Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?

Timotej Kapus, Martin Nowack, and Cristian Cadar

Imperial College London, UK
{t.kapus,m.nowack,c.cadar}@imperial.ac.uk

Abstract. Dynamic symbolic execution is a technique that analyses programs by gathering mathematical constraints along execution paths. To achieve bit-level precision, one must use the theory of bitvectors. However, other theories might achieve higher performance, justifying in some cases the possible loss of precision.
In this paper, we explore the impact of using the theory of integers on the precision and performance of dynamic symbolic execution of C programs. In particular, we compare an implementation of the symbolic executor KLEE using a partial solver based on the theory of integers, with a standard implementation of KLEE using a solver based on the theory of bitvectors, both employing the popular SMT solver Z3. To our surprise, our evaluation on a synthetic sort benchmark, the ECA set of Test-Comp 2019 benchmarks, and GNU Coreutils revealed that for most applications the integer solver did not lead to any loss of precision, but the overall performance difference was rarely significant.

1 Introduction

Dynamic symbolic execution is a popular program analysis technique that aims to systematically explore all the paths in a program. It has been very successful in bug finding and test case generation [3, 4]. The research community and industry have produced many tools performing symbolic execution, such as CREST [5], FuzzBALL [9], KLEE [2], PEX [14], and SAGE [6], among others.

To illustrate how dynamic symbolic execution works, consider the program shown in Figure 1a. Symbolic execution runs the program similarly to native execution, but it introduces the notion of symbolic variables. These denote variables that are initially allowed to take any value. In our example, variable x is marked as symbolic. As we start execution, we immediately hit the branch `if (x < 3)`. Since x is symbolic, we cannot directly evaluate the branch condition but have to treat it as a symbolic expression in terms of x. At this point, we can ask an SMT solver if $x < 3$ can be *true* under the current path constraints. Since we do not have any path constraints yet, $x < 3$ can be *true*, e.g. by setting it to 1. Therefore, we explore the then branch, add $x < 3$ to our path constraints, and continue execution along that path. In our example, we simply `return 1`. To handle the else branch, we similarly consider the path where $x \geq 3$.

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

(a) Simple program

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

(b) BV constraints

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

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

(c) Int constraints

Fig. 1. Constraints KLEE gathers when symbolically executing a simple function.

To accurately model the semantics of machine integers that have a finite number of bits, one has to use the theory of bitvectors, and tools such as Angr [13] and KLEE [2] do so. However, using the theory of mathematical integers, as tools like CREST [5] and CUTE [12] do, has the potential of improving performance, justifying in some cases the possible loss of precision. This performance/precision trade-off between the bitvector and integer theories has been documented before. For instance, He and Rakamarić discuss that *"On one end of the spectrum, verifiers opt for exclusively bit-precise reasoning, which often becomes a performance and/or scalability bottleneck. On the other end, verifiers opt for imprecise modelling using integers and uninterpreted functions, which often leads to a large number of false alarms."* [7]. However, while this trade-off has been studied in the context of static verification, we are not aware of any similar study for dynamic symbolic execution.

In this paper, we examine how gathering and solving constraints in the two theories impacts symbolic execution. We build a partial integer solver on top of KLEE [2] and Z3 [10] that aims to determine which queries can be easily expressed in the theory of integers and solve them with an integer solver, while delegating all others to a bitvector solver. In particular, we delegate to a bitvector solver queries involving bitwise operations and casts, which are not easily modelled in the theory of integers.[1]

We compare a version of KLEE using this partial integer solver to a standard version of KLEE using a bitvector solver on three different sets of benchmarks: a synthetic sort benchmark involving mostly inequalities, the ECA benchmark suite from Test-Comp involving mostly equalities, and the Coreutils application suite of 105 UNIX utilities, focusing on whether the two solvers produce identical results and how performance differs between the two configurations.

Surprisingly, our results show that the integer solver disagrees with the bitvector solver for only 32 Coreutils applications, the integer solver can be up to 40% faster and 16% slower, but the overall performance differences are not significant.

