Multi-solver Support in Symbolic Execution

Hristina Palikareva and Cristian Cadar

Department of Computing, Imperial College London
London, United Kingdom
{h.palikareva, c.cadar}@imperial.ac.uk

Abstract. One of the main challenges of dynamic symbolic execution—an automated program analysis technique which has been successfully employed to test a variety of software—is constraint solving. A key decision in the design of a symbolic execution tool is the choice of a constraint solver. While different solvers have different strengths, for most queries, it is not possible to tell in advance which solver will perform better.
In this paper, we argue that symbolic execution tools can, and should, make use of multiple constraint solvers. These solvers can be run competitively in parallel, with the symbolic execution engine using the result from the best-performing solver.
We present empirical data obtained by running the symbolic execution engine `KLEE` on a set of real programs, and use it to highlight several important characteristics of the constraint solving queries generated during symbolic execution. In particular, we show the importance of constraint caching and counterexample values on the (relative) performance of `KLEE` configured to use different SMT solvers.
We have implemented multi-solver support in `KLEE`, using the `metaSMT` framework, and explored how different state-of-the-art solvers compare on a large set of constraint-solving queries. We also report on our ongoing experience building a parallel portfolio solver in `KLEE`.

1 Introduction

Symbolic execution [14] is a program analysis technique that can systematically explore paths through a program by reasoning about the feasibility of explored paths using a constraint solver. The technique has gathered significant attention in the last few years [6], being implemented in several tools, which have found deep bugs and security vulnerabilities in a variety of software applications [4].

One of the key factors responsible for the recent success of symbolic execution techniques are the recent advances in constraint-solving technology [10]. Nevertheless, constraint solving remains one of the main challenges of symbolic execution, and for many programs it is the main performance bottleneck. As a result, a key decision when designing a symbolic execution tool is the choice of a constraint solver. While this choice may be affected by the solver's supported theories, specific optimisations or software licenses, in many cases it is somewhat arbitrary, and it is not always clear which solver is the best match for a given symbolic execution tool. In fact, given two state-of-the-art solvers, it is unlikely

that one consistently outperforms the other; more likely, one solver is better on some benchmarks and worse on others. Moreover, for a given query, it is often not possible to tell in advance which solver would perform better.

In this paper, we argue that symbolic execution tools can—and should—make use of multiple constraint solvers. These solvers can be run competitively in parallel, with the symbolic execution engine using the result from the best-performing solver. We believe such an approach is particularly timely in the age of parallel hardware platforms, such as multi-core CPUs [5].

The idea of using a *portfolio* of solvers is not new: this technique was already employed in the context of SAT solving [13, 24], SMT solving [23], and bounded model checking [9], among others. However, as far as we know, this is the first paper that reports on how different SMT solvers compare in the context of symbolic execution.

The main contributions of this paper are:

1. A discussion of the main characteristics of the constraint-solving queries generated in symbolic execution, accompanied by detailed statistics obtained from real symbolic execution runs;
2. An analysis of the effect of constraint caching and counterexample values on the performance of symbolic execution;
3. A comparison of several state-of-the-art SMT solvers for closed quantifier-free formulas over the theory of bitvectors and bitvector arrays (`QF_ABV`) on queries obtained during the symbolic execution of real-world software;
4. An extension of the popular symbolic execution engine `KLEE` [2] that supports multiple SMT solvers, based on the `metaSMT` [12] solver framework.
5. A discussion of our ongoing experience building a portfolio solver in `KLEE`.

The rest of the paper is organised as follows. Section 2 provides background information on symbolic execution and the `KLEE` system. Section 3 presents the `metaSMT` framework and its integration with `KLEE`. Then, Section 4 analyses the constraint-solving queries obtained during the symbolic execution of several real applications, and discusses how different solvers perform on these queries. Finally, Section 5 discusses how symbolic execution could benefit from a parallel portfolio solver, Section 6 presents related work and Section 7 concludes.

2 Background

Dynamic symbolic execution is a program analysis technique whose goal is to systematically explore paths through a program, reasoning about the feasibility of each explored path using a Satisfiability Modulo Theory (SMT) constraint solver. In addition, symbolic execution systematically checks each explored path for the presence of generic errors such as buffer overflows and assertion violations.

At a high level, the program is executed on a *symbolic* input, which is initially unconstrained. For example, in the code in Figure 1, the symbolic input is the integer variable x, which is in the beginning allowed to take any value. Then, as the program executes, each statement that depends on the symbolic input adds

```
1   int main() {
2       unsigned a[5] = {0, 1, 1, 0, 0};
3       unsigned x = symbolic();
4       unsigned y = x+1;
5       if (y < 5) {
6           if (a[y])
7               printf("Yes\n");
8           else printf("No\n");
9       }
10      else printf("Out of range\n");
11      return 0;
12  }
```

Fig. 1. Code example illustrating some of the main aspects of symbolic execution.

