Automatic Test Case Generation for, and Verification of, a Single Array Implementation of a Binary Search Tree using KLEE

Omar Mostafa

University of Waterloo, Waterloo ON N2L3G1, Canada
omostafa@uwaterloo.ca

Abstract. Symbolic execution tool KLEE was used to verify an implementation of a binary search tree. The first attempt was at verifying an implementation that allocated memory on the heap via calls to `malloc()` but working with pointers appeared to increase the complexity and difficulty of the verification process. Alternatively, the BST was implemented as a single array of nodes, accepting only distinct positive 32-bit key values. The purpose was to verify the functions used to manipulate the data structure, such as functions for adding and deleting nodes, finding the nodes with maximum and minimum key values, and searching for a specific key. The clang compiler was used to produce LLVM bytecode files from the C files, which KLEE then used to run its symbolic and concrete tests. A KLEE-specific version of the *uClibc* library replaced *libc* library in the runs.

At first, the implementation was tested manually with simple inputs to ensure the basic skeleton of the code is functional, but KLEE was used subsequently for finding all serious bugs. The code was verified incrementally, where only a subset of the required functionality is verified before continuing with verification of other functionality. After numerous runs, more than half a million files, mostly test cases, were generated by KLEE. KLEE was successful in revealing many bugs and achieved a full verification of the implementation, with the restriction that the number of symbolic inputs is not large. This is because the run-time of the tool increases non-linearly with increased symbolic input size.

1 Introduction

This project was started with the intention to explore and utilize the capabilities of KLEE, which is a symbolic execution tool that works on LLVM bytecode files. All descriptions in this paper of the KLEE tool and how to use it are derived from the paper written by the tool's creators titled "*KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*" [1]. KLEE has the ability to find bugs in the code as it runs symbolically, exploring all paths that can be entered, which it can define based on whether the path condition is satisfiable or not. The path condition is initially true, and at each conditional statement execution is forked and two different states are created that follow both paths. For one state, the constraint that satisfies the conditional statement is added to the path condition, while for the other state the

constraint that makes the conditional false is added to the path condition. If the path condition is satisfiable, it means that the execution can, after reaching this part of the code, enter the path in the specified direction. If the path condition is unsatisfiable, KLEE does not need to worry about exploring the path. To determine the satisfiability of the path condition, KLEE uses a constraint solver, such as STP or Z3. Instead of just giving the constraints as they are to the solver, KLEE employs multiple optimizations that have proven to greatly increase its efficiency, as described in the paper [1].

For the user, KLEE provides functions to convert normal program variables, including arrays, into symbolic variables. It also has the ability to simulate interactions with the environment by generating symbolic command line input arguments to a program. In this project, the first step was to write the program to be tested. It was found that writing the program in C would require the least amount of setup work and will make the use of additional tools unnecessary. This is because KLEE was designed originally to work with C programs. After determining the programming language, the program to be written had to be determined. The choice was made to write an implementation of a binary search tree, because of the expected manageable size and complexity of the implementation, and included the data structure itself and the functions that interact with it. The implementation was based on the algorithm descriptions in the textbook *"Introduction to Algorithms"* [2]. The first version of the implementation was based on allocating memory in the heap using calls to `malloc()` function. Nodes are represented as a `struct` that has a key value and three (3) pointers for: the parent node, left child node, and right child node. The functions implemented were for creating a node (by allocating memory for it and returning the pointer), inserting a node into the tree, deleting a node from the tree, finding the node with the maximum key value, finding the node with the minimum key value, finding the successor of a node *("If all keys are distinct, the successor of a node x is the node with the smallest key greater than key[x]."*[2]), searching for a node by its key, and walking over the tree in order. Implementing those functions was easy, as dealing with a pointer-based representation of the tree made translating the algorithms from the textbook [2] to actual code straightforward. However, difficulties appeared when trying to discover and eliminate all bugs in the code, as will be described in the next sections.

In the second attempt, the code was rewritten to represent the BST using a single local array of nodes (allocated on the stack). Each node was represented as a `struct` containing four 32-bit signed integer variables: `key`, `parentIndex`, `leftChildIndex`, and `RightChildIndex`. Since we are dealing with an array, using node indices instead of pointers was more practical. Instead of representing the tree using only the array, a `struct` was created that has an `int32_t` field `root-Index` for keeping track of the index of the current tree root as it changes with insertions into and deletions from the tree, along with the array of nodes itself.