[1] Symbolic execution tools that use integer solvers typically do not handle such operations, e.g. CREST reverts to the concrete case when encountering a bitwise operation.

2 Partial Integer Solver

We chose to implement our partial integer solver in KLEE [2], a popular symbolic execution engine for LLVM [8] bitcode. KLEE gathers constraints in the theory of bitvectors and arrays. Therefore, we need to translate queries from this theory into the theory of integers and arrays.

While translating some constructs from bitvectors to integers is trivial—for example replacing bitvector addition with integer addition—there are two major challenges: translating arrays of bytes into mathematical integers and handling sign. Before we can show how we address these challenges, we need to describe precisely how KLEE gathers constraints.

Based on the symbolic execution of `lessThanThree` in Figure 1a, consider the bitvector query that KLEE generates in Figure 1b. KLEE represents each memory object as an array of bytes. So the variable `x` in our example is a two-byte memory object. This memory object is presented to the solver as `array x`. Arrays have a domain (type of index, `w32`) and range (type of value, `w8`). In KLEE, all arrays map 32-bit bitvector indices to 8-bit bitvector values.

The expression `x < 3` is then represented by the `SignedLessThan` construct, with a concatenation of two read expressions and the constant `3` as arguments. Because KLEE has a byte-level view of memory, it has to construct wider types such as `short` or `int` as a concatenation of byte-level reads.

2.1 Translating queries to the integer theory

To translate the query in Figure 1b to integers as shown in Figure 1c, we first change `array x` from a two-byte array to an array containing a single integer, which reflects the fact there is a single integer in the program. We could have opted to keep the byte-level view in integer constraints, but that would have led to a large amount of expensive non-linear constraints due to concatenate operations.

We infer the type of `array x` from LLVM's type information of the allocation site of the memory object that `array x` refers to. In our example, we would know `x` is an array of `short`, therefore we obtain a solver array with a single integer. We also constrain each element of the array to the numerical limits of the type (in our case `[-32768, 32767]`).

During execution, we collapse the concatenation of stridden read expression into a single read expression by pattern matching. This strategy is already employed by KLEE and its solver component Kleaver to print expressions in a more human-friendly form.

LLVM type information can be imprecise due to casting, which can occur for multiple reasons, such as viewing memory both as a struct and an array of chars (in a `memcpy`); or casting to the "correct" type only after allocation, which would be the case if the program has a special allocation function. Since we choose to view the memory as having only a single type, we can detect this case by comparing the width of the (collapsed) read with the type of the array. If there is a mismatch, we delegate the query to the bitvector solver.

2.2 Sign of variables

In the theory of bitvectors, variables do not have a sign. The sign is only inferred through operations on variables (i.e. `SignedLessThan` vs `UnsignedLessThan`). When translating to integers, we need to know the sign of variables to infer their ranges.

We first considered inferring the sign from the operations performed on the variable. While this can be done in some cases for a single expression, it is hard to keep it consistent across multiple expressions. Since KLEE queries the solver with a set of expressions representing the current path constraints, the variables common across expressions need to have a consistent sign. Unfortunately, inferring the sign from the operations performed can lead to contradictions across expressions.

Therefore, we had to decide whether to keep all variables signed or unsigned. Since mistakenly treating signed variables as unsigned can lead to more problems in practice (e.g. it causes `-1` to be greater than `0`), we decided to treat all variables as signed.

2.3 Operations delegated to the bitvector solver

As first discussed in Section 1, operations that cannot be easily translated to the theory of integers are delegated to the bitvector solver. We discuss each operation type below:

1. *Bitwise and, not, or, xor, shift.* Encoding them in the theory of integers would produce a large amount of non-linear constraints. We still keep any boolean versions of these operations.
2. *Extract expressions* arise due to casting (i.e. from `int` to `short`). Encoding them in the theory of integers would involve a lot of expensive division and module operations.
3. *Type mismatches*, which were described in Section 2.1.

