Running Symbolic Execution Forever

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

Martin Nowack
Imperial College London
United Kingdom
m.nowack@imperial.ac.uk

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

ABSTRACT

When symbolic execution is used to analyse real-world applications, it often consumes all available memory in a relatively short amount of time, sometimes making it impossible to analyse an application for an extended period. In this paper, we present a technique that can record an ongoing symbolic execution analysis to disk and selectively restore paths of interest later, making it possible to run symbolic execution indefinitely.

To be successful, our approach addresses several essential research challenges related to detecting divergences on re-execution, storing long-running executions efficiently, changing search heuristics during re-execution, and providing a global view of the stored execution. Our extensive evaluation of 93 Linux applications shows that our approach is practical, enabling these applications to run for days while continuing to explore new execution paths.

CCS CONCEPTS

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

KEYWORDS

symbolic execution, memoization, KLEE

ACM Reference Format:
Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397360

1 INTRODUCTION

For testing real-world software systems, symbolic execution is often proposed as a method for thoroughly enumerating and testing every potential path through an application. While achieving full enumeration is usually impossible due to the fundamental challenge of the state-space explosion problem, even a subset of paths can be used to find bugs or generate a high-coverage test suite [4, 7, 14]. And typically, the more paths are explored, the better the outcome.

With the multitude of paths, performing symbolic execution on a modern machine quickly consumes all available memory. For instance, in Figure 1, we use the symbolic execution engine KLEE [4] to run 87 real-world applications from the *GNU Coreutils* suite with a timeout of 2 h, using the default memory limit of 2 GB. For more than two thirds of the runs (65 out of 87), KLEE prematurely terminates a substantial amount of paths as the given memory limit is reached. Each of those paths could have spawned a large number of new paths if exploration was allowed to continue. Even if the memory limit is increased to 10 GB, more than half of the benchmarks prematurely terminate at least 80% of the paths they started to explore. And worse, for some applications, the premature killing of paths causes KLEE to run out of paths entirely after a relatively short time. For example, with a limit of 2 GB, there are 14 applications where KLEE completely runs out of paths before the 2 h timeout. Therefore, for these benchmarks and configurations, no matter how much time one has at their disposal, KLEE won't be able to explore more than a certain number of paths.

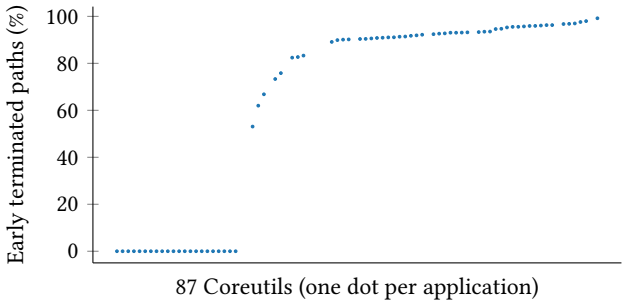


Figure 1: When running KLEE[1] on 87 *Coreutils* for 2 h each with the default search heuristic and memory limit (2 GB), most paths are terminated early due to memory pressure.

One solution for dealing with this problem is to store the paths being terminated early to disk and then replay them later incrementally. Previous work has proposed *memoized symbolic execution* [26], where executed paths are recorded to disk as a trie, and then paths of interest are brought back to memory on replay, reusing the recorded constraint solving results to speed up the re-execution. The approach was shown to be applicable to iterative deepening, regression analysis and coverage improvement. But it was applied to rather small Java applications (<5000 LOC) and short runs (on the order of minutes), and has the important limitation that the same search heuristic needs to be used during re-execution.

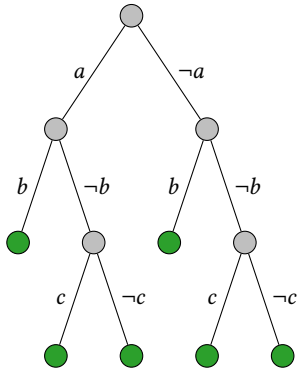
In this paper, our ambition is to build upon this idea to design a technique capable of running symbolic execution on large programs *indefinitely*, while continuing to explore new paths through the program using any search heuristic. We show that to scale up

[1] To generate this graph, we use our own extension of KLEE that implements memoization, but results are similar when using mainline KLEE.

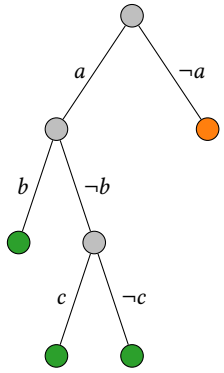
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '20, July 18–22, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397360

```
1  int main(void) {
2    int a, b, c; // symbolic
3
4    if (a) { ... }
5    else { ... }
6
7    if (b) { ... }
8    else {
9      if (c) { ... }
10     else { ... }
11   }
12 }
```

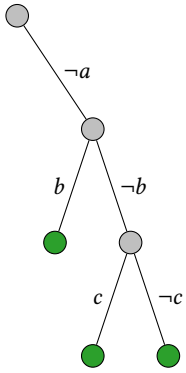
(a) Simple code example



(b) Fully explored tree



(c) Interrupted exploration



(d) Replay with path pruning

Figure 2: A simple code example and associated execution trees.

memoization to larger applications (tens of kLOC) and analysis times (hours and days), we need to overcome several research challenges:

(1) Real-world applications often interact with the environment. Changes in the environment between the original and replay runs are frequent, and without a robust detection of such changes (*divergences*), re-execution of memoized runs can lead to the exploration of infeasible paths.
(2) Long runs often involve millions of paths that need to be stored to disk. Storing these paths efficiently while keeping enough information to detect divergences caused by the environment is critical.
(3) Providing a global view of the stored execution tree is important in many applications, but restoring the whole memoized execution tree consisting of millions of nodes to memory is infeasible.
(4) Overcoming the restriction of using the same search heuristic during the original and replay runs is important, as it can allow one to be oblivious to the way in which the original tree was created, can speed-up replay, and can allow dynamically changing search heuristics as needed.

In this work, we propose a novel memoization approach for symbolic execution which is designed to overcome the research challenges discussed above. We implement our technique on top of the state-of-the-art symbolic execution engine KLEE [4] and perform an extensive evaluation in which we show that the technique can enable KLEE to run large applications for long periods of time while incurring acceptable space and time overheads.

In the remainder of the paper, we present our approach in Section 2, discuss its implementation in Section 3 and comprehensively evaluate it in Section 4. We then discuss related work in Section 5 and conclude in Section 6.

2 APPROACH

Symbolic execution aims to explore all (interesting) execution paths of a program by treating inputs as symbolic. When a symbolic execution engine reaches a conditional statement (e.g. an `if`) whose condition involves symbolic values (*a symbolic branch*), it forks the execution and continues to explore all possible paths. The branch condition and its negation are attached to the respective sides and form a unique *path condition* along each execution path. The set of all paths form an *execution tree*. An example program and its execution tree are shown in Figures 2a and 2b. Many symbolic execution engines maintain a trie-like data structure to store the execution tree in memory, where intermediate nodes encode symbolic branch feasibility and leaf nodes represent pending execution states. We use the term *execution state*, or simply *state* to denote the representation of a (pending) path in memory.

Constraint solving often incurs a significant computational cost, as it is heavily used to check branch feasibility, to verify safety properties and to generate test cases. Especially when testing applications repeatedly, these costs accumulate quickly as paths have to be re-executed and queries have to be re-solved.

