1 Project title

Java StarFinder: Symbolic Execution with Separation Logic for Testing and Verifying Heap-manipulating Programs.

2 Provide a description of the project

Symbolic PathFinder (SPF) [5] has been very successful in testing and verifying Java bytecode programs with numeric inputs. However, its capability is very limited when coping with programs that make extensive use of heap data structures. The underlying *lazy initialization* algorithm [3] exhaustively enumerates all heap objects that can bind to the structured inputs accessed by the program. This exhaustive enumeration may identify many invalid heap configurations that violate properties of the data structures in the heap, which leads to a huge amount of false alarms.

We aim to tackle this problem by leveraging recent advances in separation logic [6], a well-established assertion language designed for reasoning about heap-manipulating programs. We will build a system, Java StarFinder (JSF), that enables users to describe properties of the data structures in the heap using separation logic. JSF is a symbolic execution engine, built on top of SPF, that generates path conditions (PCs) in the form of separation logic. Similar to numeric PCs, these PCs will be checked by a (separation logic) solver for satisfiability and test input generation. In addition, JSF can also verify program correctness by first collecting PCs satisfying some given preconditions and then verifying whether these PCs satisfy user-provided assertions.

3 What is the development methodology you propose to use for the project?

Our first step is to extend the annotations that enable users to specify preconditions, postconditions, and loop invariants. Then, we will develop a new symbolic execution engine, extending from SPF, based on the theoretical results proposed in Smallfoot [1] and HIP [2]. To solve separation logic path conditions, we will integrate our symbolic execution engine with the $\texttt{S2SAT}_{\texttt{SL}}$ solver [4]. In the following, we illustrate our approach through an example.

```
1  public class Node {
2    public Node next;
3  }
4
5  public class TestNode {
6    public Node foo(Node x)
7    {
8      if (x == null) return null;
9      else if (x.next == null) return x;
10     else return x.next;
11   }
12 }
```

Here the input x is a single-linked list. However, without considering this property SPF (with HeapSymbolicListener and lazy initialization being turned on) exhaustively enumerates all possible references, and generates four PCs, including a redudant one for the case x is a circularly linked list: x.next = x. The number of redundant PCs will rapidly explode when the function foo involves more nodes, or in the case the input is a tree, and lazy initialization includes assumptions of a cyclic graph.

On the other hand, our JSF system will enable the user to provide the following inductive predicate sll to define a singly-linked list.

$$\texttt{pred}\ sll(\texttt{root}) \equiv \texttt{emp} \wedge \texttt{root} = \texttt{null} \\ \vee \exists\, n \cdot \texttt{root} \mapsto Node(n) * sll(n)$$

JSF generates the test cases for all paths where the input x satisfying the pre-condition $sll(x)$ which states that the input x must be an instance of a singly-linked list.

Our illustrative example has three paths corresponding to return statements at line 8, line 9, and line 10 respectively. For the first path, JSF collects the PC corresponding to the conjunction of the precondition and the condition of the then branch as: $sll(n) \wedge x = \texttt{null}$. After that, JSF generates the test input $x = \texttt{null}$ for the path by using $\texttt{S2SAT}_{\texttt{SL}}$ solver. In the second path, there exists a memory access x.next at line 9. To obtain states in which there is no memory error, following the approach presented in [1, 2], JSF unfolds the inductive predicate relating to x to expand its footprints. In this example, JSF unfolds the predicate $sll(x)$ of the state $sll(x) \wedge x \neq \texttt{null}$ right before the memory access to obtain the memory-safe state: $\exists\, n \cdot x \mapsto Node(n) * sll(n) \wedge x \neq \texttt{null}$. By doing so, the PC generated for this paths is: $\exists\, n \cdot x \mapsto Node(n) * sll(n) \wedge x \neq \texttt{null} \wedge n = \texttt{null}$. By using $\texttt{S2SAT}_{\texttt{SL}}$, JSF obtains the test input as: $x \mapsto Node(n) \wedge n = \texttt{null}$. Similarly, JSF generates the following test input for the third program path: $x \mapsto Node(n) * n \mapsto Node(n_1) \wedge n_1 = \texttt{null}$.

4 What are the goals that you hope to accomplish in your project?

At the end of this project, we aim to achieve the following goals:

- Separation logic-style symbolic execution engine for a fraction of Java bytecode.
- A component to generate path conditions in form of separation logic.
- Integration of $\texttt{S2SAT}_{\texttt{SL}}$ to generate test inputs from these conditions.
- (Optional) Supporting postconditions and loop invariants to verify the programs.

5 Tell us about yourself. What are your qualifications, interests, and expectations?

I am Long H. Pham, a second year Ph.D. student at the Singapore University of Technology and Design in Singapore. My research interests include program verification, program analysis, and software engineering. In particular, I have a strong background in symbolic execution and separation logic. I participated in the development of the HIP/SLEEK [2] verification system for separation logic when I was a research assistant at the School of Computing - National University of Singapore. As a programmer, I have eight years of experience with Java as well as other programming languages. I believe that my background and experience will guarantee the success of this project.

6 Briefly describe any discussions you have had with JPF mentors about your project

I have discussed with Quang Loc Le and Quoc-Sang Phan, the mentors of this project. We have clarified the goals of the project, technical challenges and solutions to achieve these goals. We also made an estimation for the project, and agreed on the following milestones:

- 07/14/2017: generating test cases for loop-free intra-procedural programs.
- 08/07/2017: generating test cases for intra-procedural programs with loops.
- 08/29/2017: generating test cases for inter-procedural programs.

7 Have you worked on an open-source project in the past, whether through JPF, Summer of Code, or otherwise?

Yes, I have made several contributions to the following project on Github: `https://github.com/sunjun-group/Ziyuan`.

References

1. J. Berdine, C. Calcagno, and P. W. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.
2. 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. *Science of Computer Programming*, 77(9):1006–1036, 2012.
3. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.
4. Q. L. Le, J. Sun, and W.-N. Chin. Satisfiability modulo heap-based programs. In *International Conference on Computer Aided Verification*, pages 382–404. Springer, 2016.
5. 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*, 20(3):391–425, 2013.
6. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.