Pending Constraints in Symbolic Execution for Better Exploration and Seeding

Timotej Kapus
Imperial College London
United Kingdom
t.kapus@imperial.ac.uk

Frank Busse
Imperial College London
United Kingdom
f.busse@imperial.ac.uk

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

Symbolic execution is a well established technique for software testing and analysis. However, scalability continues to be a challenge, both in terms of constraint solving cost and path explosion. In this work, we present a novel approach for symbolic execution, which can enhance its scalability by aggressively prioritising execution paths that are already known to be feasible, and deferring all other paths. We evaluate our technique on nine applications, including *SQLite3*, *make* and *tcpdump* and show it can achieve higher coverage for both seeded and non-seeded exploration.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Symbolic execution, KLEE

ACM Reference Format:
Timotej Kapus, Frank Busse, and Cristian Cadar. 2020. Pending Constraints in Symbolic Execution for Better Exploration and Seeding. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416589

1 INTRODUCTION

Symbolic execution is a dynamic program analysis technique that has established itself as an effective approach for many software engineering problems such as test case generation [4, 12], bug finding [6, 13], equivalence checking [10, 11], vulnerability analysis [8, 27] and debugging [14, 20].

Even with well-engineered tools like KLEE [4], symbolic execution still faces important scalability challenges. These fall into two broad categories: constraint solving and path explosion. As symbolic execution proceeds, the complexity of constraints and the number of paths typically increase, often making it difficult to make meaningful progress.

In this work, we propose a novel mechanism that aggressively explores paths whose feasibility is known via caching or seeding. Our approach tackles both scalability challenges of symbolic execution. On the one hand, it enables more efficient use of solved constraints, thus reducing the burden on the solver. And on the other hand, it provides a meta-search heuristic that gives a way to guide the exploration towards interesting parts of the program.

Before presenting our idea, we briefly summarise symbolic execution. We focus here on the *EGT-style* of dynamic symbolic execution [5], embodied in tools such as KLEE [4], which unlike *concolic execution* tools [12, 24], store partially explored paths in memory. Symbolic execution works by running the program on some symbolic inputs, which means they can initially take any value, as they are unconstrained. During execution, if a branch condition depends on a symbolic value, symbolic execution queries an SMT solver for the feasibility of each of the two branches (under the current path condition which is initially empty). If both the `then` and the `else` branches are feasible, it forks the execution exploring both paths and adding the respective branch conditions to each path condition (PC). After every fork, symbolic execution uses a search heuristic to decide what path to explore next. Each path explored in symbolic execution is encoded by a *state* which keeps all the information necessary to resume execution of the associated path (PC, program counter, stack contents, etc.).

The core of our idea revolves around *inverting* the forking process. Instead of doing an (expensive) feasibility check first and then forking the execution, we fork the execution first. The branch condition is then added as a *pending constraint*, which means its feasibility has not been checked yet. We refer to states (or paths) with pending path constraints as *pending states*.

The responsibility for feasibility checking of pending path constraints is passed to the search heuristic. This gives the search heuristic the capability to decide when and for which states it wants to pay the price of constraint solving. For example, it could solve pending states immediately, thus restoring the original algorithm, or could take into account the (estimated) cost of constraint solving in its decisions.

In our approach, we take advantage of an important characteristic of symbolic execution runs: the feasibility of some paths/states can be quickly determined without using a constraint solver. There are two common cases. First, modern symbolic execution systems like KLEE make intensive use of caching and many queries can be solved without involving the constraints solver [1, 4, 26]. Second, symbolic execution is often bootstrapped with a set of seeds from which to start exploration: these can come from regression test suites [18, 19] or greybox fuzzers in hybrid greybox/whitebox fuzzing systems [9, 21, 25]. By aggressively following paths for which feasibility can be quickly determined without using a constraint solver, our approach can minimise the constraint solving

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416589

Algorithm 1 Standard symbolic execution.

```
 1: Set $states$
 2:
 3: function Fork(State $s$, SymExpr $condition$)
 4:     if IsSat($s.pc \land condition$) $\land$ IsSat($s.pc \land \neg condition$) then
 5:         $falseState \leftarrow s$
 6:         $falseState.pc \leftarrow s.pc \land \neg condition$
 7:         $s.pc \leftarrow s.pc \land condition$
 8:         SearcherAdd($s$, $falseState$)
 9:     end if
10: end function
11:
12: function SearcherAdd(State $s1$, State $s2$)
13:     $states \leftarrow states \cup \{s1, s2\}$
14: end function
15:
16: function SearcherSelect()
17:     return SearchHeuristic($states$)
18: end function
```

Algorithm 2 Symbolic execution with pending constraints.

```
 1: Set $feasibleStates, pendingStates$
 2:
 3: function Fork(State $s$, SymExpr $condition$)
 4:     $feasibleStates \leftarrow feasibleStates \setminus \{s\}$
 5:     $s.pendingCondition \leftarrow condition$
 6:     $falseState \leftarrow s$
 7:     $falseState.pendingCondition \leftarrow \neg condition$
 8:     SearcherAdd($s$, $falseState$)
 9: end function
10:
11: function SearcherAdd(State $s1$, State $s2$)
12:     foreach State $s \in \{s1, s2\}$ do
13:         if FastIsSat($s$, $s.pc \land s.pendingCondition$) then
14:             $s.pc \leftarrow s.pc \land s.pendingCondition$
15:             $s.pendingCondition \leftarrow \emptyset$
16:             $feasibleStates \leftarrow feasibleStates \cup \{s\}$
17:         else
18:             $pendingStates \leftarrow pendingStates \cup \{s\}$
19:         end if
20:     end foreach
21: end function
22:
23: function SearcherSelect()
24:     while $feasibleStates = \emptyset$ do
25:         $s \leftarrow$ SearchHeuristic($pendingStates$)
26:         if IsSat($s.pc \land s.pendingCondition$) then
27:             $s.pc \leftarrow s.pc \land s.pendingCondition$
28:             $s.pendingCondition \leftarrow \emptyset$
29:             $feasibleStates \leftarrow feasibleStates \cup \{s\}$
30:         end if
31:         $pendingStates \leftarrow pendingStates \setminus \{s\}$
32:     end while
33:     return SearchHeuristic($feasibleStates$)
34: end function
```

costs as well as provide an effective exploration of the program search space.

The rest of the paper is organised as follows. In §2 we present in detail the design of our technique. Then in §3 we give further implementation-specific details in the context of building our technique in KLEE. We finally evaluate our approach in §4, discuss related work in §5 and conclude in §6.

2 ALGORITHM

Algorithm 1 presents, in simplified form, the forking and search heuristics components of the standard symbolic execution algorithm. The symbolic executor is exploring program paths encoded as a set of *states* (line 1). As discussed before, a state keeps track of all the information necessary to resume execution of the associated path (program counter, stack contents, etc.) and particularly its path condition *pc*. When a symbolic state *s* encounters a symbolic branch *condition*, the Fork function is called (line 3).

Fork first checks if the *condition* can be both *true* and *false* under the current path condition (line 4). If so, the state is duplicated into a *falseState* (line 5), which will be the state representing the execution of the false branch. The path conditions of the two states are then updated accordingly (lines 6 and 7) and the new state is added to the set of states of the search heuristic by calling SearcherAdd (line 8). If the *condition* cannot be both true and false, the path condition is not updated and *s* continues to execute the only feasible side of the branch (the updates to the program counter are omitted for ease of exposition).