further constraints on the input. For instance, the statement `y=x+1` on line 4 constrains y to be equal to $x+1$. When a branch that depends on the symbolic input is reached, if both branch directions are feasible, symbolic execution follows them both, constraining the branch condition to be *true* on the true path, and *false* on the other. In our example, when the program reaches the branch on line 5, execution is forked into two paths: on the `then` path, the constraint $y < 5$ is added and execution proceeds to line 6, while on the `else` path, the constraint $y \geq 5$ is added and execution proceeds to line 10. However, note that constraints are added not in terms of intermediate variables such as y, but in terms of the initial symbolic inputs. That is, in our example, the constraints being added on each path are $x+1 < 5$ and $x+1 \geq 5$. There is one case in which constraints cannot solely be expressed in terms of the original inputs. This happens when a concrete array is indexed by a symbolic variable. In our example, the concrete array a is indexed by the symbolic variable y on line 6. In order to reason about the symbolic access $a[y]$, the constraint solver needs to know all the values of the array a. With this knowledge, the solver can determine that the branch at line 6 can be both *true* (when x is 0 or 1) and *false* (when x is 2 or 3).

Finally, when a path ends or an error is discovered, one can take all the constraints gathered along that path and ask the constraint solver for a concrete solution. This solution, also called a *counterexample*, represents a test case that exercises the path. In the context of software testing, these test cases can be used to form high-coverage test suites, as well as to generate bug reports.

2.1 Constraint Solving in Symbolic Execution

In this section, we discuss some of the most important characteristics of the constraint-solving queries generated during symbolic execution:

Large number of queries. This is perhaps the most important characteristic of constraint solving in symbolic execution. Unlike other constraint-based program testing and verification techniques that generate a small number of queries,[1] on a typical run, symbolic execution generates queries at every sym-

[1] For instance, in bounded model checking [7] all paths up to a particular length are encoded as a single query.

bolic branch and every potentially-dangerous symbolic operation it encounters, amounting to a large number of overall queries. As a result, in order to efficiently explore the program space, these queries need to be solved quickly, at a rate of tens or even thousands of queries per second.

Concrete solutions. Symbolic execution often requires concrete solutions for satisfiable queries. These are needed to create test cases, interact with the outside world (e.g. before calling an external function, all symbolic bytes need to be replaced by concrete values), simplify constraints (e.g., double-pointer dereferences [3]), and reuse query results (e.g., KLEE's counterexample cache [2]).

Array operations. Arrays play an important role in symbolic execution. Many programs take as inputs arrays (in one form or another, e.g., strings are essentially arrays of characters), and concrete arrays often become part of the symbolic constraints when they are indexed by a symbolic input, as we have shown above. Furthermore, pointer operations in low-level code are also modelled using arrays. As a result, efficiently reasoning about arrays is extremely important in symbolic execution [3].

Bit-level accuracy. Many programs require bit-level accurate constraints, in order to reason about arithmetic overflow, bitwise operations, or integer and pointer casting. In particular, an important characteristic of the KLEE tool analysed in this paper is that it generates queries with bit-level accuracy.

2.2 Constraint Solving in KLEE

The experiments presented in this paper use KLEE [2], a modern symbolic execution engine available as open source from `http://klee.llvm.org`. KLEE works at the level of LLVM bitcode [15] and uses the constraint solver STP [11]. We now elaborate on some key issues related to constraint solving in KLEE.

The queries issued by KLEE are of two main types: *branch* and *counterexample queries*. Branch queries are issued when KLEE reaches a symbolic branch, to decide whether to follow only the *then*, only the *else*, or both sides of the branch. They are of the form (C, E) where C represents the set of constraints that hold on the current path (the *path constraints*), and E is a branch expression whose validity KLEE tries to establish. The possible answers are *provably true*, *provably false*, and *neither* (i.e., under the current set of constraints C, E could be both *true* and *false*). Branch queries are broken down into one or two satisfiability (or validity[2]) queries. For instance, to conclude that E is neither provably true nor provably false, KLEE needs to determine that both $\neg E$ and E are satisfiable.

Counterexample queries are used to request a solution for the current path constraints, e.g. when KLEE needs to generate a test case at the end of a program path. In addition, the counterexample cache in KLEE (described below) asks for a counterexample for all queries that are found to be satisfiable.

Before invoking STP, KLEE performs a series of constraint solving optimisations, which exploit the characteristics of the queries generated during symbolic

[2] In our context, a satisfiability query can be transformed into a validity query and vice-versa: a formula F is satisfiable iff $\neg F$ is not valid.

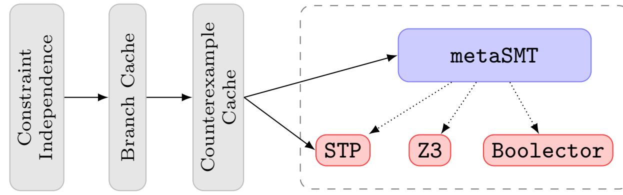

Fig. 2. Solver passes in KLEE, including the new metaSMT pass.

execution. These optimisations are structured in KLEE as a sequence of solver passes, which are depicted in Figure 2; most of these passes can be enabled and disabled via KLEE's command-line options.

One of the key solver passes in KLEE is the elimination of redundant constraints, which we call *constraint independence* [3]. Given a branch query (C, E), this pass eliminates from C all the constraints which are not (transitively) related to E. For example, given $C = \{x < 10, z < 20, x + y = 10, w = 2z, y > 5\}$ and $E = x > 3$, this pass eliminates from C the constraints $z < 20$ and $w = 2z$, which don't influence E.