Early attempts relied upon passing symbolic arguments from the command line when invoking KLEE. To use this method of passing arguments, the argument `-posix-runtime` had to be passed to KLEE to enable the symbolic environment. Since C programs interpret these arguments as strings of characters, they had to be converted, using the standard `atoi()` function, into integer values that can be use as node keys. As each argument passed had the length of 4 bytes, the `atoi()` function

would always return values in the range of -999 to 9999. The function also returns 0 for non-numerical input, and since there are more ASCII characters that are not numbers than numbers, the 0 value always dominated KLEE runs. The small range of possible integer values limited the range of inputs that the program attempts to insert, and hence prevents KLEE from rigorously testing the full functionality of the program and how it handles edge cases. To use the full range of possible inputs, the KLEE-provided function `klee_make_symbolic()` was used to turn an array of 32-bit signed integers into separate symbolic input variables.

KLEE revealed multiple bugs, mainly invalid memory accesses, that were fixed one by one. After a number of bugs were fixed, additional code was added utilizing the `assert()` function to verify the correctness of the program, instead of only checking for memory access errors. KLEE discovers if any path would lead to a failed assertion, revealing errors in the implementation or, more rarely, errors in the definition of the correctness criteria as implemented in the code. This makes KLEE a powerful tool that combines both memory error discovery and proving correctness of programs. On a small symbolic input of 6 symbolic variables, that are signed 32-bit integer when run concretely, and a binary search tree data structure that has room for only 5 nodes, KLEE successfully completely verified that the written program is error-free and logically correct. The next sections explain the process in more details showing optimizations performed and lessons learned.

2 Hardware, Environment, and Installation

The machine on which KLEE was attempted to be used is powered by a 2.6 GHz 6-core Intel Core i7-9750H chip and has 16 GB of 2667 MHz DDR4 memory. It runs the macOS Monterey operating system. Attempting to install KLEE on the machine was accompanied with difficulties and many failed attempts. In the official KLEE website [3] the developers also point out that the POSIX runtime environment cannot run on macOS. Going with the developers' recommended installation option, I decided to build KLEE locally on Ubuntu, which has the ability to run the POSIX runtime environment, by means of a virtual machine running on the macOS. I used VirtualBox 6.1 as the virtual machine platform and used version 20.04.4 LTS of Ubuntu. Installing KLEE on Ubuntu was easier, although some CMake files needed to be modified for the build to be successful.

3 Version 1: Heap-allocated BST

This section describes the attempt to implement and verify the binary search tree by allocating its nodes on the heap using calls to `malloc`. We will call this version of the code *version 1*. The project repository contains the .c files for version 1, and the files' names have the format *project_trial*.c*, where '*' is replaced by the number of the trial. Each trial is the same as the previous trial except for some modifications, bug fixes or both. For each KLEE run, there exists a folder *klee-out-trial**, where '*' is number of the trial, the same number used for the .c file.

3.1 KLEE Runs

The general sequence for using KLEE in any of the runs described in this paper does not vary greatly. The first step, after having installed KLEE, is to use the *clang* compiler on the target C file. From the terminal, this line is used:

```
$ clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone
project_trial*.c
```

where the '*' is replaced by the trial number.

The argument `-emit-llvm` directs the clang compiler to produce LLVM bytecode files (with the extension *.bc*). The rest of the arguments are explained in [3]: "...`-c` is used because we only want to compile the code to an object file (not to a native executable), and finally `-g` causes additional debug information to be stored in the object file, which KLEE will use to determine source line number information. `-O0 -Xclang -disable-O0-optnone` is used to compile without any optimisation, but without preventing KLEE from performing its own optimisations, which compiling with `-O0` would."

The second step, after having the *project_trial*.bc* file, is to run klee. From the command line, we use the following commands:

```
$ klee --libc=uclibc -posix-runtime project_trial*.bc
<optional arguemnts>
```

where the '*' is replaced with the trial number, and <optional arguments> is replaced with optional command line arguments, like the ones for creating symbolic command line input arguments. The `--libc=uclibc` argument tells KLEE to use uClibc instead of the normal libc library. An implementation of uClibc that is tailored to work with KLEE is provided by the developers and so was used in the runs. The argument `-posix-runtime` enables us to input symbolic command line arguments to the program.