After each instruction, the symbolic executor calls SearcherSelect (lines 16–18) to select the state to be executed next. In the case of standard symbolic execution, SearcherSelect simply forwards calls to the SearchHeuristic. For instance, the search heuristic might choose states according to depth or breadth-first strategies or randomly pick a state. Below, we introduce the three search heuristics which we explore in this project:

Depth-first search is a standard graph traversal algorithm that explores states as deep as possible before backtracking.

Random path search, introduced in the original KLEE paper [4], works by randomly selecting a path through the execution tree of explored states. The algorithm starts at the root of the tree and with 50% probability follows the left-hand subtree and 50% probability the right-hand one. The process repeats until a leaf state is reached, which is selected for further exploration. By design, this search heuristic favours states closer to the root of the execution tree.

Depth-biased search is a form of non-uniform random search provided by KLEE. It works by randomly selecting states weighted by their depth—the higher the depth of a state, the more likely for it to be selected. By design, this search heuristic favours states deeper in the execution tree.

2.1 Pending Constraints Algorithm

Note that the calls to IsSat in standard symbolic execution are potentially very expensive and thus are the optimisation target of our proposed approach.

Algorithm 2 shows the same three functions in the pending constraints version of symbolic execution. In this version, we maintain two disjoint sets of states: *feasibleStates*, which stores regular states which we know are feasible; and *pendingStates*, which stores pending states for which feasibility is still unknown (line 1).

Algorithm 3 Fast satisfiability checking.

```
1: function FastIsSat(State $s$, SymExpr $condition$)
2:     $assignments \leftarrow$ GetAssignmentsForState($s$)
3:     foreach Assignment $a \in assignments$ do
4:         if Substitute($a$, $condition$) = true then
5:             return true
6:         end if
7:     end foreach
8:     return false
9: end function
```

In our version of Fork, the IsSat feasibility checks are skipped and execution is forked unconditionally. First, we remove the current state s from the list of *feasibleStates* (line 4), and assign the *condition* to a special pending condition field associated with s (line 5). Second, we duplicate s into a new state *falseState* (line 6) and assign the negation of *condition* to its pending condition field (line 7). When a state becomes pending, as s and *falseState* here, it means that it should not continue execution until its pending condition is checked for feasibility. We call the process of checking the pending condition and adding it to the path condition as *reviving* the state.

Finally, we call SearcherAdd (line 8) to let the searcher decide what to do with the newly created pending states. For each of the two states, SearcherAdd checks whether the pending condition is feasible using a fast satisfiability checker (line 13). If so, the pending condition is added to the path condition (line 14), the *pendingCondition* field is reset (line 15) and the state is added to the set of *feasibleStates* (line 16). If the fast satisfiability check is unsuccessful, the state is added to the set of *pendingStates* (line 18). The fast satisfiability solver is discussed in §2.2.

The SearcherSelect function operates as in standard symbolic execution as long as there are feasible states available (line 33). However, when all the feasible states have been exhausted, it keeps picking a pending state (line 25) until it finds one that can be revived successfully. This is done by asking the solver whether the pending condition is feasible (line 26) and if so, by adding the pending condition to the state’s path condition (line 27), clearing the *pendingCondition* field (line 28), and adding the state to the set of *feasibleStates* (line 29).

2.2 Fast Satisfiability Checking

As discussed in the introduction, the fast satisfiability checker relies on the fact that we often have existing assignments (solutions) to symbolic constraints. There are two common scenarios that occur in practice: first, modern symbolic execution engines use a smart form of caching to speed up constraint solving (§2.2.1) and second, symbolic exploration often starts with a set of seeds (§2.2.2).

Algorithm 3 shows the fast satisfiability checking algorithm. More formally, an *assignment* is a mapping from symbolic variables to concrete values, e.g. $\{x \leftarrow 2, y \leftarrow 3\}$. The Substitute function takes a symbolic condition and an assignment and evaluates the condition under the assignment (line 4). That is, it substitutes all the symbolic variables in the condition with the values specified in the assignment. If all the symbolic variables in the expression are mapped by the assignment, Substitute will return a concrete value (either *true* or *false*). Otherwise it will return *false*.

FastIsSat first gets all the assignments associated with the given state using the GetAssignmentsForState function (line 2); we will discuss how this function works below. The condition is then evaluated on every assignment returned by GetAssignmentsForState and if the evaluation results in *true*, FastIsSat returns successfully (line 5). If none of the returned assignments satisfy the condition, then FastIsSat returns unsuccessfully (line 8).

2.2.1 Caching. When issuing satisfiability queries to the solver, modern symbolic execution engines typically also ask for a satisfying assignment when the query is satisfiable. This is because caching these assignments can be highly effective [1, 4, 26]. In this work, we similarly use cached assignments to implement our fast satisfiability checks. Substituting all assignments that were returned by the core solver at any previous point in the execution can be expensive and here we want to ensure the check is fast. So instead, we use the same mechanism used by assignment caching in KLEE, and have GetAssignmentsForState (line 2) return only the assignments associated with a subset of the path condition of state s.

The idea behind the pending constraints approach is to use existing assignments as long as it is possible, and only ask for expensive new solutions when absolutely necessary. As a result, the search goes deep into the execution tree, because it explores the solutions it has to completion. Therefore, combining this strategy with a search heuristic that goes wide when picking a pending state can be really effective. Note that the search heuristics that often perform best, such as random path search, tend to behave like this [4]. Another way of understanding our approach is as a meta-searcher that enhances a given search heuristic, enabling it to explore deeper parts of the program. Intuitively, this is achieved by picking a path and sticking to it, while normally the same searcher would keep changing between paths. Importantly, the path it sticks to issues no additional queries to the SMT solver and therefore completes quickly.

To make things more concrete, consider the program in Figure 1 with `DFS_FRIENDLY` undefined. It has two symbolic variables: a boolean `isSpace` and a string `str`. It first branches on `isSpace` on line 14 and writes space as the first character of `str` on the `then` branch or a zero on the `else` branch. The two loops between lines 21–23 then introduce a large number of symbolic branches. After these loops, it computes the fifteenth Fibonacci number using an inefficient recursion. This represents some large concrete workload. Finally, we assert `isSpace` is `false`. Note that reaching this assertion failure only depends on the first branch and is completely independent of the large number of paths spawned in the loops.

Vanilla KLEE executes around 4 million instructions, taking around 10 seconds to reach the assertion failure using the random path search strategy. By contrast, the pending constraints approach executes only around 67k instructions, taking less than a second. The reason is that vanilla KLEE needs to compute a large number of solutions to the symbolic branches in the loops, which are not needed to find the assertion failure. Whereas the pending constraints approach only needs one solver solution for `str`. Because random path search is the underlying search strategy, it is

```
 1 unsigned fib(unsigned n) {
 2   if (n == 0) return 0;
 3   if (n == 1) return 1;
 4   return fib(n - 1) + fib(n - 2);
 5 }
 6
 7 int main() {
 8   bool isSpace;
 9   klee_make_symbolic(&isSpace, sizeof(isSpace), "isSpace");
10   char str[6];
11   klee_make_symbolic(str, sizeof(str), "str");
12
13 #ifndef DFS_FRIENDLY
14   if (isSpace) str[0] = '␣';
15   else str[0] = '\0';
16 #else
17   if (isSpace == 0) str[0] = '\0';
18   else str[0] = '␣';
19 #endif
20
21   for (int i = 1; i < 6; i++)
22     for (char j = 0; j < str[i]; j++)
23         str[i-1]++;
24
25   fib(15);
26   assert(!isSpace);
27   return 0;
28 }
```

Figure 1: An example program where pending constraints find the failing assertion faster.

also more likely to pick the pending states produced by the branch on line 14 as it is closer to the root of the process tree.