In the following, we present a framework that significantly reduces re-execution times by memoizing solver results (§2.1) and pruning fully-explored subtrees from re-executions (§2.2). Divergence detection (§2.3) ensures that re-executed paths execute the same instructions as their recorded counterparts.

2.1 Overview

Memoization is a well-known technique to substitute run-time with storage costs, with results of computationally expensive operations stored and re-used later to avoid re-computations. Prior work [26] memoizes the sequence of choices taken during path exploration.

In our approach, we memoize the execution tree information differently. In general, every node has an *ID*, which is used to associate data with it, and knows the potential IDs of its direct children. We differentiate different node types of the tree, as shown in Figure 3. *Active nodes* are associated with a state, while all *intermediate nodes*—from the root node to an active node—represent symbolic branch decisions that must be made to reach the same state. Therefore, intermediate nodes only represent feasible decisions.

We annotate all *active nodes* with metadata necessary to re-execute a path from its last fork. Specifically, we store solver results, the number of executed instructions, symbolic branches, and basic block hashes to detect divergences (see §2.3), in addition to debug information and basic statistics. We write an active node's data to a relational database if the associated state is terminated or reaches

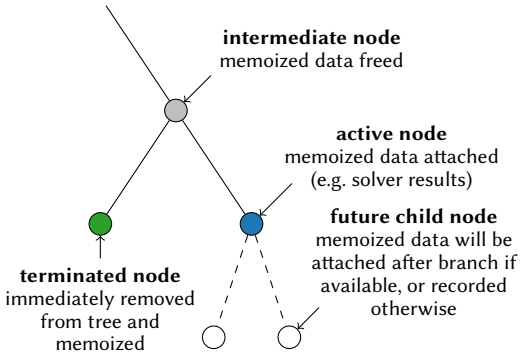


Figure 3: Subtree with different node types.

Algorithm 1 Satisfiability checking

```
1:  global Solver                                              ▷ SMT solver
2:  function isSAT(state, condition)
3:      if inReexecutionMode(state) then
4:          if state.node.solverResults.hasNext() then
5:              return state.node.solverResults.getNext()
6:          end if
7:      end if
8:      result ← Solver.isSAT(state.constraints, condition)
9:      state.node.solverResults.append(result)
10:     return result
11: end function
```

a symbolic branch. In the latter case, the active node becomes an intermediate node with its child nodes being the new active nodes. In either case, the stored data is associated with the node's ID, which on re-execution will provide fast selective access to parts of the execution tree. To keep the memory overhead low, we free the memoized data of intermediate nodes (see Figure 3).

2.2 Memoization and Re-execution

During re-execution, we need to associate the memoized data with the new run. A program starts with its initial state and its associated active node with ID 1. As it is a re-execution, data associated with this node can be loaded, particularly solver results up to the next symbolic branch. Such results include those associated with checks for buffer overflows and other errors, as well as queries for concretizing part of the symbolic input (e.g., when calling an external function).

Algorithm 1 shows the function for determining whether a *condition* is satisfiable in a given *state*. If we are re-executing that part of the code and results are memoized, we simply retrieve the next result from the node associated with *state* (lines 3–5). Otherwise, the underlying solver is called (line 8) and the result is recorded in the node (line 9).

The moment a state reaches a symbolic branch, the execution tree needs to be updated. Algorithm 2 shows the code responsible for forking the execution of the current state into up to two child states and the handling of the execution tree nodes. Specifically, function branch takes as input the current *state* and a branch

Algorithm 2 Branching

```
1:  function branch(state, condition)
2:      trueSAT ← isSAT(state, condition)
3:      falseSAT ← isSAT(state, ¬condition)
4:      bothFeasible ← trueSAT ∧ falseSAT
5:      if ¬bothFeasible then
6:          ▷ Only one side is feasible, we continue with current state
7:          if trueSAT then
8:              checkOrAppendBranch(state, true)                  ▷ §2.3
9:              return (state, NULL)
10:         else
11:             checkOrAppendBranch(state, false)                 ▷ §2.3
12:             return (NULL, state)
13:         end if
14:     else                                          ▷ both branches feasible
15:         checkOrSetFork(state)                                 ▷ §2.3
16:         trueState ← createNewState(state, condition)
17:         if state.node.trueID then
18:             trueState.node ← initFromDB(state.node.trueID)
19:         else
20:             trueState.node ← createNewActiveNode()
21:         end if
22:         state.node.trueID ← trueState.node.ID
23:         ...                                   ▷ similarly for falseState
24:         writeNodeToDB(state.node)
25:         freeData(state.node)
26:         return (trueState, falseState)
27:     end if
28: end function
```

condition that was encountered during execution and returns a pair of (*trueState, falseState*) as result, with either state set to *NULL* if that side of the branch is infeasible.

The algorithm assumes that *state.node* contains a reference to the corresponding execution tree node retrieved from the database. We first determine the satisfiability, in the current *state*, of the *condition* (line 2) and its negation (line 3). If only one side is feasible, we can continue to use the current state and active node and avoid updates of the database (lines 5–13).

If both branches are feasible (line 14), we create a new state (line 16). On line 17, we check whether the corresponding node has its *trueID* set, meaning that the child state where *condition* holds is already in the database. If this is the case, we associate the new state with the corresponding node from the database (line 18). Otherwise, if there is no existing node in the database associated with this state, we create a new node with a unique ID (line 20) and set the *trueID* of the node associated with the current *state* to point to it (line 22).

We repeat the same steps for the other child state (line 23), after which we write the updated *state.node* to the database (line 24), free it from memory (line 25) and return the child states (line 26).

Switching search heuristics. Prior work simply memoized the sequence of choices taken during path exploration [26]. Instead, we keep the information about the structure of the execution tree in the database (via the *trueID* and *falseID* relations), which makes our approach completely independent of the chosen exploration

strategy.[2] That is, exploration strategies can vary between the original and re-execution runs. For instance, an interrupted depth-first exploration (Fig. 2c) can be re-executed and completed with a breadth-first traversal (Fig. 2b) without using the constraint solver for the previously explored subtree.

This is useful for various reasons. Firstly, it allows the memoized part of the execution tree to be re-executed much faster. For instance, one might want to use a search heuristic that tries to reach uncovered code during the initial run. However, such heuristics are expensive (e.g. as they may involve shortest paths algorithms), and it would be wasteful to use them during re-execution. Instead, one could use a lightweight heuristic such as depth-first search (*DFS*) for the memoized parts of the execution, switching to a more effective but expensive heuristic for non-memoized parts.

Secondly, changing the search heuristic can help alleviate memory pressure on re-execution. Suppose that an initial execution is performed using breath-first search (*BFS*)—which consumes a lot of memory—and then saved to disk. If *BFS* is used again during re-execution, the same amount of memory would be used. Instead, one could use a heuristic such as *DFS* which has a small memory footprint during re-execution and only switch to different heuristics for the non-memoized parts.

Thirdly, search heuristics can be used to limit the re-execution to interesting paths that might lead to uncovered code or select narrow subtrees for iterative deepening. While prior work on memoized symbolic execution [26] could also accomplish that, it did so by statically marking nodes for re-execution, which required bringing the entire execution tree into memory.