Now, we can explore the different KLEE runs and what each run teaches us about our implementation and about KLEE:

Project 1 Trial 1. The first trial is provided in *project_trial1.c* and the KLEE output files are in *klee-out-trial1* folder. The main information about the run is found inside the KLEE folder in a text file titled *info*. If we visit the info file, we can see the commands used to invoke KLEE:

```
$ klee --libc=uclibc -posix-runtime project_trial1.bc
-sym-arg 4
```

The argument `-sym-arg 4` tells KLEE to generate one symbolic command line argument to input to the program with the length of 4 characters.

KLEE only takes 2 seconds of execution to find the out of bound pointer error and specifies line 53 of project_trial1.c as the location of the error. KLEE realizes that it

cannot complete any path with this bug present, as all program paths are required to execute it, and quickly terminates. We can examine the line to find the bug:

```
52: treeNode* treeInsertNode(treeNode* ptr_tree_root,
treeNode* ptr_node){
53:     if(ptr_tree_root->ptr_parent != NULL){
54:         fprintf(stderr, "Not tree root!\n");
55:         return ptr_tree_root;
56:     }
…
81: }
```

Fig. 1. Code snippet from project_trial1.c

In Figure 1, the function `treeInsertNode()` takes a pointer to a tree root and a pointer to the new node to be inserted as arguments. It is designed based on the algorithm description in CLRS [2] that preserves the structure of the binary search tree. By examining line 53, we can quickly spot the problem. `ptr_root_tree` is accessed to make sure the parent pointer is `NULL`, ensuring that `ptr_root_tree` is indeed a pointer to a root of a tree since the root node should not have any parent. The problem with this is that `ptr_root_tree` is dereferenced without making sure it is not a `NULL` pointer.

There is one more observation to be made here. KLEE provides a tool called ktest-tool that can display the files containing the test cases generated. These files have the extension .ktest. In this run, KLEE produced one test file test000001.ktest. By using the command `$ ktest-tool ./klee-out-trial1/test000001.ktest`, the file is displayed in the terminal. We can see that KLEE produced one command line argument, which it calls `arg00`, with size 5 instead of 4. It seems that KLEE expects us to use the convention of putting a dash (-) before each command line argument and so adds one additional character to the argument.

Project 1 Trial 2. In the second trial, the line 53 is changed to:

```
53: if(ptr_tree_root != NULL && ptr_tree_root->ptr_parent
!= NULL){…
```

And KLEE is run with the following commands:

```
$ klee --libc=uclibc -posix-runtime project_trial2.bc
-sym-args 0 5 3
```

Here, `-sym-args 0 5 3` instructs KLEE to input at least 0 and at most 5 symbolic command line arguments, each with length 3. In this run, 2 errors are printed to the terminal. KLEE produces files that give info about the error, specifying the line on which the error occurred in the C and assembly files and the stack trace. The test file with the same number gives the exact concrete input that caused the failure. Both errors

are out-of-bound pointer errors on lines 137 and 61 of the C file. For line 61, it is easy to spot the bug.

```
52: treeNode* treeInsertNode(treeNode* ptr_tree_root,
treeNode* ptr_node){
53:     if(ptr_tree_root != NULL && ptr_tree_root-
>ptr_parent != NULL){
54:         fprintf(stderr, "Not tree root!\n");
55:         return ptr_tree_root;
56:     }
57:     treeNode* ptr_current_parent = NULL;
58:     treeNode* ptr_current_node = ptr_tree_root;
59:     while(ptr_current_node != NULL){
60:         ptr_current_parent = ptr_current_node;
61:         if(ptr_node->key < ptr_current_node->key){
…
```

Fig. 2. Code snippet of `treeInsertNode()` function from project_trial2.c

In Figure 2, `ptr_node` is accessed before checking that it is not a `NULL` pointer. But the error on line 137 is not immediately apparent. In Figure 3, line 137, `ptr_tree_root` is dereferenced to access the value of its left child pointer. However, line 136 does check whether `ptr_tree_root` is a `NULL` pointer or not! We can examine the rest of the code to make sure we have not made other mistakes that may cause this error.

```
135: void inorderTreeWalk(treeNode* ptr_tree_root){
136:     if(ptr_tree_root != NULL){
137:     inorderTreeWalk(ptr_tree_root->ptr_left_child);
138:     printf("%d ", ptr_tree_root->key);
139:     inorderTreeWalk(ptr_tree_root->ptr_right_child);
140:     }
141:     return;
142: }
```

