Enhancing Symbolic Execution of Heap-based Programs with Separation Logic for Test Input Generation

Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin

Abstract. Symbolic execution is a well established method for test input generation. Despite of having achieved tremendous success over numeric domains, existing symbolic execution techniques for heap-based programs are limited due to the lack of a succinct and precise description for symbolic values over unbounded heaps. In this work, we present a new symbolic execution method for heap-based programs based on separation logic. The essence of our proposal is context-sensitive lazy initialization, a novel approach for efficient test input generation. Our approach differs from existing approaches in two ways. Firstly, our approach is based on separation logic, which allows us to precisely capture preconditions of heap-based programs so that we avoid generating invalid test inputs. Secondly, we generate only fully initialized test inputs, which are more useful in practice compared to those partially initialized test inputs generated by the state-of-the-art tools. We have implemented our approach as a tool, called Java StarFinder, and evaluated it on a set of programs with complex heap inputs. The results show that our approach significantly reduces the number of invalid test inputs and improves the test coverage.

1 Introduction

Symbolic execution [22] is getting momentum thanks to its capability of discovering deep bugs. It is increasingly used not only in academic settings but also in industry, such as in Microsoft, NASA, IBM and Fujitsu [11]. Despite having achieved tremendous success, symbolic execution has limited impact on testing programs with inputs in the form of complex heap-based data structures (a.k.a. *heap-based* programs). The dominant approach to symbolic execution of heap-based programs is lazy initialization [21], which postpones the initialization of reference variables and fields until they are accessed (either assigned to or de-referenced). However, lazy initialization makes no assumption on the shape of the input data structure, and explicitly enumerates all possible heap objects that may bind to the input. This approach has the following fundamental limitations. Firstly, due to the lack of a succinct and precise description of the shape of the input data structures, they often generate a large number of invalid test inputs. Secondly, due to the enumeration of all possible heap objects that may bind to the input, they often worsen the path explosion problem of symbolic execution. Lastly, due to lazy initialization, the generated test inputs may be partially initialized (if some fields are never accessed) and need to be further concretized.

In the context of logic-based verification, the problem of specifying and reasoning about heap-based programs has been studied for nearly five decades. The dominant approaches are based on separation logic [20,32]. The strength of separation logic lies in its separating conjunction $*$, which splits the heap into disjoint regions or *heaplets*.

This enables *local reasoning*, i.e., specification and reasoning are kept confined within heaplets, independent of the rest of the heap. This is in contrast to *global reasoning*, i.e., the specification describes properties of the global heap, which "suffers from either limited applicability or extreme complexity, and scales poorly to programs of even moderate size" [32].

Surprisingly, there has been limited effort on using separation logic to enhance symbolic execution for test input generation. In this work, we start filling this gap. Firstly, we propose a novel method for symbolic execution of heap-based programs based on separation logic. In particular, we adopt a logic that combines separation logic with existentially quantified variables, inductive predicates and arithmetic which allows us to encode path conditions effectively in heap-based programs. Secondly, we enhance our method with *context-sensitive* lazy initialization, i.e., we use preconditions written in separation logic to guide the search, and only explore the states that are reachable when the input satisfies the preconditions. As a result, all generated test inputs are valid.

In summary, we make the following main contributions.

1. We develop a symbolic execution engine for heap-based programs based on separation logic.
2. For efficiency, we present context-sensitive lazy initialization with a *least fixed point* analysis to generate valid test inputs during symbolic execution.
3. We have implemented the proposed approach as a tool, called Java StarFinder[1], built on top of Symbolic PathFinder [31], to generate test inputs for Java bytecode.
4. We have evaluated our tool on a set of Java programs including complex and mutable data structures. All generated test inputs are valid and we achieve 98.98% branch coverage on average.

The rest of this paper is organized as follows. Sect. 2 presents some background and illustrates our proposal via an example. Sect. 3 describes the syntax of our core language and its operational semantics. Our first contribution, a symbolic execution engine based on separation logic, is presented in Sect. 4. Our second contribution, the context-sensitive lazy initialization, is shown in Sect. 5. We present our implementation and evaluation in Sect. 6. Sect. 7 presents related work. Finally, we conclude and discuss future works in Sect. 8.

2 Motivation and Illustration

In the following, we illustrate how our approach works with an example. Consider a program that represents a big non-negative integer in the form of a singly linked list, i.e., each node of the lists contains a single digit of the number. Suppose we want to generate test inputs for the method `add` shown in Figure 1, which computes the sum of two numbers in this representation.

This method is designed to take two parameters `x` and `y` satisfying the following preconditions: (1) all the digits of `x` and `y` are less than 5, and thus there is no carry; (2) `x` and `y` have the same number of digits. Condition (1) is a simple numerical constraint which can be handled by existing symbolic execution engines. We thus leave it out of

[1] `https://github.com/star-finder/jpf-star`

```
1 node add(node x, node y) {
    node dummyHead = new node(0, null); node z = dummyHead;
3   while (x!=null) {
      z.next = new node(x.val + y.val, null);
5     x = x.next;  y = y.next;  z = z.next;
    }
7   return dummyHead.next;
}
```

Fig. 1. Adding two numbers represented by linked lists

the discussion for the sake of simplicity. To capture condition (2), we define an inductive predicate `pre` based on our fragment of separation logic as follows.

$$\begin{aligned}\texttt{pred}\ \texttt{pre}(a,b) \equiv\ & (\texttt{emp} \wedge a = \texttt{null} \wedge b = \texttt{null}) \\ & \vee (\exists n_1, n_2 . a{\mapsto}\texttt{node}(\_, n_1) * b{\mapsto}\texttt{node}(\_, n_2) * \texttt{pre}(n_1, n_2))\end{aligned}$$

Predicate `emp` means the heap is empty; predicate $a{\mapsto}\texttt{node}(\_, n_1)$ states a points to an allocated object; and $*$ is the separation operator in separation logic. The data type `node` corresponds to the class `node` in the program, which has two instance fields: `val` containing the digit, and `next` referencing to another `node` object. The wildcard "_" is used to indicate a "don't care" value.

Intuitively, a linked list is recursively defined as a head points-to predicate with its next field pointing to a sublist. In the base case of the definition, the heap is empty, and both parameters a and b are `null`. In the recursive case, $a{\mapsto}\texttt{node}(\_, n_1)$ signifies that a points to an allocated object composed of a certain value (represented by "_") and its `next` field n_1. Similarly, b points to an allocated object with its `next` field n_2. Furthermore, n_1 and n_2, i.e., the sublists of a and b respectively, satisfy `pre` as well.

In this definition, the separating conjunction operator $*$ splits the global heap into three heaplets. The first two heaplets contain the `node` objects referenced respectively by a and b, and the third one contains the sublists. This separation enforces that a and b refer to two distinct objects and their sublists are disjoint too. Since a and b must be either both `null` or both `not null`, and likewise for the objects in their sublists, a and b have the same length.

