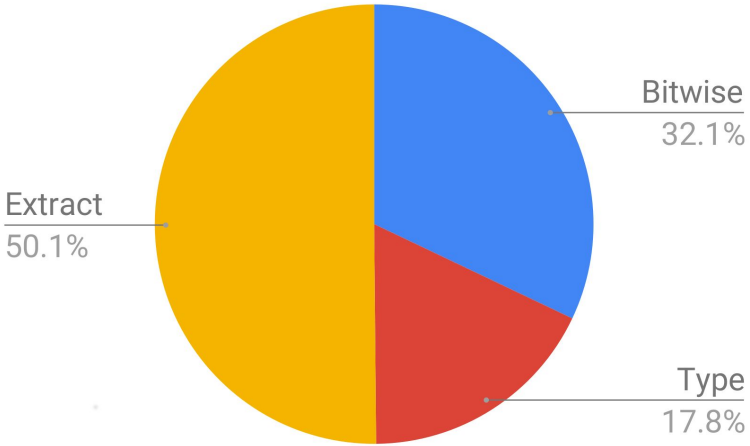
RQ3a: What portion of the queries can the partial integer solver handle?



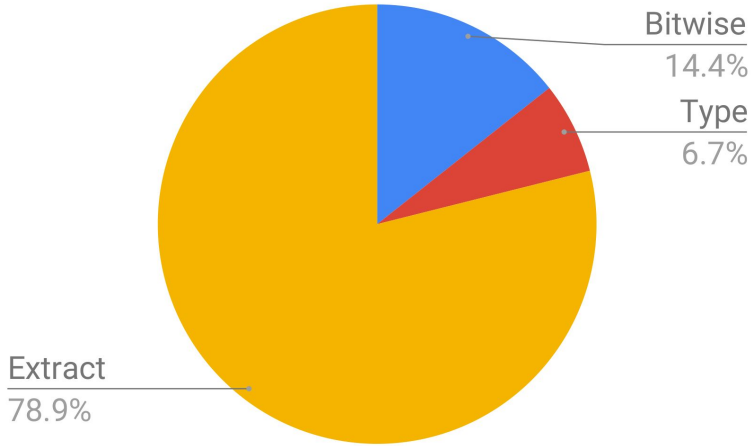
RQ3a: What portion of the queries can the partial integer solver handle?



RQ3b: Reasons for delegating queries to the bitvector solver?



