LISSA: Lazy Initialization with Specialized Solver Aid

Anonymous Author(s)

ABSTRACT

Programs taking as input heap-allocated structures are hard to deal with for symbolic execution (SE). Lazy Initialization approach (LI) handle such programs by starting SE over a fully symbolic heap, and initializing input's fields *on demand*, as the program under analysis accesses them. However, when the program's assumed precondition has structural constraints over the input, operationally captured via `repOK` routines, LI may produce spurious symbolic structures, leading to spurious paths that undermine SE performance. Previous work relied on manually crafted specifications to avoid producing symbolic structures violating program's precondition ("hybrid" `repOK`s, or declarative specifications equivalent to `repOK`).

In this work, we introduce SymSolve, a novel approach (inspired by the test case generator Korat) to *decide* whether a partially symbolic structure can be extended to a fully concrete structure satisfying `repOK`. In contrast to former approaches, SymSolve reduce the specification requirements by relying only on traditional `repOK` routines (and upper-bounds of the input's size). SymSolve explore feasible concretizations of partially symbolic structures in a bounded-exhaustive manner, until it finds a fully concrete structure satisfying `repOK`, or it exhausts the search space and deems the input structure spurious. SymSolve's search algorithm can prune large parts of the search space that are known to contain only concrete structures violating `repOK`. It also includes a symmetry breaking approach to discard isomorphic structures that significantly improves its efficiency.

We implemented LISSA, an approach based on LI employing SymSolve to identify spurious symbolic structures and prune spurious paths. We experimentally assessed LISSA against related techniques over various case studies, consisting of programs that manipulate heap-allocated structures with complex constraints. The results show that LISSA is faster and scales better than related techniques.

KEYWORDS

Symbolic Execution, Lazy Initialization, Structural Constraint Solving

ACM Reference Format:

Anonymous Author(s). 2022. LISSA: Lazy Initialization with Specialized Solver Aid. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Symbolic execution (SE) [5, 19, 20] is a well known technique for program analysis that has been successfully applied to software verification [14, 17, 21] and automated test input generation [3, 6, 13, 18, 27], among other applications [12, 15, 22, 24]. SE employs symbolic inputs instead of concrete ones and systematically explores feasible (bounded) paths in a target program. To achieve this, SE constructs a formula for each program path, called the path condition, holding the constraints on symbolic inputs that concrete inputs must satisfy to exercise the corresponding path. In this way, a symbolically executed path can be thought of as representing an often large set of concrete executions. Constraint solvers [7, 8, 10] can then determine the feasibility of a path condition, and prune those paths where the corresponding conditions become infeasible. Pruning infeasible paths is crucial for the performance and scalability of SE.

Many programs take as input heap-allocated data, such as instances of user-defined class-based data representations. Dealing with such structures in a symbolic way is a major challenge, since constraint solvers cannot directly handle constraints on these structures that are part of the program's precondition. There exist many approaches to tackle this problem [3, 4, 12, 17, 24, 26, 29]. One approach consists of initializing the heap as empty, and use a harness that non-deterministically populates the heap (satisfying program's precondition) before symbolically executing the target program. This approach, however, significantly reduces SE automation, since the harness has to be manually provided. Moreover, this approach is "eager" in the sense that heap-allocated data is constructed prior to the SE of the target program, and in principle without consideration of what parts of the heap the target program will actually access.

In contrast, the so-called *lazy initialization* approach [17] addresses this problem by assuming that SE starts on a fully symbolic heap, and non-deterministically initializes the heap *on demand* as the target program accesses it. This approach favors an assume-guarantee analysis, but it also comes with its own limitations: when the assumed program's precondition contains constraints over the data representation being manipulated, usually a representation invariant operationally captured via a `repOK` routine, then further effort from the developer is required to effectively execute symbolically the target program. The main issue in this situation is how to determine if a partially symbolic input structure (incrementally concretized during SE) can be extended to a fully concrete one satisfying the assumed `repOK`. Otherwise, we say the partially symbolic structure is *spurious*. When not identified properly, spurious symbolic structures make LI waste resources in exploring spurious paths, which is detrimental for LI's efficiency. They also might result in false positives in the analysis of the program.

Some techniques employ the so-called `HybridRepOKs` [17, 28], i.e., user-crafted adaptations of given `repOKs`, to detect spurious partially symbolic structures. Other approaches require the developer to provide an additional specification, equivalent to the original `repOK`, but written in a logical declarative language amenable to constraint solving [4, 5, 24]. These additional specification efforts are non-trivial, and reduce the automation of SE.

In this paper, we improve the above-described problems of lazy initialization via a novel technique to efficiently identify spurious symbolic structures. Our approach, called SymSolve (inspired by the test input generator Korat [2]), receives a partially symbolic structure and decides if this symbolic structure can be extended into at least one fully concrete structure that satisfies the `repOK`. In contrast to previous approaches, SymSolve does not require any additional specification to be provided by the user. SymSolve employs the operational `repOK` for concrete structures and user-provided bounds on the maximum size allowed for the structures (often called scopes, also required by LI). SymSolve explores the search space of concrete structures that are concretizations of its partially symbolic input, in a bounded-exhaustive manner. In this process, SymSolve either finds out a witness showing that the symbolic structure can be fully concretized into a structure satisfying `repOK`, or the structure is deemed spurious.

We also define a symmetry breaking approach for SymSolve, to efficiently get rid of isomorphic structures throughout SymSolve's search process. As shown in our experimental assessment, this approach contributes significantly to SymSolve's efficiency and scalability to larger structures (see Section 4.2).

We implemented SymSolve and incorporated it as a solver for heap-allocated partially symbolic structures in the LI engine of Symbolic PathFinder (SPF) [20]. We call this SE approach LISSA. LISSA employs SymSolve to identify spurious structures produced by LI, and prune the corresponding spurious paths. We experimentally assessed LISSA against related techniques in several case studies. The results show that for many programs dealing with complex heap-allocated structures LISSA is faster, and scales better than related techniques.

In summary, the main contributions of our paper are:

- SymSolve, an efficient solver for partially symbolic structures, that requires only a standard `repOK` and scopes for the analysis.
- A symmetry breaking approach for SymSolve that significantly contributes to its efficiency and allows it to scale up to larger scopes.
- A SE approach, LISSA, that employs SymSolve to identify spurious symbolic structures and prune spurious paths. Compared to previous work, LISSA has lower specification requirements (a standard `repOK`).
- An experimental assessment showing that, for programs manipulating heap-allocated inputs with rich structural constraints, LISSA performs better than related approaches.

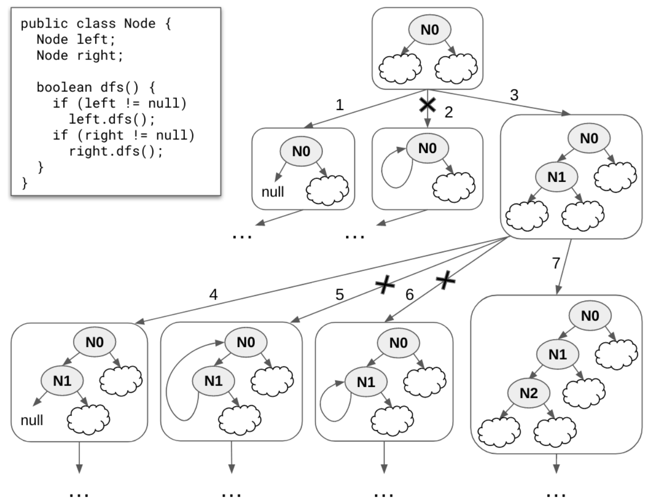
2 BACKGROUND

2.1 Symbolic execution with lazy initialization

In this section, we introduce lazy initialization (LI) [17] by means of an example. Figure 1 shows the starting fragment of how LI incrementally concretizes a partially symbolic structure during the symbolic execution of method `dfs`, a depth-first search traversal of a binary tree. LI starts by instantiating the receiver object `this` with a `Node` object (`N0`) with all its fields initialized as symbolic. Symbolic fields of partially symbolic structures are concretized when they are first-accessed by `dfs`. LI considers all the feasible options for initializing symbolic fields (of reference type): (1) the special value `null`; (2) an object of the corresponding type already present in the structure (allocated in previous lazy initialization steps); (3) a newly allocated object of the corresponding type with all its fields initialized as symbolic. Fields of primitive types are dealt with as in traditional SE.