To generate test inputs, we perform symbolic execution of method $\texttt{add}(\texttt{x}, \texttt{y})$ with precondition $\texttt{pre}(\texttt{x},\texttt{y})$. In the proposed symbolic execution, path conditions are formulae in the fragment of separation logic with inductive predicates and arithmetic. Reference variables are initialized by values obtained from a procedure, called `enum`. Initially, `x` and `y` are initialized to symbolic (stack) values X and Y respectively, and the path condition Δ is initialized to $\texttt{pre}(X,Y)$. When variable `x` is first accessed at line 3, our engine, through procedure `enum`, examines precondition $\texttt{pre}(X,Y)$[2] for possible heap values for `x`. Procedure `enum` gets possible values through the least fixed point analysis with procedure `LFP`. `LFP` unfolds predicate `pre` until the set of values reaches a fixed point. In this example, `LFP` only needs to unfold predicate `pre` once and reach fixed point with two formulae corresponding to the two disjuncts in the definition of

[2] In all path conditions in this example we only show the constraints over those variables which are relevant to the inputs `x` and `y`; the constraints over local variables `z` and `dummyHead` are separated from `x`, `y` and thus are omitted for simplicity.

predicate pre. Then the engine substitutes the predicate $\texttt{pre}(X,Y)$ in the Δ by these two formulae (with α-renaming, i.e., substitutions of formal/actual parameters and of bound variables to avoid name collisions) to obtain two non-deterministic choices, and symbolic execution case splits. It first explores the path corresponding to the base case and hence the constraint over X and Y in Δ becomes $X{=}\texttt{null} \wedge Y{=}\texttt{null}$. We use the constraint solver $\texttt{S2SAT}_{\texttt{SL}}$ [24,35,26] to check that Δ is satisfiable. There is no further case splitting in this path, and we have a test input where x and y are both null.

After exploring the base case, our symbolic executor explores the path corresponding to the recursive case. The updated path condition Δ over X and Y is:

$$\exists n_1, n_2 . X{\mapsto}\texttt{node}(\_, n_1) * Y{\mapsto}\texttt{node}(\_, n_2) * \texttt{pre}(n_1, n_2)$$

Executing the body of the loop, at line 5 x.next is dereferenced; hence n_1 is to be accessed. Since n_1 is constrained by pre, our engine again tries two possible values for n_1 resulting from LFP. For the base case, the path condition Δ over X, Y, n_1 and n_2 is:

$$\exists n_1, n_2 . X{\mapsto}\texttt{node}(\_, n_1) * Y{\mapsto}\texttt{node}(\_, n_2) \wedge n_1{=}\texttt{null} \wedge n_2{=}\texttt{null}$$

Then, n_2 is accessed via y.next. Since it has been assigned to null already, there is no case splitting. $n_1{=}\texttt{null}$ violates the looping condition so symbolic execution finishes exploring the path and backtracks. We obtain a test input where x and y both have one digit. Likewise, we generate test inputs where x and y both have two digits, three digits and so on. Note that we put a bound on the number of unfolding for loops.

In contrast to ours which always generates valid test inputs, the existing lazy approaches [36,9,10] would generate invalid test inputs such as (i) x and y have different number of digits; or (ii) x and y are aliasing; or (iii) x (or y) is a cyclic linked list.

3 A Core Language

In [34], Schwartz *et al.* described the algorithm of symbolic execution as an extension to the run-time semantics of a general language. The language, called SimpIL, is simple but "powerful enough to express typical languages as varied as Java and assembly code" [34]. In this work, we use a similar presentation to describe our new symbolic execution engine. This section introduces our core language, which is an extension of SimpIL with operations on heap memory. Note that our implementation is for Java bytecode, and our approach extends to other languages.

Syntax The syntax of the language is defined in Figure 3. A program in our core language consists of multiple data structures and statements. The primitive types include integer, boolean and void; statements consist of assignment, memory store, goto, assertion, conditional goto, memory allocation, and memory deallocation; expressions are side-effect free and consist of typical non-heap expressions and memory load. We use $\texttt{op}_\texttt{b}$ to represent typical binary operators, e.g., addition, subtraction. Similarly, $\texttt{op}_\texttt{u}$ is used to represent typical unary operators, e.g., logical negation. k represents either a 32-bit integer constant or a boolean value (true or false). The expressions used together with goto should not contain variables. For the sake of simplicity, we assume the programs are in the form of static single assignments (SSA) and are well-typed in the standard way.

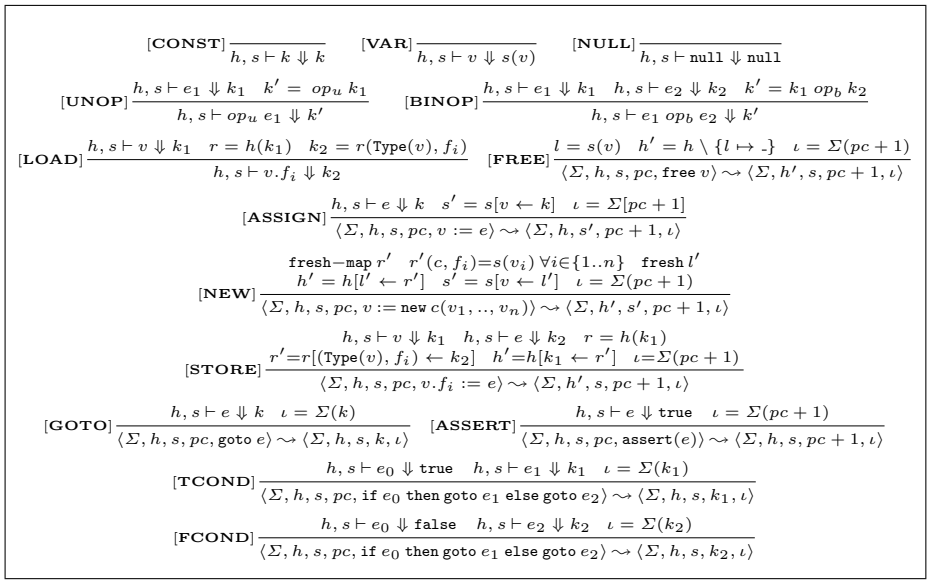

Fig. 2. Operational semantics of the core language

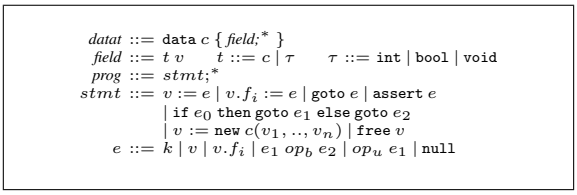

Fig. 3. A core intermediate language

Operational semantics The *concrete* execution configuration of a program defined by the syntax shown in Figure 3 is a tuple of five components $\langle \Sigma, h, s, pc, \iota \rangle$. Σ is the list of program statements; h is the current memory state (i.e., the heap); s records the current value of program variables (i.e., the stack); pc is the program counter; and ι is the current statement. Among these, Σ, h and s are mapping functions: Σ maps a number to a statement; h maps a memory location to its content; s maps a variable to its value.