Note that DFS reaches our one-hour timeout in this case. However, it performs similarly to the pending states if the order of branches on line 14 is swapped (by defining `DFS_FRIENDLY`). With `DFS_FRIENDLY` defined, both pending states and upstream KLEE find the assertion in about 33k instructions. This shows that while DFS can be as efficient, it is highly dependent on the order of branches, unlike the pending constraints approach.

2.2.2 Seeding. A seed is a concrete input to a program which is given to symbolic execution to bootstrap exploration. A seed can make symbolic execution of the program easier for two reasons. First, it is used to guide symbolic execution toward parts of the program that are executed by real inputs. Second, it provides a solution for complex path constraints, therefore removing the need for expensive constraint solving. Seeding has been shown to be effective when symbolic execution is combined with regression test suites [18, 19, 22] as well as when it is integrated with a greybox fuzzer [9, 21, 25].

Our pending constraints approach can be easily extended to support seeding. In FastIsSat, we modify the GetAssignmentsForState function to return the seeds as assignments (line 2). As a result, only the states that satisfy the seeds are followed first, and there are no calls to the SMT solver until all the paths followed by the seeds are explored. In this use case, pending states represent the possible divergences from the seeds, from where we can pick up exploration once the seeding is completed.

When seeds are available, random path search is unlikely to be the most effective strategy. Once the paths followed by the seeds are completed, random path search would pick a state very close to the start of the program, as the probability of picking a state halves at every symbolic branch. This means that the exploration will behave similarly as to when no seeds are available.

On the other hand, depth-biased search would most likely pick a state toward the end of the path executed by the seeds, meaning it will start to explore paths that are very hard to reach from the start of the application. In other words, it benefits from the seed and can start exploring code out of reach of normal symbolic execution.

However, there are two limitations to this search heuristic. First, by design, this strategy misses code that is easy to reach from the start of the program. Therefore, it is best combined with a more wide-reaching strategy such as random path search. Second, this heuristic is more likely to select pending states that are infeasible, as the path conditions for those states have more constraints and thus the pending condition is more likely to be infeasible in that state.

3 IMPLEMENTATION

We implemented our approach in KLEE [4], a state-of-the-art symbolic execution system that operates on LLVM bitcode [16]. Our prototype is based on KLEE commit `0fd707b` and is configured to use LLVM 7 and STP 2.3.3. We discuss below some of the most important implementation aspects.

We make our prototype and associated artefact available at https://srg.doc.ic.ac.uk/projects/pending-constraints and https://doi.org/10.6084/m9.figshare.12865973.

3.1 Fast Satisfiability Solver

KLEE's constraint solving chain consists of a series of caches and other partial solvers, finally delegating the query to an external SMT solver. For our fast solver we simply used this solver chain without the final, potentially expensive, call to STP.

Seeding is easily implemented with KLEE's counterexample (assignment) cache, as we simply add the seed to the cache as an additional assignment at the start of execution. Note that this is different from how vanilla KLEE implements seeding. However, to make the comparison between vanilla KLEE and pending constraints fair, we also added the seed to the cache in vanilla KLEE.

3.2 Error Checks

During execution, KLEE performs various checks to find errors. Examples include checks for division by zero and out-of-bounds memory accesses. One option would be to treat these checks as we treat regular branches and create pending states. However, that would mean that errors wouldn't be caught as soon as that code is covered, as in vanilla KLEE (and in fact they might never be caught if those pending states are killed later on due to memory pressure). Therefore, we treat these checks specially by always performing them instead of creating pending states.

However, if higher coverage is more important than bug finding, deferring these checks to pending states might make more sense. Therefore, we also run our benchmarks without these checks in a version that we call pending constraints with *relaxed checks*. We explore this version in §4.5 and observe significant coverage gains.

3.3 Branching without a Branch Instruction

In KLEE there are several cases where forking happens without a branch instruction. For instance, in the default configuration, a switch instruction is handled as a branch with multiple targets. This fits less neatly in our model therefore we simply configure KLEE to lower all switch instructions to a series of `if-else` statements.

When a symbolic pointer is encountered, KLEE scans the address space to find all memory objects to which it can point. If multiple objects are found, it forks execution for each object, adding appropriate constraints [4, 15]. In the default version of our approach, we don't create pending states in this case; instead we eagerly fork as necessary, as in vanilla KLEE. However, with relaxed checks, for each memory object to which the pointer can refer, the state gets forked into two pending states: one in-bounds of the object and the other one out-of-bounds. This second pending state encompasses all other resolutions of the symbolic pointer.

3.4 Releasing Memory Pressure

KLEE often hits the memory limit for large programs due to the vast size of the search space and generally broad search heuristics [3]. When exceeding the memory limit, KLEE terminates a number of randomly-chosen states to get back below the memory limit. We could follow the same approach with pending constraints, but this could delete both feasible and infeasible pending states. Feasible pending states potentially represent large parts of the search space and we should avoid their termination. By reviving pending states, we can select and terminate infeasible pending states. But this comes at a price, as reviving states requires expensive constraint solving. In our implementation, we decided to stop reviving states when the remaining number of pending states equals the number of states we still need to terminate to fall below the memory limit. At that point, we start to randomly choose states to terminate as in vanilla KLEE.

4 EVALUATION

Our evaluation is structured as follows. We present our benchmarks in §4.1 and explain why we evaluated them on internal coverage in §4.2. We evaluate pending constraints on their ability to enhance symbolic execution in a non-seeding context in §4.3 and then in a seeding context in §4.4. We also evaluate the version with relaxed checks in §4.5. Finally, we discuss our approach in further detail in §4.6, via a case study on *SQLite3*.

We run all experiments in a Docker container on a cluster of 18 identical machines with Intel i7-4790 @ 3.60GHz CPU, 16GiB of RAM and Ubuntu 18.04 as operating system. We repeat our experiments three times and where possible plot the average and the minimum and maximum as an interval. In cases where we combine experiments, there are no intervals shown as we merge all the repetitions.

4.1 Benchmarks

Our aim is to demonstrate the capability of pending constraints to increase the exploration potential of symbolic execution. Therefore, we chose benchmarks that are challenging for symbolic execution.

SQLite3[1] is one of the most popular SQL database systems. We used version 3.30.1 via its default shell interface with symbolic input and without custom drivers. However, we adjusted some compile-time flags, such as simplifying the allocation model, to make it easier for symbolic execution.

magick[2] is a command-line utility for performing various operations on images. We used version 7.0.8-68 in two configurations, converting (symbolic) jpeg images to png images and back, which makes use of *libjpeg*[3] and *libpng*[4].

tcpdump[5] is used for capturing and printing network packets. We used version 4.10.0 to print information from a symbolic capture file.

GNU m4[6] is a macro processor. We used version 1.4.18 to process a symbolic file.

GNU make[7] is a widely-used build system. We used version 4.2 on a symbolic makefile.

oggenc[8] is a tool from the Ogg Vorbis audio compression format reference implementation. We used version 1.4.0 to convert a symbolic `.wav` file to the `.ogg` format.

GNU bc[9] is an arbitrary precision calculator. We used version 1.07 on symbolic standard input.