The first LI step occurs when the target program checks whether `left != null`. As `N0.left` is symbolic, the execution branches for each of the aforementioned possibilities: (1) `null` (branch 1 in Fig. 1); (2) the only existing node at this point, `N0` (branch 2); (3) a new node (`N1`) with symbolic fields (branch 3). Continuing with branch 3, as now `N0.left != null`, the program makes the recursive call `left.dfs`. Then, `dfs` checks whether `N1.left != null`. This time, a LI step originates the four branches in the Figure: `N1.left` is initialized to `null` (branch 4); to the previously created nodes `N0` (5) and `N1` (6); and to a new node `N2` (7).

Figure 1: `dfs` program and a fragment of its symbolic execution tree.



As symbolic structures can grow infinitely large, the user needs to specify a maximum number k of nodes to be created by LI. This number is referred to as the *scope* of the analysis. The exploration continues until all the feasible paths of `dfs` are executed, using structures with up to k nodes.

The (partially) symbolic structure that LI maintains, and more precisely its *concrete* part, captures the constraints that concrete structures must satisfy for the program to exercise the corresponding path. For example, to exercise branch 7 of Figure 1, the concrete structures must satisfy `N0.left=N1` and `N1.left=N2`.

Very often, programs under analysis require preconditions to be met. Particularly, programs with heap-allocated objects as input must satisfy the representation invariants of those objects, typically captured by an operational `repOK` routine. We say that a partially symbolic structure S is satisfiable (`sat`) if there exists at least one fully concrete structure satisfying the constraints imposed by S for which the `repOK` returns true. Otherwise, we call S spurious (or `unsatisfiable`). For example, for the depth-first search traversal of the binary tree, we assume the `repOK` shown in Figure 2 as the precondition, which rules out non-tree structures (i.e. containing cycles or with nodes with more than one parent). With this precondition, branches 2, 5, and 6 (marked with a cross) in Figure 1 are spurious given that they can't be concretized into valid trees due to the existing cycles.

Paths in the symbolic execution tree that lead to a spurious structure are spurious paths. It is easy to see that the number of spurious paths can grow exponentially with respect to the scopes,

Figure 2: A representation invariant for binary trees

```
1  public boolean isBinaryTree() {
2    Set<Node> visited = new HashSet<Node>();
3    List<Node> worklist = new LinkedList<Node>();
4    visited.add(this);
5    worklist.add(this);
6    while (!worklist.isEmpty()) {
7      Node node = worklist.remove(0);
8      Node right = node.right;
9      if (right != null) {
10       if (!visited.add(right))
11         return false;
12       worklist.add(right);
13     }
14     Node left = node.left;
15     if (left != null) {
16       if (!visited.add(left))
17         return false;
18       worklist.add(left);
19     }
20   }
21   return true;
22 }
```

as is the case in our example. Thus, efficiently identifying spurious symbolic structures, and pruning their corresponding paths, is essential to improve the performance of symbolic execution and to avoid false positives.

Furthermore, spurious structures can generate infinite loops in the target program, further degrading the performance of the SE. For example, all the spurious branches depicted on figure 1 lead to infinite recursions in `dfs`.

2.2 Representation Invariants as Decision Procedures

As mentioned before, `HybridRepOK`s are manual adaptations of traditional `repOK`s to support partially symbolic structures [17, 28]. Implementing good `HybridRepOK`s is not trivial, as they should be able to identify invalid fields over the concrete parts of the symbolic structure, and ignore symbolic fields for as long as possible.

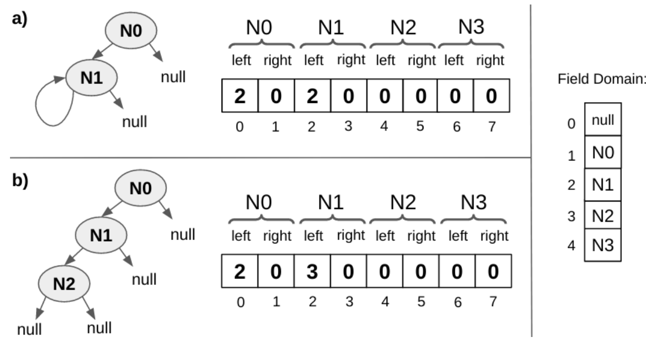
An algorithmic approach to derive a `HybridRepOK` from the `BinaryTree` `repOK` of Figure 2 is to make a `HybridRepOK` that returns true as soon as symbolic field is accessed. The resulting `HybridRepOK` is conservative, as it always returns true for satisfiable partially symbolic structures, but it accepts many spurious structures. For example, for the symbolic structure in branch 2 of Figure 1, it returns true when `N0.right` is accessed in line 8. For the same reason, the spurious structures after branches 5 and 6 are incorrectly classified as satisfiable.

This example illustrates that manual effort is needed to create `HybridRepOK`s that are precise in identifying spurious structures. An additional problem is that the use of `HybridRepOK` bears considerable risk of introducing specification errors. Ensuring that a `HybridRepOK` is sound with respect to the original specification is a non-trivial problem.

2.3 Korat

Korat is a framework to automatically generate structurally complex test inputs [2]. Given a boolean predicate in an imperative programming language (`repOK`), and bounds on the size of the inputs, it exhaustively generates all the non-isomorphic inputs within the bounds for which `repOK` returns true.

Figure 3: Two binary trees and their corresponding candidate vectors



To use Korat, the user needs to provide a `Finitization`, an imperative routine that specifies the maximum number of objects allowed for each class. Korat uses the `Finitization` to create a class domains, defining the sequence of objects of the class that will be employed to generate structures. For instance, assuming a maximum of 4 `Node` objects for our binary tree example, the class domain for `Node` would be `[null,N0, N1,N2,N3]` (one can specify whether to include `null` in class domains in the `Finitization` [2]). Class domains are sorted in Korat, this is why we represent them with sequences. Thus, specific values from class domains can be accessed by indexing the sequence: `null` has index 0, `N0` has index 1, and so on.

The user must also provide a field domain for each field in the `Finitization`. A field domain defines the set of feasible values for the field, and is often defined as the union of one or more class domains (concatenation of corresponding class domains' sequences). Hence, the values of field domains are also sorted in Korat. In our example, as fields `left` and `right` have `Node` type, we set `[null,N0,N1,N2,N3]` as the domain for both fields.

Korat sorts the fields of every object within the bounds (that is, in each class domain), and assigns each field a unique identifier. Thus, Korat represents structures as vectors of integers, called candidate vectors, mapping unique fields identifiers into indices of the corresponding field domains. Figure 3 shows two binary tree instances along with their corresponding candidate vectors. For example, in Figure 3 a), we have that the field `N0.left` (unique identifier 0) has value 2, meaning that `N0.left` references `N1` (`N1` has index 2 in the field domain). The values for the remaining fields can be interpreted similarly.

