Precise Lazy Initialization for Programs with Complex Heap Inputs

Juan Manuel Copia
IMDEA Software Institute and
Universidad Politécnica de Madrid
Madrid, Spain
juanmanuel.copia@imdea.org

Facundo Molina
IMDEA Software Institute
Madrid, Spain
facundo.molina@imdea.org

Nazareno Aguirre
Universidad Nacional de Río Cuarto and
CONICET
Río Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar

Marcelo F. Frias
Instituto Tecnológico de Buenos Aires and
CONICET
Buenos Aires, Argentina
mfrias@itba.edu.ar

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain
alessandra.gorla@imdea.org

Pablo Ponzio
Universidad Nacional de Río Cuarto and
CONICET
Río Cuarto, Argentina
pponzio@dc.exa.unrc.edu.ar

***Abstract*—Lazy initialization enables symbolic execution for programs with heap-allocated inputs. It starts the program execution with a symbolic heap and concretizes it on demand as the program accesses it. However, the main challenge of lazy initialization is efficiently determining whether the current symbolic heap becomes infeasible with respect to the program's precondition. Pruning infeasible heaps is crucial to avoid significant runtime overhead and false alarms.**

In this paper, we propose PLI (Precise Lazy Initialization), an approach that precisely decides whether there exists a concretization of the current symbolic heap that satisfies the program's precondition. Unlike previous approaches, PLI also takes into account the constraints in the path condition to determine the feasibility of the current symbolic heap. Furthermore, PLI allows preconditions to be specified as standard operational predicates for concrete structures, eliminating the need for additional specifications tailored to symbolic heaps.

In our empirical evaluation, PLI demonstrated comparable performance to existing lazy approaches while reducing the number of explored paths by 43% (all infeasible) and eliminating all false alarms in the analysis. Moreover, PLI exhibited faster execution and better scalability compared to "eager" (enumeration-based) approaches, achieving a 67% reduction in explored paths.

I. Introduction

Symbolic Execution (SE) [1] is a widely recognized technique for program analysis, with successful applications in software verification [2], [3] and automated test input generation [?], [4]–[8], among others [9], [10]. It involves systematically exploring different paths in a target program using symbolic inputs instead of concrete values. When SE encounters a decision point in the program, such as a conditional statement or loop termination condition, the execution branches into two paths: one for the true case and another for the false case. These branches introduce constraints on the inputs, which are accumulated along the execution path, forming what is known as *path condition*. Intuitively, the path condition represents the set of constraints that the inputs must satisfy for the program to follow a particular path. As these constraints typically involve arithmetic and logical properties, their satisfiability can be often be solved resorting to SMT solvers [11]. An unsatisfiable path condition implies that no concrete input can exercise the path, which can be pruned from further analysis. Pruning infeasible paths is crucial for enhancing the speed and scalability of symbolic execution.

Modern programs frequently operate with heap-allocated structures, such as library collections or user-defined classes. These structures often need to satisfy specific constraints, whose satisfiability cannot be straightforwardly decided by SMT solvers. For instance, binary search trees have constraints related to the structure of the heap-allocated objects, particularly their reference-typed fields, such as acyclicity. Additionally, these structures often have constraints pertaining to their primitive-typed fields, such as the requirement to maintain sorted keys. Symbolic execution of such programs poses a significant challenge. There are two main approaches to enable symbolic execution in such scenarios. On one hand, eager approaches enumerate all feasible concrete heap configurations and use them as inputs for symbolic execution. However, this method can be inefficient and computationally expensive due to the potentially large number of structures that need to be considered. On the other hand, the *Lazy Initialization* (LI) [2] approach starts the symbolic execution with a fully symbolic heap, and the concretization of the symbolic fields is deferred until they are accessed during program execution. This approach allows LI to collapse multiple concrete executions into a single symbolic path, reducing the overall number of paths that need to be explored. In order to ensure finite exploration, LI requires an upper bound on the number of objects that can be created, known as the scope.

Similarly to the constraints gathered in the path condition, each lazy branch introduces a new constraint on the symbolic heap by assigning a specific value to the field. Some assignments may lead to an infeasible symbolic heap with respect to the program precondition. For instance, introducing a cycle in a tree. Branches with infeasible symbolic heaps can be safely discarded, so that symbolic execution can focus solely on feasible and meaningful program states.

Different techniques have been developed to tackle the feasibility analysis of symbolic heaps. Some of these approaches require users to provide additional specifications using specialized languages capable of expressing constraints over symbolic structures [2], [6], [12]. However, this places an additional burden on users, demanding extra effort and introducing the possibility of errors in the specifications. Other approaches, such as LISSA [13], rely on standard operational predicates over concrete structures. However, they address the feasibility of the symbolic heap separately from the path condition. As a result, they fall short in detecting infeasible branches where both the path condition and the heap are independently satisfiable but not when considered together. This limitation leads to the exploration of infeasible symbolic states, which are prone to generate false alarms during the analysis (see Section II).

We introduce a novel symbolic execution approach called Precise Lazy Initialization (PLI) to address this issue. The core of PLI is a novel solver capable of determining the satisfiability of symbolic heaps with respect to a specification and the scopes of the analysis. The solver takes into account the interplay between the path condition and the heap constraints, and accurately identifies symbolic heaps that are infeasible with respect to the specification. In this case, PLI can safely prune the current branch from further exploration, reducing unnecessary analysis overhead and eliminating false alarms. The specification must be expressed in terms of an operational predicate pre, which is a conjunction of two predicates: $pre = preP \wedge preH$. $preH$ describes the constraints about the shape of the heap (e.g., acyclicity), while $preP$ specifies constraints over primitive-typed fields of the structure (e.g., ordered keys). Importantly, pre is expressed in the same programming language as the analyzed code. This eliminates the need for specialized languages, leveraging the familiarity and existing knowledge of the target programming language. Is worth to mention that specifications are often expressed as a conjunction of several properties, and that there are other techniques that leverage this idea [14], [15].

To determine the satisfiability of the current symbolic heap and path condition, the PLI solver employs a two-step process. Firstly, it conducts a bounded exhaustive search within the provided scopes to find a concretization of the symbolic heap satisfying $preH$. The result is a candidate heap satisfying $preH$, with all its reference fields concrete and possibly some primitive fields set to symbolic values. Secondly, it performs a symbolic execution of the $preP$ predicate using as inputs the candidate heap and the path condition. This step tries to find out concrete values for the remaining primitive symbolic fields of the candidate, such that it satisfies $preP$ and the path condition. If this procedure succeeds, it finds a witness of the satisfiability of the symbolic heap and the path condition. Otherwise, the path is deemed unsatisfiable.

We experimentally assessed PLI against related techniques. The results demonstrate that PLI performs on par with existing lazy approaches while significantly reducing the number of explored paths by 43% in average. PLI effectively eliminates all infeasible paths explored by the related lazy techniques, leading to the removal of all false alarms. In comparison to traditional "eager" (enumeration-based) approaches, PLI exhibits better performance and scalability; it achieves a 67% reduction in the number of explored paths in average. This improvement is attributed to the over-concretization of heap inputs typically performed by eager approaches. Additionally, PLI successfully identified a known bug in a Binomial Heap implementation. This bug required the analysis to scale up to a scope of 13 nodes for the defect to manifest, which other techniques were not able to reach.