The concrete heap h of type *Heaps* assumes a fixed finite collection *Node*, a fixed finite collection *Fields*, a disjoint set *Loc* of locations (i.e., heap addresses), a set of non-address values *Val*, such that $\texttt{null} \in \mathit{Val}$ and $\mathit{Val} \cap \mathit{Loc} = \emptyset$. We define *Heaps* as:

$$\mathit{Heaps} \stackrel{\text{def}}{=} \mathit{Loc} \rightharpoonup_{fin} (\mathit{Node} \rightarrow \mathit{Fields} \rightarrow \mathit{Val} \cup \mathit{Loc})$$

Further, a concrete stack s is of type *Stacks*, defined as follows:

$$\mathit{Stacks} \stackrel{\text{def}}{=} \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}$$

$$
\begin{array}{rl}
\Phi & ::= \Delta \mid \Phi_1 \vee \Phi_2 \\
\Delta & ::= \exists \bar{v}.\, (\kappa \wedge \alpha \wedge \phi) \\
\kappa & ::= \texttt{emp} \mid v \mapsto c(f_i{:}v_i) \mid P(\bar{v}) \mid \kappa_1 * \kappa_2 \\
\alpha & ::= \texttt{true} \mid v_1{=}v_2 \mid v{=}\texttt{null} \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2 \\
\phi & ::= \texttt{true} \mid a_1{=}a_2 \mid a_1 \leq a_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \\
a & ::= k \mid v \mid k \times a \mid a_1 + a_2 \mid -a \\
\mathit{Pred} & ::= \texttt{pred}\ P_1(\bar{v}_1) \equiv \Phi_1 ;\ \cdots ;\ \texttt{pred}\ P_N(\bar{v}_N) \equiv \Phi_n
\end{array}
$$

Fig. 4. Syntax of separation logic

is a mapping from a variable to a value or a memory address. We use $[x \leftarrow k]$ to denote updating a variable x with value k for mapping functions; for example, $s[x \leftarrow 13]$ denotes a new stack that is the same as stack s except that it maps variable x to the value 13. The operational semantics of our language is shown in Figure 2. The rules are of the following form:

$$\frac{\text{computation}}{\langle \text{current state} \rangle \ \leadsto \ \langle \text{end state} \rangle}$$

The computation in a rule is read from the top to the bottom, the left to the right, and are applied based on syntactic pattern-matching. Given a statement, our engine finds a rule to execute the computation on the top and returns the end state in the case of success. If no rule matches (e.g., accessing a dangling pointer), the execution halts. In these rules, `fresh` is used as an overloading function to return a new variable/address. Similarly, `fresh-map` returns a new mapping and `Type` returns the type of a variable.

For the evaluation of expressions, we use $h, s \vdash e \Downarrow k$ to denote the evaluation of expression e to a value k in the current context h and s. The application of these rules is also based on pattern-matching similar to the application of the statements above.

For example, rule [NEW] describes the operational semantics of the command that allocates dynamic heaps. Firstly, it creates a new mapping r' to relate fields of the new object to their stack values. Next, it generates a new heap entry at the fresh address l'. Lastly, it updates the stack value of the variable with the heap address.

4 Symbolic Execution

This section presents details on symbolic execution using a separation logic-based language to encode path conditions in heap-based programs.

Separation logic We use separation logic [20,32] to capture symbolic heaps and expressions. Separation logic, an extension of Hoare logic, is a state-of-the-art assertion language designed for reasoning about heap-based programs. It provides concise and precise notations for reasoning about the heap. In particular, it supports the separating conjunction $*$ that splits the global heap into disjoint sub-heap regions, each of which can be analysed independently. Combined with inductive predicates, separation logic has been shown to capture semantics of unbounded heaps, loops and recursive procedures naturally and succinctly [24,35,26].

In the following, we define the separation logic formulae used in this work to encode path conditions of heap-based programs. A separation logic formula is defined by the syntax presented in Figure 4. We assume that $c \in \mathit{Node}$ is a heap node; $f_i \in \mathit{Fields}$ is a

$$[\mathbf{S-CONST}]\ \frac{}{\Delta, s \vdash k \Downarrow k} \quad [\mathbf{S-VAR}]\ \frac{}{\Delta, s \vdash v \Downarrow s(v)} \quad [\mathbf{S-NULL}]\ \frac{}{\Delta, s \vdash \mathtt{null} \Downarrow \mathtt{null}}$$

$$[\mathbf{S-UNOP}]\ \frac{\Delta, s \vdash e_1 \Downarrow \pi_1}{\Delta, s \vdash op_u\ e_1 \Downarrow\ op_u\ \pi_1} \quad [\mathbf{S-BINOP}]\ \frac{\Delta, s \vdash e_1 \Downarrow \pi_1 \quad \Delta, s \vdash e_2 \Downarrow \pi_2}{\Delta, s \vdash e_1\ op_b\ e_2 \Downarrow \pi_1\ op_b\ \pi_2}$$

$$[\mathbf{S-LOAD}]\ \frac{\exists \bar{w}.l{\mapsto}c(v_1,..,v_i,..,v_n){*}\Delta, s \vdash v \Downarrow l \quad \exists \bar{w}.l{\mapsto}c(v_1,..,v_i,..,v_n){*}\Delta, s \vdash v_i \Downarrow \pi_i}{\exists \bar{w}.l{\mapsto}c(v_1,..,v_i,..,v_n){*}\Delta, s \vdash v.f_i \Downarrow \pi_i}$$

$$[\mathbf{S-FREE}]\ \frac{\Delta, s \vdash v \Downarrow l \qquad \iota = \Sigma(pc+1)}{\langle \Sigma, \exists \bar{w}.l{\mapsto}c(\ldots){*}\Delta, s, pc, \mathtt{free}\ v\rangle \rightsquigarrow \langle \Sigma, \exists \bar{w}.\Delta, s, pc+1, \iota\rangle}$$

$$[\mathbf{S-ASSIGN}]\ \frac{\Delta, s \vdash e \Downarrow \pi \quad s'{=}s[v \leftarrow \pi] \quad \iota = \Sigma[pc+1]}{\langle \Sigma, \Delta, s, pc, v := e\rangle \rightsquigarrow \langle \Sigma, \Delta, s', pc+1, \iota\rangle}$$

$$[\mathbf{S-NEW}]\ \frac{\mathtt{fresh}\ l' \quad \Delta' {\equiv} \Delta * l'{\mapsto}c(v_1,..,v_n) \quad s'{=}s[v \leftarrow l'] \quad \iota = \Sigma(pc{+}1)}{\langle \Sigma, \Delta, s, pc, v = \mathtt{new}\ c(v_1,..,v_n)\rangle \rightsquigarrow \langle \Sigma, \Delta', s', pc{+}1, \iota\rangle}$$