Global view and memoization across runs. Besides being exploration-strategy-agnostic, a big advantage of our memoization framework is the persistent global view of all runs. With every re-execution, newly explored paths are added to the tree stored on disk and provide a more complete picture of the tested application over time. Such a view allows one to reason about an application more thoroughly. Moreover, knowing which paths have been explored, re-executing them to evaluate different properties of an application becomes easier. Metadata of paths that were not fully explored during re-execution is kept, and new paths explored during re-execution automatically start recording new metadata as soon as they progress beyond memoized data. In this way, the execution tree stored on disk can grow across multiple re-execution runs, as we show in our experimental evaluation.

Path pruning. An optimisation for re-execution runs is it to remove (prune) all fully-explored subtrees from the exploration. Therefore, we extend the set of metadata to record the termination type for each path (e.g. program exit, bug found, interrupted due to memory pressure) and propagate this information to the parent nodes when a state is terminated. During re-execution, when a path branches, only a single value per branch needs to be checked to determine if its subtree solely contains fully-explored paths. If this is the case, the corresponding branch is terminated (its metadata still stored in the database). That way, only interrupted paths are re-executed, as illustrated in Figure 2d.

[2]We use the terms *search heuristic* and *exploration strategy* interchangeably.

```
1:  symbolic a
2:
3:  if a > 10 then          ▷ Spawned state1 follows direction "true"
4:    ...
5:  end if
6:
7:  if a > 5 then
8:    ...                   ▷ only direction "true" feasible in state1
9:  end if
10:
11: if a > 100 then         ▷ both directions feasible in state1
12:   ...                   ▷ state2 spawned, following direction "true"
13: end if
```

Figure 4: Code fragment to illustrate divergence detection.

2.3 Divergence Detection

Modern symbolic execution engines mix concrete and symbolic execution [5]. That is, the program under test is allowed to interact directly with the environment, e.g. by calling uninstrumented libraries or performing OS system calls. But during re-execution, changes in the environment—e.g. file timestamps or amount of available memory—can make the path deviate from the original path. Furthermore, because symbolic executors often interleave the execution of different paths, changing the order of execution during re-execution—e.g. by using a different search heuristic—can lead to a different exploration. The reason is that for performance, the execution of single paths is not fully isolated and as such leakage can happen between paths. Finally, symbolic executors often add their own sources of non-determinism, e.g. by using hash tables indexed by concrete memory addresses which may vary across executions.

One key property of symbolic execution is that only feasible paths are explored. To preserve this property, it is necessary that a re-executed path executes the same instruction sequence as its corresponding memoized path. Otherwise, any *divergence* could change the set of collected path conditions and hence invalidate memoized solver results, or even re-use solver results in wrong code locations. In that case, the symbolic executor would explore infeasible paths, potentially leading to false alarms.

To prevent such cases, we memoize and compare against additional safeguarding information: the instruction count, a hash sum of traversed basic blocks, and a vector of symbolic branch directions. Specifically, each execution tree node keeps information summarising the execution between the time the node was first created and the time its associated state branches into two child states. This information consists of:

(1) A vector of branch directions (*true, false*), which starts with the direction that was followed by this state, followed by zero or more entries for any symbolic branches encountered where only one direction was feasible. To make things concrete, consider the code in Figure 4. At line 11, the vector of branches for *state1* is *[true, true]* because the state is spawned as a *true* state on line 3 and only the *true* branch is feasible at line 7.

Algorithm 3 Divergence detection

```
1:  function updateSafeguardingData(state)
2:      state.node.instructions ← state.instructionCounter
3:      state.node.bbHash ← state.bbHash
4:      state.node.branches ← state.branches
5:  end function
6:
7:  function checkOrAppendBranch(state, branch)
8:      if inReexecutionMode(state) then
9:          if branch ≠ state.node.branches.getNext( ) then
10:             markStateAsDiverging(state)
11:             updateSafeguardingData(state)
12:             state.node.branches.append(branch)
13:         end if
14:     else
15:         state.node.branches.append(branch)
16:     end if
17: end function
18:
19: function CheckOrSetFork(state)
20:     if inReexecutionMode(state) then
21:         i ← (state.node.instructions = state.instructions)
22:         h ← (state.node.bbHash = state.bbHash)
23:         b ← ¬state.node.branches.hasNext( )
24:         if ¬i ∨ ¬h ∨ ¬b then
25:             markStateAsDiverging(state)
26:             updateSafeguardingData(state)
27:         end if
28:     else
29:         updateSafeguardingData(state)
30:     end if
31: end function
```

(2) The number of instructions that were executed before the state forked into two states. In Figure 4, for *state1* this would be the number of instructions executed by that path up to the branch at line 11.
(3) A cumulative hash of all the basic blocks that were executed before the state forked into two states. In Figure 4, for *state1* this would be a hash of all the basic blocks executed from the start of the program up to and including line 11. A state's basic block hash gets updated whenever a new basic block is reached and relies on a pre-computed map that contains hash sums for all basic blocks.

Function updateSafeguardingData in Algorithm 3 shows how we track this data as part of the state and propagate it in lockstep with the associated execution tree node. As part of the memoization, this data is stored when the tree node is written to the database (Alg. 2, line 24).

During re-execution, the run needs to be crosschecked against the information on disk, ensuring that the same branches are taken. The key is to validate this data efficiently. Algorithm 3 shows function checkOrAppendBranch, which is called if a symbolic branch has a single outcome (Alg. 2, lines 8 and 11). If the state is in re-execution mode, it checks whether the same memoized branch is taken (line 9). If not, it marks the state as diverging (line 10), meaning that all memoized information is discarded and the state continues in recording mode. We then update the safeguarding data (line 11) and append the *branch* to the vector of branches (line 12). In recording mode, the function simply appends the branch to the vector of branches (line 15).

When a state is about to fork into two feasible branches (Alg. 2, line 14), function checkOrSetFork from Algorithm 3 checks in re-execution mode whether the same number of instructions have been executed until that point (line 21), the same basic block hash has been computed (line 22) and all recorded branches up to the fork have been consumed (line 23). If not all checks pass (line 24), the state is marked as divergent (line 25) and the safeguarding data is updated (line 26). In recording mode, the function simply updates the safeguarding information associated with the database node (line 29).

Finally, we also perform similar checks when a state terminates, but for space reasons we do not show them in Algorithm 3.

Limitations. The algorithm does not detect all divergences. First, it misses rare cases where hash collisions occur, and second, it is not able to detect diverging behaviour that does not change the control flow. An example of the latter is given in Figure 5. When an execution is recorded with an external value $ext = 0$, a re-execution with $ext \geq 10$ re-uses the stored feasibility check results and incorrectly follows the now infeasible `true` branch. Nevertheless, such situations are quite rare and it is unlikely for our algorithm to completely miss a divergence, given that we perform checks at every single symbolic branch.

```
1: external ext                    ▷ concrete
2: symbolic sym                    ▷ range [0..9]
3:
4: if sym + ext < 10 then
5:     output("true")
6: else
7:     output("false")
8: end if
```

Figure 5: Code to illustrate divergence detection failure.

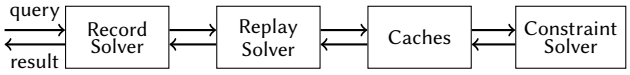


Figure 6: KLEE's main solving chain elements when memoization is used.