GNU datamash[10] is a tool for performing command-line calculations on tabular data. We used version 1.5 both with symbolic input and symbolic arguments.

Table 1 gives a more detailed overview of the benchmarks, showing their size, arguments we used and a brief description of the seeds. The program size is measured in terms of LLVM instructions as reported by KLEE. The third column shows the arguments we used to run the benchmarks. The arguments prefixed with `-sym` are replaced with symbolic data. For example, `-sym-arg 3` is replaced with a three-character symbolic argument, `-sym-stdin 20` is a 20-byte symbolic standard input stream, and `-sym-file 100` creates a file named `A` with 100 bytes of symbolic content.

The last two columns of Table 1 give an overview of the seeds we used. Where possible, we give the exact data used as in the case of *bc*. However, in the case of *magick* and similar utilities that would not be too informative, so we give instead a brief description of the seed. We took the smallest valid files from https://github.com/mathiasbynens/small. Some seeds were padded with zeroes as necessary to make the seed size match the number of symbolic bytes as specified by the arguments.

Figure 2 shows the percentage of time spent in the solver by vanilla KLEE on our benchmarks, on 2-hour runs, using different

[1] https://www.sqlite.org/
[2] https://imagemagick.org/
[3] http://libjpeg.sourceforge.net/
[4] http://www.libpng.org/
[5] https://www.tcpdump.org/
[6] https://www.gnu.org/software/m4/
[7] https://www.gnu.org/software/make/
[8] https://xiph.org/vorbis/
[9] https://www.gnu.org/software/bc/
[10] https://www.gnu.org/software/datamash/

Table 1: Overview of (symbolic) arguments and used seeds for our benchmarks. The benchmark size is given as number of LLVM instructions in the bitcode file.

Benchmark	Size	Arguments	Seed set 1 ($s1$)	Seed set 2 ($s2$)
bc	34,311	`-sym-stdin 20`	`435 / 4 + 6 - 3421`	`x=6 sqrt(2 * x + 5)`
datamash	63,405	`-sym-arg 3 -sym-arg 1 -sym-arg 4 -sym-arg 1 -sym-stdin 20`	`sum 1 mean 3` on a 3×3 matrix	`md5 1 sha1 2` on 2×3 matrix
magick$_{jpg}$	1,368,912	`jpeg:fd:0 png:fd:1 -sym-stdin 285`	smallest valid JPEG file	1×1 JPEG image created with *GIMP*
magick$_{png}$	1,368,912	`png:fd:0 jpg:fd:1 -sym-stdin 70`	smallest valid PNG file	1×1 PNG image created with *GIMP*
make	80,790	`-n -f A -sym-arg 2 -sym-arg 5 -sym-file 20`	`-n -Bfds` with `$(info $$helo ther)`	`-n -Bfds` with `a:=4 $(info $a)`
oggenc	87,142	`A -sym-file 100`	smallest valid WAV file	sine wave WAV file created by *SciPy*
SQLite3	283,020	`-sym-stdin 20`	`SELECT * FROM t;`	`CREATE TABLE t (i);`
tcpdump	353,196	`-r A -K -n -sym-file 100`	100 byte captured packet	another 100 byte captured packet

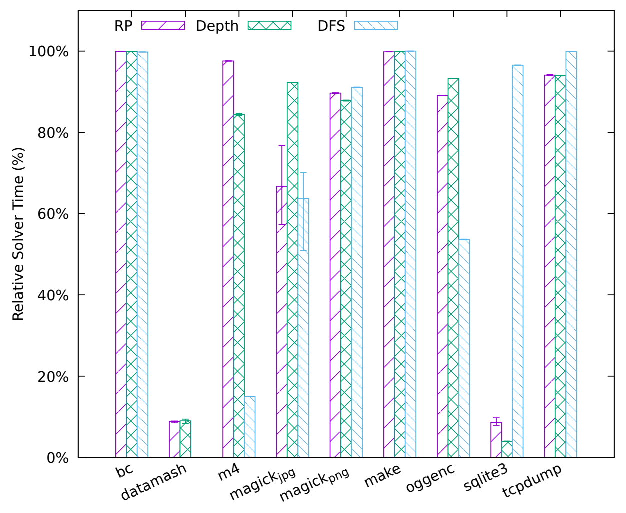


Figure 2: Relative time spent in the SMT solver (STP) by vanilla KLEE in our non-seeded experiments with the random path (RP), depth-biased (Depth) and DFS strategies.

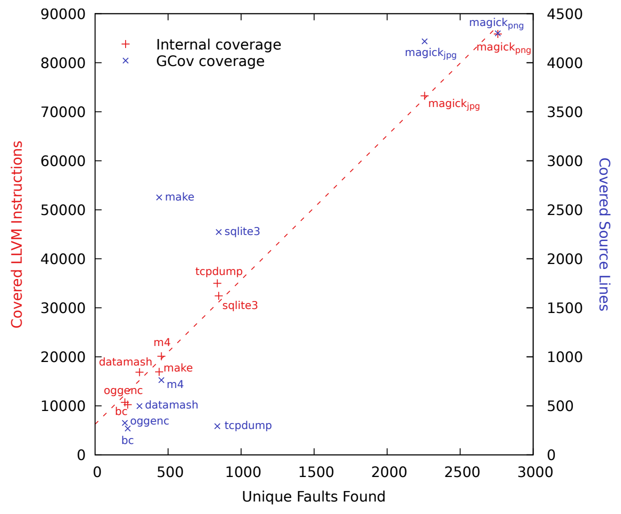


Figure 3: Dual-axis scatter plot of internal coverage (left y-axis) and *GCov* coverage (right y-axis) against the number of injected faults found.

search strategies. As can be seen, most of our benchmarks are solver-bound, meaning they spent a very high proportion of time in the solver, for example *bc* and *make*. *datamash* is the only benchmark that is not solver-bound in any search strategy considered, meaning KLEE spends very little time constraint solving. *SQLite3* is only solver-bound with the DFS search strategy.

4.2 Internal Coverage vs. *GCov* Coverage

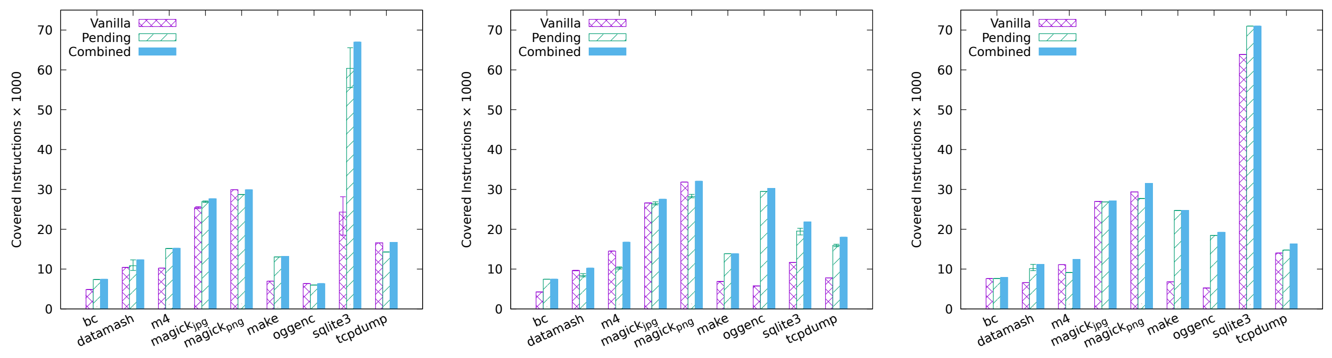
When measuring the coverage achieved by vanilla KLEE and our pending constraints extension, we can use either the internal coverage reported by KLEE or the coverage reported by *GCov*[11] when running the test inputs generated by KLEE.