2.3.1 *Korat's state space exploration.*

Korat explores the state space of candidate vectors within the specified bounds. Initially, it starts the exploration from a vector with all its fields set to zero, which corresponds to the first index in all field domains (usually `null` for reference types).

For each candidate vector, Korat runs `repOK` on the object represented by the vector, while saving the object's accessed fields in a stack (in the order they are accessed by repOK). Korat outputs all structures for which repOK returns true and discards those for which repOK is false. For instance, consider the invocation of repOK in Figure 2 over the binary tree of Figure 3 a). The accessed fields are [N0.right, N0.left, N1.right, N1.left] before returning false, leaving the accessed fields stack with [1,0,3,2].

To obtain the next candidate, Korat backtracks on the sequence of accessed fields. It pops the accessed field of the top of the stack and increments its value in the candidate vector by 1, to make the field point to the next feasible object for the field. If the new value exceeds the limits of the domain, Korat resets the field to zero and continues with the next field in the stack. Continuing with our example, from the candidate vector of Figure 3 a) Korat takes the last accessed field N1.left (with unique identifier 2), and increments its value by 1. This gives to N1.left the value N2 , producing the next candidate shown in Figure 3 b). Notice that this step prunes from the search all the candidate vectors with the form [2,0,2,0,_,_,_,_], where underscores can be filled with any value from the corresponding field domains (5^4 candidates).

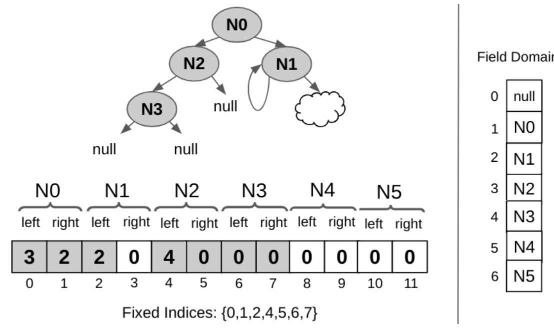
Korat's pruning mechanism is sound, as repOK did not access the last four fields in the vector, it would have returned false irrespective of the values assigned to those fields. This pruning approach allows Korat to efficiently explore huge search spaces [2, 25].

Korat continues the search process described above until the accessed fields stacks becomes empty. At that point, it is guaranteed that all candidate vectors within the bounds satisfying repOK have been explored.

2.3.2 Korat's symmetry breaking approach. Symmetry breaking avoids the generation of isomorphic structures [16, 23]. Two structures are isomorphic when they represent the same structure but have different identifiers assigned to their nodes. For example, if we assign identifier N3 to the node tagged N2 in Figure 3 b), we obtain a structure that is isomorphic to the one we started with. Node identifiers represent the memory addresses of nodes, but in languages without explicit memory manipulation like Java these do not add any useful information for program analysis. Thus, considering a single representative for each set of isomorphic structures is enough from the analysis point of view. Efficiently choosing only one representative for isomorphic structures is what symmetry breaking is about.

To implement symmetry breaking, before increasing the value of a field, Korat computes the largest value of the corresponding field domain (according to the field domain ordering) that is present in the structure. For this, Korat only has to explore the values of fields in the accessed fields stack. The Korat search algorithm guarantees that fields that are not in the stack either they are not part of the structure or its value is not relevant to the structure's validity. That is, let fd be the field domain of the field, let mf be the largest value from fd present in the accessed fields stack, and let i be the current value of the field being considered. If $i <= mf$ the value of the field can be incremented by one to obtain a new candidate. Otherwise, it means that increasing i would lead to a candidate that is isomorphic to the current vector, and thus Korat resets the value of the field to zero and continues by backtracking on the stack of accessed fields.

Figure 4: Partially symbolic structure and the corresponding candidate vector generated by createVector



3 LISSA

In this section we introduce our symbolic execution approach, LISSA, implemented as an extension to SPF's LI engine [20]. LISSA symbolically executes the program under analysis using lazy initialization. After each LI step performed by SPF, LISSA encodes the symbolic structure as a vector, and employs the specialized solver SymSolve to decide about its satisfiability.

SymSolve explores the search space of possible concretizations of its partially symbolic input, in a bounded-exhaustive manner. In this process, SymSolve either finds out a witness showing that the symbolic structure can be fully concretized into a structure satisfying repOK, and returns sat, or the structure is deemed spurious, and returns unsat. In the latter case, the path being explored is pruned, and the symbolic execution is forced to backtrack to continue with the next path.

Below we introduce the contributions of this work in more detail. We refer the reader to the literature for more information on symbolic execution and lazy initialization [5, 17, 20]. Section 3.1 explains how LISSA encodes symbolic structures as candidate vectors. Then, Section 3.2 introduces the SymSolve solver for symbolic structures. Finally, Section 3.3 discusses a novel symmetry breaking approach for SymSolve, to significantly improve its performance and scalability.

3.1 Encoding Symbolic Structures as Candidate Vectors

As mentioned before, SymSolve requires an operational repOK routine and bounds on the size of the structures. As Korat, it represents partially symbolic structures as candidate vectors. However, to handle partially symbolic structures, SymSolve makes the concrete part of the structures fixed during the search. That is, SymSolve explores the state space of concrete structures without allowing the search to change the concrete part of the partially symbolic structure. For instance, Figure 4 shows a partially symbolic binary tree along with its vector representation, computed for a scope of 6 Node objects. Shadowed cells in the vector represent concrete fields of the structure. Thus, the encoding process takes into account both the resulting representation vector and the set of concrete fixed indices.

Figure 5: `createVector` algorithm: conversion of symbolic structures to candidate vectors

```
1  (int[], Set<Integer>) createVector(Object root, int size) {
2    vector = new int[size];
3    fixedIndices = new Set<Integer>();
4    idMap = new Map<Object, Integer>();
5    maxIdMap = new Map<Class, Integer>();
6    idMap.put(root, 0);
7    maxIdMap.put(root.getClass(), 0);
8    worklist = new List<Object>();
9    worklist.add(root);
10   while (!worklist.isEmpty()) {
11     current = worklist.remove(0);
12     for (Field field: current.sortedFields()) {
13       fieldValue = field.getValue(current);
14       if (fieldValue.isSymbolic())
15         continue;  // already set to zero
16       index = uniqueIndex(field, current.getClass());
17       fixedIndices.add(index);
18       if (fieldValue == null)
19         continue;  // already set to zero
20       if (idMap.contains(fieldValue))
21         // previously visited object
22         vector[index] = idMap.get(fieldValue) + 1;
23       else { // first time visited
24         objectClass = fieldValue.getClass();
25         id = 0;
26         if (maxIdMap.contains(objectClass))
27           id = maxIdMap.get(objectClass) + 1;
28         idMap.put(fieldValue, id);
29         maxIdMap.put(objectClass, id);
30         vector[index] = id + 1;
31         worklist.add(fieldValue);
32       }
33     }
34   }
35   return (vector, fixedIndices);
36 }
```

A pseudocode for the encoding algorithm, `createVector`, is shown in Figure 5. We now run the algorithm over the symbolic structure of Figure 4. `createVector` receives a reference to the root of the symbolic structure to be encoded and the vector size (computed from the provided bounds). First, the method creates the candidate vector with the corresponding size, starting with all its fields initialized to zero (line 2). It also initializes an empty integer set to keep track of the concrete indices, which we call `fixedIndices` (line 3). The encoding process must assign identifiers to the objects visited during the traversal of the structure, and given that structures can contain aliasing, it must also keep track of the identifiers of previously visited objects. Therefore, the routine builds a map between objects and identifiers, `idMap` (line 4), and another map, `maxIdMap`, to keep track of the largest identifiers assigned to objects of each class (line 5). The `root` is assigned identifier 0 (lines 6-7), and is added to `workList` to start the traversal of the structure (lines 8-9).