3 IMPLEMENTATION

We implemented our memoization framework, MoKlee, on top of KLEE [4] version 1.4,[3] a modern symbolic execution engine for LLVM [16] bitcode. The framework consists of three components: a persistence layer for KLEE's execution tree (*process tree*) that stores and loads arbitrary metadata transparently, and two new stages in KLEE's solver chain (Figure 6). The `Record Solver` appends solver results to nodes whenever paths progress into unmemoized

[3] Some bug fixes and improvements were backported from version 2.0

subtrees and its counterpart, the `Replay Solver`, returns memoized results to the engine. Although both solvers complement each other, they can be used independently.

As shown in Figure 6, KLEE uses caching in front of the actual constraint solver to reduce solving time. It might seem beneficial to place the `Replay Solver` behind the caches to populate them on replay. In practice however, accessing the caches can be expensive—for instance, a 7-day run of *split* from *Coreutils* spends more than 99% in the caches. MoKLEE still supports both solver placements but it is an essential design decision to override the caches for faster re-execution times. Progressing states still benefit from the fact that caches typically fill quickly.

Another important implementation aspect concerns solver queries that contain memory addresses. Our framework does not implement means for deterministic memory allocation and forwards affected queries to the solving chain without memoizing them.

A large part of the engineering effort also went into other parts of KLEE, to make it more reliable on long-running experiments. For instance, initially our implementation was not capable of re-executing even 1% of the instructions of *readelf*. The reason for that was its heavy use of function pointers and KLEE's non-deterministic assignment of these addresses. KLEE internally re-uses addresses of the corresponding LLVM function objects which are allocated differently after restarting KLEE. We modified KLEE in such a way that it assigns the same addresses in every run.

Some mitigation strategies are already implemented in KLEE, such as providing a fixed set of environment variables to the program under test and, most notably, deterministic memory. This mechanism does not ensure that pointers have the same value in re-executions, but at least have the same ordering. Internally, KLEE pre-allocates a memory map and incrementally assigns memory from that map for program allocations. The disadvantage of the deterministic mode is that allocated space is never freed, rendering it impractical for many applications. Hence, we refrained from using this mode although it reduces the occurrences of divergences significantly.

4 EVALUATION

We have extensively evaluated our approach on 93 Linux applications. The evaluation section is structured as follows. Section 4.1 discusses the experimental setup, Section 4.2 presents the runtime savings achieved during re-execution, Section 4.3 reports the space and runtime overhead of our framework, and Section 4.4 discusses divergences and their impact in re-executions. Finally, Section 4.5 shows how our framework enables effective iterative deepening with symbolic execution, and Section 4.6 how it allows applications to run for days in a row.

An artifact with our experiments is available at https://srg.doc.ic.ac.uk/projects/moklee/.

4.1 Experimental Setup

For our experiments, we use a set of homogeneous machines with Intel Core i7-4790 CPUs at 3.6 GHz, 16 GiB of RAM, and 1 TB hard drive (7200 rpm). All experiments run in Docker containers that use the same operating system as the host machines (Ubuntu 18.04). If not stated otherwise, we use KLEE's default memory limit of 2 GB. MoKLEE is built against LLVM 3.8 and uses Z3 [8] as SMT solver with a timeout of 10 s.

As benchmarks, we selected a variety of different applications: from the GNU software collection we selected the *Coreutils* suite, *diff*, *find* and *grep*. These are non-trivial systems applications used by millions of users. With *libspng*, *readelf* and *tcpdump*, we additionally selected applications that process more complex input formats such as images, binaries and network packets.

In more detail, the applications are:[4]

GNU Coreutils [10] (version 8.31; 66.2k LOC) provide a variety of tools for file, shell and text manipulation. Used in the paper introducing KLEE [4], they have become the de-facto benchmark suite for KLEE-based tools. The suite consists of 106 different applications, from which we excluded five tools that are very similar to *base64* (*base32*, *sha1sum*, *sha224sum*, *sha384sum*, *sha512sum*) and one tool (*cat*) whose execution with KLEE runs out of memory after a few seconds due to large internal buffers in recent versions. Furthermore, we excluded 13 tools (*chcon*, *chgrp*, *chmod*, *chown*, *chroot*, *dd*, *ginstall*, *kill*, *rm*, *rmdir*, *stdbuf*, *truncate*, *unlink*) that behave non-deterministically under symbolic execution, as executing paths through these tools can affect the execution of other paths or even Docker's and KLEE's behaviour without additional isolation in place (which is possible, but whose implementation is orthogonal to the goals of the project). This leaves us with a total of 87 tools. As in the original KLEE paper [4], we patched *sort* to reduce the size of a large buffer. *Coreutils* link against the GNU Portability Library (as do *diff*, *find* and *grep* discussed below). We use the same revision of this library (git `#d6af241`; 484.5k LOC) in all applications.

GNU diff from GNU Diffutils [11] (version 3.7; 7.9k LOC) is a command-line tool to show differences between two files.

GNU find from GNU Findutils [12] (version 4.7.0; 15.7k LOC) searches for files in a directory hierarchy.

GNU grep [13] (version 3.3; 4.1k LOC) searches for text in files that matches specified regexes.

GNU readelf from GNU Binutils [9] (version 2.33; 78.3k LOC and 964.4k of library code[5]) displays information about ELF object files.

libspng [18] (git `#2079ef6`; 4k LOC) is a library for reading and writing images in the Portable Network Graphics (PNG) file format. We wrote a small driver (24 LOC) that reads a symbolic image and links with the zlib [23] compression library (version 1.2.11; 21.5k LOC).

tcpdump [24] (version 4.9.3; 77.5k LOC) analyses network packets. We link against git `#f030261` of its accompanying *libpcap* library (44.6k LOC).

4.2 Speed of Re-execution

MoKLEE implements two mechanisms to reduce runtime costs of re-executions: 1) memoization of constraint solver results, and 2) path pruning. To evaluate their effectiveness, we run MoKLEE in recording mode on our 93 benchmarks for 2 h and measure the time it

[4] Lines of code (LOC) are reported by *scc* [1] and are for the whole application suites from which our benchmarks come, as deciding which lines of code "belong" to an individual application is difficult, if at all possible.

[5] We only consider lines in the `binutils` folder and the following dependencies: *libbfd*, *libctf*, *libiberty*, and *libopcodes*.

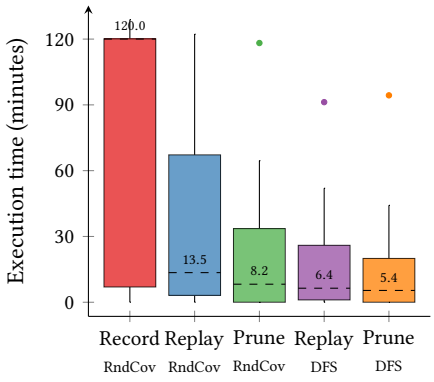


Figure 7: Distribution of execution times for 37 Coreutils, when *RndCov* is used during the initial run.

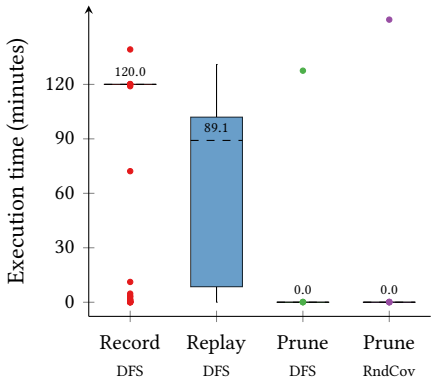


Figure 8: Distribution of execution times for 74 Coreutils, when *DFS* is used during the initial run.