2.4 Extending KLEE's solver chain

One of KLEE's strengths is its extensibility. For example, a set of solvers form a chain that is typically ordered from the most lightweight to the most expensive solver [2, 11]. Each solver chain element tries to solve a query. If this is unsuccessful, the next chain element is queried. While the first elements in the chain contain simple optimisation and independent-set calculations, further elements cache queries and their solutions. The final element is the actual solver, e.g. Z3.

We added a new chain element before the final solver, which consists of an expression analyser and our integer-based solver. The analyser, as discussed in Section 2.3, decides if the integer solver should handle the query. If so, the query is delegated to the Z3-based integer solver, which can use solving tactics specific to the integer theory. If not, the query is forwarded to the bitvector solver.

3 Evaluation

We evaluated our partial solver on several benchmarks: a synthetic sort benchmark, which produces inequality constraints (§3.1); the ECA set of benchmarks from the Test-Comp 2019 competition, which produces equality constraints (§3.2), and GNU Coreutils, a set of UNIX utilities, which produces a wide mix of constraints (§3.3).

Our implementation[2] was based on top of KLEE commit `44325801`, built with LLVM 3.8. For the ECA set of benchmarks, we also rebased the commits with which KLEE participated in Test-Comp 2019 to be reasonably competitive on that benchmark suite.[3]

In all our experiments, we used Z3 version 4.8.4 as the backend for both the integer and bitvector theories, as we felt it makes more sense to compare within a single solver than across multiple solvers.

3.1 Synthetic sort benchmark

Our synthetic sort benchmark crosschecks an implementation of insertion sort with one of bubble sort on a small array of N symbolic integers, as shown in Figure 2a. This benchmark generates mostly inequalities, making it a favourable benchmark for solving in the theory of integers.

Figure 2b shows the results of running the simple sort benchmark as we increase N up to 8 on a machine with Intel i7-6700 @ 3.60GHz CPU and 16GB RAM running Ubuntu 16.04. For each N we run KLEE twice, once with the default bitvector solver and once with our partial integer solver. We then plot the time it takes KLEE to fully explore all the paths in each case. We omit the runs for N between 1 and 4 as they terminate in under a second for both solvers.

Over the four array sizes shown, the integer solver makes KLEE explore all the paths between 49% and 90% faster, with a median of 77%. The integer solver causes no loss of precision in these experiments.

3.2 ECA set

The ECA set, part of the SV-COMP/Test-Comp benchmark suite,[4] consists of benchmarks that represent event condition action systems. We picked the ECA set because it contains fairly large programs (several thousand lines of code) that mostly consist of simple int assignments and conjunctions of equalities. Therefore, the ECA set illustrates the performance of the integer solver on a distinct set of constraints when compared to the synthetic sort benchmark. We used the same hardware as in Section 3.1 for this experiment.

We performed a preliminary run of the whole ECA set of 412 benchmarks and excluded those for which KLEE times out with both solvers. That left us

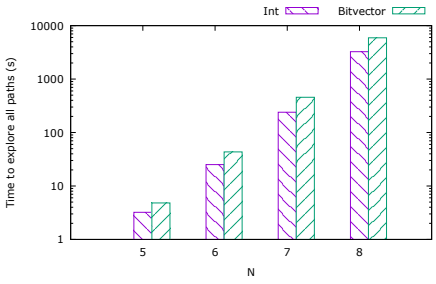
[2] Available at https://github.com/kren1/klee/tree/int_constraints
[3] Available at https://github.com/kren1/klee/commits/int_testcomp
[4] https://github.com/sosy-lab/sv-benchmarks

```
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]);
```

(a) Pseudo code for the sort benchmark: two identical arrays of size N are sorted via insertion and bubble sort respectively and their results are crosschecked.



(b) Runtime of KLEE to fully explore the sort benchmark with integer and bitvector solvers for different number of elements (log scale).

Fig. 2. Synthetic sort benchmark and its KLEE runtime.

with 247 benchmarks. We ran each one 14 times with both the integer and the bitvector solvers. The relative performance difference between the two solvers, with 95% confidence intervals included, is shown in Figure 3. The benchmarks are sorted by the performance difference between the two solvers. There are more programs that are faster with the integer solver, but the average speedup is only 0.37%, which is quite negligible (however, using a t-test gives us a p-value of 0.00, showing that the difference is statistically significant). The integer solver causes no loss of precision in these experiments either.