`createVector` traverses the structure in a breadth-first manner; the main while loop of lines 10-34 implements the traversal. For each visited object (referenced by `current` in line 11), the loop at lines 12-33 traverses all its fields (in the order they appear in candidate vectors).

The value to be stored in the vector depends on the value of the field, which is stored in `fieldValue` in line 13. In the following, we assume that all fields are of reference type. If the field has a symbolic value, we have to set `vector[index]` to 0 in the candidate vector for SymSolve to start the exploration for the field from the first value of its field domain. As the vector is already initialized with all zeros from the beginning, the algorithm just continues with the next field (lines 14-15).

If the field is not symbolic, then its unique index in the candidate vector is retrieved by `uniqueIndex` at line 16, and added to the set of fixed indices (line 17). If the field value is `null`, the algorithm also proceeds with the next field (lines 18-19), as `vector[index]` is already set to 0 (the index of `null` in field domains). If the field value is a reference to an object, `createVector` checks whether it has been visited before (line 20). For previously visited objects, the previously assigned identifier is set as the field value in the vector (line 22). Notice from Figure 4 that the field domain index for node with identifier `Ni` is `i+1` (since `null` has index 0). Thus, we set `vector[index]` to `idMap.get(fieldValue) + 1` in line 22. The algorithm creates and assigns a new identifier for objects not yet visited (lines 24-29). For the first object found for a given class, `id` is set to 0 (line 25). Afterwards, the new identifier is obtained by retrieving the largest identifier from `maxIdMap` and increasing it by 1 (line 26-27). The object is assigned the newly created identifier (line 28), and `maxIdMap` is updated to include the new `id` (line 29). The field value is set to `id + 1` in the vector (line 30, for the same reason explained above), and the object is added to `workList` to continue the breadth-first traversal (line 31).

Continuing with the example of Figure 4, as the field `N0.right` points to an object not visited previously, the else statement of line 23 is executed. The largest identifier for a node was 0 (assigned to the root), thus `id = 1` is created. Then, the value of the vector for `N0.right` (index 1) is set to `id + 1`, leaving `vector[1] = 2`.

The algorithm ends when all the fields of the structure have been traversed and returns the created candidate vector along with the set of concrete indices (`fixedIndices`) (line 35).

3.2 SymSolve: A Satisfiability Solver for Symbolic Structures

In this section we introduce SymSolve, our satisfiability solver for symbolic structures. Figure 6 shows a pseudocode of the SymSolve's algorithm. SymSolve receives as inputs the encoding of a partially symbolic structure as a candidate vector (`initialVector`), and the set of concrete fields in the structure (`fixedIndices`), generated by the `createVector` algorithm of the previous section.

The concrete fields of the partially symbolic structure will remain fixed during the search. Intuitively, we want to find out whether the constraints imposed by the partially symbolic structure, represented by its concrete fields, are satisfiable. To decide about satisfiability, SymSolve needs to figure out whether there exists a valuation for the symbolic fields that makes `repOK` return true.

SymSolve starts the search from the candidate vector encoding the partially symbolic input structure, `initialVector` (line 2). SymSolve iteratively builds candidate vectors until the search space of (bounded) concrete structures has been exhausted and no new vector can be created (`vector == null` in line 3). At this point, no valid concretization has been found for the partially symbolic input structure, and SymSolve returns `unsat` (line 9).

Figure 6: SymSolve's algorithm

```
1  boolean SymSolve(int[] initialVector, Set fixedIndices) {
2    vector = initialVector;
3    while (vector != null) {
4      Object structure = buildObject(vector);
5      if (structure.repOK())
6        return true;  // SAT!!
7      vector = getNextVector(vector, accessedIndices, fixedIndices);
8    }
9    return false;     // UNSAT!!
10 }
11
12 int[] getNextVector(int[] vector, Set accessedIndices, Set
        fixedIndices) {
13   while (!accessedIndices.isEmpty()) {
14     int lastIndex = accessedIndices.pop();
15     if (!fixedIndices.contains(lastIndex)) {
16       FieldDomain fd = getFD(lastIndex);
17       Set u = union(fixedIndices, accessedIndices);
18       if (vector[lastIndex] < fd.size() &&
19             vector[lastIndex] <= maxId(fd, u)) {
20         vector[lastIndex]++;
21         return vector;
22       }
23       vector[lastIndex] = 0;   // Backtrack
24     }
25   }
26   return null;
27 }
```

For each explored candidate vector (variable `vector` in the code) SymSolve creates the structure represented by the vector (line 4), and invokes `repOK` over the structure (line 5), while monitoring the structure's accessed fields. As was the case with Korat, and was explained in Section 2.3, the unique field identifiers representing the fields are saved in a stack. We assume the accessed fields stack is saved in global variable `accessedIndices` after executing `repOK`.

If `repOK` returns true, a valid concretization of the partially symbolic input structure has been found, and `SymSolve` returns `sat` (line 6). Otherwise, the search continues by invoking `getNextVector` to obtain the next candidate vector (line 7).

`getNextVector` (line 12) tries to create the next candidate vector by backtracking on the stack of accessed fields, `accessedIndices` (in the while loop of lines 13-25). If there are accessed fields in the stack, the algorithm pops the index of the last accessed field, `lastIndex` (line 14), and tries to increase the value of that field in the vector, if feasible. As mentioned before, only non-fixed indices are modified, so if `lastIndex` is fixed it is ignored (line 15) and the search continues with the next index in the stack. Notice that this helps `SymSolve` to prune large parts of the search space, as it does not need to try out any other values for fixed fields. For example, for the vector in Figure 4, `repOK` returns false and the stack of accessed field indices is `[1,0,3,2]`. Then, as 2 is a fixed index (it is shadowed in the Figure), the algorithm proceeds with the next field in the stack.

For non-fixed indices, the algorithm needs to determine if it's feasible to increment the current value of the field with index `lastIndex` to create a new candidate vector. There are two conditions that must be satisfied for a new vector to be created. First, the new value for the field must reference a valid object within the field's domain (`vector[lastIndex] < fd.size()` in line 18). Second, the new value for the field must not generate an isomorphic input (lines 17 and 19). We leave the explanation of the symmetry breaking algorithm of SymSolve for the next section.

If the next value for the field is feasible, SymSolve increases the field value by 1 (line 20) and the newly created candidate vector is returned (line 21). Otherwise, SymSolve backtracks by setting the value of the field to 0 (line 23), and it continues with the next field in the stack. Similarly to Korat, when SymSolve increases a field value after `repOK` returns false for the current vector, large parts of the search space are pruned that contain only invalid structures with respect to `repOK`. An example of this kind of pruning was shown in Figure 3, Section 2.3.

When the `accessedIndices` stack becomes empty, no more vectors can be created from the current vector (line 13). Then, `getNextVector` returns `null` (line 26) and SymSolve's search finishes.