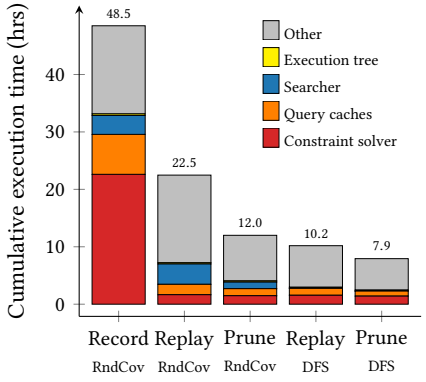


Figure 9: Cumulative execution times for different subsystems of MoKlee for the runs in Figure 7.

takes to re-execute the same execution trees. As the selected exploration strategy has an impact on the effectiveness of each mechanism, we evaluate our implementation with two different heuristics: the memory-friendly depth-first search (*DFS*) and KLEE's default strategy, a combination of random path selection and coverage-guided search (*RndCov*).

We set the memory limit to 2 GB; if this limit is reached, states are terminated prematurely to stay within it. Although the memory overhead of our framework is small, it might be enough to trigger an earlier state termination during a full replay with a different heuristic. When this happens, fewer paths are replayed, making it meaningless to compare the speed of re-execution. To mitigate the problem, we slightly increase the memory limit of re-execution runs from 2 GB to 2.2 GB and terminate states as soon as they are fully replayed. (Note that we only do this for the purpose of this experiment, none of those adjustments are needed when using MoKlee in a real setting, where one would typically only restore states of interest instead of performing a full replay.)

Coreutils with *RndCov* during recording. We first memoize 87 Coreutils with the default exploration strategy (*RndCov*) and re-execute these runs with path pruning disabled (*replay*) or enabled (*prune*) using each of the two search strategies (*DFS*, *RndCov*). To keep different re-executions comparable, we remove all tools that diverge or terminate states early due to memory pressure in at least one of the runs.

The results for the 37 remaining tools are shown in Figure 7. The median execution time decreases significantly from 120 min, our chosen timeout, to 13.5 min when re-executed with the same exploration strategy and even further to 8.2 min when path pruning is enabled. 11 applications that terminate within our time limit benefit most from path pruning: the whole tree is pruned and not a single instruction needs to be re-executed. When the less expensive *DFS* exploration strategy is used during re-execution, the median execution times are further reduced to 6.4 min and 5.4 min respectively. The outlier in this graph is *yes*, whose core functionality is an infinite loop with no constraint solving, and thus does not benefit from memoization (including path pruning, due to the broad exploration of the *RndCov* heuristic used during recording).

Coreutils with *DFS* during recording. We repeat this experiment using *DFS* in the recording run. As before, to keep different re-executions comparable, we remove all tools that diverge or terminate states early due to memory pressure in at least one of the runs. The results for the 74 remaining tools are shown in Figure 8. Note that we do not report results for the *RndCov* re-execution without path pruning: while *DFS* shapes an execution tree that consists of large fully-explored subtrees along an active path, a *RndCov* exploration without path pruning of that same tree creates too many states, and KLEE terminates most of them due to memory pressure.

Without path pruning, recording and replaying the execution tree with *DFS* results in lower savings than when *RndCov* is used in both stages: this is because the DFS run spends less time solving constraint queries. However, when path pruning is used, the re-execution time is usually under a minute, as the shape of the DFS-generated execution tree is ideal for pruning: re-executions only need to follow the active path and can prune all adjacent subtrees, reducing the re-execution time significantly.

The outlier taking more than 2 h during recording is *split*; we are currently investigating the reasons. The outlier that exceeds the timeout of 2 h during re-execution is *shred*. This application ends up in a tight loop in *DFS* mode, creating more than 1.9×10^{10} instructions with a negligible amount of constraint solving (0.1%).

Impact on MoKlee components. To show which components of MoKlee benefit most from the memoization during re-execution, we break down the cumulative execution time of each experiment configuration into the time spent in each component.

The results are shown for *RndCov* in Figure 9 and for *DFS* in Figure 10. As expected, memoization significantly reduces the time spent in constraint solving and also the time spent in the query caches, as our framework is placed in front of the solving chain. Comparing Figure 9 and 10 one can see that *RndCov* spends significant time in the searcher, due to its expensive coverage calculations and tree traversals. By contrast, *DFS* spends insignificant time in the searcher.

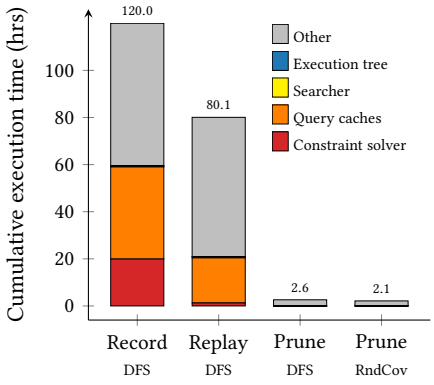


Figure 10: Cumulative execution times for different subsystems of MoKLEE for the runs in Figure 8.

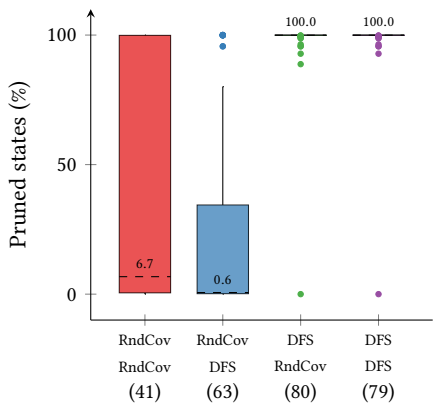


Figure 11: Distribution of pruned states across all path-pruning runs (number of *Coreutils* in each experiment in parentheses).

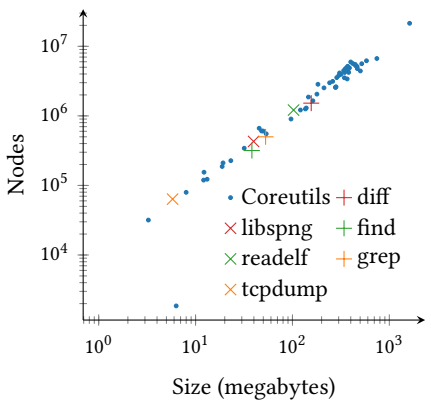


Figure 12: Execution tree sizes on disk for 66 runs that reach the 2 h limit. Sizes range between 3.10 MiB (*pathchk*) and 1.50 GiB (*echo*).

Finally, we report the distribution of pruned states for all re-executions in Figure 11. The more states can be pruned, the fewer instructions have to be re-executed. Obviously, the set of pruned states in a *complete* re-execution is independent of the search strategy used during re-execution. The difference between *RndCov*/*RndCov* and *RndCov*/*DFS* is merely caused by the different sets of non-diverging applications. The outlier in the experiments with a recorded *DFS* execution is *yes*. By executing an endless loop, *yes* creates a very deep path that never terminates and hence can't be pruned.

Non-Coreutils applications. For these applications, we memoize 2 h runs only with *RndCov* and use both exploration strategies on re-execution. Table 1 shows for each run the number of recorded instructions, the time needed by the re-execution, the number of instructions successfully replayed and the number of diverging states.

libspng and *readelf* have no divergences. Here, our implementation significantly reduces the execution time to between 1.39% and 15% of the initial execution time.

diff, *grep* and *tcpdump* suffer from divergences, but MoKLEE is able to re-execute most of the instructions in the recorded runs. In all three cases, more than 99% of the instructions could be replayed, in a time ranging from 3.45% to 25.76% of the initial execution time.

find has several thousand divergences, and thus MoKLEE is only capable of replaying 78.07% (79.27%) of the instructions in 9.87% (24.29%) of the initial execution time for *DFS* (*RndCov*). We will discuss this case and possible mitigations in Section 4.4.