3.3 GNU Coreutils

Our main evaluation was performed on GNU Coreutils,[5] a popular application suite for evaluating symbolic execution-based techniques. We used version 8.29, which consists of 105 utilities. In our experiments, we exclude those runs for which KLEE did not terminate successfully (e.g., due to resource limitations or imprecision of the integer solver).

We pose four research questions. We first analyse the performance impact of our partial integer solver (RQ1) and if it can be explained by the proportion of inequality queries a benchmark produces (RQ2). We then evaluate what proportion of queries our partial solver handles (RQ3) and look into the imprecision of our solver (RQ4).

We ran all experiments in a Docker container, on a cluster of 14 identical machines with Intel i7-4790 @ 3.60GHz CPU with 16GB of RAM running Ubuntu 18.04. We used the same flags for all Coreutils, as discussed on the KLEE website;[6] in particular each utility was run for one hour.

[5] https://www.gnu.org/software/coreutils/
[6] http://klee.github.io/docs/coreutils-experiments/

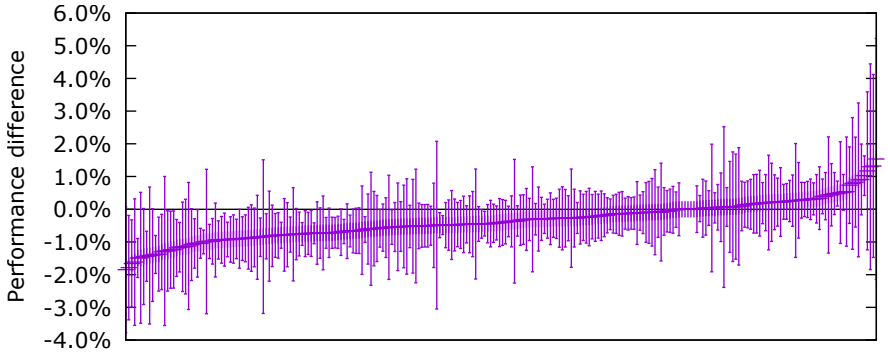


Fig. 3. Speedups (negative values) and slowdowns (positive values) recorded when using the integer solver instead of the bitvector solver on the ECA benchmarks (with 95% confidence intervals).

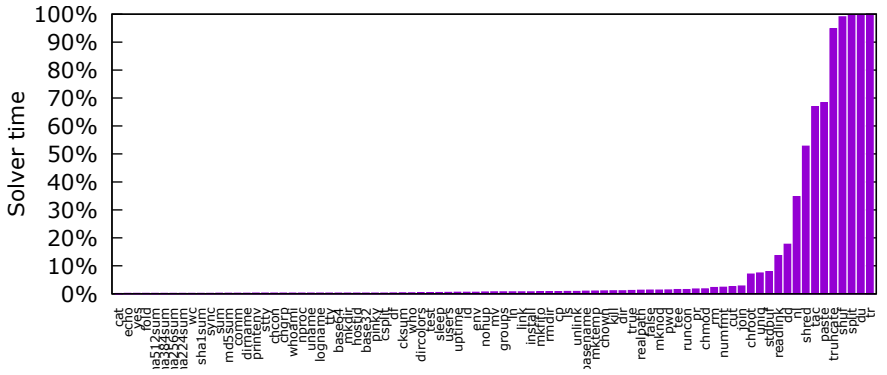


Fig. 4. Percentage of execution time spent in the bitvector solver for Coreutils.

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

To answer this question, we first run KLEE with the bitvector solver for one hour, recording the number of instructions executed. We then rerun KLEE with the integer solver for the same number of instructions and record the time taken. To ensure deterministic execution, we configure KLEE to use the DFS heuristic.

Figure 4 shows the time spent by each benchmark in the solver, when using the bitvector theory. These numbers indicate the extent to which optimising the solver can affect the overall runtime of KLEE. As can be seen, most benchmarks are not solver-bound in these experiments, but in several cases, the solver time dominates the execution time (note however that KLEE performs several other constraint-solving activities such as caching, but here we are only interested in the time spent in the SMT solver).