Continuing further with our example of Figure 4, as the spuriousness of the symbolic structure is caused by the loop in the fixed field `N1.left`, SymSolve will exhaust the options for the non-fixed fields (`N1.right, N4.left, N4.right, N5.left, N5.right`) and will never be able to find a concrete structure satisfying `repOK`, thus determining the input structure to be unsatisfiable.

3.3 A Symmetry Breaking Approach for SymSolve

Let us start by remarking that the symmetry breaking approach of Korat does not work for symbolic structures. As Korat search is performed according to the fields accessed by `repOK`, the fields that `repOK` did not access are not reachable from the root of the structure. But when deciding satisfiability of a partially symbolic structure, the candidate vector might have fields reachable from the root that `repOK` did not access: the concrete fields from the input symbolic structure that are set as fixed throughout the search. Thus, when assigning a new value for a field, breaking symmetries only considering `repOK`'s accessed fields may cause the search to miss feasible assignments of values to fields.

For instance, after executing `repOK` for the vector in Figure 4, the stack of accessed fields is `[1,0,3,2]`. At this point, field with index 2 is popped from the stack and ignored because it's fixed, and the field `N1.right` (with identifier 3) is popped next, leaving the stack of accessed indices with `[1,0]`. Now the algorithm has to decide whether making `N1.right` point to the next node generates an isomorphic input or not. Looking only at fields in the stack (`[1,0]`), `N2` (index 3) is the largest node identifier accessed for the field domain. Thus, following Korat's symmetry breaking approach, `N3` (index 4) is the largest node identifier that is allowed to be assigned to `N1.right`. However, this would make SymSolve miss the valid possibility of setting `N1.right` to `N4` (index 5, which does not generate an isomorphic structure). Missing feasible assignment of values to fields can lead to SymSolve reporting a symbolic structure as `unsat` when it's in fact `sat`, which in turn can make the symbolic execution of the program under analysis to prune feasible paths and miss faults.

To correctly break symmetries in SymSolve, we have to consider fields in the stack and fixed fields when computing the largest accessed node identifier for the field domain. Thus, SymSolve computes the union of the set of fixed indices and the stack of accessed indices, called `u` (line 17), and then computes the largest node identifier assigned to the fields in `u` (`maxId(fd, u)` at line 19). Then, the symmetry breaking condition allows increasing the field (`vector[lastIndex]`) if it's lesser or equal than `maxId(fd, u)` (line 19). This symmetry breaking approach is sound, i.e., it only prunes isomorphic structures.

In our previous example, `N3` (index 4) is the largest node identifier in the union of accessed and fixed fields (`maxId(fd, u) = 4`), and therefore the value of `N1.right` can be incremented until it receives `N4` (index 5) as its value (line 20). Furthermore, the symmetry breaking algorithm does not allow `N5` (index 6) as a value for `N1.right`. This is correct, since the result would be a structure isomorphic to the one with `N1.right = N4`.

4 EXPERIMENTAL ASSESSMENT

The goal of our experimental evaluation is to answer the following research questions:

- RQ1: How does LISSA perform in comparison to existing approaches in the analysis of programs manipulating complex heap-allocated structures with rich constraints?
- RQ2: How much does the proposed symmetry breaking approach for SymSolve contributes to the performance of LISSA?

Section 4.1 presents the experiments performed in order to answer RQ1, and Section 4.2 discusses the experiments for RQ2.

As case studies, we include several widely-used data structure implementations from the Java standard library (`java.util`). We analyze a linked list implementation (`LinkedList`); red-black tree based implementations of sets and maps (`TreeSet` and `TreeMap`, respectively); and a map implemented using a hash table (`HashMap`). We also include five classes from different projects of the SF110 benchmark [11], that are clients of the aforementioned data structure implementations. `Template` from the `templateit` project, which stores data in a `LinkedList` (of `Parameter` type), indexed by name using a `HashMap`. `TransportStats` from the `vuze` project, which keeps track of bytes read and written in two separate `TreeMaps`. `DictionaryInfo` from `fixsuite`, which stores data (`FieldInfo`) indexed by name and by tag using two different `TreeMaps`. `SQLFilter` from `squirrel-sql`, which defines a `HashMap` of `HashMap`'s to store information about database queries. `CombatantStatistic` from the `twfbplayer` project, defines a `HashMap` of `HashMap`'s for storing game statistics. Finally, we include a scheduler implementation, `Schedule`, from the well known SIR benchmark [9] (implemented with four linked lists).

The experiments were run in a workstation with a Xeon Gold 6154 CPU (72 virtual cores running at 3GHz), and Debian Linux 11 OS. The assessed approaches only use a single CPU core, and were executed with Java's default maximum heap size of 4Gb. We set a maximum time of 2 hours (7200 seconds) for each individual run. Executions exceeding this time were interrupted, and we report them as TO in Table 1.

4.1 LISSA vs related approaches

For this assessment, we considered related approaches that do not require further specification effort beside writing a `repOK` in the same programming language as the code under analysis. Thus, we ruled out approaches that require significant additional effort from the developer, like writing a manually tailored `HybridRepOK`, or creating additional declarative specifications. Following this criteria, the approaches included in the evaluation are:

Driver. One of the most common approaches to symbolically execute programs taking heap-allocated structures as inputs. The user must write a "driver" program, that employs methods from the API and non-deterministic constructs to populate the heap before symbolic execution the program under analysis. For completeness, the driver should generate all the valid structures with up to k nodes (using symbolic values for fields of primitive type in the structures). Notice that, if methods employed in the driver are correct, the generated structures satisfy the precondition of the program by construction (`repOK` in our experiments). In many cases, using a constructor and an insertion method suffices for the driver. For example, a typical driver for `TreeSet` executes the constructor first, and then the `add()` method a non-deterministically selected number of times, up to a maximum of k times. Drivers employ symbolic values for primitive type parameters, like the integer parameter of `add()` in a `TreeSet` of integers.

LIHybrid. This approach is SPF's built-in lazy initialization exploration, augmented with a `HybridRepOK` that is automatically derived from a concrete `repOK` (as explained in Section 2.2).

IFrepOK. This technique consists of symbolically executing `repOK` using lazy initialization to generate all the bounded heap-allocated structures with up to k nodes that satisfy `repOK`, previous to the symbolic execution of the method under analysis. The approach can be summarized by the following simplified pseudocode: `if repOK(str) { M(str); }`. Similarly to Driver, IFrepOK ends up exhaustively enumerating all valid bounded structures and running the code under test with all of them.

LISSA. Our symbolic execution approach introduced in section 3. We ran all experiments in this section with `SymSolve`'s symmetry breaking enabled.

LISSA-M. It adds memoization capabilities to LISSA. Each time a new candidate vector is generated by LISSA, we first search a cache and return the previously computed answer for the vector (`sat` or not) if it exists. Otherwise, SymSolve is invoked and the vector and its result are saved in the cache.

All the approaches above were either built-in or implemented by the authors in the (SPF) tool [20].

4.1.1 Metrics. We ran all the approaches in all our case studies for increasingly large scopes, until a maximum scope of 50 is reached or a timeout occurs. For each run, we report the runtime of the approach (`time` columns in Table 1) and the number of paths generated in its symbolic execution tree (`paths` columns in Table 1)

With respect to symbolic paths generated, the less the better, as all techniques only prune infeasible paths, although with different degrees of precision. Basically, if a technique produces more symbolic paths, it either explores redundant paths (due to treating some data concretely) or infeasible paths, that do not represent any concrete execution. Time is also highly relevant, as more precise pruning techniques may not pay off due to their cost; the objective here is to produce the fewer total paths possible (guaranteeing that feasible paths are not missed, of course) in the least time possible. Full symbolic path coverage is in fact a kind of worst case scenario for symbolic execution, thus being the motivation of our evaluation.