The internal coverage reported by KLEE includes the lines of code that KLEE executed symbolically, at the LLVM bitcode level. By contrast, the *GCov* coverage is the coverage reported by *GCov* at the source code level, when rerunning the test inputs generated by KLEE on a version of the program compiled with *GCov* instrumentation. Both approaches have advantages and disadvantages.

On the one hand, *GCov* coverage has the advantage of being disconnected from KLEE and LLVM; often developers just want a tool like KLEE to generate a high-coverage test suite and assess this coverage independently, at the source code level. On the other hand, *GCov* coverage includes parts of the program that KLEE did not execute symbolically; these parts of code were not error-checked by KLEE, and thus bugs could be missed. It is important to note here that when KLEE performs an error check, it finds a bug if there are *any* values that can trigger it on that path. By contrast, a test case that covers the buggy path might not necessarily reveal the bug. Therefore, if KLEE is used primarily to reveal bugs, internal coverage is more appropriate.

To illustrate this issue, we injected a division-by-zero error in every basic block of some of our benchmarks. These division-by-zero errors are triggered by a different value in each basic block. We then run vanilla KLEE for two hours on these fault-injected programs and measure both internal and *GCov* coverage.

[11]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html



(a) Random Path **(b) Depth-Biased** **(c) DFS**

Figure 4: Instructions covered by vanilla KLEE and pending constraints alongside their combination in 2h non-seeded runs.

Figure 3 shows a dual-axis scatter plot of both internal coverage on the left y-axis and *GCov* coverage on the right y-axis against the number of unique injected faults found.

As can be seen, internal coverage seems to highly correlate with unique faults found, whereas *GCov* coverage shows no such correlation. We discuss this in more detail in a blog post [7].

In our work, we observed that pending constraints do not necessarily improve *GCov* coverage, but significantly improve internal coverage. Therefore, while accepting the former, we show in our experiments that it can improve the latter, and thus report internal coverage in our experiments.

4.3 Non-seeded Experiments

To evaluate pending constraints in a setting without seeds, we ran each of our benchmarks for 2 hours with the random path, DFS and depth-biased strategies. The results are shown in Figure 4. For some benchmarks, such as *bc* and *make*, pending constraints consistently cover more instructions for all search strategies. For others, such as *m4* and *tcpdump*, the relative performance is dependent on the search strategy.

If we look at the combined coverage of vanilla KLEE and pending constraints, we see that there is some complementarity to the approaches. For instance, while vanilla KLEE obtains better coverage for *m4* with the depth-biased search, the pending constraints reach code that is not covered by vanilla KLEE. Overall, pending constraints reached 35%, 24% and 34% more instructions across our benchmarks with random path, DFS and depth-biased search respectively. These results show that pending constraints can significantly increase the power of symbolic execution on some benchmarks by themselves and/or cover different parts of the code when compared to vanilla KLEE. Therefore they seem to be an effective tool for non-seeded exploration in symbolic execution.

Comparing the search strategies for pending constraints, we observe that for these experiments DFS performs best overall, covering 11% more instructions than random path, which in turns covers 14% more instructions than depth-biased search.

One advantage of the pending constraints approach is that it usually spends less time solving queries that turn out to be infeasible. Figure 5 shows the time spent constraint solving queries that turn out to be infeasible relative to total constraint solving time. We averaged across all search strategies for brevity and clarity. For most benchmarks, pending constraints spend significantly less time constraint solving queries that are infeasible. The only exception is *SQLite3*, where the absolute time spent solving infeasible queries is still lower for pending constraints across all three search strategies.

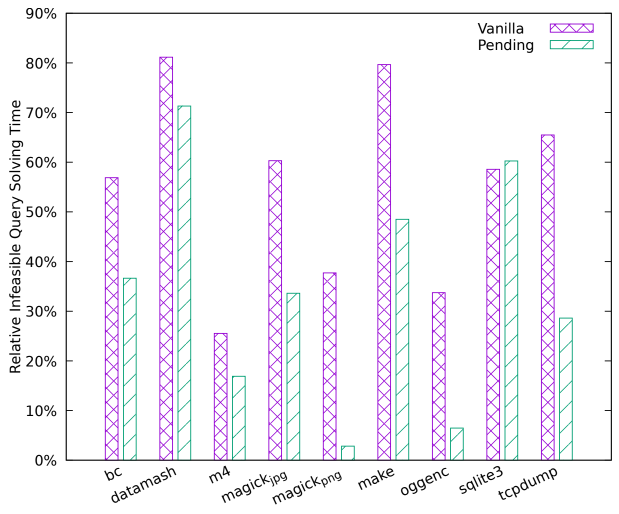
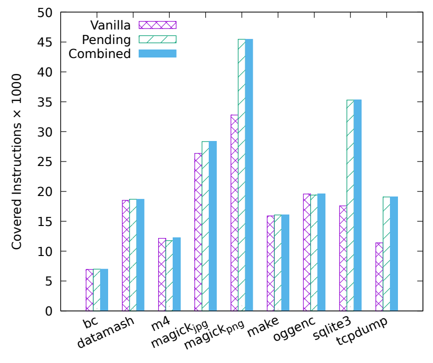


Figure 5: Relative time spent solving queries that were infeasible, averaged across all three search strategies.

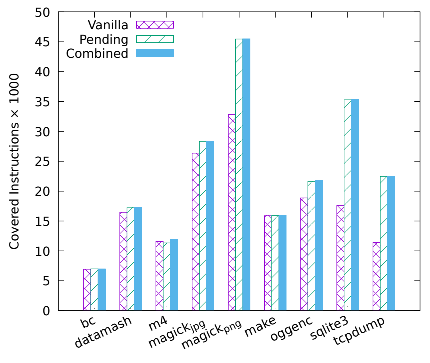
4.4 Seeded Experiments

To evaluate our approach in the context of seeding, we ran both vanilla KLEE and pending constraints for 2 hours with each seed from Table 1. Both versions are given the seeds—vanilla KLEE as assignment in the (counter-example) cache and additionally as concrete input.

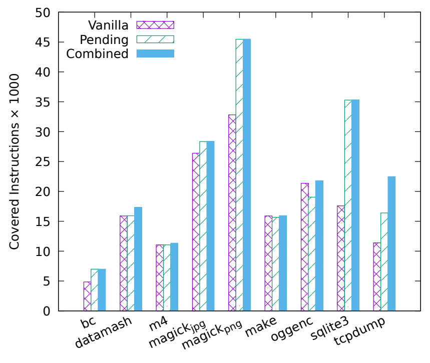
Figure 6 shows the coverage achieved by vanilla KLEE and pending constraints, for each search strategy. Coverage results for the two seeds are merged in each configuration.



(a) Random Path



(b) Depth-Biased



(c) DFS

Figure 6: Covered instructions by vanilla KLEE and pending constraints alongside their combination in seeded runs on our benchmarks. Each seed set is run for 2 hours in each configuration and the coverage results for the two seeds are then merged.

For some benchmarks, such as *magick$_{png}$*, *SQLite3* and *tcpdump*, pending constraints significantly outperform vanilla KLEE for all search strategies. For others, such as *datamash* and *make*, there does not seem to be a large difference. Finally, for *m4*, pending constraints perform slightly worse than vanilla KLEE. This is due to dereferences of symbolic pointers being very common in *m4*, which do not benefit from pending constraints.