In summary, this paper makes the following contributions:

- A novel solver that utilizes an operational predicate as a specification to determine the satisfiability of symbolic states involving a symbolic heap and a path condition.
- The PLI lazy symbolic execution approach that utilizes the aforementioned solver to eliminate paths containing infeasible symbolic states. Our implementation is built on top of the Symbolic Pathfinder tool [16].
- An experimental assessment comparing PLI with related approaches, and demonstrating that PLI exhibits comparable performance to existing lazy approaches, but discards all infeasible paths and yields no false alarms.

II. Motivating Example

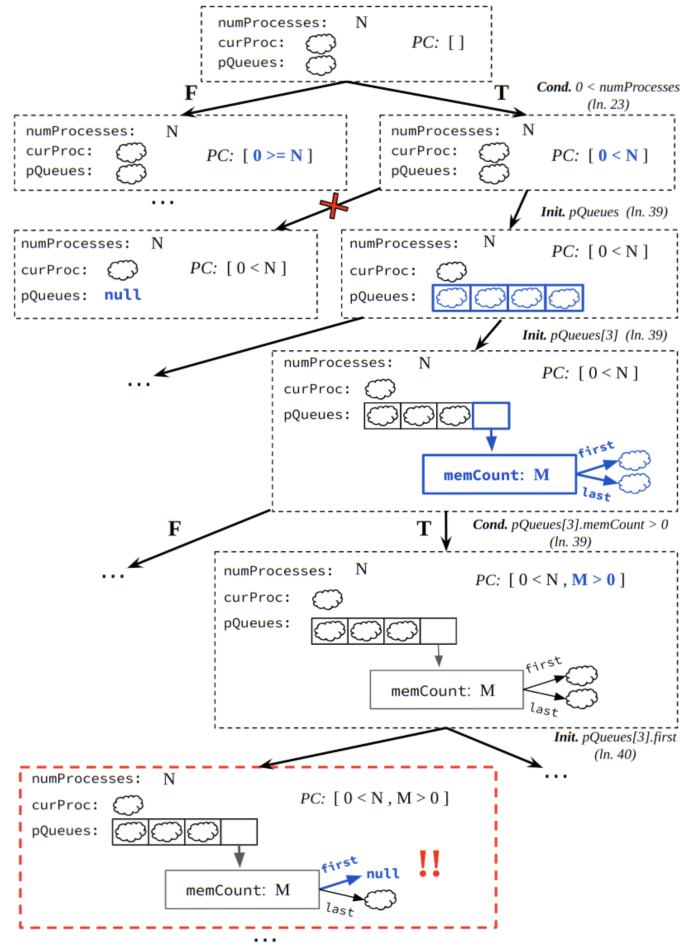
We now present an example that emphasizes the consequences of separately solving the symbolic heap and the path condition in lazy initialization, as occurs in LISSA [13]. Fig. 1 illustrates the `Schedule` class, taken from the SIR benchmark [17], which implements a scheduler of processes (`Job`) with three priority queues stored in the `pQueues` array. The highest priority queue is represented by `pQueues[3]`, followed by `pQueues[2]` and `pQueues[1]` for the next and lowest priorities, respectively. Particularly, the method `finishAllProcesses()` is in charge of terminating all scheduled processes. It executes a loop that, at each iteration, terminates the current running process `curProc` and schedules the next process with the highest priority. The precondition of `finishAllProcesses()` establishes that the receiver object, an instance of `Schedule`, must satisfy the following properties.

- *sched-1* (heap constraint): The `pQueues` array must not be null.
- *sched-2* (heap constraint): `pQueues[0]` must always be null.
- *sched-3* (heap constraint): the priority queues `pQueues[1]`, `pQueues[2]`, and `pQueues[3]` are non-nullable doubly linked lists (`List`).
- *sched-4* (primitive constraint): The `memCount` field of `List` holds the number of processes in each list.

Fig. 2 depicts a partial view of the symbolic execution tree explored when using lazy initialization. Each node in the tree corresponds to a symbolic state. The initial symbolic state

```
1  class Schedule {
2
3    class List {
4      Job first;
5      Job last;
6      int memCount;  // # processes in the list
7    }
8
9    class Job {
10     Job next;
11     Job prev;
12     int val;
13     int priority;
14   }
15
16   final static int MAXPRIO = 3;
17
18   int numProcesses; // # running processes
19   Job curProc;      // current running process
20   List[] pQueues = new List[MAXPRIO + 1];
21
22   public void finishAllProcesses() {
23     for(int i = 0; i < numProcesses; i++) {
24       finishProcess();
25     }
26   }
27
28   public void finishProcess() {
29     schedule();
30     if(curProc != null) {
31       curProc = null;
32       numProcesses--;
33     }
34   }
35
36   void schedule() {
37     curProc = null;
38     for(int i = MAXPRIO; i > 0; i--) {
39       if(pQueues[i].memCount > 0) {
40         curProc = pQueues[i].first;
41         pQueues[i] = delEle(pQueues[i], curProc);
42         return;
43       }
44     }
45   }
46
47   List delEle(List dList, Job dEle) {
48     if(dList == null || dEle == null)
49       return null;
50     ... // the method continues
51   }
52 }
```

Fig. 1. `finishAllProcesses` method from `Schedule`



Clouds in the picture represent symbolic reference values, while upper case letters represent symbolic primitive values.

Fig. 2. Symbolic execution tree of `finishAllProcesses`

of the analysis comprises an empty path condition (PC) and an instance of `Schedule` with symbolic fields. To represent existing lazy approaches, we assume the presence of an oracle capable of deciding the feasibility of a symbolic heap with respect to the program precondition.

The first decision point in the symbolic execution arises at line 23 within the condition of the loop: `i < numProcesses`. Taking the true branch, since `i` $= 0$, the constraint $0 < N$ is added to the path condition, where N represents the symbolic value associated with `numProcesses`. The execution proceeds until line 39, where it accesses the field `pQueues`, which contains a symbolic value. In lazy initialization, whenever the program under analysis accesses a symbolic field of reference type *T*, the execution branches for each possible initialization of that field: (1) to the special value `null`; (2) to each instance of type T allocated in previous lazy initializations; (3) to a new instance of type T with symbolic fields. We will refer to these ramifications as lazy branches. For `pQueues` the execution generates two lazy branches: one initializing the array as null and the other initializing it with a new array populated with symbolic values. The length of the array is determined by the bounds specified by the user, which in this case is 4. However, the null initialization violates constraint *sched-1*, and therefore the oracle discards it. Continuing with the feasible branch, the analysis encounters another lazy branch point on the same line, specifically for the initialization of index 3 of the array (`i` $=$ 3). Let us consider the branch that initializes `pQueues[3]` as a new `List` instance with symbolic fields, which satisfies constraint *sched-3*. Next, a branch occurs on the decision point `pQueues[3].memCount > 0` in line 39. Following the true branch, the constraint $M > 0$ is added to the path condition, where M represents the symbolic value associated with the `memCount` field of the list. Notice that these approaches treat all the primitive fields of the structures as symbolic. Finally, another lazy branch assigns `null` to the field `first` of the list when accessed at line 40. Without considering the path condition, this initialization is feasible for `pQueues[3]`. However, when considering the constraint $M > 0$ from the path condition in conjunction with *sched-4*, the field `first` cannot be `null`; the conjunction indicates the list must have at least one process.