The other key solver passes are concerned with caching. KLEE uses two different caches: a branch cache, and a counterexample cache. The branch cache simply remembers the result of branch queries. The counterexample cache [2] works at the level of satisfying assignments. Essentially, it maps constraint sets to either a counterexample if the constraint set is satisfiable, or to a special sentinel if it is unsatisfiable. Then, it uses subset and superset relations among constraint sets to determine the satisfiability of previously unseen queries. For example, if the cache stores the mapping $\{x > 3, y > 2, x + y = 10\} \to \{x = 4, y = 6\}$, then it can quickly determine that the subset $\{x > 3, x + y = 10\}$ of the initial constraint set is also satisfiable, because removing constraints from a set does not invalidate any solutions. Similarly, the counterexample cache can determine that if a constraint set is unsatisfiable, any of its supersets are unsatisfiable too.

An important optimisation of the counterexample cache is based on the observation that many constraint sets are in a subset/superset relation. For example, as we explore a particular path, we always add constraints to the current path constraints. Furthermore, we observed that many times, if a subset of a constraint set has a solution, then often this solution holds in the original set too [2]. That is, adding constraints often does not invalidate an existing solution, and checking whether this holds (by simply substituting the solution in the constraints) is usually significantly cheaper compared to invoking the constraint solver. For example, if the cache stores the mapping $\{x > 3, y > 2, x+y = 10\} \to \{x = 4, y = 6\}$, then it can quickly verify that the solution $\{x = 4, y = 6\}$ also satisfies the superset $\{x > 3, y > 2, x+y = 10, x < y\}$ of the original constraint set, thus saving one potentially-expensive solver call. Given a constraint set, the counterexample cache tries all of its stored subsets, until it finds a solution (if any exists).

An important observation is that the cache hit rate depends on the actual counterexamples stored in the cache. For example, if instead the cache stored the mapping $\{x > 3, y > 2, x + y = 10\} \rightarrow \{x = 7, y = 3\}$, it would not be able to prove that the superset $\{x > 3, y > 2, x + y = 10, x < y\}$ is also satisfiable, since the cached assignment $\{x = 7, y = 3\}$ is not a solution for the superset.

3 The metaSMT framework

One of the factors that has contributed to the recent progress in constraint-solving technology is the development of SMT-LIB [18], a common set of standards and benchmarks for SMT solvers. In particular, SMT-LIB defines a common language for the most popular SMT logics, including the fragment of *closed quantifier-free formulas over the theory of bitvectors and bitvector arrays* (`QF_ABV`), which is used by `KLEE`. Unfortunately, communicating via the textual constraint representation offered by the SMT-LIB format is not a feasible option in practice. The overhead of having the symbolic execution engine output SMT-LIB constraints and the solver parsing them back would be excessive. For example, we measured for a few benchmarks the average size of a single `KLEE` query in SMT-LIB format, and we found it to be on the order of hundreds of kilobytes, which would add significant overhead given the high query rate in symbolic execution (see §4).

As a result, it is critical to interact with solvers via their native APIs, and in our work we do so by using the `metaSMT` framework. `metaSMT` [12] provides a unified API for transparently using a number of SMT (and SAT) solvers, and offers full support for the `QF_ABV` logic used by `KLEE`. The unified C++ API provided by `metaSMT` is efficiently translated at compile time, through template meta-programming, into the native APIs provided by the SMT solvers. As a result, the overhead introduced by `metaSMT` is small, as we discuss in Section 4.

The solvers supported by `metaSMT` for the `QF_ABV` fragment are `Boolector` [1] and `Z3` [17]. `STP` only had support for `QF_BV`, and we extended it to fully handle the `QF_ABV` fragment. We contributed back our code to the `metaSMT` developers.

In `KLEE`, we added support for using the `metaSMT` API by implementing a new core solver pass, as depicted in Figure 2.

4 Experimental Evaluation

We evaluated our multi-solver `KLEE` extension on 12 applications from the `GNU Coreutils 6.10` application suite, which we used in prior work [2,16]. We selected only 12 out of the 89 applications in the `Coreutils` suite, in order to have time to run them in many different configurations (our experiments currently take days to execute). Our selection was unbiased: we first discarded all applications for which either (a) our version of `KLEE` ran into unsupported `LLVM` instructions or system calls,[3] (b) `KLEE` finished in less than one hour (e.g. `false`), or c) the

[3] Currently, `KLEE` does not fully support the `LLVM` 2.9 instruction set nor certain system calls in recent versions of Linux.

symbolic execution runs exhibited a significant amount of nondeterminism (e.g. for date, which depends on the current time, or for kill, where we observed very different instructions executed across runs). Out of the remaining applications, we selected the first 12 in alphabetical order.

We used LLVM 2.9 and metaSMT 3 in our experiments, and the SMT solvers Boolector v1.5.118, Z3 v4.1 and STP 32:1668M, for which we used the default configuration options as provided by metaSMT (see also the threats to validity in §4.3). We configured KLEE to use a per-query timeout of 30s and 2 GB of memory. We ran all of our experiments on two similar Dell PowerEdge R210 II machines with 3.50 GHz Intel Xeon quad-core processors and 16 GB of RAM.