$$[\mathbf{S-STORE}]\ \frac{\begin{array}{c} l{\mapsto}c(v_1,..,v_i,..,v_n) \in \Delta \quad \Delta, s \vdash v \Downarrow l \\ \Delta, s \vdash e \Downarrow \pi \quad s'{=}s[v_i \leftarrow \pi] \quad \iota = \Sigma(pc+1) \end{array}}{\langle \Sigma, \Delta, s, pc, v.f_i = e\rangle \rightsquigarrow \langle \Sigma, \Delta, s', pc+1, \iota\rangle}$$

$$[\mathbf{S-GOTO}]\ \frac{\Delta, s \vdash e \Downarrow k \quad \iota = \Sigma(k)}{\langle \Sigma, \Delta, s, pc, \mathtt{goto}\ e\rangle \rightsquigarrow \langle \Sigma, \Delta, s, k, \iota\rangle} \quad [\mathbf{S-ASSERT}]\ \frac{\Delta, s \vdash e \Downarrow \pi \quad \Delta' {\equiv} \Delta \wedge \pi \quad \iota = \Sigma(pc{+}1)}{\langle \Sigma, \Delta, s, pc, \mathtt{assert}(e)\rangle \rightsquigarrow \langle \Sigma, \Delta', s, pc{+}1, \iota\rangle}$$

$$[\mathbf{S-TCOND}]\ \frac{\Delta, s \vdash e_0 \Downarrow \pi_0 \quad \Delta, s \vdash e_1 \Downarrow k_1 \quad \Delta' {\equiv} \Delta {\wedge} \pi_0 \quad \iota = \Sigma(k_1)}{\langle \Sigma, \Delta, s, pc, \mathtt{if}\ e_0\ \mathtt{then\ goto}\ e_1\ \mathtt{else\ goto}\ e_2\rangle \rightsquigarrow \langle \Sigma, \Delta', s, k_1, \iota\rangle}$$

$$[\mathbf{S-FCOND}]\ \frac{\Delta, s \vdash e_0 \Downarrow \pi_0 \quad \Delta, s \vdash e_2 \Downarrow k_2 \quad \Delta' {\equiv} \Delta {\wedge} \neg \pi_0 \quad \iota = \Sigma(k_2)}{\langle \Sigma, \Delta, s, pc, \mathtt{if}\ e_0\ \mathtt{then\ goto}\ e_1\ \mathtt{else\ goto}\ e_2\rangle \rightsquigarrow \langle \Sigma, \Delta', s, k_2, \iota\rangle}$$

Fig. 5. Symbolic operational execution rules

field; and v, v_i represent variables. We notice that each kind of heap nodes c corresponds to a data structure declared by the user using the keyword `data` in our core language. We write $\bar{v}$ to denote a sequence of variables. A separation logic formula is denoted as Φ, which can be either a symbolic heap Δ or a disjunction of them. A symbolic heap Δ is an existentially quantified conjunction of some spatial formulae κ, some pointer (dis)equality α, and some formulae in arithmetic ϕ [16]. All free variables in Δ, denoted by function $FV(\Delta)$, are either program variables or implicitly universally quantified at the outermost level. The spatial formula κ may be a separating conjunction ($*$) of `emp` predicate, points-to predicates $v{\mapsto}c(f_i{:}v_i)$, and predicate applications $P(\bar{v})$. Whenever possible, we discard f_i of the points-to predicate and use its short form as $v{\mapsto}c(\bar{v})$. Note that $v_1{\neq}v_2$ and $v{\neq}\mathtt{null}$ are short forms for $\neg(v_1{=}v_2)$ and $\neg(v{=}\mathtt{null})$ respectively. Each inductive predicate is defined by a disjunction Φ using the keyword `pred`. In each disjunct, we require that variables which are not formal parameters must be existentially quantified. We use $\Delta[v_1/v_2]$ for a substitution of all occurrences of v_2 in Δ to v_1.

Symbolic execution Recall that the concrete execution configuration is a 5-tuple. The *symbolic* execution configuration is also a tuple of five components: $\langle \Sigma, \Delta, s, pc, \iota\rangle$ where Δ is a path condition in the form of a separation logic formula defined above and s is used to map every variable to a symbolic value[3]. The rest of the components

[3] We use the same symbol s as in concrete setting. From the context, it should be clear as to whether we are referring to symbolic stack or concrete stack.

are similar to those of the concrete execution configuration, except that symbolic values of variables are captured in Δ and s. We use π (and π_i) to denote symbolic values. Memory allocations are symbolically captured in the path condition Δ. A symbolic execution configuration $\langle \Sigma, \Delta, s, pc, \iota \rangle$ is *infeasible* if Δ is unsatisfiable. Otherwise, it is *feasible*.

All operational symbolic execution rules over our language are shown in Figure 5. In these rules, similar to Figure 2 we use function `fresh` to return a fresh variable. We illustrate the execution through rule [S–NEW]. This rule allocates a new object of type c and assigns to variable v. Firstly, it generates a fresh symbolic address l' and updates the stack to map v to this address. Secondly, it creates new symbolic heap for l' by separately conjoining the current path condition with a new points-to predicate $l' \mapsto c(v_1, .., v_n)$. Lastly, it loads the next statement using the program counter. Note that we assume each variable v_i is only used to create at most one new object.

Rule [S–FREE] symbolically de-allocates the heaps. To capture the de-allocated heaps for test input generation, we keep track the corresponding points-to predicates by storing them in a "garbage" formula. At the end of execution, those predicates are plugged into the current path condition before being used to generate test inputs.

5 Lazy Test Input Generation

In this section, we present the test input generation based on the symbolic execution engine we depicted in the previous sections. The inputs of our method are a program `prog` in the language we defined in Section 3 and an optional precondition Δ_{pre} in the form of a separation logic formula defined in Section 4. The output is a set of fully initialized test inputs that satisfy the precondition and often achieve high test coverage. Our method is based on lazy initialization. The main difference between our method and previous approaches based on lazy initialization is that we generate values of reference variables and fields in a context-sensitive manner.

Our symbolic execution engine starts with the configuration: $\langle \Sigma, \Delta_{pre}\sigma, s_0, pc_0, \iota_0 \rangle$ where σ is a substitution of input variables to their corresponding symbolic values, s_0 is an initial mapping of input variables to symbolic values, pc_0 and ι_0 denote the first value of the program counter and the first statement respectively. The engine systematically derives the strongest postcondition of every program path (with a bound on the number of loop unfolding), by applying the symbolic operational execution rules (i.e., shown in Figure 5). After obtaining the strongest post-state, our engine invokes $\texttt{S2SAT}_{\texttt{SL}}$ to check whether or not the resultant symbolic heap is satisfiable, and generate a model if it is. The model is then transformed into a test input.