The execution of this infeasible path continues with the invocation of method `delEle()` at line 41, with a null `Job` as second parameter. This causes the method to return and assign `null` to `pQueues[3]`. In the next iteration of the loop in line 23, when `schedule()` is called with this infeasible symbolic structure, a `NullPointerException` is thrown at line 39 when trying to access the `memCount` field of `pQueues[3]`, which is set to `null`. This exception is a false alarm. The analysis reports a bug where there is none. False alarms significantly impact the usability of program analysis techniques as they require manual inspection by users for their dismissal. By evaluating the feasibility of the path condition and the heap together, PLI efficiently identifies and discards infeasible branches, and avoids false alarms like this one.

III. The Precise Lazy Initialization Approach

In this section we present Precise Lazy Initialization (PLI), a novel symbolic execution technique based on lazy initialization for programs manipulating complex heap-allocated inputs.

A. Specifying Preconditions for PLI

Fig. 3 shows the precondition *pre* as an operational predicate for the `Schedule` case study. The idea is that $preH$ must check all the heap related constraints. Notice that the heap constraints *sched-1* and *sched-2* from Section II are checked at line 6, and *sched-3* in method `checkList` at line 19. Priority queues are implemented with doubly-linked lists, thus method `isDoublyLinkedList` (not included for space reasons) checks that the structural properties of doubly-linked lists are satisfied. On the other hand, $preP$ includes constraints on primitive-typed fields, such as that the `priority` field of the processes matches the index of the priority queue they belong to (line 39), and *sched-4* from Section II (line 44). The reason for requiring separated preconditions is that PLI delegates the solving of $preH$ to *SymSolve*, as it solves heap constraints very efficiently (see Section III-B). However, *SymSolve* usually does not perform well for primitive-typed constraints, given that primitive fields usually can have a large range of values that *SymSolve* would resolve in a bounded-exhaustive manner. Hence, PLI resorts to symbolic execution of $preP$ to solve primitive-typed constraints (see Section III-C).

Currently, determining which constraints to include in $preH$ and $preP$ is a task for the user. For PLI to work best we recommend following the simple guidelines discussed below.

- $preH$ includes all constraints over reference-typed fields and all constraints over primitive-typed fields that are strongly related to the shape of the heap (in our experiments, only size and balance constraints) and that can assume a small set of values (booleans, enumerations, small ranges of integers).

```java
1  public boolean pre() {
2    return preH() && preP();
3  }
4
5  public boolean preH() {
6    if (pQueues == null || pQueues[0] != null)
7      return false;
8    Set<List> visitedQ = new HashSet<>();
9    Set<Job> visitedJobs = new HashSet<>();
10   for (int i = 1; i <= MAXPRIO; i++)
11     if (!checkList(pQueues[i], visitedQ, visitedJobs))
12       return false;
13   if (!checkList(blockQueue, visitedQ, visitedJobs))
14     return false;
15   return numProcesses == visitedJobs.size();
16 }
17
18 boolean checkList(List queue, Set<List> visitedQ, Set<
       Job> visitedJobs) {
19   if (queue == null || !visitedQ.add(queue))
20     return false;
21   if (!isDoublyLinkedList(queue, visitedJobs))
22     return false;
23   return true;
24 }
25
26 public boolean preP() {
27   if (curProc != null && (curProc.priority < 1 ||
         curProc.priority > MAXPRIO))
28     return false;
29   for (int i = 1; i <= MAXPRIO; i++)
30     if (!checkPriority(pQueues[i], i))
31       return false;
32   return checkPriorityBlockQueue();
33 }
34
35 boolean checkPriority(List prioQueue, int priority) {
36   Job current = prioQueue.first;
37   int size = 0;
38   while (current != null) {
39     if (current.priority != priority)
40       return false;
41     size++;
42     current = current.next;
43   }
44   return size == prioQueue.memCount;
45 }
```

Fig. 3. Operational specification for `Schedule`

- $preP$ includes all the rest of the constraints over primitive-typed fields, which are solved via SMT.

The reason for including in $preH$ constraints over primitive fields with a bounded set of values, when they are related to the shape of the heap, is that it can improve the performance of PLI. This is because those constraints reduce the amount of candidate concretizations that *SymSolve* produces. For example, the colors of nodes in red-black trees have bounded domains (are either red or black), and checking that the tree is correctly colored ensures the balance of the tree. Including this constraint in $preH$ allows *SymSolve* to discard many heaps that represent imbalanced trees.

It is important to note that the soundness and completeness of PLI can be compromised if the user wrongly includes into $preP$ constraints over primitive-typed fields that might assume an unbounded set of values. For instance, a constraint stating that keys of a tree are sorted. This would cause the fields to be treated by *SymSolve* as if they were bounded. Thus, we bear the risk of missing feasible symbolic states that require values that are outside the provided bounds for the fields.

Algorithm 1 Precise Lazy Initialization Pseudocode

```
1:  function NextHeap(predicate, scopes, symH, initial)
2:    candidate ← initial
3:    repeat
4:      candidate ← next(scopes, symH, candidate)
5:      if candidate ≠ null ∧ predicate(candidate) then
6:        return candidate
7:      end if
8:    until candidate = null
9:    return null
10: end function
11:
12: function PLISolver(pre, scopes, symState)
13:   where pre = preH ∧ preP and
14:     scopes are the maximum numbers of allowed objects and
15:     symState = (symH, pathCond)
16:
17:   concH = NextHeap(preH, scopes, symH, null)
18:   while concH ≠ null do
19:     primC ← getPrimitiveConstraints(concH)
20:     conj ← primC ∧ pathCond
21:     if SMT(conj) = SAT then
22:       solutionPC ← SymbolicExec(preP, (concH, conj))
23:       if solutionPC ≠ null then
24:         return (concH, solutionPC)
25:       end if
26:     end if
27:     concH = NextHeap(preH, scopes, symH, concH)
28:   end while
29:   return null
30: end function
31:
32: class SymbolicExecutionTreeNode
33:   branchType        ▷ type of branch of this node: {PRIMITIVE, LAZY}
34:   parent            ▷ parent node in the symbolic execution tree
35:   solHeap           ▷ heap solution for this node
36:   solPC             ▷ path condition solution for this node
37: end class
38:
39: function PLIAlgorithm(pre, scopes, node, symState)
40:   if node.parent ≠ null then
41:     if node.branchType = PRIMITIVE then
42:       if SMT(node.parent.solPC ∧ pathCond) = SAT then
43:         node.solPC ← node.parent.solPC ∧ pathCond
44:         node.solHeap ← node.parent.solHeap
45:         return True
46:       end if
47:     else                                        ▷ Is a LAZY branch
48:       if node.parent.solHeap is solution of symH then
49:         node.solPC ← node.parent.solPC
50:         node.solHeap ← node.parent.solHeap
51:         return True
52:       end if
53:     end if
54:   end if
55:   solution ← PLISolver(pre, scopes, symState)
56:   if solution ≠ null then
57:     node.solHeap ← solution.solHeap
58:     node.solPC ← solution.solPC
59:     return True
60:   end if
61:   return False
62: end function
```

Given the simplicity of splitting preconditions in our experimental assessment, we believe that this procedure can be automated. Furthermore, we expect it can be done in such a way that optimal performance in PLI is achieved. A possible research line in this direction is the exploration of transcoping-based techniques, similar to the one employed in HyTeK [15]. By doing so, we would relieve the user from this task. Exploring this possibility is part of our future work.