Fig. 3. Code snippet of `inorderTreeWalk()` function from project_trial2.c

Examining the main function in Figure 4, we see it starts by creating a `treeRoot` pointer and initializing it to `NULL`, creating another pointer named `tempNode` to be reused in the function, and an unsigned `tempKey` integer variable. The code checks if there are any command line arguments (other than the name of the executable) and then enters the `for` loop. The loop takes each argument, sends it to `atoi()` function and stores the integer value in `tempKey`. A function that allocates the required memory, `createTreeNode()`, returns the pointer to the newly allocated node and stores it in `tempNode`. `treeInsertNode()` is then called with the current root, `treeRoot`, and the new node and returns the tree root address which we store in the `treeRoot` pointer variable. This is important since the root can change with insertions into the

tree. After all insertions, we call the function `inorderTreeWalk()` with the root of the tree as an argument. Since the error happens in this function when dereferencing the `treeRoot` pointer, we need to examine the functions `createTreeNode()` and `treeInsertNode()`.

```
144: int main(int argc, char** argv){
145:     treeNode* treeRoot = NULL;
146:     treeNode* tempNode;
147:     uint32_t tempKey;
148:
149:     if(argc > 1){
150:     for(int i = 1 ; i < argc; i++){
151:     tempKey = (uint32_t) atoi(argv[i]);
152:     tempNode = createTreeNode(tempKey);
153:     treeRoot = treeInsertNode(treeRoot, tempNode);
154:     }
155: }
156:
157:     inorderTreeWalk(treeRoot);
…
```

Fig. 4. Code snippet showing the `main()` function in project_trial2.c

```
14: treeNode* createTreeNode(uint32_t key){
15:     treeNode* newNode = (treeNode*) mal-
loc(sizeof(treeNode));
16:     if (newNode == NULL){
17:     fprintf(stderr, "Memory allocation failed!\n");
18:     exit(-1);
19:     }
20:     newNode->key = key;
21:     return newNode;
22: }
```

Fig. 5. Code snippet showing `createTreeNode()` function in project_trial2.c

In Figure 5, we see that we allocate the memory using malloc and store the returned address in a pointer called `newNode`. We check to see if malloc returned `NULL`, in which case we print an error message and terminate the program. Otherwise, we store the appropriate key in the `key` struct member of the new node and return the address of the new node. If we pay attention, the three pointers to the parent, left child, and right child were not initialized. We can see the pointers in Figure 6.

Since `ptr_left_child` was not initialized, it will probably have a garbage value. When we look back at Figure 3, we can see that `inorderTreeWalk()` recursively goes down the left branch of the tree to get to the smallest key value node. When the

line `inorderTreeWalk(ptr_tree_root->ptr_left_child);` is invoked, `ptr_tree_root` will have a garbage value and so the check for a `NULL` value is not sufficient. Dereferencing this garbage value can definitely cause an out-of-bound pointer error.

```
7:  typedef struct treeNode{
8:      uint32_t key;
9:      struct treeNode* ptr_parent;
10:     struct treeNode* ptr_left_child;
11:     struct treeNode* ptr_right_child;
12: }treeNode;
```

Fig. 6. Code snippet showing `treeNode` struct

To see the full picture, assume a new node is created and inserted into an empty tree. The tree now has one node with garbage values for all its internal pointers, and the `treeRoot` pointer is updated to point to this node. Next, `inorderTreeWalk()` is called with `treeRoot` as the argument. and the check for a `NULL` pointer passes. `inorderTreeWalk()` is called again recursively with the garbage value of the left child pointer. The `NULL` pointer check passes again, and on the attempt to call `inorderTreeWalk()` again, the garbage value pointer is dereferenced, and the error occurs.

```
148: int main(){
149:     treeNode* treeRoot = NULL;
150:     treeNode* tempNode;
151:     uint32_t arrKeys[10];
152:     klee_make_symbolic(arrKeys, 10*4, "arrKeys");
153:
154:     for(int i = 0 ; i < 10; i++){
155:     tempNode = createTreeNode(arrKeys[i]);
156:     treeRoot = treeInsertNode(treeRoot, tempNode);
157:     }
158:
159:     inorderTreeWalk(treeRoot);
160:
161:     return 0;
162: }
```