4.3 Space and Runtime Overhead

Memoization does not come for free: execution tree nodes use additional memory during runtime, large execution trees require significant disk space, and there is additional computational overhead for managing the execution tree and the database. A big advantage of our implementation is the trie-like structure: states share common prefixes that are only stored once in memory and on disk.

Fortunately, the additional memory overhead of memoization is negligible. As discussed before, paths that are skipped by our path-pruning mechanism are never attached to the execution tree and terminated subtrees are freed from memory immediately. Every intermediate node (Fig. 3) consumes 24 B extra per node and active and leaf nodes require an additional 208 B without any progress. This includes some statistics and debugging information. The exact amount gathered throughout execution in leaf nodes depends on the number of queries issued and the number of symbolic branches taken by the respective state. Each memoized satisfiability query result requires two bits. For divergence detection, we store one bit for the single outcome of a symbolic branch point.

The required disk space is roughly linear to the number of stored nodes, as tree nodes only keep enough information for a state to reach the next node. Figure 12 shows the storage sizes for all

Table 1: Results for re-executions of memoized runs (2 h, *RndCov*) with different exploration strategies and path pruning disabled. Although some tools have diverging states, most instructions could be re-executed successfully.

Tool	Search	Recorded Instrs (M)	Re-execution Time (%)	Re-execution Instrs (%)	Diverging states
diff	*DFS*	65.1	22.01	99.13	64
diff	*RndCov*	65.1	25.76	99.27	60
find	*DFS*	1,105.3	9.87	78.07	9,518
find	*RndCov*	1,105.3	24.29	79.27	8,633
grep	*DFS*	32.5	4.45	99.99	3
grep	*RndCov*	32.5	5.86	99.99	3
libspng	*DFS*	22.0	1.39	100.00	0
libspng	*RndCov*	22.0	2.44	100.00	0
readelf	*DFS*	99.4	9.01	100.00	0
readelf	*RndCov*	99.4	15.00	100.00	0
tcpdump	*DFS*	10.0	3.45	99.96	2
tcpdump	*RndCov*	10.0	3.68	99.95	3

RndCov memoization runs from Section 4.2 which reached the 2 h timeout. Sizes vary between 3.10 MiB (*pathchk*) and 1.50 GiB (*echo*). On average, a single node requires 85 B of disk space.[6] The outlier in the bottom-left corner of Figure 12 is again *yes*. The core of *yes* is basically a tight print loop guarded by a symbolic condition with one feasible branch. As a result, it branches fewer than 1000 states in 2 h but some of the corresponding nodes have to store more than 100 million branch decisions for divergence detection. However, this type of behaviour is not representative of other applications.

To measure the runtime overhead, we use the *DFS* re-executions without path pruning (*replay*) from Figure 10, where *DFS* was also used during recording. This is close to the worst case for memoization: the paths reached with *DFS* are typically deeper such that more intermediate nodes have to be memoized and stored to disk more often. Still, the computational overhead for a path explored via *DFS* is typically smaller as the cache utilisation for constraint solving is much higher reducing the overall solving costs while at the same time the search heuristic has almost no overhead. This leaves only little room to offset performance using memoization.

In the first experiment, we measure the cumulative time spent in managing the metadata for memoization during runtime and time spent in reading and writing the execution tree to disk (I/O) for both the recording and replaying runs. Figure 13 shows the distribution of these times relative to the overall execution times. As can be seen, this time is never higher than 2% during recording and replay, with the median values of only 0.19% and 0.36% respectively. Moreover, our experiment setup uses traditional rotating hard disks. The overhead could be reduced further by using faster solid-state drives (SSDs).

To quantify the impact of I/O operations, we perform a run identical to the recording run, except that we disable storing the tree to disk. We keep the 42 applications that behave in the same way when memoization is disabled. Memoization without I/O adds a median overhead of only 1.22%, which is similar to what we observed when measuring the extra time spent writing the execution tree to disk in recording mode. The overhead in this experiment varies between -1.9% and 5.5%. The negative overhead is likely due to some measurement noise arising from running our experiments on a cluster of machines and any unexpected influences of our implementation on machine-level caches.

4.4 Divergences

In this section, we assess how common divergences are and how they affect re-execution. We first note that some applications suffer from divergences regardless of the exploration strategy used. Many of these applications rely on the execution environment. For instance, they print the system time (*date*), show free (*df*) or used (*du*) disk space or the time for which the system has been running (*uptime*).[7]

We again re-use the experiments from Section 4.2, with path pruning disabled and consider all four combinations of exploration strategies between recording and re-execution runs. We remove applications that on replay terminate states due to memory pressure, as our focus here is on runs where the replay would incorrectly follow infeasible paths due to divergences, rather than the case where different paths are executed due to memory pressure.

Coreutils. Figure 14 shows the data for *Coreutils*. The figure plots the distribution of the number of *lost instructions*, which are instructions in subtrees that were discarded when a divergence occurred. Execution trees resulting from *RndCov* explorations are typically less deep than their *DFS* counterparts. This means that in case of divergences only small incomplete subtrees are removed for *RndCov* instead of large fully-explored subtrees for *DFS*. Figure 14 clearly shows this difference: whereas a *RndCov*/*RndCov* combination loses only a small number of instructions (median 6.22%), a *DFS*/*DFS* run not only has a higher median value (27.61%) but also a much larger maximum (99.95% vs. 61.38%).

libspng and **readelf** do not encounter divergences.

diff, **grep**, and **tcpdump** have a small number of diverging states, as shown in Table 1. Despite these divergences, more than 99% of instructions could be re-executed successfully.

find suffers most from divergences and only re-executes 78.07% of instructions with *DFS* and 79.27% with *RndCov*. Most of the divergences occur in its `memmove` function, which we suspect are due to different memory allocation schemes between the record and replay runs. To confirm this hypothesis, we re-ran *find* with KLEE's deterministic allocation mode (`-allocate-determ`, see §3 for a discussion of why this mode is often impractical), with an additional 1 GB of memory. As expected, the number of re-executed instructions increased significantly, to 96.96% for *DFS* and 99.17% for *RndCov*.

Finally, we re-emphasise the fact that divergences have a much worse impact than losing memoized instructions. Without detecting them, symbolic execution can start exploring infeasible paths, potentially wasting a lot of time without making any meaningful progress and even generating false positives in the process.

4.5 Iterative Deepening

A natural use case for memoization is iterative deepening, which is able to mimic *BFS* exploration with much lower memory consumption by *repeatedly exploring* a program with *DFS* up to a certain (increasing) depth in the execution tree.

For this experiment, we select and run the set of non-*Coreutils* applications with *BFS* exploration until KLEE's default memory limit of 2 GB is exceeded. At this point, we cannot run in *BFS* mode anymore without losing paths. Instead, we employ iterative deepening, making use of the memory-friendly *DFS* in each iteration. Two of the benchmarks, *diff* and *readelf*, had very long runtimes and we decided to reduce the solver timeout in both cases to 1 s. Also, we excluded *find* from the selected benchmarks because of its large number of divergences (see Table 1).