Context-sensitive lazy initialization Recall that lazy initialization [21] leaves a reference variable or field uninitialized until it is first accessed and then enumerates all possible valuations of the variable or field. For instance, as shown in Figure 5, a reference variable or field can be accessed in the rules [S–VAR], [S–LOAD], [S–STORE], or [S–FREE]. The problem is that many valuations obtained through enumeration are invalid (i.e., violating the precondition). Thus, in this work, we propose context-sensitive enumeration. When an uninitialized reference variable or field with symbolic value

Algorithm 1: Procedure LFP

```
1  SV_i ← {P_i(t̄_i)} ; A_i ← false ;                          /* i ∈ {1..N} */
2  while true do
3  |  SV'_i ← {} ;                                            /* i ∈ {1..N} */
4  |  foreach i ∈ {1..N} do
5  |  |  foreach Δ_j ∈ SV_i do
6  |  |  |  SV'_i ← SV'_i ∪ unfold(Δ_j) ;
7  |  |  A'_i ← ⋁{∃w̄.abs(Π(κ∧α, t̄_i), t̄_i) | (∃w̄.κ∧α∧φ) ∈ SV'_i} ;
8  |  if ∀i ∈ {1..N}. A'_i ⇒ A_i then
9  |  |  return S̄V ;                                          /* fixed point */
10 |  else
11 |  |  SV_i ← SV'_i ; A_i ← A'_i ;                          /* i ∈ {1..N} */
```

v is accessed during symbolic execution, procedure $\texttt{enum}(v, \Delta, s)$ is invoked to non-deterministically initialize v with values derived from the symbolic execution context, i.e., Δ and s. For each value of the set, the symbolic execution engine creates a new branch for exploration. Procedure $\texttt{enum}(v, \Delta, s)$ works based on the following three scenarios.

1. If Δ, s implies that v has previously been initialized to either `null` (i.e., $v = \texttt{null}$) or a points-to predicate (i.e., $v \mapsto \_ \in \Delta$), initializing again is not necessary.
2. If v is uninitialized and there does not exist a predicate $P_i(\bar{v}_i)$ in Δ such that $v \in \bar{v}_i$ (i.e., v is not constrained by the context), v is initialized to `null`, to a new points-to predicate with uninitialized fields, or to a points-to predicate in Δ.
3. If v is uninitialized and there exists some predicate $P_i(\bar{v}_i)$ in Δ such that $v \in \bar{v}_i$ (i.e., v is constrained by the predicate), we substitute P_i by SV_i to get a set of possible values of v which are consistent with the context Δ. SV_i is computed using the least fixed point analysis as shown below.

Least fixed point analysis We assume that there are N predicate definitions $P_1(\bar{t}_1)$, ..., $P_N(\bar{t}_N)$ ($\mathcal{P}$ for short). We then use the procedure LFP to compute SV_1, ..., SV_N ($\bar{SV}$ for short) according to each predicate. Each SV_i stores all possible contexts (in the form of separation logic formulae) for $\bar{t}_i$ that could be derived from $P_i(\bar{t}_i)$. After having all $\bar{SV}$, in scenario 3 mentioned above, we substitute $P_i(\bar{v}_i)$ with $SV_i[\bar{v}_i/\bar{t}_i]$ to get all possible values for v. Because of the substitution, new variables may be introduced in Δ, we update s accordingly by using the variables' names as their symbolic values. Note that we only compute $\bar{SV}$ once before running the symbolic execution engine.

The details of LFP are shown in Algorithm 1. LFP takes the set of predicate definitions $\mathcal{P}$ as input. It outputs the set of symbolic heap formulae $\bar{SV}$ of all predicates. In this algorithm, each A_i, a disjunctive base formula (i.e., a formula without any occurrence of inductive predicates), captures the abstraction of all formulae in accordance with SV_i. In intuition, LFP iteratively explores each SV_i (initialized with $\{P_i(\bar{t}_i)\}$ at line 1) into a set of disjoint, complete and "smaller" contexts. At the same time, it computes an abstraction A_i over heap allocations and (dis)equality constraints over $\bar{t}_i$. If

the fixed point (at lines 8-9) is achieved, LFP stops. Otherwise, it moves to the next iteration.

In particular, for the first task LFP enumerates all possible symbolic heap locations which $\bar{t}_i$ can be assigned to. The enumeration is performed through the function $\texttt{unfold}(\Delta_j)$ (at line 6), which replaces every occurrence of inductive predicates in Δ_j by their corresponding definitions with α-renaming. As a result, each disjunct in SV_i generates a new context.

For the second task, after the new contexts have been derived, at line 7, LFP computes an abstraction on the set of symbolic values which every parameter $\bar{t}_i$ can be assigned to. This abstraction is critical for the termination of the algorithm and is computed with two functions. Intuitively, these two functions compute constraints on parameters of each inductive predicate. The first function $\texttt{abs}(\kappa \wedge \alpha, \bar{t}_i)$ captures the reference values of $\bar{t}_i$, while the second function $\Pi(\kappa \wedge \alpha, \bar{t}_i)$ captures (dis)equality constraints on $\bar{t}_i$.

In particular, the function abs is defined as $\texttt{abs}(\texttt{emp} \wedge \alpha, \bar{v}) = \texttt{emp} \wedge \alpha$. Otherwise,

$$\texttt{abs}(v{\mapsto}c(\bar{w}) * \kappa_1 \wedge \alpha, \bar{v}) = \begin{cases} v{\mapsto}c(\bar{w}) * \texttt{abs}(\kappa_1 \wedge \alpha, \bar{v}) & \text{if } v{\in}\bar{v} \\ \texttt{abs}(\kappa_1 \wedge \alpha, \bar{v}) & \text{otherwise} \end{cases}$$

$$\texttt{abs}(P(\bar{w}) * \kappa_1 \wedge \alpha, \bar{v}) = \begin{cases} \texttt{false} & \text{if } \bar{w} \cap \bar{v} \neq \emptyset \\ \texttt{abs}(\kappa_1 \wedge \alpha, \bar{v}) & \text{otherwise} \end{cases}$$

In principle, this function retains all heap nodes allocated by variables in $\bar{v}$, maps inductive predicates with arguments in $\bar{v}$ to false , and discards other constraints in κ.

The function $\Pi(\kappa{\wedge}\alpha, \bar{v})$ eliminates (dis)equality constraints in $\kappa{\wedge}\alpha$ on all variables which are not in $\bar{v}$. In particular, $\Pi(\kappa{\wedge}\texttt{true}, \bar{v}){=}\kappa{\wedge}\texttt{true}$, $\Pi(\kappa{\wedge}\texttt{false}, \bar{v}){=}\texttt{false}$, and $\Pi(\kappa \wedge v_1{\neq}v_1 \wedge \alpha_1, \bar{v}){=}\texttt{false}$. Otherwise,

$$\Pi(\kappa \wedge v_1{=}e \wedge \alpha_1, \bar{v}) = \begin{cases} \Pi(\kappa \wedge \alpha_1[\texttt{null}/v_1], \bar{v}) & \text{if } e{=}\texttt{null} \wedge v_1{\notin}\bar{v} \\ \Pi((\kappa \wedge \alpha_1)[v_2/v_1], \bar{v}) & \text{if } e{=}v_2 \wedge v_1{\notin}\bar{v} \\ \Pi((\kappa \wedge \alpha_1)[v_1/v_2], \bar{v}) & \text{if } e{=}v_2 \wedge v_2{\notin}\bar{v} \wedge v_1{\in}\bar{v} \\ \Pi(\kappa \wedge \alpha_1, \bar{v}) \wedge v_1{=}e & \text{otherwise} \end{cases}$$

$$\Pi(\kappa \wedge v_1{\neq}e \wedge \alpha_1, \bar{v}) = \begin{cases} \Pi(\kappa \wedge \alpha_1, \bar{v}) \wedge v_1{\neq}e & \text{if } e{=}\texttt{null} \wedge v_1{\in}\bar{v} \vee e{=}v_2 \wedge v_1{\in}\bar{v} \wedge v_2{\in}\bar{v} \\ \Pi(\kappa \wedge \alpha_1, \bar{v}) & \text{otherwise} \end{cases}$$