B. A Constraint Solver for Symbolic Heaps

SymSolve [13] is a specialized solver designed to *decide* the satisfiability of symbolic heaps in relation to a provided specification for concrete structures (such as $preH$) and scopes within the data domains of these concrete structures. *SymSolve* performs an efficient bounded-exhaustive exploration over the space of concrete structures within the specified scopes. Consequently, *SymSolve* produces witnesses in the form of concrete structures that act as concretizations of the input symbolic heap while also adhering to the specification.

The NextHeap function at line 1 of Algorithm 1 outlines an abstraction of the behavior of *SymSolve* offering crucial insights for comprehending this paper. This function takes in several parameters: the symbolic heap denoted as `symH`, the specification that needs to be fulfilled identified as `predicate`, the specified scopes denoted as `scopes`, and an optional initial concrete structure identified as `initial`. Notably, *SymSolve* can resume the search process from any specified concrete structure, as facilitated by the `initial` parameter. *SymSolve* and PLI require the typical scopes definition for bounded-exhaustive approaches [13], [14]: the maximum number of objects to be created for each class (also required by lazy initialization [2]), and value ranges for primitive fields accessed by $preH$. In scheduler we set a maximum of 4 objects, and [0..3] as ranges for the primitive fields.

SymSolve explores concretizations in a bounded-exhaustive manner and in a deterministic order, guided by the order in which `predicate` visits the structure's fields. This approach was first implemented by bounded-exhaustive test generator *Korat* [14]. We assume the `next` function, invoked at line 4, abstracts this procedure and simply returns the next concrete candidate heap that satisfies the constraints of the symbolic heap `symH`, which is assigned to `candidate`. In the case that `initial` is `null`, the method `next` retrieves the first candidate. Conversely, if `initial` is provided, the search commences from the designated `candidate` to generate the subsequent concrete heap, without re-exploring previous states. This is a key feature that greatly contributes to the efficiency of *PLI*'s solver. The algorithm loops over concrete candidates in lines 3-8, until it finds a candidate satisfying `predicate` (line 6) or until it exhausts the search space and returns `null` in line 9.

Importantly, the search performed by *SymSolve* is efficient, as it discards large portions of the state-space that contain invalid structures. We refer the reader to LISSA's paper for details [13]. The main problem of *SymSolve* is that it cannot reason about the program path condition. Thus, any constraint present in the path conditions is completely ignored when deciding feasibility of a symbolic heap (see Section II).

C. An Integrated Solver for Heap and Primitive Constraints

A pseudocode for PLI's solver is provided in function `PLISolver` of Algorithm 1. It takes as inputs the specification `pre` = $preH \wedge preP$, the `scopes` for the analysis, and a symbolic state `symState`, composed of the symbolic heap (`symH`) and the program path condition (`pathCond`).

The purpose of this function is to search for a concrete instance within the given `scopes` that demonstrates the satisfiability of the symbolic state with respect to `pre`. The algorithm starts at line 17, calling NEXTHEAP (*SymSolve*) to search for a concretization `concH` of the symbolic heap satisfying $preH$. If no candidate is found (line 18), then the symbolic heap cannot satisfy $preH$ and the entire symbolic state can be deemed unsatisfiable, returning *null* at line 29. Otherwise, all the fields that were accessed by $preH$ will have concrete values, and the remaining fields would remain symbolic. In the scheduler example, all the reference-typed fields will have concrete values. All the primitive-typed fields will have symbolic values, except `numProcesses` that is accessed by $preH$ (line 15 of Fig. 3) and will be assigned a concrete integer by NEXTHEAP. Continuing with the algorithm, at line 19, `getPrimitiveConstraints` creates a formula that is a conjunction of equalities of the form $f = v$, for all primitive-typed fields f that have a concrete value v in `concH`. In our scheduler example, for a concrete heap `concH` with k processes, the formula would be $numProcesses = k$.

Next, the algorithm uses an SMT Solver to check whether the conjunction `conj` of `primC` and the program's path condition `pathCond` is SAT (lines 20-21). This step ensures that the concretized primitive values of the current candidate do not conflict with the constraints of the program's path condition. If the conjunction is UNSAT, the solver discards the candidate and proceeds to search for the next candidate that satisfies $preH$ (line 27). If `conj` = `primC` ∧ `pathCond` is SAT, the algorithm employs symbolic execution of $preP$ to determine whether `concH` and `conj` can also satisfy $preP$ (line 22). For this, $preP$ is executed symbolically with `concH` as input, using `conj` as the initial path condition. If the symbolic execution explores a path where $preP$ returns true, then the remaining symbolic values in `concH` can be assigned concrete values in such a way that $preP$ is satisfied. In that case, the algorithm retrieves the path condition of the path, `solutionPC`, which contains the constraints over the primitive symbolic fields of `concH` that lead to the satisfiability of $preP$. The algorithm returns `solutionPC` along with `concH` at line 24. These components together are a witness of the feasibility of the symbolic state `symState` with respect to `pre`. If no path in the symbolic execution of $preP$ returns true, `concH` and `conj` cannot satisfy $preP$. Then, the candidate is discarded and NEXTHEAP is queried for the next one. If no candidate satisfies both $preH$ and $preP$, the symbolic state is deemed infeasible with respect to `pre` and `scopes`, and *null* is returned at line 29.

For example, to decide about the satisfiability of the last symbolic state in Fig. 2, $preP$ is symbolically executed on candidates `concH` using as initial path condition: $0 < numProcesses \wedge memCount > 0$. As `first = null` in all `concH`, all paths in the symbolic execution will return false at line 44 of $preP$ in Fig. 3. That is because $memCount = 0$ cannot be true, as the path condition states that $memCount > 0$. Thus, the symbolic state is correctly identified as infeasible.

D. PLI's symbolic execution approach

PLI invokes the solver whenever a symbolic state is modified during symbolic execution. The solver is used not only to verify the feasibility of lazy branches but also for traditional symbolic execution branches. We refer to the latter kind of branches as "primitive branches". This is crucial because constraints added to the path condition can conflict with the representation invariants of heap-allocated inputs, and thus violate the program's precondition. However, invoking the solver often has an impact on the overall performance of PLI. For this reason, PLI also implements an optimization to avoid unnecessary solver queries. It consists of saving the solutions found by a previous solver call and checking whether they are still valid for subsequent symbolic states. In such cases, it reuses previous solutions to avoid calling the solver.