4.1 Solver comparison using the DFS strategy and no caching

For the first set of experiments, we ran each benchmark for one hour using KLEE's default STP solver and recorded the number of executed LLVM instructions. In the following experiments, we ran each benchmark with KLEE for the previously recorded number of instructions, configured to use four different solver configurations: default STP, metaSMT with STP, with Boolector and with Z3.[4]

The main challenge is to configure KLEE to behave deterministically across runs; this is very difficult to accomplish, given that KLEE relies on timeouts, time-sensitive search heuristics, concrete memory addresses (e.g. values returned by malloc), and counterexample values from the constraint solver. To make KLEE behave as deterministically as possible, we used the depth-first search (DFS) strategy, turned off address-space layout randomisation, and implemented a deterministic memory allocator to be used by the program under testing. With this configuration, we have observed mostly deterministic runs with respect to the sequence of instructions executed and queries issued by KLEE—in particular, for all 12 benchmark considered, we observed that KLEE's behaviour is very similar across metaSMT runs: e.g. modulo timeouts, KLEE consistently executed the same number of instructions with different solvers, and the number of queries only rarely differed by a very small number. We believe that for these benchmarks, the effect of any remaining nondeterminism is small enough to allow for a meaningful comparison.

MetaSMT overhead. To measure the overhead introduced by metaSMT, we compared KLEE using directly the STP API with KLEE using STP via the metaSMT API. For 9 out of the 12 benchmarks, the overhead was small, at under 3%. For ln it was 6.7%. For chmod and csplit, the overhead was substantial (72.6% and 42.0% respectively), but we believe this is mainly due to the way STP expressions are exactly constructed via the two APIs.

Statistics for KLEE with STP. Table 1 presents some statistics about the runs with KLEE using its default solver STP, invoked via the metaSMT API.

[4] Note that re-running KLEE with STP for the same number of instructions does not always take one hour; this is due to the fact that on exit KLEE performs different activities than when run with a timeout. However, the goal of this initial run is only to obtain a fixed number of instructions for which to re-run each benchmark.

Table 1. Statistics for the DFS runs without caching, using STP via the metaSMT API: the instructions per second rate, the number of queries, the average query size, the queries per second rate in KLEE and STP respectively, and the percentage of time spent in all constraint-solving activities and STP respectively.

Application	**Instrs/sec**	**Queries**	**Q-size**	**Queries/sec**		**Solver(%)**	
				total	STP	total	STP
[	3,914	197,282	2,868	55.1	60.0	97.8	89.8
base64	18,840	254,645	546	73.8	76.6	97.0	93.4
chmod	12,060	202,855	7,125	36.4	40.2	97.2	87.9
comm	73,064	586,485	120	189.0	201.9	88.4	82.7
csplit	10,682	244,803	2,179	49.7	52.7	98.3	92.7
dircolors	8,090	175,531	1,588	49.3	50.5	98.6	96.4
echo	227	114,830	6,852	34.8	41.7	98.8	82.3
env	21,955	379,421	664	109.1	119.8	97.2	88.5
factor	1,897	19,055	2,213	5.3	5.3	99.7	99.4
join	12,649	131,947	1,391	36.6	37.2	98.1	96.3
ln	13,420	366,926	786	103.8	115.3	97.0	87.4
mkfifo	25,331	221,308	2,144	62.3	67.4	96.6	89.3

The number of LLVM instructions executed per second varied between 227 for echo and 73,064 for comm, with a median of 12,355 instructions per second. The number of queries issued by each benchmark to the solver ranges between 19,055 for factor and 586,485 for comm, with a median of 212,082. The Queries/sec column shows two query rates for each benchmark: the first is the overall rate, i.e., number of queries over the total time spent by KLEE in constraint-solving activities (i.e. all activities in Figure 2), while the second is the rate seen by STP, i.e., the number of queries over the total time spent in STP.[5] The overall rate varies between 5.3 queries per second for factor and 189.0 for comm, with the median at 52.4 queries per second, while the STP rate varies between 5.3 queries per second for factor and 201.9 for comm, with the median at 56.4 queries per second. This is a high query rate; as we discussed in Section 2, constraint solving in symbolic execution has to handle a high number of queries per unit of time.

The column Q-size measures the average query size in terms of number of expression nodes, where shared expressions are only counted once. The set of the four smallest Q-size values corresponds to that of the four largest queries per second rates, but we did not observe any correlation for the other benchmarks.

The last column of Table 1 shows the percentage of time spent in constraint solving: the first number shows the percentage of time spent by KLEE in all constraint-solving activities, while the second number shows the percentage of time spent in STP. The key observation is that with the exception of comm, KLEE spends over 96% of the overall time in constraint-solving activities, with over 82% of the overall time spent in STP. This shows that for these benchmarks,

[5] STP times include the overhead incurred by using metaSMT.