An equality $v_1{=}v_2$ is retained if both v_1 and v_2 are in $\bar{v}$. Otherwise, it is eliminated and one of the variables must be eliminated via a substitution. A disequality $v_1{\neq}v_2$ is retained if both v_1 and v_2 are in $\bar{v}$. Otherwise, it is eliminated. Similarly, $v_1{=}\texttt{null}$ and $v_1{\neq}\texttt{null}$ are retained if v_1 is in $\bar{v}$. After applying the above two functions, formulae may contain redundant variables in $\exists\bar{w}$, which may be eliminated.

Correctness Correctness of the proposed enumeration method follows the correctness of the procedure LFP. We argue that LFP is sound (i.e., all generated values are correct), terminating, and complete (i.e., all possible heap and (dis)equality constraints between reference parameters in each predicate are captured at fixed point).

Theorem 1 (Soundness). *If $\Delta_j \in SV_i$ and $h, s \models \Delta_j$ then $h, s \models P_i(\bar{t}_i)$.*

i	SV^i	A^i
0	$\texttt{pre}(a, b)$	$\texttt{false}$
1	$\texttt{emp} \wedge a{=}\texttt{null} \wedge b{=}\texttt{null}$ $\vee \exists n_1, n_2.a{\mapsto}\texttt{node}(\_, n_1){*}b{\mapsto}\texttt{node}(\_, n_2){*}\texttt{pre}(n_1{,}n_2)$	$\texttt{emp} \wedge a{=}\texttt{null} \wedge b{=}\texttt{null}$ $\vee \exists n_1, n_2.a{\mapsto}\texttt{node}(\_, n_1){*}b{\mapsto}\texttt{node}(\_, n_2)$
2	$\texttt{emp} \wedge a{=}\texttt{null} \wedge b{=}\texttt{null}$ $\vee \exists n_1, n_2.a{\mapsto}\texttt{node}(\_, n_1){*}b{\mapsto}\texttt{node}(\_, n_2) \wedge n_1{=}\texttt{null} \wedge n_2{=}\texttt{null}$ $\vee \exists n_1, n_2, n_3, n_4.a{\mapsto}\texttt{node}(\_, n_1){*}b{\mapsto}\texttt{node}(\_, n_2){*}$ $n_1{\mapsto}\texttt{node}(\_, n_3){*}n_2{\mapsto}\texttt{node}(\_, n_4){*}\texttt{pre}(n_3{,}n_4)$	$\texttt{emp} \wedge a{=}\texttt{null} \wedge b{=}\texttt{null}$ $\vee \exists n_1, n_2.a{\mapsto}\texttt{node}(\_, n_1){*}b{\mapsto}\texttt{node}(\_, n_2)$ $\vee \exists n_1, n_2.a{\mapsto}\texttt{node}(\_, n_1){*}b{\mapsto}\texttt{node}(\_, n_2)$

Fig. 6. LFP for the motivating example

Proof. The soundness of LFP follows the correctness of the unfolding, i.e., Δ_j is derived through the unfolding of $P_i(\bar{t}_i)$ and $P_i(\bar{t}_i) \equiv \bigvee\{\Delta_j \mid \Delta_j \in SV_i\}$. Hence, Δ_j is an under-approximated formula of $P_i(\bar{t}_i)$. □

Theorem 2 (Termination). *Suppose M be the maximal arity among the inductive predicates $P_i(\bar{t}_i)$, $i \in \{1..N\}$. Then* LFP *runs in $\mathcal{O}(N2^{M^2+M})$.*

Proof. The complexity of LFP relies on the number of disjuncts computed by two functions $\texttt{abs}(\kappa{\wedge}\alpha, \bar{t}_i)$ and $\Pi(\kappa{\wedge}\alpha, \bar{t}_i)$. It comes from the following three sub-components.

- As the maximal arity of $\bar{t}_i$ is M, the number of (dis)equalities among these variables is $\mathcal{O}(M^2)$. The number of its subsets is $\mathcal{O}(2^{M^2})$.
- Furthermore, the maximum number of points-to predicates is M. Hence, the number of its subsets is $\mathcal{O}(2^M)$.
- There are N predicate definitions in the system.

Hence, the implication at line 8 in Algorithm 1 holds in a finite number of iterations. □

Theorem 3 (Completeness). *If $h, s \models P_i(\bar{t}_i)$ then $\exists \Delta_j \in SV_i$ s.t. $h, s \models \Delta_j$. Moreover, SV_i captures all possible heap and (dis)equality constraints between reference variables in $\bar{t}_i$.*

Proof. The first part follows the correctness of the unfolding. For the second part, notice that at least one of $A_1, ..., A_N$ gets weaker via each iteration until all of them reach fixed point. Each A_i is derived from its according SV_i by two functions $\texttt{abs}(\kappa{\wedge}\alpha, \bar{t}_i)$ and $\Pi(\kappa \wedge \alpha, \bar{t}_i)$, which captures all heap and (dis)equality constraints between $\bar{t}_i$.

□

Example 1. We demonstrate the computation of SV for predicate $\texttt{pre}(a, b)$ in Sec 2. The computation is summarized in Figure 6 where i is the number of the iteration.

Since $A^2 \Rightarrow A^1$, LFP stops after two iterations and produces SV^1 as the set of new contexts for this example. After that, the engine substitutes SV^1 into Δ to obtain the two corresponding symbolic heaps.