From the combined coverage bars, we can observe there is very little complementarity between the two approaches, with the exception of *tcpdump* under DFS, where the combined coverage is significantly higher. In this case, both runs explore the seed, but pending constraints go deeper in the exploration. We discuss in more detail the advantages provided by pending constraints with seeding in the *SQLite3* case study of §4.6.

Overall, pending constraints cover 25% more instructions with random path search, 30% more with depth-biased search, and 23% more with DFS. In the pending constraints experiments, depth-biased search covered 2% more instructions overall when compared to random path search, which in turn covered 3.5% more than DFS.

Figure 7 shows the coverage achieved with only seed set 1 between our pending constraints approach and vanilla KLEE with the random path search strategy. For most utilities, we can draw similar conclusions as for the experiments of Figure 6 where the coverage for the two seed sets is merged. *oggenc* is an exception as it shows no coverage improvement on only seed set 1.

4.5 Pending Constraints with Relaxed Checks

Figure 7 also shows the coverage achieved with relaxed checks for critical instructions (Section 3.2). As can be seen for some tools like *bc* or *SQLite3*, relaxing these checks can lead to large increases in number of covered lines. In the case of *bc* the instruction coverage more than doubles. For most other utilities the increase is smaller, but not insignificant, with *oggenc*, *magick$_{png}$* and *tcpdump* being the notable exceptions showing no or small decreases in coverage.

These results show that pending constraints with relaxed checks can be used to effectively reach deeper into the program, but with the downside of errors being missed during exploration.

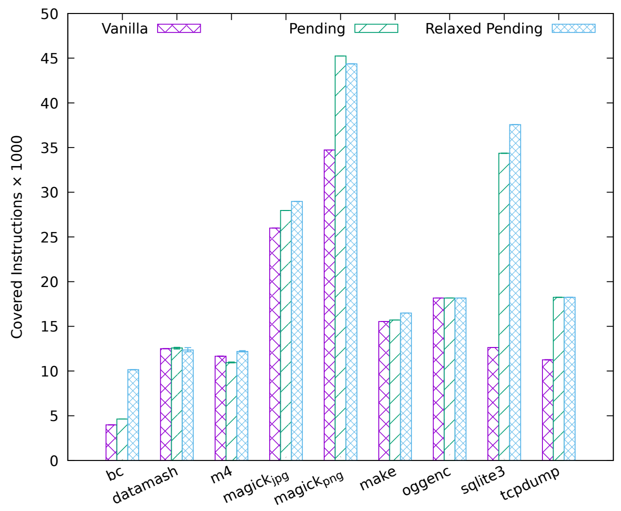


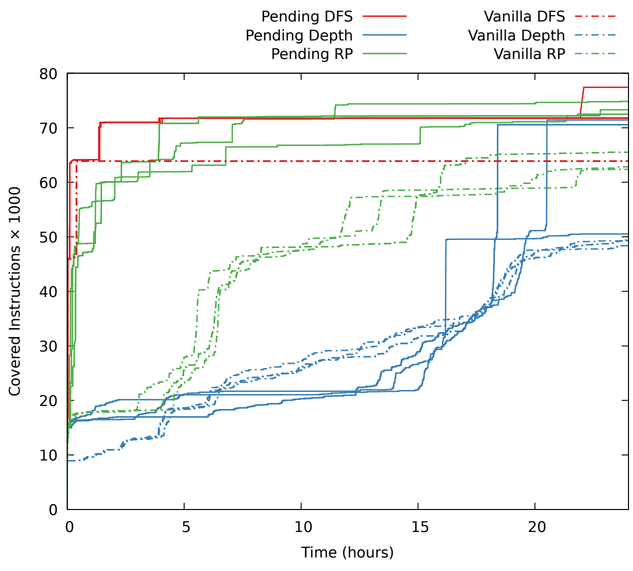
Figure 7: Covered instructions on seed set 1 of vanilla KLEE against pending constraints with both strong and relaxed checks with the Random Path search strategy.

4.6 Case Study: *SQLite3*

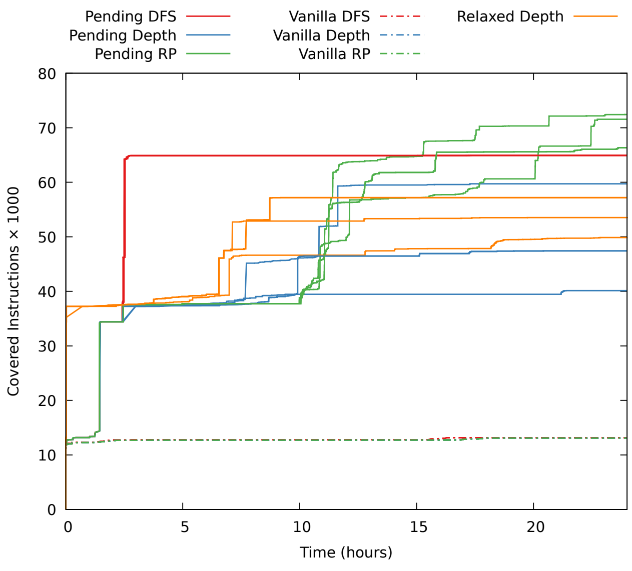
To provide more insight into our approach, we now discuss one of our benchmarks, *SQLite3*, in more detail. In particular, we look at the evolution of coverage on long-running experiments and examine various pairwise combinations of configurations to uncover how different seeding approaches and search strategies complement each other.

4.6.1 24-hour Non-seeded Runs. We first performed 24-hour experiments with *SQLite3* without seeding, with each of the three considered search strategies. Each experiment was repeated 3 times to account for the randomness of the depth-biased and random path strategies. The results are shown in Figure 8a.

DFS performs surprisingly well for this benchmark. However, with vanilla KLEE, it achieves no additional coverage after the first



(a) Non-Seeded



(b) Seeded

Figure 8: Coverage of *SQLite3* over a 24-hour run. The experiments were repeated 3 times, represented as lines of the same type. Both non-seeded and seeded runs are shown.

hour. By contrast, pending constraints with DFS continue to make progress and end up achieving the highest coverage overall.

We inspected the test inputs generated using DFS to understand why it performs so well. Vanilla KLEE with DFS appears to be lucky, as it manages to find two short keywords, such as `AS` and `BY`, which seem to be located close to the edge of the search space. These two keywords are not found by vanilla KLEE using the other two search strategies. Pending constraints with DFS does not find these short keywords, but instead finds longer ones such as `INSERT`, `FILTER`, `VACUUM` and `WINDOW`. This is not surprising, as the search space of pending constraints with DFS is often quite different from that of regular DFS.

Pending constraints with random path gain coverage quickly due to its broader exploration, but makes limited progress in the last 10 hours. Pending constraints with the depth-biased strategy make little progress for a long time, but gain a lot of coverage towards the end of the run, achieving coverage similar to that of random path. Vanilla KLEE makes steady progress with both search strategies, but does not overtake pending constraints.

4.6.2 24-hour Seeded Runs. We also performed 24-hour experiments with *SQLite3* with seed set 1, with each of the three considered search strategies. In addition, we also run in this setting pending constraints with relaxed checks, using the depth biased strategy. As in the non-seeded runs, each experiment was repeated 3 times.