Function PLIALGORITHM of Algorithm 1 describes the behavior of PLI in the different kinds of nodes in the symbolic execution tree. The optimization is performed if a solution exists for the parent of the current node, thus it requires that the node has a parent (line 40). If the current branch is primitive (line 41) the symbolic heap remains the same, and only the program path condition has changed. Thus, the algorithm checks if the new path condition is satisfiable in conjunction with solution of the path condition found for the parent node (line 42). In such a case, the newly added constraint does not violate the constraints that make the current heap solution feasible with respect to the precondition. Thus, the algorithm saves the current solution in the current node and returns true without calling the solver (lines 43-45). When the current branch is due to a lazy initialization of a reference field (a lazy branch in line 47), the symbolic heap must have been changed, but the path condition remains the same as in the parent node. The algorithm checks if the solution found for the symbolic heap in the parent node is also a solution for the current heap (line 48). In this case, PLI stores the solutions for the previous symbolic state in the current node, and returns true without calling the solver (lines 49-51).

If a previous solution cannot be reused, PLISOLVER must be invoked (line 55). If it finds a solution (line 56), the symbolic state is feasible with respect to the program's precondition. The solution is stored in the current node and PLIALGORITHM returns true (lines 57-59), indicating that the exploration of the current branch continues. If the solver finds no solution, the current symbolic state is infeasible with respect to the precondition and PLIALGORITHM returns false (line 61). Thus, the branch is infeasible and is pruned out.

E. Soundness and Completeness of PLI

PLI is *sound*: it only prunes symbolic states that are infeasible with respect to the precondition and the given scopes. PLI is *complete*: as it examines all feasible symbolic states explored by LI. Below, we sketch a proof of the *soundness* and *completeness* of PLI with respect to LI.

Theorem 1. Let p be the program under analysis with precondition $pre = preH \wedge preP$, and let $scopes$ be the bounds for the analysis. Let s be a symbolic state that is feasible with respect to pre and $scopes$, s is explored along the execution of p using LI if and only if s is explored along the execution of p using PLI.

Proof (soundness) ($\implies$): Assume that there exists a symbolic state s $(symH, pathCond)$ satisfying pre within $scopes$ that is explored by LI, but is deemed infeasible and thus discarded by PLI. We will show that this assumption leads to a contradiction. There are two cases in which PLI may discard s. In the first case, no bounded concrete heap for s satisfying $preH$ and $symH$ is found by *SymSolve* (line 17 of the pseudocode). Notice that *SymSolve* performs a bounded exhaustive search with the same scopes as LI. Hence, if it cannot find a concrete heap satisfying $preH$, then s is not feasible with respect to pre for the given scopes, and we have arrived to a contradiction. In the second case, every bounded concrete heap s' satisfying $preH$ found by *SymSolve* either does not satisfy $pathCond$ ($SMT(conj) = UNSAT$ at line 21), or does not satisfy $preP$ in conjunction with $pathCond$ ($solutionPC = null$ at line 23). Let s' be a concrete heap candidate returned by *SymSolve*. If s' contradicts the path condition, then s' is not a feasible concretization of s because it cannot exercise the current path. If there is no way for s' to satisfy $preP$ and $pathCond$, then it cannot satisfy pre in a way that exercises the current path. Therefore, s is not feasible with respect to pre for the scopes, and we have a contradiction.

Proof (completeness) ($\impliedby$): This is trivial since PLI traverses the same symbolic execution tree as LI but prunes infeasible states along the way. Therefore, if a feasible state is explored by PLI, it is guaranteed to be explored by LI.

IV. Evaluation

Our evaluation is guided by three research questions:

1) RQ1: How does PLI compare to related approaches in terms of explored symbolic execution paths? We assess the number of paths explored by PLI against related techniques on 25 subjects. Our hypothesis is that PLI can reduce this number significantly, and therefore improve scalability.

2) RQ2: How does the execution cost of PLI compare to related approaches? PLI's costlier solving approach may have a negative impact on the execution time. We thus compare PLI against related techniques in this respect on the same subjects. Our hypothesis is that PLI is in line with the state-of-the-art.

3) RQ3: How do the optimizations affect the performance of PLI? We evaluate the impact of the optimizations implemented in PLI by enabling and/disabling them on each subject. Our hypothesis is that our optimizations have a positive impact on both aspects.

A. Subjects

We evaluate PLI on a set of subjects from the literature. Our subjects consist of 12 classes and 25 methods with complex heap-allocated inputs. The subjects include four implementations of data structures from the `java.util` standard library: `LinkedList` (circular, doubly-linked list), `HashMap` (hash tables based map implementation) and `TreeSet` and `TreeMap` (Red-Black Tree based implementations of set and map respectively). We also consider five client programs of the aforementioned classes, originally from the SF110 benchmark [18]: `Template` (stores data in a `LinkedList` and uses a `HashMap` for indexing), `TransportStats` (uses two `TreeMaps` to keep track of transferred bytes), `DictionaryInfo` (relies also on `TreeMaps` to store indexed data), `SQLFilterClauses` (defines a `HashMap` of `HashMaps` to store information about database queries), and `CombatantStatistic` (also uses a `HashMap` of `HashMaps` to store game statistics). Finally, we include an implementation of a scheduler of processes, `Schedule`, originally from the SIR benchmark [17]; and two implementations of data structures, `AvlTree` from the book [19] and `BinomialHeap` from the evaluation of BLISS [20]. All case studies but `AvlTree` and `BinomialHeap` were used in the experimental assessment of LISSA [13], where $preH$ and scopes were already specified. We manually implemented $preP$ for all cases and $preH$ and scopes only for `AvlTree` and `BinomialHeap`.

B. Experimental Procedure

We compare PLI against related symbolic execution approaches that at most require an operational specification for concrete structures, in the same programming language as the programs (notice that *DRIVER* that employs the implementation of routines from the API).

DRIVER: an eager approach that implements a *driver* program to enumerate the symbolic inputs. The driver repeatedly executes insertion methods from the API of the program under analysis, interleaving their executions using SPF's non-deterministic operators [16]. The insertion routines of the driver are executed symbolically, generating symbolic heaps with fully concrete reference fields. The drivers used in our experiments were taken from [13], except for `AvlTree` and `BinomialHeap`, which we manually implemented.

IFPRE: another eager approach, commonly used in the literature [6], [12]. It enumerates heap-allocated inputs by performing a symbolic execution with lazy initialization of the operational specification `pre` before the symbolic execution of the target program *p*. In other words, it symbolically executes **`IF`** `(pre(s))` **`THEN`** `p(s)`.

LIHYBRID: the baseline for automated lazy initialization-based approaches. It automatically derives a hybrid precondition [13], [20] from `pre` to decide about the feasibility of symbolic heaps. The hybrid precondition works like `pre` when it accesses fields with concrete values (and it might be able to discard some infeasible heaps), but it returns that the heap is feasible as soon as it finds a field with a symbolic value. It can thus produce many false positives.

LISSA: the most recent *lazy* approach, employs the specialized solver *SymSolve* to identify and prune lazy branches that violate `preH`. It addresses the satisfiability of symbolic heaps independently of the program path condition, which can result in the exploration of infeasible symbolic states (see Section II).