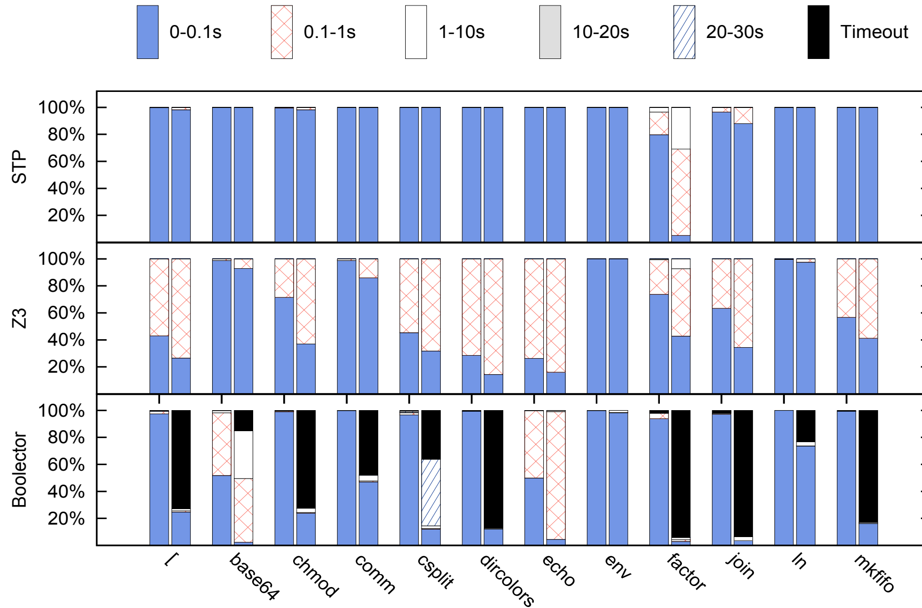

Fig. 3. Distribution of query types for the DFS runs without caching. For each benchmark, the left bar shows the percentage of queries that are processed by the core solver (via `metaSMT`) in 0–0.1s, 0.1–1s, 1–10s, 10–20s, 20–30s or reach the 30s timeout, while the right bar shows the percentage of time spent executing queries of each type.

constraint solving is the main performance bottleneck, and therefore optimising this aspect would have a significant impact on `KLEE`'s overall performance.

The upper chart in Figure 3 shows the distribution of query types for `KLEE` with `STP`. There are two bars for each benchmark: one showing the percentage of queries that finish in 0–0.1s, 0.1–1s, 1–10s, 10–20s, 20–30s, and time out; and one showing the percentage of time spent executing queries of each type. With the exception of `factor` and `join`, almost all queries (over 99%) take less than 0.1 seconds to complete, and `STP` spends almost all of its time (over 98%) solving such cheap queries. For `factor`, almost 80% of the queries still take less than 0.1s, but they account for only around 5% of the time.

Solver comparison. To compare the performance of different solvers, we ran `KLEE` via `metaSMT` with `STP`, `Z3` and `Boolector` for a fixed number of instructions, as discussed above. Besides the 30s timeout set for each query, we also set an overall timeout of three hours for each run. `STP` and `Z3` have no query timeouts, while `Boolector` has query timeouts on all benchmarks with the exception of `echo` and `env`. Note that per-query timeouts may have a significant impact on the instructions executed by `KLEE`, since on a query timeout, we terminate the current execution path (which may have later spawned a large number of paths), and follow alternative parts of the execution tree instead.

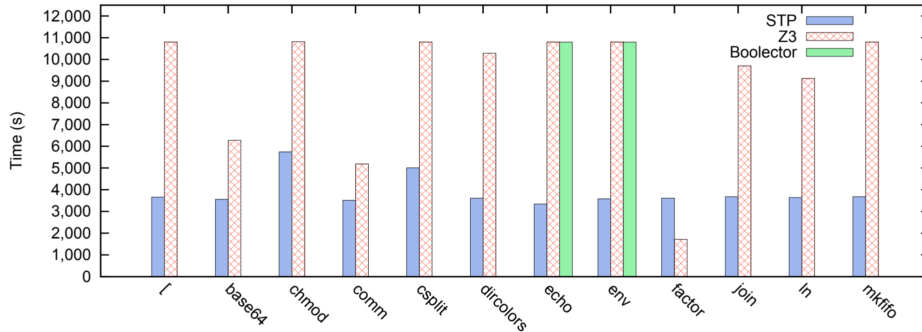

Fig. 4. Time taken by KLEE with metaSMT for STP, Z3 and Boolector, using DFS and no caches. We set a timeout of 30s per query and an overall timeout of 10,800s (3h) per run. Boolector had query timeouts on all applications apart from echo and env and always reached the overall timeout except for factor. Query timeouts affect the subsequent queries issued by KLEE, which is why we only show two bars for Boolector.

Figure 4 shows the results. STP emerges as the clear winner: it is beaten on a single benchmark, and it has no query timeouts nor overall timeouts. The one notable exception is factor, where Z3 finishes more than twice faster than STP (1,713s for Z3 vs. 3,609s for STP). This categorical win for STP is not that surprising: STP was initially designed specifically for our symbolic execution engine EXE [3], and its optimisations were driven by the constraints generated by EXE. As a re-design of EXE, KLEE generates the same types of queries as EXE, and thus benefits from STP's optimisations.