Figure 8b shows the results. The seed covers 37.2k instructions. Vanilla KLEE performs similarly across the three search strategies (all the lines for the vanilla KLEE runs are at the bottom of Figure 8b) and never manages to complete the seeded path, stalling around 13k covered instructions. It makes very slow progress, covering less than 500 new instructions in the final 20 hours of its run. This is due to KLEE's eager feasibility checks, with large constraints that take a long time to solve.

Our pending constraints approach manages to cover all instructions on the seeded path. With error checks in place, it takes up to 3 hours to complete the seeded path, whereas with relaxed checks it only takes 4 minutes. There are about 303 memory bounds checks on the seeded path and solving the associated constraints accounts for the majority of the time difference between the configuration without and with relaxed checks. Note that 4 minutes is still significantly slower than the pure interpretation time of KLEE on this input. Our approach still performs some solving activities such as computing independent sets and substituting the seed in the constraints.

There is no difference between search strategies in the initial seeding phase as the exploration is guided by the seed. After pending constraints finish with the seeding, they make little progress for up to several hours. During this time they are attempting to revive pending states, most of which are infeasible and therefore killed, thus giving no additional coverage. Finally, depending on the search strategy, a feasible state is revived, leading to the execution of a different path, potentially giving large coverage gains.

The depth-biased search, which picks good states to revive in 2 out of 3 of our runs, achieves significant new coverage. However, random path search outperforms the depth-biased search in all of the 3 runs and seems to generally achieve higher coverage. Due to determinism of DFS, there is no difference between its runs. They all achieve coverage in-between the random path and depth biased search strategies.

We inspected the generated test inputs for the depth-biased, DFS and random path runs. The test inputs generated by the depth-biased runs were mostly very similar to the seed, with only a small

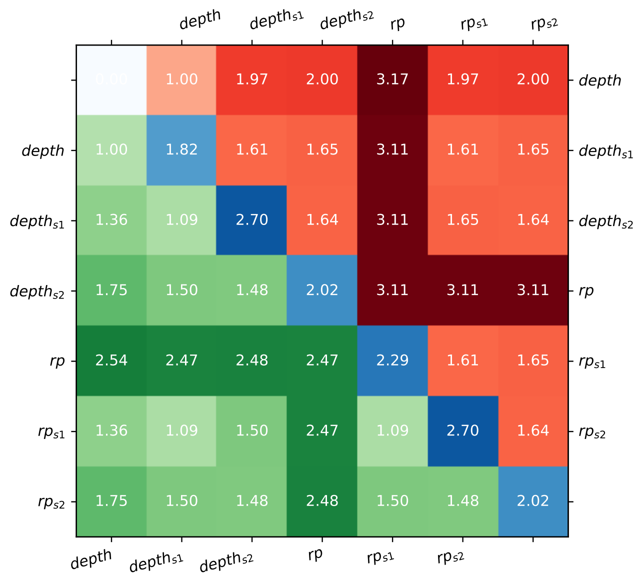


Figure 9: Heatmaps of coverage combination pairs for vanilla KLEE (lower triangle) and pending constraints (upper triangle) for *SQLite3*.

number of characters changed (e.g. `SELECT * TEMp ;. .` instead of the seed `SELECT * FROM t;`).[12] Even when it manages to find new keywords, such as `FILTER`, `HAVING`, `UPDATE`, `VACUUM`, and `WINDOW`, the test input is close to the initial seed (e.g. `uPdATE * FROM t;....`). Similarly, DFS-generated test inputs are almost identical to the seed, with only the last 2 characters changing in a depth-first fashion. The random path search strategy, on the other hand, also finds several more new keywords, such as `BEFORE`, `HAVING`, `INSERT`, `FILTER`, and `VACCUM`, but diverges early from the seed and finds additional keywords by exercising different code in the parser (e.g. `./.c....;expLain....`). This difference between the generated test inputs across the three search strategies is expected, as depth-biased search and DFS pick states towards the end of the path, where the seed has a greater impact.

4.6.3 Pairwise Coverage Combinations. Finally, we wanted to explore how the different combinations (non-seeded, seeded with seed sets 1 and 2, and different search strategies) complement each other. Therefore, we looked at the pairwise combinations of coverage for 2-hour runs of each configuration. We decided to omit showing the DFS runs in this case to illustrate the complementary aspects of seeding versus non-seeding runs more clearly. As shown in Figure 4c, DFS is really effective in *SQLite3* without seeds. The union of non-seeded DFS and other approaches cover only 10-100 more instructions than DFS by itself. However, we note that DFS is not always the best-performing strategy, as shown by other benchmarks.

[12]Inputs shorter than the specified symbolic input size are padded with `\0` and newlines (\n, \r) are shown as '.'

Analysing this amount of data is hard, so we devised the visualisation in Figure 9, which shows pairwise comparisons of combined coverage across the 6 different configurations for *SQLite3*. The labels indicate the search strategy used and the seed set is indicated in the subscript. Un-subscripted labels indicate non-seeding runs.

The figure consists of three different heatmaps. The upper right triangle (red) shows the coverage achieved by pending constraints. The lower left triangle (green) shows the coverage achieved by vanilla KLEE. The diagonal (blue) shows the ratio between pending states and vanilla for non-combined runs. The data in the two triangles is normalised with respect to their upper-left corner, that is the 2h depth-biased run without seeds. The hypotenuses of the triangles therefore show the non-combined coverage.

As this graphic is dense with information, we will walk through a couple of examples. Looking at the top-left corner of our triangles, we see that both pending and vanilla KLEE with depth-biased strategy have the value of 1.00—this is because everything is normalised to this value. However, the blue 1.82 value tells us that our pending constraints approach covered 82% more instructions than vanilla KLEE with depth-biased search strategy.

Now focusing on two configurations from the lower green triangle: depth-biased without seeds and depth-biased with seeds for vanilla KLEE. The coverage of the run with a seed from set 1 is 9% (1.09) higher than that of the non-seeded run, while the combined coverage of both achieve 36% (1.36) more coverage. This indicates that there is complementarity between the coverage achieved with and without a seed.

Looking at the same 2 combinations in the upper red triangle, we can see that with pending constraints, seeding with set 1 achieves 61% (1.61) more coverage than the non-seeding run. Furthermore, their combination achieves 97% (1.97) more coverage than solely the non-seeding run. That indicates again that there is complementarity between seeding and non-seeding runs as with vanilla KLEE.

The random path search strategy without seeding achieves more than twice the coverage of the depth-biased search strategy in both vanilla KLEE (2.47) and with pending constraints (3.11). However, pending constraints cover 129% (2.29) more lines than vanilla KLEE.

There is also some complementarity of coverage between depth-biased search and random path with pending constraints. Their union covers 217% (3.17) more instructions than just depth-biased as opposed to 211% (3.11) achieved by random path. Vanilla KLEE behaves similarly.

Finally, looking at the union of coverage between non-seeded and seeded runs, we can see that for pending constraints seeding complements well with depth-biased exploration, achieving over 30% points (1.61 to 1.97 and 1.65 to 2.00) more coverage when combined. With random path, we don't see any such complementarity (3.11). Vanilla KLEE follows a similar pattern.

4.7 ZESTI

ZESTI [18] is a promising extension of KLEE that combines EGT-style symbolic execution with seeding via regression test suites. Unfortunately, its original implementation was never upstreamed, partly because it is quite large and intrusive. In this section we show that pending constraints can be used to build a lightweight and effective version of ZESTI.