We executed each technique on every subject program, incrementally increasing the scopes. We used a workstation with a Xeon Gold 6154 CPU (3GHz) and Debian Linux 11 OS. Each execution used a single CPU core, a maximum heap size of 4GB and a 1 hour timeout. PLI's implementation and

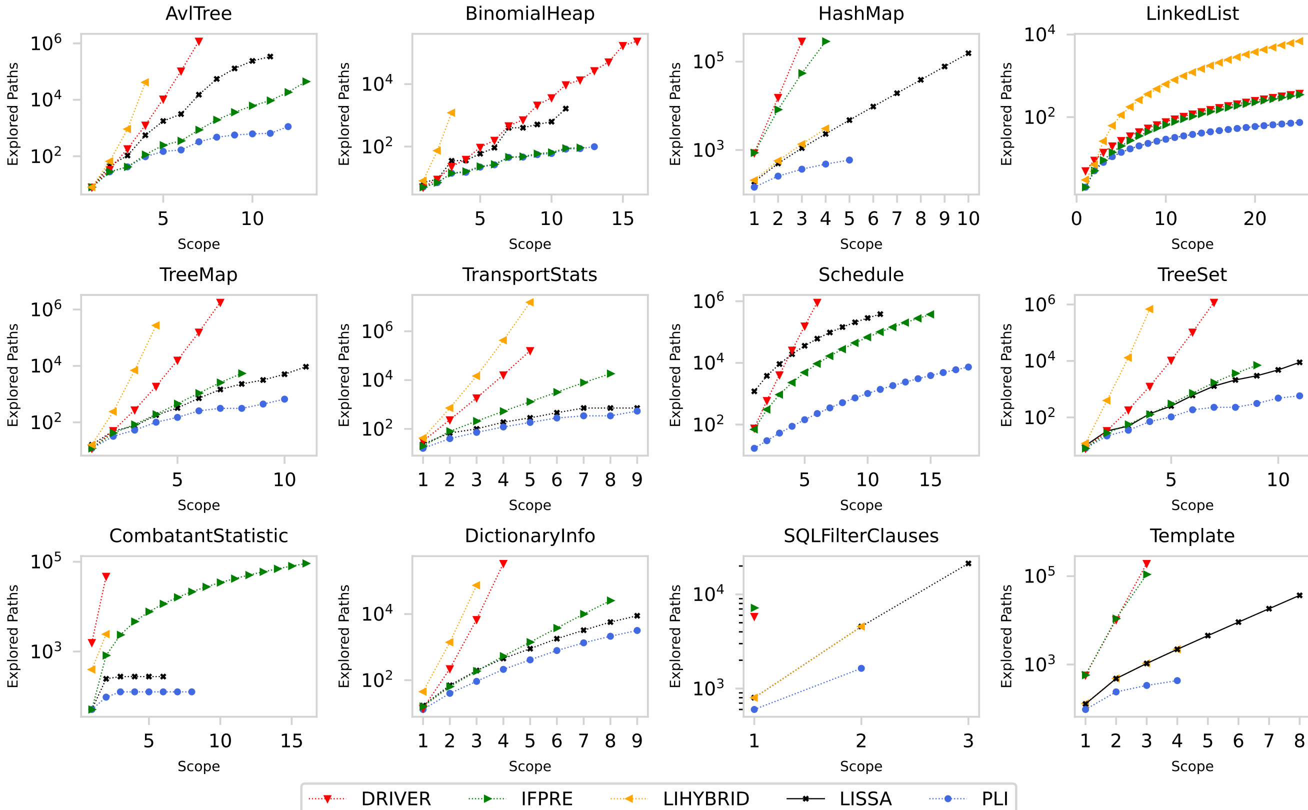


Fig. 4. Number of paths explored by each technique and for each scope, within a timeout of one hour.

replication package to reproduce the experiments can be found online [21], [22].

C. Experimental Results

RQ1: Number of Explored Paths: Fig. 4 plots the number of paths explored by each technique in each subject (an average over the considered methods) for increasing scopes, in logarithmic scale. Notice that PLI explores significantly fewer paths than related techniques in 10 out of 12 cases, while for `BinomialHeap` and `LinkedList` it ties with IFPRE and LISSA.

In comparison to eager approaches, PLI explores at least one order of magnitude fewer paths in most cases, particularly when considering larger scopes. This reduction can be attributed to the over-concretization of the heap performed by eager techniques. Eager techniques enumerate all the possible concretizations of the heap, which often grow exponentially with the scope. In contrast, PLI is a lazy approach that concretizes the heap on-demand as the program accesses it during analysis. In summary, *PLI achieves a significant reduction of 78% and 67% in the number of explored paths compared to DRIVER and IFPRE, respectively*. DRIVER explores more paths than IFPRE because it employs symbolic execution of API calls, which can generate repeated inputs. This allows PLI to scale better than eager techniques in most of the cases.

PLI outperforms the lazy approaches LIHYBRID and LISSA, reducing the number of paths by 66% and 43% respectively. This is attributed to PLI's ability to eliminate infeasible paths. LIHYBRID's automatically derived hybrid preconditions contribute to a substantial over-approximation of the state space, leading to a high number of false positives. Although LISSA is more precise than LIHYBRID, it still fails to prune infeasible paths arising from inconsistencies between the symbolic heap and the path condition.

As infeasible paths tend to grow exponentially with respect to the scope, the scalability of the analysis is significantly compromised. This is particularly evident in the case of LIHYBRID, which scales poorly due to a high number of false positives. PLI maintains a similar scalability to LISSA while eliminating false positives during analysis. In four cases, PLI outperforms LISSA in terms of scalability, while LISSA performs better than PLI in four other cases. Both techniques achieve the same scope in the remaining cases.

Furthermore, infeasible paths usually lead to false alarms. LIHYBRID produced many false alarms in most cases. Considering the highest scope reached for each case, LISSA produced 1075 false alarms in `Schedule`, 10,378 in `AvlTree`, and 60 in `CombatantStatistic`. This places a significant burden on users, who must manually inspect and discard these false alarms. In contrast, PLI analysis did not produce any false alarms, as all the explored paths are feasible.