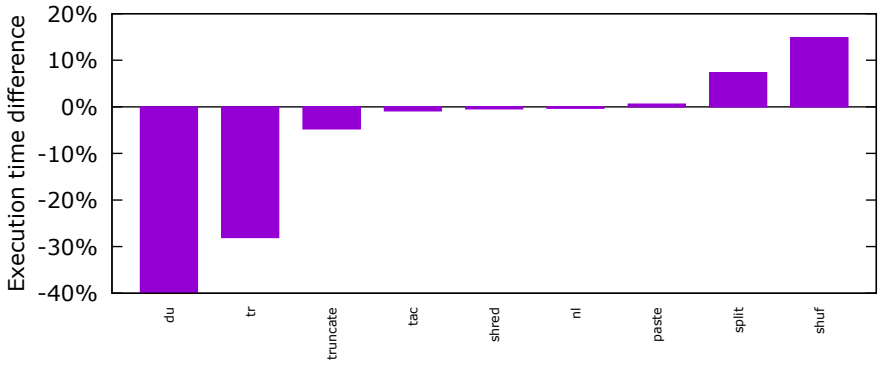


Fig. 5. Speedups (negative values) and slowdowns (positive values) for the whole execution when using the integer solver instead of the bitvector solver for the Coreutils benchmarks.

Based on the results, we select for investigation the applications that spend significant time in the solver—we choose 30% as our lower bound. Figure 5 shows for each utility the speedup or slowdown for the whole execution when the partial integer solver is used instead of the bitvector one for these applications. There are notable improvements in `du` and `tr` utilities, which see a 39.7% and 28.0% speedup respectively when using the integer solver. On the other end of the spectrum, `shuf` and `split` experience slowdowns of 14.8% and 7.3% respectively.

Figure 6 shows the speedups or slowdowns achieved for the solver time only. In broad strokes, the results match those in the previous figure, which makes sense, given that the utilities experiencing larger speedups or slowdowns spend most of their time in the solver.

From these results, we can conclude that there can be significant performance differences when symbolically executing GNU Coreutils in DFS mode with an integer vs a bitvector solver, but for most applications the impact is negligible.

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

The results from Section 3.1 indicate that the integer solver seems to perform significantly better when solving inequality ($<$, $>$,$\leq$,$\geq$) queries.

To explore this, we measured the proportion of inequality constructs in the queries. For this, we inspected the type of each node of every expression in the queries solved by the integer solver. To answer our research question, we computed the ratio of the number of inequalities over the number of all other constructs.

The results are shown in Figure 7. As one can see, there is no correlation between the proportion of inequalities shown in this figure and the performance numbers of Figure 6. For example, a significant fraction of inequalities are generated by `shred` and `paste`, but their performance changes are minor. In contrast,

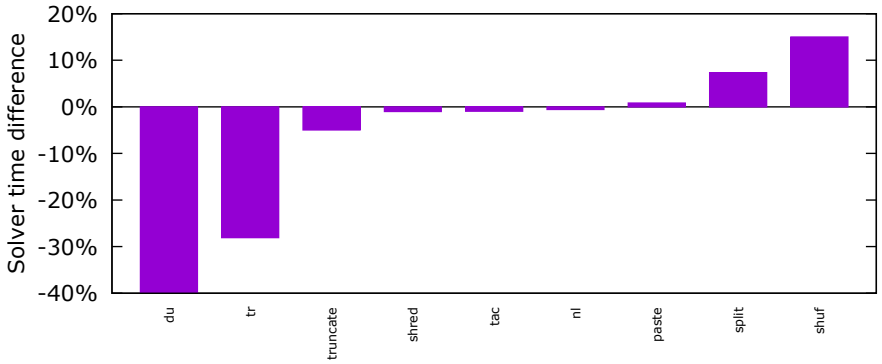


Fig. 6. Speedups (negative values) and slowdowns (positive values) for the solver time only when using the integer solver instead of the bitvector solver for the Coreutils benchmarks.