Figure 3 presents the distribution of query types for all solvers. Note that because of timeouts, the three solvers do not always execute the same queries. So although a precise comparison across solvers is not possible, this figure does show the mix of queries processed by each solver. While STP solves most of its queries in under 0.1s, the situation is different for the other two solvers: Z3 often spends a lot of its time processing queries in the 0.1–1s range, while Boolector processes a wider variety of query types. In particular, it is interesting to note that while Boolector often spends most of its time in queries that time out, the percentage of these queries is quite small: for example, for dircolors there are only 0.5% queries that time out, but these consume 87.3% of the time. This illustrates the effect of the timeout value on the overall performance of symbolic execution: one may observe that decreasing the timeout value to 10s would not change the results for STP and Z3, but would help Boolector solve more queries.

4.2 Solver comparison using the DFS strategy and caching

We repeated the same experiment with KLEE's caches enabled, to understand the effect of caching on solver time. That is, we ran KLEE using the STP API for one hour per benchmark, and we recorded the number of instructions executed in

Table 2. Statistics for the DFS runs with caching, using STP via the metaSMT API: the instructions per second rate, the number of queries and the queries per second rate in KLEE and STP respectively, and the percentage of time spent in all constraint-solving activities and STP respectively.

Application	Instrs/sec	Queries		Queries/sec		Solver(%)	
		total	STP	total	STP	total	STP
[	695	30,838	30,613	7.9	58.3	99.6	13.4
base64	20,520	184,348	47,600	42.2	42.6	98.7	25.3
chmod	5,360	46,438	37,911	12.6	75.7	99.2	13.5
comm	222,113	1,019,973	21,720	305.0	83.5	87.9	6.8
csplit	19,132	285,655	33,623	63.5	28.0	98.1	26.2
dircolors	1,091,795	5,609,093	2,077	4,251.7	64.0	36.3	0.9
echo	52	16,318	764	4.5	52.7	99.7	0.4
env	13,246	96,425	38,047	26.3	63.8	98.5	16.1
factor	12,119	80,975	6,189	22.6	1.8	99.1	97.6
join	1,033,022	5,362,587	4,963	3,401.2	34.2	43.9	4.0
ln	2,986	91,812	40,868	24.5	62.7	99.4	17.3
mkfifo	3,895	26,631	25,622	7.2	58.1	99.3	11.9

each case. We then re-ran KLEE on each benchmark for the previously recorded number of instructions using STP, Z3 and Boolector via the metaSMT API.

We begin again by showing, in Table 2, some statistics about the runs with KLEE using STP via the metaSMT API. The table presents the same information as in Table 1, except that we omit the Q-size metric, and we show two numbers under the Queries column: the first is the number of queries issued by KLEE to the branch cache,[6] while the second is the number of queries issued to STP (i.e. those that miss in both caches).

First of all, it is interesting to compare the rate of instructions and queries issued by KLEE with and without caching. While the actual instructions executed in each case may be quite different, Tables 1 and 2 clearly show that caching sometimes helps significantly, while sometimes hurts performance. For example, dircolors goes from a relatively modest 8,090 instructions per second without caching to 1,091,795 with caching, and from only 49.3 queries per second without caching to 4,251.7 with caching. At the other end of the spectrum, mkfifo decreases its processing rate from 25,331 to 3,895 instructions per second, and from 62.3 to 7.2 queries per second. This illustrates the need for better, more adaptive caching algorithms.

The Solver column shows that KLEE still spends most of its time in constraint-solving activities (which include the caching code): for 9 out of the 12 benchmarks, KLEE consumes more than 98% of the time solving queries. On the other hand, the amount of time spent in STP itself decreases substantially compared to the runs without caching, with a median value at only 13.5%.

[6] Recall that the branch queries stored in this cache sometimes summarise the result of two solver queries.

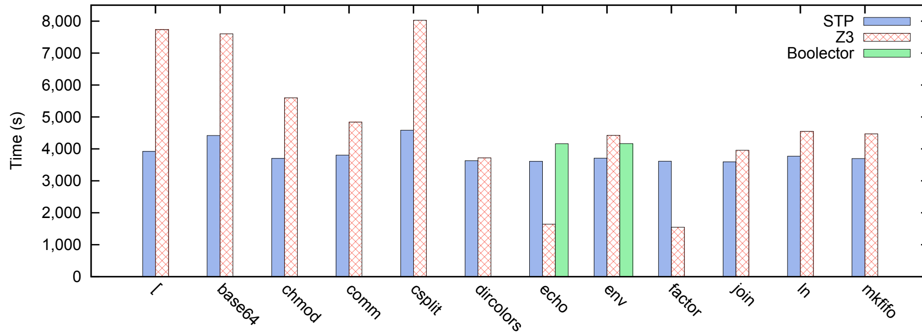

Fig. 5. Execution times for the DFS runs with caching. We set a timeout of 30s per query and an overall timeout of 10,800s (3h) per run. `Boolector` had individual query timeouts on all applications apart from `echo` and `env`. Query timeouts affect the subsequent queries issued by `KLEE`, which is why we only show two bars for `Boolector`.