DFS

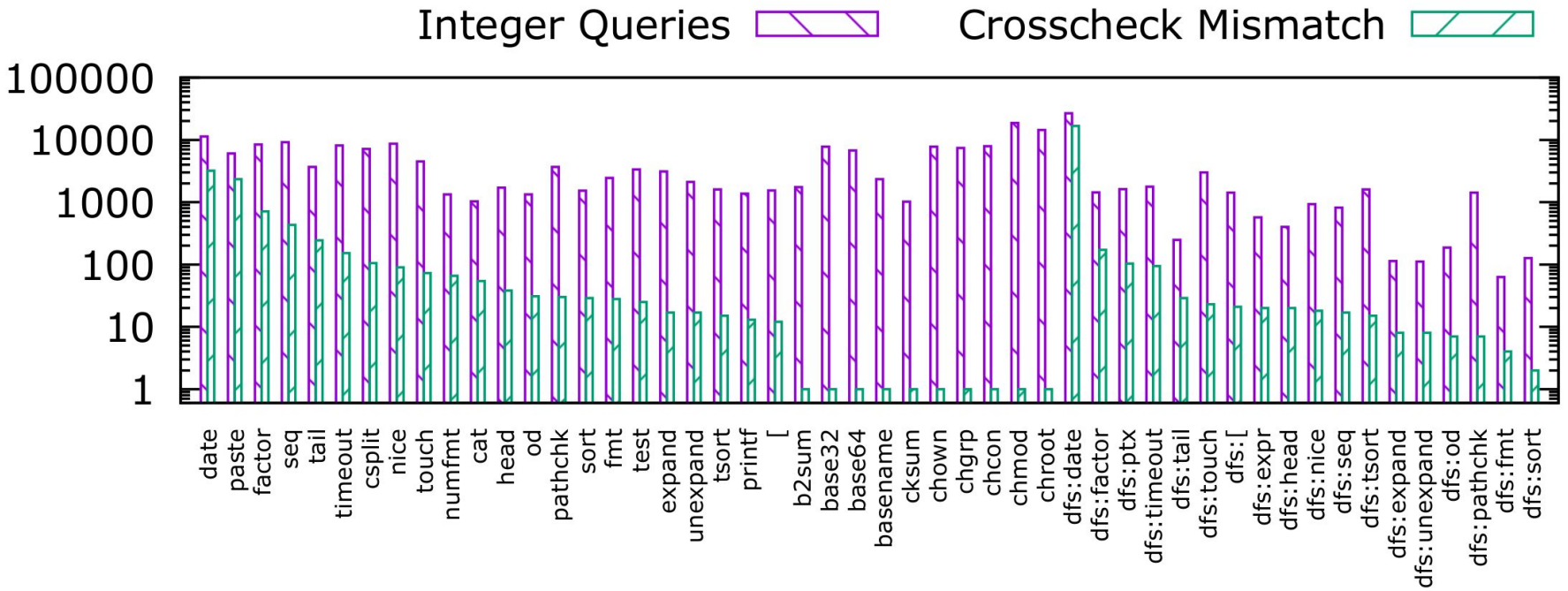


Default

RQ4: How many queries does the integer solver handle correctly with respect to the theory of bitvectors?

- Explorations runs
  - DFS
  - Random path + coverage
- All Coreutils
- Crosscheck integer solver with a bitvector one
  - Ignore integer solver

RQ4: Queries integer solver handles correctly



Mismatches only in 34 programs!

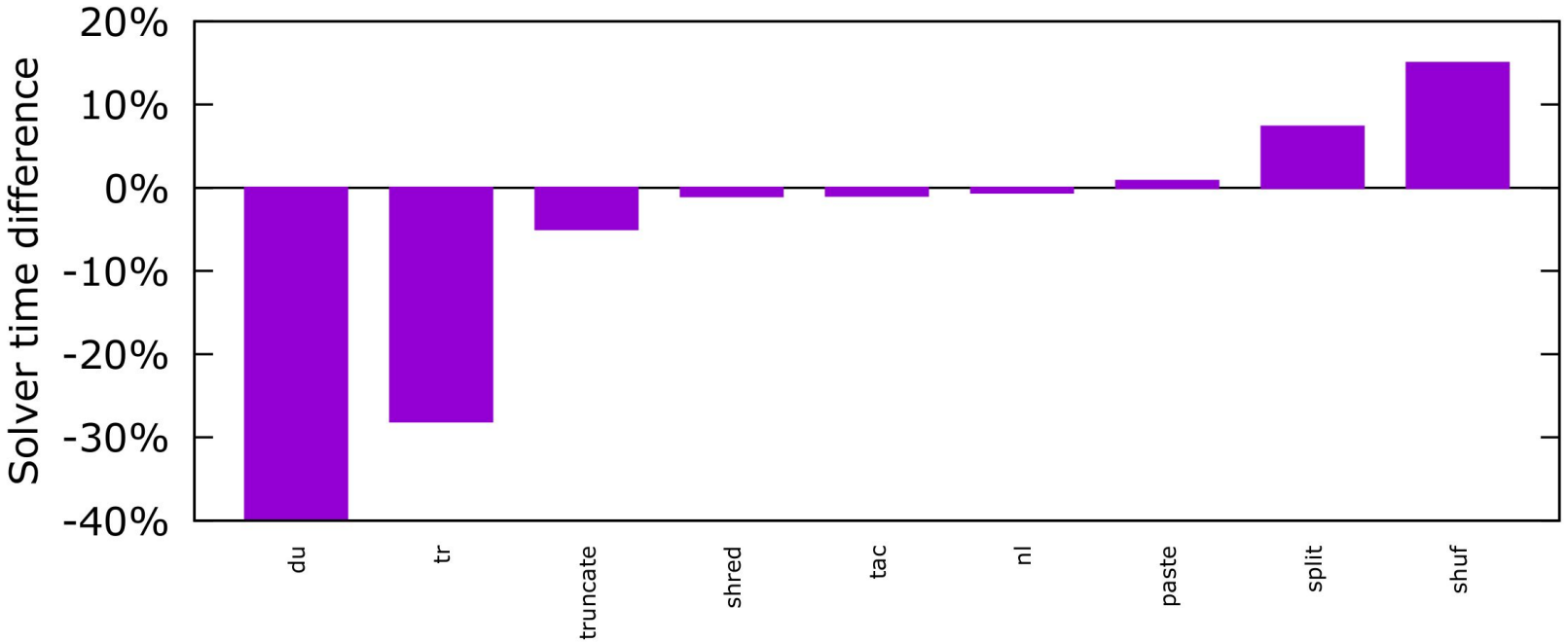
Threats to Validity

- Design decisions
  - Operation we encode in theory of integers
- KLEE’s dependencies
  - Same approach to Z3 tactics as KLEE
- Z3 based
  - Other solvers might behave differently
- Benchmark selection

Conclusion

- Evaluated integer constraint solving for symbolic execution in real programs
- Applicable to many queries
- No loss of precision in majority of cases
- Can have a performance benefit in some cases

Solver Time


Solver time difference
20%
10%
0%
-10%
-20%
-30%
-40%
du
tr
truncate
shred
tac
nl
paste
split
shuf

Query time proportion