ZESTI consists of several parts. The two most important ones are the ability to use a variety of inputs as seeds and the so called ZESTI searcher, whose purpose is to explore paths around sensitive instructions. The idea of the ZESTI searcher is to take a single seed and execute it to completion, while recording sensitive instructions and divergence points. A divergence point is a symbolic branch where the path not taken by the seed is feasible. ZESTI then starts bounded symbolic execution from the divergence point closest to a sensitive instruction. It then moves to the next closest divergence point and so on.

Re-implementing ZESTI on top of pending constraints is straightforward. Consider a symbolic execution run with pending constraints after a single seed has been executed to completion. There are no normal states and many pending states representing the divergence points described above. The ZESTI searcher is implemented by considering pending and normal states separately. We prioritise normal states, but only if their depth does not exceed the depth of the last revived pending state plus some bound. Pending states are revived in the order of distance to sensitive instructions. This is equivalent to the ZESTI searcher and is easy to implement. For simplicity, our implementation currently considers only memory accesses as sensitive instructions.

We found the benchmarks used for evaluation to be too hard for ZESTI. For example, as seen in §4.6.2 it takes 3 hours to execute a simple seed with *SQLite3*. Thus running the whole test suite of *SQLite3* and exploring a significant amount of paths around a seed is impractical. Therefore, we chose three tools that KLEE can execute more comfortably: *dwarfdump*[13], *readelf*[14] and *tar*[15]. These are inspired by the original ZESTI paper [18], where we replaced *GNU Coreutils* with *tar* as recent modifications in the *Coreutils* build system make it harder to use with ZESTI.

To capture the seeds, we replaced each binary with a script and ran the application test suite. We then removed large seeds of over 8.1MiB to keep the execution time associated with an individual seed short. This resulted in 1273, 313 and 5 seeds for *dwarfdump*, *tar* and *readelf* respectively. Since we wanted to run each seed for 30 minutes as per the original ZESTI paper and keep the overall time under 12 hours, we ran at most 200 seeds per benchmark.

These experiments found one bug in *dwarfdump*[16] and one in *tar*[17] which have already been confirmed and fixed by the developers. Both of these bugs were found by the ZESTI searcher and were not triggered by the original seed. Vanilla KLEE seeded with the seed ZESTI mutated was not able to find these bugs with a 2-hour timeout.

5 RELATED WORK

Concolic executors such as CREST [2], DART [12] or SAGE [13] also drive each test input to completion, which is similar to the behaviour of pending constraints. However, these tools suffer from the disadvantages of concolic executors such as re-executing path prefixes and exploring a single path at a time. Our approach brings some of the strengths of concolic execution to EGT-style tools like KLEE while maintaining their advantages.

KLEE [4] has an existing seeding mechanism, which is also used by ZESTI [18] and KATCH [19]. However, when following a seed, KLEE eagerly performs feasibility checks at every symbolic branch, unlike pending constraints which defer these checks for when a pending state is revived. This in turn can have a big impact on coverage, as we have shown in §4.4.

KLUZZER [17], a whitebox fuzzer based on KLEE, implements a similar idea of delaying the satisfiability checks by only following the currently satisfied branch given by a seed. However, their approach goes further by trading off the benefits of EGT-style symbolic execution completely and reverting to concolic execution.

UC-KLEE [23] introduces *lazy constraints*. Here, the executor continues exploring paths although the underlying constraint solver can't determine the feasibility in a given amount of time. The corresponding expensive constraint is added as lazy constraint to the respective path condition and only evaluated when some goal is satisfied, e.g. a bug is found, to suppress false positives. This leads to more states in memory and thus to more solver queries but can also reduce the overall solver time as the additional constraints along such paths narrow down the solution space for the constraint solver. Our approach explores a different design point, which always favours states that are known to be feasible, either via caching or seeding, and when just pending states are left, they are only explored further if they are determined to be feasible.

Hybrid fuzzing approaches such as Driller [25], QSYM [28] or SAVIOR [9] pass concrete inputs between a greybox fuzzer and a symbolic executor. These approaches could directly benefit from pending constraints to achieve a tighter integration between fuzzing and symbolic execution.

6 CONCLUSION

We have presented pending constraints, a strategy that achieves a more efficient use of constraint solving and a deeper exploration of programs with symbolic execution. The key idea is to aggressively follow paths that are known to be feasible either via caching or seeding, while deferring all other paths by storing states with *pending constraints*. We implemented this approach in KLEE and evaluated it on nine hard benchmarks, including *make*, *SQLite3* and *tcpdump*. Our evaluation shows that pending constraints can significantly increase the coverage achieved by symbolic execution in both seeded and non-seeded exploration.

ACKNOWLEDGEMENTS

We thank Martin Nowack, Jordy Ruiz and the anonymous reviewers for their helpful feedback on the paper. This research has received funding from the EPSRC UK via a DTA studentship, the DSO National Laboratories, Singapore, and from European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 819141).

[13]https://www.prevanders.net/dwarf.html
[14]https://www.gnu.org/software/binutils/
[15]https://www.gnu.org/software/tar/
[16]https://www.prevanders.net/dwarf.html (28 June 2020 update)
[17]http://git.savannah.gnu.org/cgit/tar.git/commit/?id=dd1a6bd37a0d57eb4f002f01f49c51fa5c6bb104

REFERENCES

[1] Andrea Aquino, Giovanni Denaro, and Mauro Pezzè. 2017. Heuristically Matching Solution Spaces of Arithmetic Formulas to Efficiently Reuse Solutions. In *Proc. of the 39th International Conference on Software Engineering (ICSE'17)*.

[2] Jacob Burnim. 2020. CREST: A Concolic Test Generation Tool for C. https://www.burn.im/crest/

[3] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)*.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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)*.

[5] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*.

[6] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*.

[7] Cristian Cadar and Timotej Kapus. 2020. Measuring the coverage achieved by symbolic execution. https://ccadar.blogspot.com/2020/07/measuring-coverage-achieved-by-symbolic.html.

[8] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'12)*.

[9] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. 2019. SAVIOR: Towards Bug-Driven Hybrid Testing. arXiv:cs.SE/1906.07327

[10] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Crosschecking of Floating-Point and SIMD Code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)*.

[11] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Testing of OpenCL Code. In *Proc. of the Haifa Verification Conference (HVC'11)*.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*.

[13] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*.

[14] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*.

[15] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*.

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*.

[17] Hoang M. Le. 2019. KLUZZER: Whitebox Fuzzing on Top of LLVM. In *Automated Technology for Verification and Analysis (ATVA)*.

[18] Paul Dan Marinescu and Cristian Cadar. 2012. make test-zesti: A Symbolic Execution Solution for Improving Regression Testing. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*.

[19] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*.

[20] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*.

[21] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach. (April 2018).

[22] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences Between Software Versions. In *Proc. of the 38th International Conference on Software Engineering (ICSE'16)*.

[23] David A. Ramos and Dawson Engler. 2015. Under-constrained Symbolic Execution: Correctness Checking for Real Code. In *Proc. of the 24th USENIX Security Symposium (USENIX Security'15)*.

[24] Koushik Sen, Darko Marinov, and Gul Agha. 2005. 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)*.

[25] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. of the 23rd Network and Distributed System Security Symposium (NDSS'16)*.

[26] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)*.

[27] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. 2006. Automatically generating malicious disks using symbolic execution. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'06)*.

[28] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proc. of the 27th USENIX Security Symposium (USENIX Security'18)*.