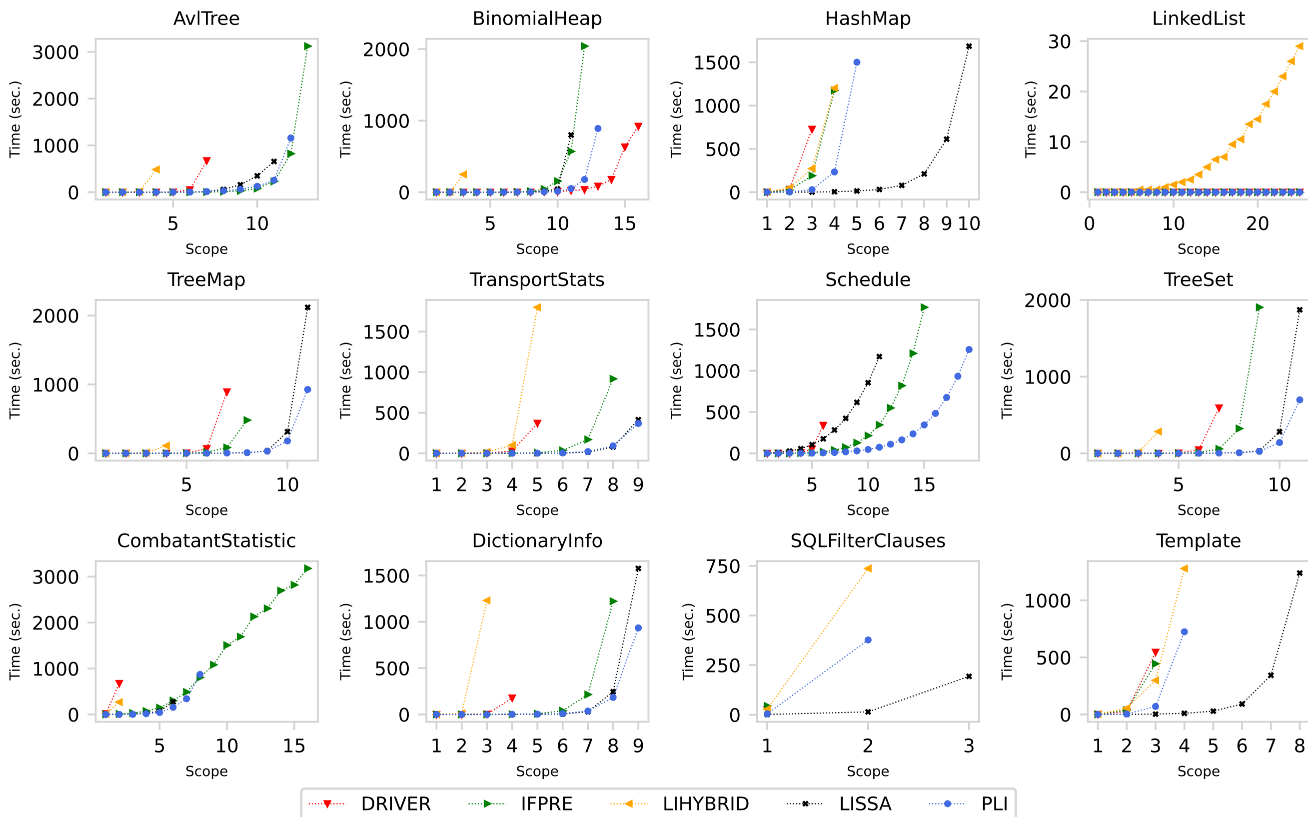


Fig. 5. Symbolic execution time of each technique for each scope, within a timeout of one hour.

RQ2: Execution Time: Figure 5 shows the average execution time of the approaches for increasingly larger scopes in our subjects. PLI performs better than lazy approaches in the majority of cases, due to a reduction in the number of explored paths through the elimination of infeasible ones. Notice that PLI uses a more expensive solver, which can negatively affect performance. Despite this, PLI's precise nature still yields improved performance in most cases. LISSA only outperforms PLI in `HashMap` and it's clients (`SQLFilterClauses`, `Template`). In these cases, $preP$ performs bitwise operations for hash table lookups, which are very expensive for the SMT solver in the current version of SPF. As LISSA does not use preP, it does not suffer this performance overhead. We are currently looking for ways of fixing this issue. Eager approaches perform well in a few cases (e.g. DRIVER in `BinomialHeap`, IFPRE in `CombatantStatistic`), but in most cases the costs of enumerating inputs and performing symbolic execution on each of them outweigh the costs of using a precise lazy approach as PLI.

Finally, notice that method `extractMin()` of `BinomialHeap` contains a known a bug [23] that only manifests when using binomial heap inputs with at least 13 nodes. This defect leads to inconsistencies between the size of the binomial heap and the actual number of nodes in the structure. PLI can successfully identify this bug. It is the only approach, beside DRIVER, which is able to reach scope 13.

TABLE I
PERFORMANCE IMPROVEMENT OF THE APPLIED OPTIMIZATION FOR EACH SUBJECT.

Subject	**Reduction ⇓**	
	Solver Calls (%)	**Time (sec.)**
AvlTree	25%	2378
BinomialHeap	11%	1409
CombatantStatistic	45%	900
DictionaryInfo	18%	2544
HashMap	46%	35
LinkedList	14%	1
Schedule	22%	1544
SQLFilterClauses	66%	68
Template	55%	63
TransportStats	20%	849
TreeMap	18%	2141
TreeSet	18%	3728
Average	29.8%	1305

RQ3: Impact of the Optimization: Table I shows the performance improvement achieved by enabling the PLI's optimization described in Section III-D. The results demonstrate that the optimization effectively avoids a significant number of solver calls across various cases. Specifically, *it prevents more than 40% of solver calls in 4 out of 12 cases and over 15% of solver calls in 10 out of 12 cases*, with an average of 29.8%. Regarding execution time, the optimization consistently saves time in all cases, with an *average time*

savings of over 20 minutes. The optimization does not provide a significant improvement in `LinkedList` because it has the least complex precondition among all cases, and specifically it lacks constraints over the primitive fields (contents) of the list. In summary, the proposed optimization significantly improves performance across the majority of cases.

V. Threats to Validity

Threats to external validity may arise from the selection of our subjects. Our experiments consider on a relatively small set of classes from the literature for which heap preconditions were available. Due to the limited number of subjects, our findings cannot be confirmed with significant statistical confidence. However, the selected subjects are highly representative of programs that manipulate complex heap inputs, exhibiting complex preconditions over the heaps. Moreover, some of the classes were taken from real-world programs (SF110 [18]). We believe that our results provide initial evidence of the capabilities of PLI to enable efficient symbolic execution of this kind of programs. Another threat to validity concerns the correctness of our PLI prototype implementation. To mitigate this threat, we conducted differential testing between PLI and the IFPRE approach. The testing procedure involved executing PLI on the program precondition `pre(s)` and comparing the number of explored paths with those obtained by executing the IFPRE technique with no subject program (**IF** `(pre(s))` **THEN** `skip;`) so that the `IF` statement acts as a ground truth, ruling out all infeasible inputs. By verifying that the number of paths is the same for both approaches, we gain a higher degree of confidence that PLI's implementation adequately explores all the intended paths (neither more nor less). Moreover, we make our dataset and implementation available at [21], [22].

VI. Related Work

Symbolic execution of heap-allocated data has been widely studied in the literature. [2], [6] introduced the idea of LI, using operational hybrid preconditions to prune infeasible paths. As these hybrid preconditions can result in imprecise analysis, several techniques have been proposed to address this issue. Some approaches address the problem by complementing LI either with precomputed bounds to reduce the number of nondeterministic choices, as in [24], or with equivalent preconditions written in declarative specifications languages, such as BLISS [20], or captured via machine learning algorithms [25]. Others, such as HEX [12], replace the use of hybrid preconditions by using a unique specification provided in a declarative language specifically designed to describe properties of symbolic structures. PLI differs from these approaches since it only requires an operational precondition written in the same language as the target program.

LISSA [13] introduced a solver for symbolic structures that relies exclusively on operational predicates for concrete structures. This has the advantage of not requiring additional specifications. However, LISSA faces another drawback of lazy initialization: treating heap and path conditions separately leads to the exploration of infeasible paths. In fact, this is the main limitation that motivated our approach. PLI addresses the feasibility of both the path condition and the symbolic heap together, enabling the detection of inconsistencies between them. As a result, PLI eliminates the occurrence of false positives and, subsequently, false alarms during the analysis.