Class	Method	Scope	LIHybrid		Driver		IFrepOK		LISSA		LISSA-M	
			time	**paths (spurious)**	**time**	**paths**	**time**	**paths**	**time (solving)**	**paths**	**time (solving)**	**paths**
Template	addParameter	1	414	514736 (513216)	3	70144	1	5120	1 (0)	1520	1 (0)	1520
		2	TO	-	925	20263680	26	108512	4 (0)	11040	4 (0)	11040
		4			TO	-	4298	9573824	191 (146)	88480	68 (21)	88480
		5					TO	-	1314 (1181)	223136	271 (125)	223136
		7							TO	-	4589 (2957)	1266592
	getParameter	2	228	224240 (224080)	42	1258000	2	6112	0 (0)	160	0 (0)	160
		5	TO	-	TO	-	1545	2835440	6 (5)	1504	2 (1)	1504
		9					TO	-	3127 (3103)	24544	452 (431)	24544
		11							TO	-	4122 (4014)	98272
TransportStats	bytesWritten	5	5322	7470714 (7470586)	729	80520	33	4038	1 (0)	128	0 (0)	128
		8	TO	-	TO	-	2628	52140	219 (215)	328	111 (108)	328
		9					TO	-	1249 (1238)	328	617 (606)	328
		10							TO	-	6099 (6016)	472
	bytesRead	5	5268	7470714 (7470586)	569	80520	32	4038	1 (0)	128	1 (0)	128
		8	TO	-	TO	-	2615	52140	13 (10)	328	8 (5)	328
		11					TO	-	3227 (2632)	792	1906 (1297)	792
SQLFilter	put	1	TO	-	16	336896	16	63104	2 (0)	6336	2 (0)	6336
		3			TO	-	TO	-	2149 (1971)	446656	870 (677)	446656
	get	2	TO	-	340	8487944	2677	3866128	1 (0)	2000	1 (0)	2000
		3			TO	-	TO	-	1785 (1778)	10032	650 (644)	10032
DictionaryInfo	addField	4	TO	-	407	298485	6	256	4 (1)	2209	4 (2)	2209
		6			TO	-	912	1440	360 (330)	13225	355 (324)	13225
		7					TO	-	3289 (3106)	32041	2935 (2765)	32041
	getField	5	1177	94 (72)	2628	2575096	78	304	0 (0)	22	0 (0)	22
		6	4334	190 (144)	TO	-	953	720	0 (0)	46	0 (0)	46
		12	TO	-			TO	-	3166 (1773)	94	2946 (1645)	94
Schedule	addProcess	6	0	9 (3)	1713	410133	2	630	0 (0)	6	0 (0)	6
		35	0	9 (3)	TO	-	6586	246753	0 (0)	6	0 (0)	6
		50	0	9 (3)			TO	-	0 (0)	6	0 (0)	6
	quantumExpire	7	2	1038 (995)	1767	781241	5	3210	0 (0)	43	0 (0)	43
		27	6842	1038 (995)	TO	-	1831	372505	250 (249)	43	276 (276)	43
		34	TO	-			6228	889665	1272 (1271)	43	1225 (1224)	43
		45					TO	-	7136 (7135)	43	7115 (7114)	43
CombatantStatistic	addData	2	3255	666 (0)	196	10638	1687	83268	2 (0)	666	2 (0)	666
		4	TO	-	TO	-	TO	-	296 (263)	12978	50 (22)	12978
		5							TO	-	5668 (5551)	53586
	ensureTypExists	2	297	80 (0)	49	4952	4934	30136	0 (0)	80	1 (0)	80
		3	1291	176 (0)	2505	126104	TO	-	1 (0)	176	1 (0)	176
		4	4849	368 (0)	TO	-			3 (0)	368	3 (0)	368
		12	TO	-					5373 (1010)	98288	4392 (18)	98288
HashMap	put	2	467	160 (0)	35	5168	18	3056	1 (0)	160	1 (0)	160
		3	2395	352 (0)	TO	-	663	21616	2 (0)	352	2 (0)	352
		11	TO	-			TO	-	3166 (2231)	98272	1493 (272)	98272
		13							TO	-	7044 (1658)	393184
	remove	3	1927	655 (255)	2173	97088	23	21616	1 (0)	400	1 (0)	400
		4	6132	1583 (735)	TO	-	176	120752	1 (0)	848	1 (0)	848
		6	TO	-			3977	2327984	7 (2)	3536	6 (1)	3536
		12					TO	-	3737 (3264)	229328	1149 (617)	229328
		13							TO	-	2753 (1566)	458704
TreeMap	put	5	921	1202220 (1202149)	5	5316	2	152	0 (0)	71	0 (0)	71
		7	TO	-	855	598444	58	855	1 (0)	179	1 (0)	179
		9			TO	-	3022	3517	15 (14)	179	15 (14)	179
		12					TO	-	4888 (4886)	427	4733 (4731)	427
	remove	4	1266	193811 (193724)	0	633	0	64	0 (0)	87	0 (0)	87
		7	TO	-	768	598444	56	855	3 (2)	1106	2 (1)	1106
		9			TO	-	2822	3517	31 (27)	2804	24 (20)	2804
		11					TO	-	1212 (1201)	8482	820 (811)	8482
TreeSet	add	5	940	1202220 (1202149)	5	5316	2	152	0 (0)	71	0 (0)	71
		7	TO	-	831	598444	59	855	1 (0)	179	1 (0)	179
		9			TO	-	2871	3517	17 (16)	179	14 (14)	179
		12					TO	-	4881 (4880)	427	4684 (4682)	427
	remove	4	1143	193811 (193724)	0	633	0	64	0 (0)	87	0 (0)	87
		7	TO	-	803	598444	57	855	6 (3)	1106	4 (1)	1106
		9			TO	-	2771	3517	27 (23)	2804	17 (14)	2804
		11					TO	-	1172 (1161)	8482	851 (840)	8482
LinkedList	add	50	0	2 (0)	0	51	0	50	0 (0)	2	0 (0)	2
	remove	50	7158	45509 (45362)	2	1326	2	1275	5 (4)	147	5 (5)	147

Table 1: Comparison of symbolic execution approaches for programs manipulating complex heap-allocated structures

LIHybrid is expected to be bad at identifying spurious structures and explore a large number of spurious paths (see Section 2.2). Thus, we report the number of spurious paths LIHybrid explores (spurious, in parentheses, in Table 1). As an oracle for spurious structures, we run SymSolve on the structures at the end of each path explored by LIHybrid. For LISSA and LISSA-M, we also measured the time expended in SymSolve's solving (solving, in parentheses, in Table 1).

4.1.2 Results and discussion. Table 1 summarizes the results of the experiment. Due to space reasons, we only display selected scopes, always including the highest scope reached by each approach. The full experimental results and a replication package for the experiments can be found online [1].

LISSA vs lazy approaches. LIHybrid is the worst performing approach. The reason is that it does not identify many spurious structures and hence a high proportion of the paths it explores are spurious. This makes LIHybrid explore a much larger number of paths than the remaining approaches in most cases, when considering the same scope. This implies that the automatically generated `HybridRepOK` precision is low in most cases.

Spurious structures containing cycles that lead to infinite loops in the method under analysis are also frequent, and this is an additional overhead for LIHybrid that other approaches do not suffer.