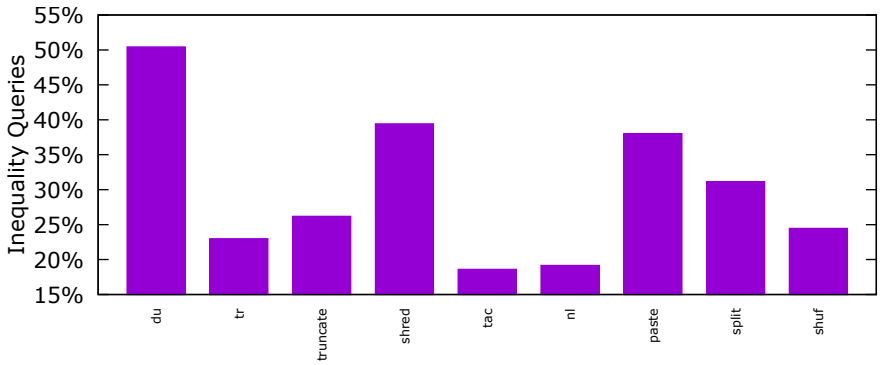


Fig. 7. Proportion of inequalities for the Coreutils experiments.

`du` and `shuf` also have a significant fraction of inequalities, and their performance changes are also significant—but `du` experiences a speedup, while `shuf` experiences a slowdown.

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

For this research question and RQ4, we also configure KLEE with its default search heuristic. This is because to answer these research questions, we do not require deterministic execution, and KLEE's default heuristic matches better how KLEE is used in practice. We run all utilities for one hour each again.

To answer RQ3, we collect the number of queries handled by our partial integer solver and the number of queries delegated to the bitvector solver. Figure 8 shows the proportion of queries that are solved by the integer solver. The

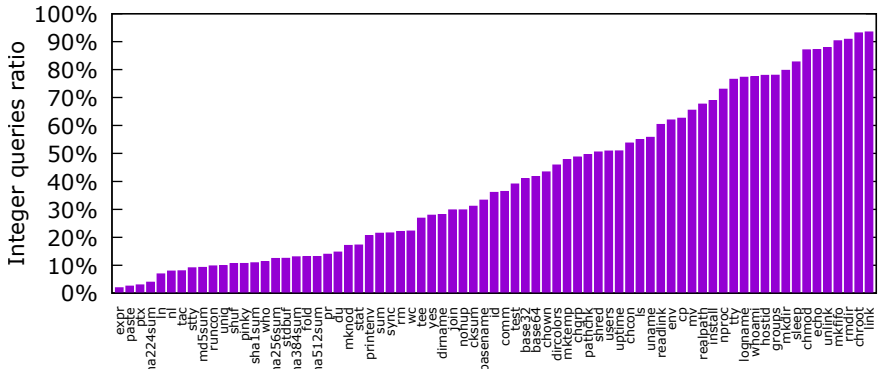
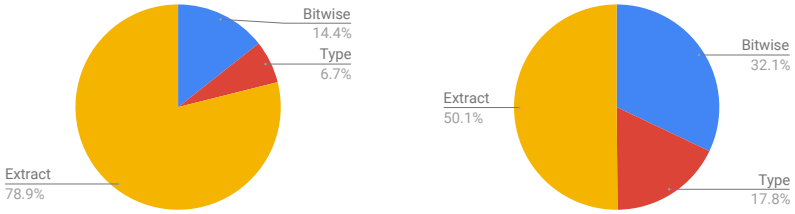


Fig. 8. Proportion of queries solved by the integer solver with default search heuristic.



(a) Default search heuristic (b) DFS search heuristic

Fig. 9. Reasons for delegation to the bitvector solver in Coreutils, for the DFS and default heuristics.

proportion varies significantly, from only 2% for `expr` to 93% for `link`, with the average across all Coreutils of 37% queries solved by the integer solver. Therefore, a significant proportion of queries for Coreutils can indeed be solved in the theory of integers.