Figure 5 shows the time taken by each solver on the DFS runs with caching. Overall, `STP` is still the clear winner, but the interesting fact is that caching can sometimes affect solvers in different ways, changing their relative performance. One such example is `echo`, where `Z3` wins: none of the solvers have timeouts, and all issue 16,318 overall queries, 764 of which reach the solver. However, `KLEE` with `Z3` spends significantly less time in the caching code, which accounts for its overall performance gain (interestingly enough, `STP` is faster than `Z3` for the 764 queries that reach the solver: 14s for `STP` vs. 74s for `Z3`). The widely different behaviour of the caching module is due to the different solutions returned by each solver: we noticed that with `Z3`, the counterexample cache tries significantly fewer subsets in its search for a solution.

Another interesting example is `env`: with all solvers, `KLEE` issues 96,425 queries. However, with `STP` and `Boolector`, only 38,047 reach the solver, while with `Z3`, 38,183 do. The 136 queries saved by the `STP` and `Boolector` runs are due to the fact that they were luckier with the solutions that they returned, as these solutions led to more hits afterwards.

To sum up, we believe these experiments demonstrate that the (relative) performance of `KLEE` running with different solvers can be affected, sometime substantially, by caching. Caching can significantly improve performance, but can also deteriorate it, so future work should focus on better, more adaptive caching algorithms. A key factor that affects caching behaviour are the solutions returned by the solver: first, different solutions can lead to different cache hit rates (as shown by the `env` runs), and second, even when the cache hit rate remains unchanged, different solutions can drastically affect the performance of the caching module, as demonstrated by the `echo` runs. Therefore, one interesting area of future work is to better understand the effect of the solutions returned by the solver on caching behaviour. In fact, we believe that this is also an aspect that could play an important role in the design of a portfolio solver: if multiple solvers finish at approximately the same time, which solution should we keep?

4.3 Threats to validity

There are several threats to validity in our evaluation; we discuss the most important ones below. First, we considered a limited set of benchmarks and only the DFS strategy, so the results are not necessarily generalisable to other benchmarks and search strategies. Second, as discussed in Section 4, `KLEE` is highly nondeterministic; while we put a lot of effort into eliminating nondeterminism, there are still variations across runs, which may affect measurements. Third, our results may be influenced by specific implementation details in `KLEE`, so we cannot claim that they are generalisable to other symbolic execution engines. Finally, we used the default configuration of the SMT solvers made available through the `metaSMT` API. Different configuration options may change the relative performance of the solvers.

5 Portfolio Solving in Symbolic Execution

Symbolic execution can be enhanced with a parallel portfolio solver at different levels of granularity. The coarsest-grained option is to run multiple variants of the symbolic execution engine (in our case `KLEE`), each equipped with a different solver. This is essentially what our experiments in Sections 4.1 and 4.2 show. This has the advantage of having essentially no overhead (the runs are fully independent, and could even be run on separate machines) and being easy to deploy. Given a fixed time budget, one can run in parallel variants of `KLEE` configured with different solvers and select at the end the run that optimises a certain metric (e.g. number of executed instructions); or given a certain objective (e.g. a desired level of coverage), one could run concurrently the different variants of `KLEE` and abort execution when the first variant achieves that objective. The key advantage here is that *without any a priori knowledge of which solver is better on which benchmarks*, the user would obtain the results associated with the best-performing variant.

Our experiments in Sections 4.1 and 4.2 already show that this can be effective in practice: for instance, for the runs with caching presented in Figure 5, where the objective is to execute a given number of instructions, the user would obtain results as soon as `KLEE` with `STP` finishes for, say, `[` and `chmod` (where `STP` wins), and as soon as `KLEE` with `Z3` finishes for `echo` and `factor` (where `Z3` wins).

Another option for integrating portfolio solving in symbolic execution is at a finer level of granularity, e.g. at the level of individual solver queries, or of groups of consecutive queries. We are currently designing a portfolio solver at the query level, and while it is too early to report any results, we include below a discussion of the most important aspects that we have encountered.

As one moves to finer granularity, one creates opportunities for the portfolio-based variant of `KLEE` to behave better than *all* single-solver variants. At the coarse granularity of `KLEE` variants, one cannot perform better than the best variant; instead, using a portfolio solver at the query level, `KLEE` may perform significantly better than when equipped with the best individual solver, since different solvers may perform better on different queries.

On the other hand, at the query level, the performance overhead can be substantial, potentially negating in some cases the benefit of using a portfolio solver. This is particularly true in light of the fact that the vast majority of queries take very little time to complete, as discussed in Section 4.1. For such queries, the time spent spawning new threads or processes and monitoring their execution may be higher than the actual time spent in the SMT solvers. As a result, one idea is to start by running a single solver, and only if that solver does not return within a small time span (e.g. 0.1s), spawn the remaining solvers in the portfolio.

Related to the point above, we noticed that spawning threads versus spawning processes to run a solver can have a significant effect on performance. On the one hand, threads are more lightweight, and therefore incur less overhead: on several runs of `KLEE` configured with a single version of `STP` we observed significant speedups (varying from 1.25x to 1.93x) by simply switching from using a process to using a thread to run `STP`. On the other hand, using processes has the advantage that memory management is not an issue: on process exit, all the memory allocated by the solver is automatically freed by the operating system. This does not happen when threads are used, and we have observed that in many cases `KLEE` equipped with a portfolio of threaded solvers ends up consuming all available memory and starts thrashing.