6 Implementation and Evaluation

We have implemented our approach described in previous sections into a tool, called Java StarFinder (JSF), consisting of 11569 lines of Java code. The architecture of JSF

Table 1. Experimental results

Program	JSF				JBSE				BBE			
	#Tests	Cov.(%)	#Calls	T(s)	#Tests	Cov.(%)	NCov.(%)	T(s)	#Tests	Cov.(%)	NCov.(%)	T(s)
DLL	74/74	**100**	325	49	121/5146	56	100	206	0/35	0	21	21
AVL	69/69	**100**	623	400	76/295	**100**	100	48	17/117	70	89	69
RBT	314/314	**100**	2070	2256	137/291	87	91	38	14/380	26	53	333
SUSHI	7/7	**100**	30	5	0/900	0	100	24	2/27	25	25	8
TSAFE	5/5	**24**	13	3	0/32	0	5	10	0/1	0	0	1
Gantt	21/21	**100**	140	25	17/887	55	90	24	0/6	0	5	2
SLL	26/26	**100**	55	11	-	-	-	-	16/50	66	71	19
Stack	18/18	**100**	31	7	-	-	-	-	11/14	84	84	6
BST	182/182	**100**	698	241	-	-	-	-	19/260	69	86	131
AAT	103/103	**100**	1179	1981	-	-	-	-	3/166	6	43	111
Tll	3/3	**100**	11	2	-	-	-	-	1/4	38	50	2

was briefly described in our previous work [30]. In the following, we evaluate JSF in order to answer three research questions (RQ). All experiments are conducted on a laptop with 2.20GHz Intel Core i7 and 16 GB RAM.

Our experimental subjects include *Singly Linked List* (SLL), *Doubly Linked List* (DLL), *Stack*, *Binary Search Tree* (BST), and *Red Black Tree* (RBT) from SIR [5]; *AVL Tree* (AVL) and *AA Tree* (AAT) from Sierum/Kiasan [4], the motivation example, *TSAFE* project, and *Gantt* project used in SUSHI [8] and a data structure called *Tll* [23]. Since JSF is yet to support string, array, and object oriented features such as inheritance and polymorphism, we exclude data structures and methods which rely on these features, i.e., *Disjoint Set* in Sierum/Kiasan and *Google Closure* in SUSHI. Supporting these features is left for future work. In total, our experiment subjects include a total of 74 methods, whose lines of code range from dozens to more than one thousand.

RQ1: Can JSF reduce invalid test inputs? To answer this question, we need to check whether a generated test input is valid or not. In the benchmark programs which we collect, six data structures contain $repOK$ methods which are designed to check if the input is valid or not, i.e., *Stack*, *DLL*, *BST*, *RBT*, *AVL*, and *AAT*. In addition, the motivation example in SUSHI is based on *DLL* and thus we use the method $repOK$ of *DLL* to validate its test inputs. We write the $repOK$ methods manually for the remaining test subjects. For *SLL* and *Tll*, we write their $repOK$ methods based on their standard definition. For *TSAFE* and *Gantt*, we write their $repOK$ methods after reading the source code, i.e., the $repOK$ encodes the condition required to avoid $RuntimeException$ such as $NullPointerException$.

For each generated test input, we check its validity by passing it as arguments to the corresponding $repOK$ method [36]. If $repOK$ returns `true`, the test input is deemed valid. As a baseline, we compare JSF with JBSE [10], which implements the HEX approach [9], and the black box enumeration (BBE) approach documented in [36]. We do not compare with the white box enumeration approach [36] as it requires user-provided *conservative* $repOK$ methods, which are missing in these benchmarks. Note that conservative $repOK$ methods are different from $repOK$ methods, and writing those methods is highly nontrivial. We do not compare our approach with SUSHI because SUSHI generates test cases in form of sequence of method calls whereas we generate test cases in form of input data structures. SUSHI and our approach are thus complementary to

each other. To run JBSE, we need invariants written in HEX. We manage to find invariants for *DLL*, *RBT*, *AVL*, *TSAFE*, and *Gantt* from [1], and thus we are able to run JBSE on these subjects. It is not clear to us how to write HEX invariants for other data structures or if HEX is expressive enough to describe them.

The experimental results are shown in Table 1, where the first column show the name of test subjects, and the last three columns show the results of JSF, JBSE, and BBE respectively. Columns #Tests show the results in form of the number of valid test inputs over the number of generated test inputs. Note that because JBSE generate partial initialized test inputs, we add an additional call to $repOK$ method after the method under test to concretize their test inputs. The results show that, as expected, every test input generated by JSF is valid. In comparison, JBSE generates 4.65% valid test inputs and BBE generates 7.83% valid test inputs. From the results, we conclude that JSF is effective in generating valid test inputs.

RQ2: Can JSF generate test inputs that achieve high code coverage? To answer this question, we use JaCoCo [6] to measure the branch coverage of test inputs generated by the tools. The results are shown in the columns titled Cov.(%) and NCov.(%) in Table 1. The columns NCov.(%) show the coverage of all generated test inputs, the columns Cov.(%) show the coverage of test inputs that satisfy $repOK$ methods. As all test inputs generated by JSF satisfy $repOK$ methods, the result of JSF has only one column Cov.(%).

For 73/74 methods (including auxiliary methods), JSF can achieve 100% branch coverage (excluding infeasible branches). The only exception is method TS_R_3 in the *TSAFE* project. It is because this method invokes native methods and handling native methods is beyond the capability of JSF at the moment. In general, JSF achieves 98.98% coverage on average. In comparison, when considering all test inputs, JBSE achieves 95.59% coverage on average and BBE achieves 54.66% coverage on average. Since many of these test inputs are invalid, these coverage are inflated. When considering only test inputs that satisfy $repOK$ method, JBSE is only able to achieve 68.54% coverage on average and BBE achieves 37.85% coverage on average. From these results, we conclude that JSF can generate test inputs with high branch coverage for the methods under test.

RQ3: Is JSF sufficiently efficient? To answer this question, we measure the time spent to generate test inputs for each method. The results are shown in the columns titled T(s) in Table 1. From the results, JBSE and BBE are clearly faster than JSF. That is, JBSE and BBE takes average 8.75 and 9.50 seconds respectively to handle each method, whereas JSF's time ranges from 1 second to half an hour, with an average of 67.29 seconds per method. We also report the number of solver calls used by JSF. In average, JSF needs 70 calls per method. The main reason JSF is slower than JBSE and BBE is JSF has to solve harder path conditions with inductive predicates. However, the efficiency of JBSE and BBE comes with the tradeoff of excessive number of invalid test inputs as discussed above, whereas JSF only generates valid test inputs. From these results, we conclude that JSF is slower than JBSE and BBE, but still sufficiently efficient to provide higher quality results.

7 Related Work

This work is based on *generalized symbolic execution* (GSE) [21], which is the state-of-the-art way for the symbolic execution [22] of heap-based programs. At the heart of GSE is the lazy initialization algorithm which executes programs on inputs with reference variables and fields being *uninitialized*. When a reference variable or field is first accessed, lazy initialization enumerates all possible heap objects that it can: (i) be `null`, (ii) point to a new object with all reference fields being uninitialized, or (iii) point to any previously initialized object of the same type. This explicit enumeration quickly leads to path explosion in any non-trivial program, and existing approaches to addressing this problem can be roughly grouped into two categories:

- *State merging* approaches group together the choices of lazy initialization. For instance, the work in [14,15] represented the choices (ii) and (iii) with a variable, while the work in [19] captured all choices (i), (ii) and (iii) in a symbolic heap using guarded value set. Those work cannot avoid path explosion, but delay it to later stage [14,15], or delegate the burden to an SMT solver [19].
- *State prunning* approaches [36,33,9] truly mitigate the path explosion problem by using a precondition to describe *some properties* of the input. After explicitly enumerating all possible choices, i.e. both valid and invalid paths, these approaches will prune the invalid paths that violate the precondition.