As starting depth values for iterative deepening, we choose the minimum tree depths for which states are terminated due to memory pressure ($d = 22$ for *diff*, 13 for *grep*, 42 for *libspng*, 22 for *readelf* and 19 for *tcpdump*). We iteratively increase the depths by five more levels (to $d+5$), once without and once with memoization (including path pruning).

[6] For the sake of simplicity we measure the database file size including metadata.

[7] Although our experiments run in Docker containers on a set of homogeneous machines using the same sandbox directories, we did not try to fully virtualize the execution to provide identical execution environments. Such virtualization could reduce the number of divergences but it would necessitate significant engineering effort and would not take advantage of the mixed concrete-symbolic execution paradigm at the core of dynamic symbolic execution [5].

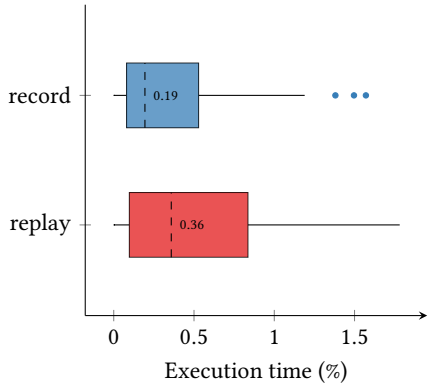


Figure 13: Execution time spent in tree operations for the *DFS* recording run and its replay from Figure 10, which is close to the worst-case scenario in terms of runtime overhead.

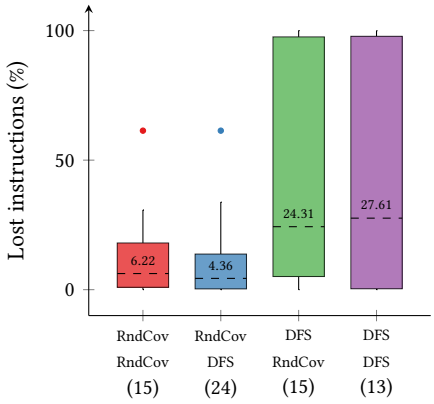


Figure 14: Distribution of lost instructions due to divergences on re-execution (no path pruning, number of *Coreutils* in each experiment in parentheses).

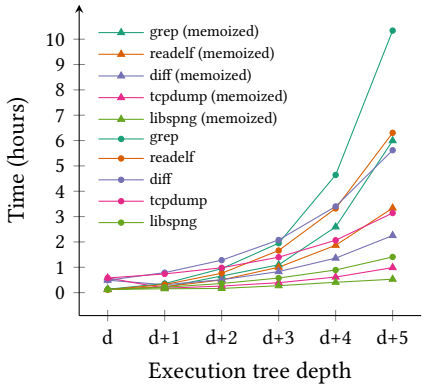


Figure 15: Iterative deepening with and without memoization. For each application and depth, we show the time it takes to perform the run with and without memoization.

Figure 15 shows how the memoized runs compare with the non-memoized ones. For all five applications, memoization significantly reduces the execution time, typically by more than 50%, which represents several hours of execution for these experiments. No divergences were observed in the re-execution runs.

Validity. It is not guaranteed in this setup that the memoized and non-memoized experiments execute the same instructions, especially with a small solver timeout and different usage of solver caches. With path pruning during memoization and no execution tree recorded in the unmemoized experiment, there is no practical way to directly compare both executions. Therefore, we use another experiment as a proxy for comparison. We repeat the run with the highest depth (d + 5) and memoize the execution tree (*record-only*). We then compare 1) the number of instructions of the non-memoized and record-only run, and 2) the execution trees of the record-only and memoized runs.

The number of instructions executed in record-only runs vary between 99.97% (*grep*) and 100.2% (*readelf*) in comparison to their non-memoized counterparts. Assuming that this reflects a high similarity between both runs, we finally compare the nodes of the execution trees of the record-only and memoized runs. The trees for *tcpdump* are identical, while the other trees have a high number of identical nodes (*readelf* 98.0%, *libspng* 99.3%, *grep* 99.94%, and *diff* 100.0%). Therefore, we believe the comparison in Figure 15 is meaningful.

4.6 Long-Running Experiments

Our final set of experiments demonstrates that our framework enables symbolic executors to run for very long periods of time while continuing to explore new paths.

We focus on the applications discussed in the introduction, for which KLEE configured with a 2 GB memory limit completely runs out of paths before the 2 h limit is reached, because too many states are killed prematurely due to memory pressure. For our *RndCov* recording run above, these applications are *base64, basenc, cut, dirname, fmt, fold, head, mktemp, paste, realpath, stty, sum, tac* and *wc*. As discussed before, no matter how much time one has at their disposal, KLEE won't be able to explore more than a certain number of paths in such cases. While increasing the memory limit could help, this is often an ineffective workaround, as important paths may still be lost on memory pressure, and symbolic execution may run out of paths at a later time. Furthermore, a low memory limit is often advantageous in a cloud deployment, where renting machines with little memory is more cost-effective. Below, we show that using our approach, one can run these applications for days, using KLEE's default memory limit of 2 GB, without losing any paths and continuously exploring new ones.

Starting with the memoized runs from the first experiment (where symbolic execution ran out of paths), we re-execute (with path pruning enabled) each of the 14 applications repeatedly from their previous recorded execution until the cumulative execution time exceeds seven days. Every time we restart the application, we run MoKlee without a timeout until it runs out of paths again and terminates. We use the default *RndCov* heuristic in each run.

Under this setup, we had to restart the applications between 20 times (for *fmt*) and 105 times (for *basenc*), with two exceptions: *dirname* and *sum* finish normally (without early path termination) after their first re-execution. Figures 16 and 17 show the number of replayed instructions and the number of new (unmemoized) instructions over time. Each point corresponds to one run. The figures show that our memoization extension is effective—each restarted run is able to progress the exploration and execute up to two orders of magnitude more new (unmemoized) instructions than already memoized ones. For six of these applications, this translates into new coverage. Figure 18 plots the additional coverage over time for these applications. After seven days, 4 of these applications achieved around 1-2% extra coverage, *stty* 3.69% more, and *tac* an impressive 17.56% more. Even more interestingly in the case of *tac*, most extra coverage is achieved in the last day, showing that in

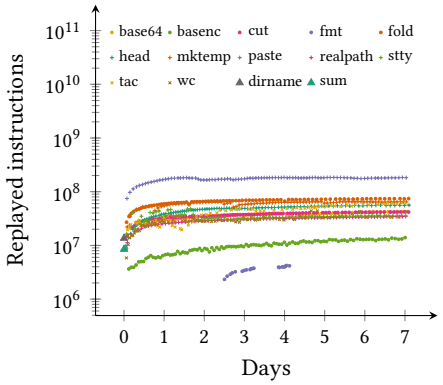


Figure 16: Replayed instructions per re-execution run.[8]

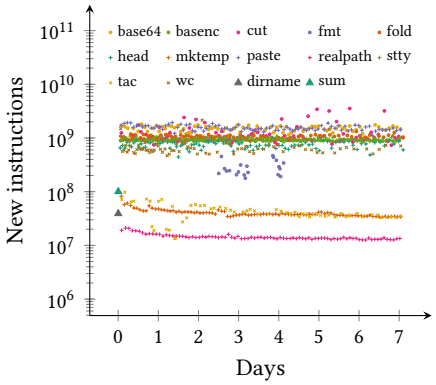


Figure 17: New (unmemoized) instructions per re-execution run.[9]

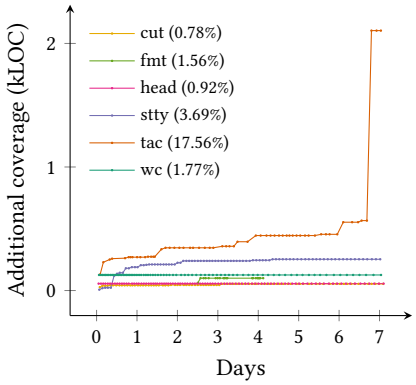


Figure 18: Additionally covered bitcode lines. Relative increase over recording runs given in parentheses.

some configurations, long-running runs are needed to achieve high coverage.