Another important consideration is caching. As we discussed in Section 4.2, the actual counterexample values returned by a solver can have a significant influence on performance, so deciding what values to keep can be important. One option is to store the values of the first solver that returns with an answer. However, for queries where multiple solvers perform similarly, one might want to wait for a small additional amount of time to see if other solvers terminate too. If this happens, one can consider keeping all the counterexamples returned by different solvers, or selecting some of them: of course, keeping more counterexamples may increase the hit rate, but degrade performance. Some SMT solvers (and the SAT solvers they use) are incremental: to benefit from this aspect, it might also be important to wait for a short amount of time to allow more solvers to finish, as discussed above.

Finally, we consider the makeup of a portfolio solver. While including different SMT solvers is an obvious choice, based on our experience, we believe it is also important to consider different variants and versions of the same solver. While solvers may overall evolve for the better, given the nature of the problem, it is not uncommon to find queries on which newer versions of a solver perform worse. As a result, multiple versions of the same solver can be good candidates for a portfolio solver.

In addition, most solvers have a plethora of configuration options, which can have a significant impact on solving time. Selecting the right configuration parameters is a difficult decision, as it is often impossible to tell in advance which parameter values will perform better on which queries. Also, many modern SMT solvers are build on top of SAT solvers. Configuring a given SMT solver with different options and SAT solvers can provide additional candidates to include in

the portfolio. Finally, SAT solvers have their own configuration options, which can be varied to create additional candidates.

6 Related Work

Constraint solving plays an important role in symbolic execution, and a significant effort has been invested in understanding and optimising constraint solving and constraint caching in a symbolic execution context, e.g. [2, 3, 19, 22, 25]. This paper provides additional information about the constraints encountered during symbolic execution (and in `KLEE` in particular), the effect of caching on solving time, and the relative performance of different solvers on several real benchmarks.

Portfolio solving has been explored in the past in the context of SAT solving [13, 24], SMT solving [23], and bounded model checking [9], among others. As far as we know, this is the first paper that reports on how different SMT solvers compare and could be combined in a portfolio solver in the context of symbolic execution.

Portfolio solving is a form of variant-based parallelization, which has been effectively used in the past to improve application performance, e.g. [8, 20, 21]. For instance, [20] proposes a general framework for competitive execution that targets multicore and multiprocessor systems, in which sequential applications are optimised by introducing competitive variants for parts of the program.

7 Conclusion

In this paper, we have discussed some of the most important characteristics of the constraints generated in symbolic execution, and identified several aspects that we believe are important for designing better SMT solvers for symbolic execution, and for combining multiple solvers using a portfolio-based approach. In particular, we have shown that counterexample values and caching can in some cases significantly affect constraint solving, and discussed several options for designing a portfolio solver for symbolic execution.

The reader can find additional information about our `KLEE` extension and experiments at `http://srg.doc.ic.ac.uk/projects/klee-multisolver`.

Acknowledgements. We would like to thank the program committee chairs of CAV 2013, Natasha Sharygina and Helmut Veith, for the opportunity to publish this invited paper. We are grateful to the `metaSMT` developers, in particular Heinz Riener and Finn Haedicke for their help with `metaSMT`. We would also like to thank Armin Biere for his help with `Boolector`. Finally, we thank Alastair Donaldson, Dan Liew, and Paul Marinescu, for their careful proofreading of our paper. This research is supported by the EPSRC grant EP/J00636X/1.

References

1. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: TACAS'09
2. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI'08
3. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: Automatically generating inputs of death. In: CCS'06
4. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice—preliminary assessment. In: ICSE Impact'11
5. Cadar, C., Pietzuch, P., Wolf, A.L.: Multiplicity computing: A vision of software engineering for next-generation computing platform applications. In: FoSER'10
6. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (2013)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS'04
8. Cledat, R., Kumar, T., Sreeram, J., Pande, S.: Opportunistic computing: A new paradigm for scalable realism on many-cores. In: HotPar'09
9. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE'09
10. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (Sep 2011)
11. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV'07
12. Haedicke, F., Frehse, S., Fey, G., Große, D., Drechsler, R.: metaSMT: Focus on your application not on solver integration. In: DIFTS'12
13. Hamadi, Y., Sais, L.: ManySAT: a parallel SAT solver. JSAT 6, 245–262 (2009)
14. King, J.C.: Symbolic execution and program testing. CACM 19(7), 385–394 (Jul 1976)
15. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO'04
16. Marinescu, P.D., Cadar, C.: make test-zesti: A symbolic execution solution for improving regression testing. In: ICSE'12
17. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08
18. Ranise, S., Tinelli, C.: The SMT-LIB format: An initial proposal. In: PDPAR'03
19. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/FSE'05
20. Trachsel, O., Gross, T.R.: Variant-based competitive parallel execution of sequential programs. In: CF'10
21. Vajda, A., Stenstrom, P.: Semantic information based speculative parallel execution. In: PESPMA'10
22. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: reducing, reusing and recycling constraints in program analysis. In: FSE'12
23. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: A concurrent portfolio approach to SMT solving. In: CAV'09
24. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. JAIR 32(1), 565–606 (2008)
25. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: ISSTA'12