A principle difference between our work and the aforementioned approaches is that we use separation logic, which is expressive enough to *define* abitrary unbounded data structures. Consequently, we are able to construct valid choices from the definition, without explicit enumeration of invalid paths, and without false positives. We discuss some notable state prunning approaches in the following.

repOK As GSE with lazy initialization results in partially initialized structures containing both concrete and symbolic values, the work in [36] propose to use a particular kind of *repOK*, called *conservative* or *hybrid repOK*, that returns `true` when running into parts of the structure that are still symbolic. This, of course, leads to false positives.

JML The BLISS approach in [33] used both hybrid *repOK* and JML [27] together as preconditions. The JML precondition is used to precompute relational bounds on the interpretation of class fields. It is translated into a SAT problem by the TACO tool [17]. As pointed out in [18], this translation introduces duplication, which undermines the benefit of eliminating invalid structures when the size is big. BLISS uses symmetry breaking and refine bounds to mitigate this problem.

HEX Braione *et al.* [9] introduced Heap EXploration Logic (HEX) as a specification language to constrain heap inputs. However, the language is not expressive enough to describe many common data structures, and users have to provide additional methods, called *triggers* [2], to check the properties that cannot be written in HEX. Moreover, HEX does not support numerical constraints, and it represents unbounded data structures using regular operators (using $(\pi)^+$ operator). Therefore, it is unable to capture

the non-regular data structures, e.g., singly-linked lists which have 2^n nodes ($n \geq 0$) and the content of each node is 0. Finally, it is unclear how the HEX solver discharges an unbounded heap formula with regular operators.

Separation logic Our work is also related to research on Smallfoot symbolic execution [7] and its following work, e.g. [13,29]. Those work are not based on lazy initialization, and it is not clear how they can be used for test input generation. Our work is the first to explore the use of separation logic for testing.

8 Conclusion and Future Work

We present a symbolic execution framework for heap-based programs using separation logic. Our novelty is the proposed context-sensitive lazy initialization for test input generation. The experimental results show that our approach significantly reduces the number of invalid test inputs and improves the test coverage. For future work, we plan to integrate our approach on a dynamic symbolic execution engine (e.g., JDart [28]). We might combine JSF with bi-abduction and frame inference tools (i.e., Infer [3,12], S2 [23,25]) to both verify safety and generate test inputs to locate/confirm real bugs in heap-based programs. Finally, we are actively investigating the use of JSF tool for automatic program repair, a preliminary results were reported in [37].

References

1. `https://github.com/pietrobraione/sushi-experiments`.
2. `https://github.com/pietrobraione/jbse`.
3. Facebook Infer. `http://fbinfer.com/`.
4. `https://code.google.com/archive/p/sireum/downloads`.
5. `http://sir.unl.edu/portal/index.php`.
6. `http://www.eclemma.org/jacoco/`.
7. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. APLAS'05, pages 52–68. Springer-Verlag, 2005.
8. P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè. Combining Symbolic Execution and Search-based Testing for Programs with Complex Heap Inputs. ISSTA, pages 90–101, 2017.
9. P. Braione, G. Denaro, and M. Pezzè. Symbolic Execution of Programs with Heap Inputs. ESEC/FSE 2015, pages 602–613. ACM, 2015.
10. P. Braione, G. Denaro, and M. Pezzè. JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs. FSE 2016, pages 1018–1022. ACM, 2016.
11. C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. ICSE '11, pages 1066–1071. ACM, 2011.
12. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*, 58(6):26:1–26:66, Dec. 2011.
13. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. *Sci. Comput. Program.*, 77(9):1006–1036, Aug. 2012.
14. X. Deng, J. Lee, and Robby. Bogor/Kiasan: A K-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. ASE '06, pages 157–166, 2006.

15. X. Deng, Robby, and J. Hatcliff. Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs. SEFM '07, pages 273–282. IEEE Computer Society, 2007.
16. H. B. Enderton. A Mathematical Introduction to Logic (Second Edition). pages 67 – 181. Academic Press, second edition edition, 2001.
17. J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of Invariants for Efficient Bounded Verification. ISSTA '10, pages 25–36. ACM, 2010.
18. J. Geldenhuys, N. Aguirre, M. F. Frias, and W. Visser. Bounded Lazy Initialization. pages 229–243. Springer Berlin Heidelberg, 2013.
19. B. Hillery, E. Mercer, N. Rungta, and S. Person. Exact Heap Summaries for Symbolic Execution. VMCAI 2016, pages 206–225. Springer-Verlag New York, Inc., 2016.
20. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. POPL '01, pages 14–26. ACM, 2001.
21. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. TACAS'03, pages 553–568. Springer-Verlag, 2003.
22. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
23. Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape Analysis via Second-Order Bi-Abduction. CAV'14, pages 52–68. Springer-Verlag New York, Inc., 2014.
24. Q. L. Le, J. Sun, and W.-N. Chin. Satisfiability Modulo Heap-Based Programs. CAV'16, pages 382–404. Springer International Publishing, 2016.
25. Q. L. Le, J. Sun, and S. Qin. Frame Inference for Inductive Entailment Proofs in Separation Logic. In *TACAS'18*, LNCS, pages 41–60. Springer, 2018.
26. Q. L. Le, M. Tatsuta, J. Sun, and W. Chin. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. CAV '17, pages 495–517, 2017.
27. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. pages 175–188. Springer, 1999.
28. K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. JDart: A Dynamic Symbolic Analysis Framework. volume 9636 of *Lecture Notes in Computer Science*, pages 442–459. Springer, 2016.
29. P. Müller, M. Schwerhoff, and A. J. Summers. Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution. In *CAV'16*, volume 9779 of *LNCS*, pages 405–425. Springer, 2016.
30. L. H. Pham, Q. L. Le, Q.-S. Phan, J. Sun, and S. Qin. Testing Heap-based Programs with Java StarFinder. ICSE '18, pages 268–269. ACM, 2018.
31. C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 1–35, 2013.
32. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. pages 55–74, 2002.
33. N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser, and M. F. Frias. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *IEEE Transactions on Software Engineering*, 41(7):639–660, July 2015.
34. E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). SP '10, pages 317–331. IEEE Computer Society, 2010.
35. M. Tatsuta, Q. L. Le, and W.-N. Chin. *Decision Procedure for Separation Logic with Inductive Definitions and Presburger Arithmetic*, pages 423–443. APLAS, 2016.
36. W. Visser, C. S. Păsăreanu, and S. Khurshid. Test Input Generation with Java PathFinder. ISSTA '04, pages 97–107. ACM, 2004.
37. G. Zheng, Q. L. Le, , T. Nguyen, and Q.-S. Phan. Automatic Data Structure Repair using Separation Logic. JPF'18, 2018.