In contrast, SymSolve's effectiveness in pruning spurious paths allowed LISSA to perform better and scale up to much higher scopes than LIHybrid, as can be noticed by the much smaller number of paths explored by LISSA (for the same scopes). Even if it's more costly than executing `HybridRepOK`, the additional overhead of employing SymSolve greatly pays off. It is important to remark that SymSolve is sound and it never prunes valid paths from the program under analysis.

Finally, LISSA-M shows an improved performance w.r.t. LISSA in most cases, and it scales up to one or more scope for 6 methods (out of 20).

LISSA vs eager approaches. Let us compare LISSA to eager approaches, i.e., approaches that enumerate structure's shapes before symbolic execution of the code under analysis (Driver and IFrepOK). First, notice that Driver explores a larger number of paths than IFrepOK and performs worse in almost all cases. We believe there are two reasons for this. First, most insertion routines in our case studies carry out complex operations (like balancing trees), and symbolically executing them is more costly than symbolically executing `repOK`. Second, there are often many ways of employing insertion routines to create exactly the same structure shape (e.g. inserting the same element once and twice in a set). This makes Driver invoke the program under analysis with the same shapes many times, unnecessarily exploring redundant program paths. Driver scales much worse than LISSA in all cases but `LinkedList` (we discuss this case below).

A comparison of LISSA against IFrepOK remains. For the most complex case studies, that involve multiple data structures (`Schedule`, `DictionaryInfo`, `SQLFilter`, `TransportStats`, `Template` and `CombatantStatistic`), LISSA is more efficient and scales much better than IFrepOK, reaching several more scopes in all cases. We believe this is because the more structures involved, the (much) larger number of structures' shapes to be enumerated by eager approaches, and in particular by IFrepOK, and this number eventually becomes intractable when the scopes grow sufficiently large.

For the most complex data structure implementations (`HashMap`, `TreeMap`, `TreeSet`), LISSA also performs better than IFrepOK, scaling up a few more scopes. The complexity of the repOKs of these structures make symbolically executing them difficult, and this seem to be hampering IFrepOK's performance.

For the simplest structure, `LinkedList`, and its `remove` method, LISSA explores an order of magnitude less paths than IFrepOK and Driver. Still, for scope 50 it takes LISSA 5 seconds to run, but IFrepOK and Driver run in 2 seconds. LISSA is still very fast for such a large scope in this case.

LISSA works best in cases where the method under test only accesses a constant number of nodes in the input structure. For example, `addProcess` from `Scheduler` appends a process at the end of a linked list. As there's a field referencing the last element of the list, appending involves setting the next field of the last element to a newly created node, and updating the last reference. The same happens with `LinkedList`'s `add` method. In such cases, LISSA's visits a constant number of paths due to its laziness, no matter the scope. In contrast, eager approaches generate all the structure's shapes for the scope.

From the results one can observe that LISSA often explores an order or magnitude or less paths than eager techniques. However, SymSolve is a sound pruning technique, so it never prunes valid paths (that arise from satisfiable partially symbolic structures) in the symbolic execution of the program under analysis. The reason for this much fewer number of explored paths is that LISSA is a lazy approach, and thus it concretizes only the part of the structure that is accessed by the program under analysis, leaving the rest symbolic. In a sense, a symbolic path of LISSA (with a partially symbolic structure) represents many symbolic paths with concrete structures generated by eager techniques. That is, a symbolic path of LISSA represents all those symbolic paths generated by eager techniques with concrete structures that match the concrete part of the partially symbolic structure. For example, while searching for a key in a binary search tree LISSA only needs to concretize a path from the root to a leaf in the input tree, leaving the remaining fields of the tree symbolic. On the other hand, eager approaches will create a large number of trees that match the symbolic tree (all the feasible concretizations of the symbolic fields within the bounds), and all of these trees would result in the (undesired) exploration of the same symbolic path of the search method repeatedly.

Class	Method	Scope	LISSA-NoSB time (solving)	LISSA time (solving)
HashMap	remove	6	63 (58)	7 (2)
		7	1101 (1090)	18 (8)
		8	TO	55 (32)
		12		3737 (3264)
TreeMap	put	8	84 (83)	4 (3)
		9	2254 (2253)	15 (14)
		10	TO	132 (131)
		12		4888 (4886)
Schedule	quantumExpire	7	43 (43)	0 (0)
		8	422 (422)	0 (0)
		9	4963 (4962)	1 (0)
		10	TO	1 (0)
		45		7136 (7135)
TreeSet	remove	8	100 (98)	10 (8)
		9	2590 (2586)	27 (23)
		10	TO	183 (178)
		11		1172 (1161)
CombatantStatistic	addData	2	2 (0)	2 (0)
		3	33 (25)	16 (6)
		4	TO	296 (263)
DictionaryInfo	addField	4	6 (3)	4 (1)
		5	84 (78)	23 (17)
		6	5532 (5498)	360 (330)
		7	TO	3289 (3106)
Template	addParameter	3	27 (12)	25 (11)
		4	388 (343)	191 (146)
		5	TO	1314 (1181)

Table 2: Impact of `SymSolve`'s symmetry breaking on the performance of `LISSA`

4.2 Impact of SymSolve's symmetry breaking

The goal of this section is to figure out how much the defined symmetry breaking approach for SymSolve (Section 3.3) contributes to the performance of symbolic execution.

Thus, we run for increasingly large scopes the LISSA approach (it has symmetry breaking enabled by default) and its version without symmetry breaking, called LISSA-NoSB. Table 2 shows the results for some selected case studies (representative of the results for the remaining cases). The full results can be found online [1]. The symmetry breaking approach of SymSolve is crucial for the performance and the scalability of LISSA. LISSA is faster, and reaches significantly higher scopes in all case studies w.r.t. LISSA-NoSB.

Without SymSolve symmetry breaking LISSA would not be able to outperform existing approaches for symbolic execution in most cases (like IFrepOK). For example, both LISSA-NoSB and IFrepOK reach the same scope (9) in about the same time for `TreeSet`'s `remove` and `TreeMap`'s `put` (see Tables 1 and 2).

5 RELATED WORK

Lazy initialization (LI) introduced a novel way of symbolically executing programs manipulating heap-allocated inputs, and the idea of employing user provided `HybridRepOK` routines to identify spurious symbolic structures [17]. The technique favors modular analysis using symbolic execution, and has a number of limitations that we have described earlier in this paper. Among the techniques that improve LI, BLISS [24] is related to our approach, as it tackles the identification of spurious symbolic structures. The approach differs from ours in various aspects. First, BLISS precomputes bounds on the feasible values for structure fields, as dictated by the representation invariant [12]. Second, it combines the execution of automatically derived `HybridRepOK` and SAT solving, for which it requires a declarative specification of the representation invariant (in addition to the `repOK`), in order to identify spurious structures during LI. This allows BLISS to be faster and scale up to larger scopes than LI, at the cost of requiring the user to provide an additional declarative specification of the representation invariant. HEX also improves over lazy initialization by introducing a new specification language to describe properties of symbolic structures [4]. The specification language allows the user to provide additional information to aid symbolic execution to perform better. Both [24] and [4] aim at improving lazy initialization by requiring a significant amount of extra effort from the user. The learning curve of declarative languages for programmers has been shown to be steep [25]. The addition of different types of specifications also bears considerable risk of introducing errors. Ensuring that specifications in different languages describe exactly the same properties is a non-trivial problem. In contrast with both BLISS and HEX, LISSA improves LI without requiring additional specification effort, besides a traditional `repOK`.