After seven days, the execution tree sizes vary between 0.6 GiB (*tac*) and 37.7 GiB (*wc*), which are reasonable values. These encode from 6.7M to over 495M nodes. Only *mktemp* encountered divergences (up to 9 per run) in this experiment.

5 RELATED WORK

Some forms of memoization, persistent execution trees, and path pruning during re-execution have been proposed in prior work. As discussed in the introduction, our work extends the excellent idea of memoized symbolic execution [26] to make it possible to detect divergences, store long-running paths without having to bring the entire execution tree back into memory and overcoming the restriction of using the same search heuristic in the record and replay runs. All these contributions make it possible for MoKlee to scale from the type of short runs on which memoized symbolic execution was previously evaluated (on the order of minutes, with small memoized trees) [22, 26] to very long runs (of hours and even days, with trees of millions of nodes).

Mayhem [7] introduced a combination of offline and online symbolic execution. When Mayhem runs out of memory, it terminates selected states after creating complete snapshots containing the path predicate and various statistics. Later, when more resources are available, states are brought back into memory from their snapshots and only associated concrete paths are re-executed. The authors give an average size of 30 kB per snapshot for *echo*. Considering that in our experiments *echo* created a large file on disk (1.50 GiB) with an average of 151 B per state and 10.5 million early terminated states, Mayhem-style checkpointing does not scale in our context. A *virtualization layer* in Mayhem intercepts and emulates system calls, preventing interferences between states and most likely divergences in re-executions.

Cloud9 [3] is a KLEE-based symbolic executor. One of its features is the distributed exploration of programs under test. For that, recorded execution path prefixes for candidate nodes are sent to workers which then re-execute those paths and resume exploration from such nodes. During re-execution of received path prefixes, all queries have to be re-solved by a constraint solver. Divergences are not detected. Our implementation currently has only limited support for distributed exploration. We always transfer complete trees between workers and run experiments incrementally.

FuzzBall [20], a symbolic execution engine for x86 binaries, explores a single path at a time. When a path terminates, a new one gets explored from the program start. An in-memory execution tree is mainly used to keep track of explored subtrees and to prevent expensive feasibility checks when reaching a symbolic branch. In case the tree becomes too large for an in-memory representation, users can optionally enable a (much slower) on-disk mode.

Orthogonal to memoization are *persistent constraint solution caches* [15, 25]. In our experiments we made three observations: 1) constraint solving time is often dominated by KLEE's caches rather than the underlying constraint solver (see Figure 10), 2) caches fill up quickly, and 3) most queries are solved quickly. It was an essential design decision to place our re-execution mechanism in front of the caches to reduce re-execution times significantly.

Seeding is another orthogonal approach to guide symbolic execution and reduce solving time [14, 19]. A seed is a concrete input (test case) for a program under test that describes a single execution path in the execution tree. Along such a path, the execution engine can omit branch-feasibility checks as a feasible input is already known. The main weakness of using seeding for memoization is that a large number of seeds would need to be stored, making the approach impractical for large execution trees. Instead of generating test cases for every execution state, SynergiSE [21] exploits the implicit ordering of tests and uses bordering tests to describe unexplored or feasible subtrees. While this representation can significantly reduce space requirements compared to keeping one seed per leaf node, it relies on a fixed search order and doesn't provide the global view of a full execution tree.

[8] *fmt* has one extra point after 9.9 days with value 4.2×10^6, which we do not show for better readability of the graph.
[9] *fmt* has one extra point after 9.9 days with value 1.6×10^8, which we do not show for better readability of the graph.

Outside of symbolic execution, memoization has been proposed in other domains. For instance, various forms of incremental and cooperative model checking rely on reusing analysis results previously saved to disk [2, 6, 17].

6 CONCLUSION

In this paper, we have presented an approach which enables symbolic execution to run indefinitely on large applications while continuing to explore new paths. Our approach is based on memoization, enhanced with support for divergence detection, switching search heuristics between record and replay time, and efficient storage which preserves the structure of the execution tree. We implemented our approach in MoKlee, an extension of the popular symbolic execution engine KLEE, and performed an extensive evaluation on 93 Linux applications. Our evaluation shows practical space and runtime overheads, high re-execution speed, effective divergence detection and applicability to iterative deepening and long-running symbolic execution analysis.

ACKNOWLEDGEMENTS

We thank Khoo Wei Ming and Cho Chia Yuan for their feedback on this project. We also thank Timotej Kapus, Jordy Ruiz and the anonymous reviewers for their comments on the paper.

This research has received funding from the DSO National Laboratories, Singapore, and the EPSRC, UK through grants EP/R011605/1 and EP/N007166/1. This project has also received funding from European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 819141).

REFERENCES

[1] Ben Boyter. 2019. *Sloc Cloc and Code (scc).* https://github.com/boyter/scc
[2] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. 2012. Conditional Model Checking: A technique to pass information between verifiers. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12).*
[3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11).*
[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 Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)* 56, 2 (2013), 82–90.
[6] Rodrigo Castaño, Víctor Braberman, Diego Garbervetsky, and Sebastian Uchitel. 2017. Model Checker Execution Reports. In *Proc. of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17).*
[7] 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).*
[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08).*
[9] GNU. 2019. *GNU Binutils.* https://www.gnu.org/software/binutils/
[10] GNU. 2019. *GNU Coreutils.* https://www.gnu.org/software/coreutils/
[11] GNU. 2019. *GNU Diffutils.* https://www.gnu.org/software/diffutils/
[12] GNU. 2019. *GNU Findutils.* https://www.gnu.org/software/findutils/
[13] GNU. 2019. *GNU Grep.* https://www.gnu.org/software/grep/
[14] 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).*
[15] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15).*
[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] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. 2008. Incremental State-space Exploration for Programs with Dynamically Allocated Data. In *Proc. of the 30th International Conference on Software Engineering (ICSE'08).*
[18] libspng [n.d.]. *libspng.* https://github.com/randy408/libspng
[19] 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).*
[20] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12).*
[21] Rui Qiu, Sarfraz Khurshid, Corina S. Păsăreanu, Junye Wen, and Guowei Yang. 2018. Using Test Ranges to Improve Symbolic Execution. In *Proc. of the 10th International Conference on NASA Formal Methods.*
[22] Rui Qiu, Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2015. Compositional Symbolic Execution with Memoized Replay. In *Proc. of the 37th International Conference on Software Engineering (ICSE'15).*
[23] Greg Roelofs and Mark Adler. [n.d.]. *zlib.* https://zlib.net/
[24] The tcpdump team. [n.d.]. *tcpdump.* https://www.tcpdump.org/
[25] 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).*
[26] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12).*