Symbolic execution has been also successfully applied in other contexts. KLEE [4] is a symbolic execution approach that specifically targets C programs. As far as our knowledge goes, KLEE does not implement lazy initialization. StarFinder [26], uses declarative predicates written in Separation Logic [27] to reason about symbolic heaps, enabling dynamic symbolic execution technique for programs that involve heap inputs. Pex [7] focuses on C# programs. When the program under analysis involves complex heap inputs, Pex requires the user to manually provide "object factories", which are responsible for generating the heap inputs to use in the dynamic symbolic execution.

Finally, symbolic execution approaches have been used in combination with other techniques to improve specific tasks, in particular test generation. Seeker [28] is built on top of Pex and uses a combination of static and dynamic analysis to achieve high-coverage testing. SUSHI [5] combines evolutionary computation and symbolic execution to produce test inputs for programs with complex heap-allocated data.

VII. Conclusions

In this paper, we introduced PLI, a complete and sound lazy initialization approach for the symbolic execution of programs with complex heap-allocated inputs. Unlike existing lazy approaches, PLI can determine the feasibility of a symbolic heap taking into account the path condition. This allows PLI to detect and prune many infeasible paths that arise when the symbolic heap and the path condition are considered together. Moreover, PLI requires only a precondition provided as an operational predicate in the same programming language as the code under analysis.

Our experimental evaluation demonstrates that PLI outperforms eager approaches in terms of execution time and scalability. Due to its lazy nature, it avoids the exploration of unnecessary concretizations of the heap, resulting in improved performance. Furthermore, the experiments reveal that PLI effectively eliminates false positives encountered in the fastest lazy initialization-based approach [13], while maintaining comparable performance and scalability. Although PLI's solving algorithm is computationally more expensive than LISSA's, the time saved by avoiding the exploration of infeasible paths outweighs the solving cost.

Acknowledgment

This work was partially supported by the Spanish Government grants Prodigy TED2021-132464B-I00 and Ramón y Cajal RYC2020-030800-I, the Madrid Regional projects SCUM S2018/TCS-4339 and MFoC 49/520608.9/18, Argentina's ANPCyT through grants PICT 2017-2622, 2019-2050, 2020-2896, an Amazon Research Award, and by EU's Marie Sklodowska-Curie grant No. 101008233 (MISSION).

REFERENCES

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[2] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems , 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 553–568. [Online]. Available: https://doi.org/10.1007/3-540-36577-X_40

[3] C. S. Pasareanu, *Symbolic Execution and Quantitative Reasoning: Applications to Software Safety and Security*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2020. [Online]. Available: https://doi.org/10.2200/S01010ED2V01Y202005SWE006

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[5] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 90–101. [Online]. Available: https://doi.org/10.1145/3092703.3092715

[6] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts , USA, July 11-14, 2004*, G. S. Avrunin and G. Rothermel, Eds. ACM, 2004, pp. 97–107. [Online]. Available: https://doi.org/10.1145/1007512.1007526

[7] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Springer, 2008, pp. 134–153. [Online]. Available: https://doi.org/10.1007/978-3-540-79124-9_10

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[9] C. S. Pasareanu, R. Kersten, K. S. Luckow, and Q. Phan, "Chapter six - symbolic execution and recent applications to worst-case execution, load testing, and security analysis," *Adv. Comput.*, vol. 113, pp. 289–314, 2019. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.10.004

[10] D. Gopinath, M. Zhang, K. Wang, I. B. Kadron, C. S. Pasareanu, and S. Khurshid, "Symbolic execution for importance analysis and adversarial generation in neural networks," in *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, K. Wolter, I. Schieferdecker, B. Gallina, M. Cukier, R. Natella, N. R. Ivaki, and N. Laranjeiro, Eds. IEEE, 2019, pp. 313–322. [Online]. Available: https://doi.org/10.1109/ISSRE.2019.00039

[11] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006. [Online]. Available: https://doi.org/10.1145/1217856.1217859

[12] P. Braione, G. Denaro, and M. Pezzè, "Symbolic execution of programs with heap inputs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 602–613. [Online]. Available: https://doi.org/10.1145/2786805.2786842

[13] J. M. Copia, P. Ponzio, N. Aguirre, A. Gorla, and M. Frias, "Lissa: Lazy initialization with specialized solver aid," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556965

[14] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, P. G. Frankl, Ed. ACM, 2002, pp. 123–133. [Online]. Available: https://doi.org/10.1145/566172.566191

[15] N. Rosner, V. S. Bengolea, P. Ponzio, S. A. Khalek, N. Aguirre, M. F. Frias, and S. Khurshid, "Bounded exhaustive test input generation from hybrid invariants," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein, Eds. ACM, 2014, pp. 655–674. [Online]. Available: https://doi.org/10.1145/2660193.2660232

[16] K. S. Luckow and C. S. Pasareanu, "Symbolic pathfinder v7," *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–5, 2014. [Online]. Available: https://doi.org/10.1145/2557833.2560571

[17] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empir. Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005. [Online]. Available: https://doi.org/10.1007/s10664-005-3861-2

[18] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 8:1–8:42, 2014. [Online]. Available: https://doi.org/10.1145/2685612

[19] M. A. Weiss, *Data Structures and Algorithm Analysis in Java (3rd. ed.)*. USA: Addison-Wesley Publishing Company, 2011.

[20] N. Rosner, J. Geldenhuys, N. Aguirre, W. Visser, and M. F. Frias, "BLISS: improved symbolic execution by bounded lazy initialization with SAT support," *IEEE Trans. Software Eng.*, vol. 41, no. 7, pp. 639–660, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2015.2389225

[21] J. M. Copia, F. Molina, N. Aguirre, M. F. Frias, A. Gorla, and P. Ponzio, "Github repository of the pli's implementation and replication package," https://github.com/JuanmaCopia/spf-pli, 2023.

[22] ——, "Website for paper "precise lazy initialization for programs with complex heap inputs"," https://sites.google.com/view/precise-lazy-initialization/home, 2023.

[23] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias, "Analysis of invariants for efficient bounded verification," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 25–36. [Online]. Available: https://doi.org/10.1145/1831708.1831712

[24] J. Geldenhuys, N. Aguirre, M. F. Frias, and W. Visser, "Bounded lazy initialization," in *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, ser. Lecture Notes in Computer Science, G. Brat, N. Rungta, and A. Venet, Eds., vol. 7871. Springer, 2013, pp. 229–243. [Online]. Available: https://doi.org/10.1007/978-3-642-38088-4_16

[25] F. Molina, P. Ponzio, N. Aguirre, and M. F. Frias, "Learning to prune infeasible paths in generalized symbolic execution," in *IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022, Charlotte, NC, USA, October 31 - Nov. 3 , 2022*. IEEE, 2022, pp. 494–504. [Online]. Available: https://doi.org/10.1109/ISSRE55969.2022.00054

[26] H. L. Pham, Q. L. Le, Q.-S. Phan, J. Sun, and S. Qin, "Enhancing symbolic execution of heap-based programs with separation logic for test input generation," 12 2017.

[27] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[28] S. Thummalapenta, T. Xie, N. Tillmann, J. Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," vol. 46, 10 2011, pp. 189–206.