ABSTRACT

We present Java StarFinder (JSF), a tool for automated test case generation and error detection for Java programs having inputs in the form of complex heap-manipulating data structures. The core of JSF is a symbolic execution engine that uses separation logic with existential quantifiers and inductively-defined predicates to precisely represent the (unbounded) symbolic heap. The feasibility of a heap configuration is checked by a satisfiability solver for separation logic. At the end of each feasible path, a concrete model of the symbolic heap (returned by the solver) is used to generate a test case, e.g., a linked list or an AVL tree, that exercises that path.

We show the effectiveness of JSF by applying it on non-trivial heap-manipulating programs and evaluated it against JBSE, a state-of-the-art symbolic execution engine for heap-based programs. Experimental results show that our tool significantly reduces the number of invalid test inputs and improves the test coverage.

KEYWORDS
Symbolic execution, separation logic, test input generation

ACM Reference Format:

1 INTRODUCTION

Symbolic execution [8] is a popular automatic approach to test input generation, error detection and security vulnerability discovery. In essence, this approach takes program inputs as symbols, instead of concrete values, and computes the effects of program statements as expressions over those input symbols. This, however, is not straightforward when the program inputs are dynamically-allocated linked data structures, such as lists and trees, and thus testing and bug finding of programs with heap inputs remain a challenge.

We present Java StarFinder (JSF), a tool that aims to address the aforementioned problem. JSF is a test case generation tool for heap-based programs, i.e., programs with inputs in the form of complex heap-based data structures. JSF takes Java bytecode programs as inputs. It performs symbolic execution of the program, and generates JUnit test cases that achieve high coverage. The generated test cases are valid data structure instances, such as doubly linked lists or red-black trees, that satisfy the invariant predicates, often called repOK, of the corresponding data structures.

Related tools. The state-of-the-art approaches to handling data structures in symbolic execution are based on lazy initialization [7], which is a brute-force algorithm that considers all possible cases of a reference variable. Lazy initialization and its variants, e.g., [4, 5], do not take into account the shape of the input data structures, and thus generate too many invalid test cases. Recently, authors in [3] introduced JBSE with preconditioned lazy initialization. In particular, JBSE uses the so-called HEX logic as a specification language to describe the input data structures, and prunes off the invalid initialization when the specification is violated. However, we found that the HEX logic is not expressive enough to describe data structures. In this work, we present JSF, which addresses this gap with separation logic. A more detailed comparison between JBSE and JSF can be found in [11].

2 TOOL DESCRIPTION

JSF is a preconditioned symbolic execution engine that uses separation logic as the specification language to describe the input data structures. JSF combines three new features. Firstly, to express the execution of heap objects, JSF uses separation logic [6, 12] combined with existential quantifiers and inductive definitions to precisely represent the input data structures and the symbolic states with unbounded heap. Secondly, JSF applies lazy initialization such that uninitialized variables/fields are instantiated only when they are accessed, e.g., assigned to another variable or heap accessed, i.e., de-referenced. Especially, instead of brute-force enumeration of all possible heap objects, our initialization is context-sensitive; JSF only enumerates those values that satisfy the preconditions. Lastly, JSF exploits recent advances in satisfiability checking of separation logic [9, 10], which enable generating a model for each feasible symbolic heap configuration. These models are then used to generate test inputs. An in-depth discussion of technical details of JSF can be found in a companion paper [11]. JSF is a freely available open-source project: https://github.com/star-finder/jpf-star.

The architecture of JSF is depicted in Figure 1. It consists of four components: jpf-core, jpf-star, starlib and S2SAT where our current focuses are jpf-star and starlib.

jpf-core is the core of NASA’s Java PathFinder (JPF) model checking platform [2]. In essence, it is a customized JVM that (concretely) executes Java bytecode. Different from a standard JVM, it allows defining non-deterministic choices during the execution, e.g., in multi-threaded programs. When there are non-deterministic choices, JPF will search all possible executions using depth-first search (by default) or other heuristics.
S2SAT is a satisfiability solver for separation logic [9, 10]. It is used to check if a (symbolic) heap configuration is satisfiable. Unlike other solvers, e.g., concolic execution, or re-used outside JPF, S2SAT is minimal. It is independent of jpf-star, and its dependence on JPF is also minimal. So that it can be re-used in other systems, e.g., concolic execution, or re-used outside JPF.

3 RESULTS

To evaluate our tool, we compare it against JBSE, a state-of-the-art symbolic execution engine for heap-based programs. We use the same benchmarks that were used to evaluate JBSE. Due to space limit, we only present the results for doubly linked list, AVL tree and red-black tree. A more thorough comparison can be found in [11].

For each generated test input, we check its validity by passing it as a parameter to the symbolic heap configuration. For example, loading a reference-type variable \( x \) (by executing the bytecode \( \text{ALOAD} \)), it first checks if \( x \) is pointing to a heap node, or if it is constrained by an inductive definition. In the latter case, it needs to instantiate the variable to all possible valid cases (non-deterministic choices).

For future work, we might investigate machine learning and/or bi-abduction techniques to synthesize separation logic preconditions.

Acknowledgments. The first author is partially supported by the Google Summer of Code 2017 program.

REFERENCES