For each query that is delegated to the bitvector solver, we record the first reason for which it is delegated. For example, if a query has both an Extract operation and a bitwise And, but the Extract is closer to the root of the parse tree, we would only report the reason for delegation as Extract. Figure 9a shows the results averaged across Coreutils. The vast majority of 79% is due to Extract operations. 14% of queries are delegated due to bitwise operations and only 7% due to type mismatches. Shift operations form an insignificant proportion of delegation reasons, only occurring in three Coreutils.

For comparison, we also report the proportion of integer queries and the reasons for delegation when using the DFS searcher in Figures 10 and 9b respectively. In this setup, there seem to be fewer queries eligible for solving with our integer solver. In terms of reasons for delegation, Extract is still the dominant reason, however less so than with the default search heuristic.

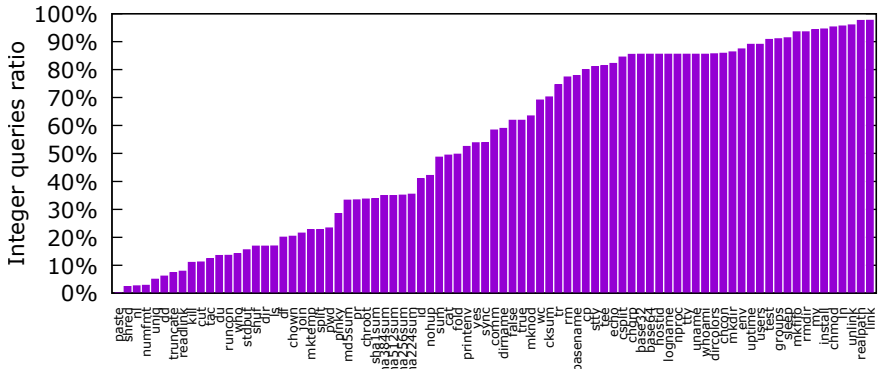


Fig. 10. Proportion of queries solved by the integer solver with DFS heuristic.

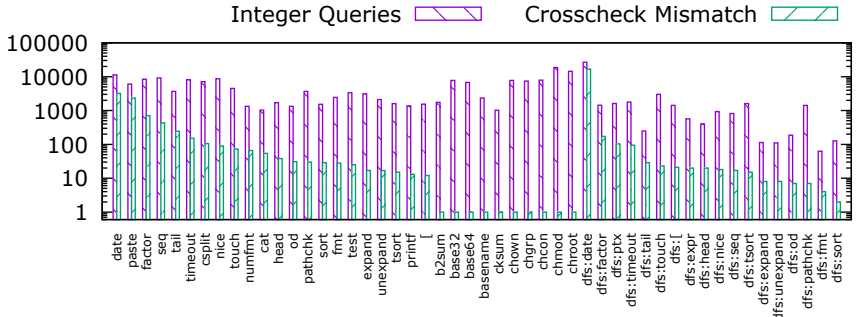


Fig. 11. Number of integer queries and their mismatches for the Coreutils experiments (log scale). Utilities with no mismatches are omitted.

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

We run another full set of Coreutils experiments for one hour per utility, with the integer solver being crosschecked by the bitvector solver. If they disagree, we record a mismatch and use the result of the bitvector solver to continue. We run this experiment with both the default heuristic and the DFS heuristic.

To our surprise, only 32 Coreutils have any mismatches with the default search heuristic, and only 18 with DFS (for a total of 34 distinct utilities). Figure 11 shows the number of mismatches for these programs; some programs appear twice if they had mismatches in runs with both the default search heuristic and DFS. We also show the number of queries solved by the integer solver in those programs for context. Since the number of mismatches is vastly different, we use a log scale. For the 32 programs where mismatches are present with the default search heuristic, they are relatively scarce. `paste` and `date` are notable

exceptions where almost one third of the queries solved by the integer solver are wrong. However, we note that even a single divergence can lead to the exploration of many infeasible paths if it occurs early in the execution. For DFS, the results look similar, with the one notable exception in `date`, where almost two thirds of the queries are not solved correctly in theory of integers.