Other symbolic execution based approaches deal with heap-allocated structures in different ways. Pex, based on dynamic symbolic execution, asks the user to manually provide a set of factory methods that create the structures, and makes these participate in the dynamic symbolic execution, thus implementing "eager" concretization [27]. Seeker builds on Pex and tries to automatically search for sequences of API method calls to build the heap-allocated structures, using static and dynamic analysis to guide the generation [26]. Seeker targets programs in C#, and also performs eager concretization. SUSHI also deals with the problem of searching for API method sequences to build heap-allocated structures, and it works with Java programs [3]. SUSHI builds on JBSE, and requires specifications of the representation invariants in the HEX declarative language, as opposed to the more traditional operational `repOK`. In any case, both Seeker and SUSHI can be employed to solve a problem that is complementary to symbolic execution (and thus also to our technique LISSA), namely the problem of producing a sequence of methods generating specific structures that symbolic execution needs to cover program paths.

KLEE is an automated test input generator for C programs based on symbolic execution [6]. KLEE does not implement lazy initialization, but rather starts symbolic execution from an empty, fully concrete heap. To the best of our knowledge, it is represented by the Driver approach assessed in our experiments (see Section 4).

A former empirical study compared several constraint solvers for complex heap-allocated structures with rich constraints [25]. The results showed that Korat was the most efficient one [25]. The impressive efficiency of Korat in the study was an important factor in motivating this work.

6 CONCLUSION

Symbolic execution is an important technique with many applications in software analysis, including test input generation and program verification. As many programs need to handle heap-allocated data, and this is known to be challenging to deal with for approaches based on symbolic execution, improving the support for such data is highly relevant for the effectiveness of symbolic execution.

We introduced LISSA, a technique that improves lazy initialization via an effective approach to detect spurious heap-allocated symbolic structures (SymSolve). Detecting such structures is important, as it allows symbolic execution to deem program paths infeasible, in a way similar to deeming path conditions unsatisfiable. SymSolve, performs an efficient bounded-exhaustive exploration over the space of concrete structures to decide if a partially symbolic structure can be fully concretized in a way that satisfies structural constraints, given as an operational routine (e.g. a `repOK`). As opposed to related techniques, LISSA does not require additional efforts from the developer, such as ad-hoc harnesses for structure construction, or logical specifications of the structural constraints.

We assessed LISSA on a benchmark of programs manipulating complex heap-allocated data, including well-known implementations of data structures, as well as larger "client" programs of such structures, taken from real-world projects. The results show that maintaining a symbolic heap (i.e. a heap that is representative of many concrete ones), as lazy approaches do, helps to significantly reduce the number of symbolically executed paths that treat the heap concretely. Moreover, the use of an efficient structural constraint solver (as LISSA does with SymSolve) to prune invalid lazy initializations is critical to achieve more scalability; the time spent in solving symbolic heaps amortizes the time costs of exploring many concrete heaps (eager techniques) or many spurious paths (as Li-Hybrid). Consequently, LISSA constitutes a convenient mechanism for symbolically executing programs that handle heap-allocated data, especially in cases where such data is assumed to satisfy structural constraints. This convenience is associated with fewer requirements for its application, and the efficiency of the resulting symbolic execution.

REFERENCES

[1] [n. d.]. Website for paper "LISSA: Lazy Initialization with Specialized Solver Aid". https://sites.google.com/view/lissa-paper/home.

[2] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. 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*, Phyllis G. Frankl (Ed.). ACM, 123–133. https://doi.org/10.1145/566172.566191

[3] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. 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*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 90–101. https://doi.org/10.1145/3092703.3092715

[4] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2015. 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*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 602–613. https://doi.org/10.1145/2786805.2786842

[5] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2016. JBSE: a symbolic executor for Java programs with complex heap inputs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 1018–1022. https://doi.org/10.1145/2950290.2983940

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. 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*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[7] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[9] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empir. Softw. Eng.* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[10] Bruno Dutertre. 2014. Yices 2.2. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

[11] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 8:1–8:42. https://doi.org/10.1145/2685612

[12] Jaco Geldenhuys, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. 2013. Bounded Lazy Initialization. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7871)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer, 229–243. https://doi.org/10.1007/978-3-642-38088-4_16

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. 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*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[14] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue* 10, 1 (2012), 20. https://doi.org/10.1145/2090147.2094081

[15] Divya Gopinath, Mengshi Zhang, Kaiyuan Wang, Ismet Burak Kadron, Corina S. Pasareanu, and Sarfraz Khurshid. 2019. 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*, Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro (Eds.). IEEE, 313–322. https://doi.org/10.1109/ISSRE.2019.00039

[16] Radu Iosif. 2002. Symmetry Reduction Criteria for Software Model Checking. In *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2318)*, Dragan Bosnacki and Stefan Leue (Eds.). Springer, 22–41. https://doi.org/10.1007/3-540-46017-9_5

[17] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2003. 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 (Lecture Notes in Computer Science, Vol. 2619)*, Hubert Garavel and John Hatcliff (Eds.). Springer, 553–568. https://doi.org/10.1007/3-540-36577-X_40

[18] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2018. Test input generation with Java PathFinder: then and now (invited talk abstract). In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 1–2. https://doi.org/10.1145/3213846.3234687

[19] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[20] Kasper Søe Luckow and Corina S. Pasareanu. 2014. Symbolic PathFinder v7. *ACM SIGSOFT Softw. Eng. Notes* 39, 1 (2014), 1–5. https://doi.org/10.1145/2557833.2560571

[21] Corina S. Pasareanu. 2020. *Symbolic Execution and Quantitative Reasoning: Applications to Software Safety and Security.* Morgan & Claypool Publishers. https://doi.org/10.2200/S01010ED2V01Y202005SWE006

[22] Corina S. Pasareanu, Rody Kersten, Kasper Søe Luckow, and Quoc-Sang Phan. 2019. Chapter Six - Symbolic Execution and Recent Applications to Worst-Case Execution, Load Testing, and Security Analysis. *Adv. Comput.* 113 (2019), 289–314. https://doi.org/10.1016/bs.adcom.2018.10.004

[23] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. 2003. Space-Reduction Strategies for Model Checking Dynamic Software. *Electron. Notes Theor. Comput. Sci.* 89, 3 (2003), 499–517. https://doi.org/10.1016/S1571-0661(05)80009-X

[24] Nicolás Rosner, Jaco Geldenhuys, Nazareno Aguirre, Willem Visser, and Marcelo F. Frias. 2015. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *IEEE Trans. Software Eng.* 41, 7 (2015), 639–660. https://doi.org/10.1109/TSE.2015.2389225

[25] Junaid Haroon Siddiqui and Sarfraz Khurshid. 2009. An Empirical Study of Structural Constraint Solving Techniques. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5885)*, Karin K. Breitman and Ana Cavalcanti (Eds.). Springer, 88–106. https://doi.org/10.1007/978-3-642-10373-5_5

[26] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 189–206. https://doi.org/10.1145/2048066.2048083

[27] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4966)*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer, 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

[28] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. 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*, George S. Avrunin and Gregg Rothermel (Eds.). ACM, 97–107. https://doi.org/10.1145/1007512.1007526

[29] Willem Visser, Corina S. Pasareanu, and Radek Pelánek. 2006. Test input generation for java containers using state matching. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, Lori L. Pollock and Mauro Pezzè (Eds.). ACM, 37–48. https://doi.org/10.1145/1146238.1146243