Furthermore, note that in the experiments where we do not perform this crosschecking step, mismatches often manifest themselves as crashes in KLEE, because wrong solver results can break KLEE's internal invariants. As mentioned before, we exclude from the figures the experiments for which KLEE did not terminate successfully.

4 Threats to validity

Our preliminary study comparing the bitvector and integer theories in dynamic symbolic execution has a number of threats to validity. We highlight the most important ones below.

Design decisions. Our results may be influenced by several design decisions we made when developing our partial integer solver. For example, while we send all the queries involving bitwise operations to the bitvector solver, constraints involving division, modulo and multiplications are still sent to the integer solver (unless other operations prevent this). This means that we use more than the theory of integer linear arithmetic.

KLEE dependencies. Some design decisions are inherited from KLEE, and may also influence results. For example, KLEE does not explicitly choose a tactic to use and leaves Z3 to choose one itself. We use the same approach.

Z3 dependencies. Our study is limited to the Z3 solver, and results may differ for other solvers. We chose Z3 as it is one of the most popular SMT solvers implementing all of the bitvector, integer and array theories. Moreover, Z3 decides on theory and solving tactics for the query it receives automatically. Unfortunately, sometimes Z3 can choose a suboptimal tactic. However, the modular design of the solver chain (see §2.4) means that the integer expression analyser is independent of Z3 and could be reused with other solvers as well, e.g. with CVC4 [1].

Benchmark selection. As for all empirical studies, the conclusion of our study is dependent on our choice of benchmarks. We decided to study two types of synthetic benchmarks—one using mostly inequalities and the other mostly equalities—and GNU Coreutils, which is one of the most popular benchmarks used to evaluate symbolic execution tools.

We make our artefact available at https://srg.doc.ic.ac.uk/projects/klee-z3-int-vs-bv, in order to facilitate future exploration of alternative design decisions.

5 Conclusion

In this paper, we have investigated the impact of using the theory of integers instead of the theory of bitvectors on the precision and performance of dynamic symbolic execution. In particular, we have conducted experiments based on KLEE and Z3, applied to several benchmarks, including GNU Coreutils.

Our results show that an integer solver can be applied to a large proportion of queries and that many benchmarks are not affected by the imprecision of the integer solver. While the performance differences can be sometimes significant—with Coreutils applications experiencing slowdowns of up to 14.8% and speedups of up to 39.7%—in most cases they are negligible.

Acknowledgements We would like to thank Yannick Moy for challenging us at the Dagstuhl Seminar 19062 to pursue this direction of research, and Frank Busse and the anonymous reviewers for their valuable feedback. This research was generously sponsored by the UK EPSRC via grant EP/N007166/1 and a PhD studentship.

References

1. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11) (Jul 2011)
2. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08) (Dec 2008)
3. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C., Sen, K., Tillmann, N., Visser, W.: Symbolic Execution for Software Testing in Practice—Preliminary Assessment. In: Proc. of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact'11) (May 2011)
4. Cadar, C., Sen, K.: Symbolic Execution for Software Testing: Three Decades Later. Communications of the Association for Computing Machinery (CACM) **56**(2), 82–90 (2013)
5. CREST: Automatic Test Generation Tool for C. https://github.com/jburnim/crest
6. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08) (Feb 2008)
7. He, S., Rakamarić, Z.: Counterexample-guided bit-precision selection. In: Proc. of the 15th Asian Symposium on Programming Languages and Systems (APLAS'17) (Nov 2007)
8. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04) (Mar 2004)
9. Martignoni, L., McCamant, S., Poosankam, P., Song, D., Maniatis, P.: Path-exploration lifting: Hi-fi tests for lo-fi emulators. In: Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12) (Mar 2012)

10. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08) (Mar-Apr 2008)
11. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13) (Jul 2013), http://srg.doc.ic.ac.uk/files/papers/klee-multisolver-cav-13.pdf
12. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05) (Sep 2005)
13. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'16) (May 2016)
14. Tillmann, N., De Halleux, J.: Pex: white box test generation for .NET. In: Proc. of the 2nd International Conference on Tests and Proofs (TAP'08) (Apr 2008)