Fig. 7. Code snippet showing the `main` function in project_trial4.c

Project 1 Trials 3 and 4. In the next trials, `createTreeNode()` is modified to initialize all pointers in the `treeNode` struct to `NULL`. Trial 3 is called with the same commands as the previous trial. For trial 4, instead of using symbolic command line arguments, we create an array and make it symbolic using the KLEE-provided function

`klee_make_symbolic()`, as presented in Figure 7. As mentioned in the Introduction section of the paper, this allows us to use the full range of values of a 4-byte variable. The command to invoke KLEE now becomes:

```
$ klee --libc=uclibc -posix-runtime project_trial*.bc
```

where the * is replaced with the trial number.

In trial 3, the program starts printing the values of keys when it reaches the `printf()` function in `inorderTreeWalk()`. As expected, the values are mostly zeros, with some key values consisting of the number 9, such as 9, 99, -99, and so on. Trial 4 did not print any values of keys to the terminal. In both trials, execution took too long and had to be halted using Ctrl+C. Looking at the info file, KLEE does not go very deep into the paths it explores and tries instead to explore a lot of paths with a shallow depth.

Project 1 Trials 5 and 6. To make the KLEE run faster, I decided to reduce the size of the input. Only a 3-node array was used instead of the 10-node array. Thankfully, the KLEE run in trial 5 was able to conclude in less than 1 second with all paths fully completed. At first glance, this might seem like proof that all our code is bug free, but in fact most of the code was not actually covered. This is because most functions were not invoked from the `main` function and even the `inorderTreeWalk()` function was commented out to reduce the run time as much as possible and for us to get any tangible results quickly.

For trial 6, the function to delete nodes from the tree is called from `main`. KLEE is run and immediately finds an out-of-bound pointer error on line 90 in function `treeDeleteNode()`. It completes 4 paths, and 2 paths are only partially completed because they encounter the erroneous line of code. The bug is the common bug of dereferencing a pointer before checking if it is `NULL` or not, and it is fixed in trial 7.

Project 1 Trial 7. Trial 7 is the last trial performed on version 1. An out-of-bound pointer error is found on line 116, but when we examine the line, we see that a proper check for a `NULL` pointer appears before it. After revising the code, no clear reason was found that explains why KLEE considers this line a possible cause of failure. There is a chance that this is a false positive due to a bug in KLEE itself, as the developers mentioned in their paper: “However, non-determinism in checked code and bugs in KLEE or its models have produced false positives in practice.” [1]. To simplify our program and avoid dealing with pointers, another version of the binary search tree program, which we will call *version 2*, was implemented. In the next section, we will present this implementation and the outcomes of running KLEE on it.

4 Version 2: Single-array-allocated BST

This section describes the second attempt to implement and verify a binary search tree, this time by allocating its nodes on the stack in a single local array. We will call this

version of the code *version 2*. The project repository contains the .c files for version 2, and the files' names have the format *project2_trial*.c*, where '*' is replaced by the number of the trial. Each trial is the same as the previous trial except for some modifications, bug fixes or both. For each KLEE run, there exists a folder *klee-out2-trial**, where '*' is number of the trial, the same number used for the .c file.

The new version was done by walking over each line in the previous version and changing it based on the new design decision. It rewrites the definition of `treeNode_t struct` to contain four (4) `int32_t` member variables: `key`, `parentIndex`, `leftChildIndex`, and `RightChildIndex`. The index variables will contain indices into the array instead of pointers. This version also has a `tree struct` that consists of an `int32_t rootIndex` member variable to carry the index of the current root, and an array `arrNodes` of `treeNode_t` objects.

The `tree struct` is instantiated in the main function, and `memset()` function is used to ensure the `rootIndex` is set to -1 and that all nodes in the array have their key values and index variables set to -1. This is straightforward as the binary representation of -1 is all 1s, so `memset()` can be used directly on the array. The logic behind using the -1 value is that index values into the array cannot be negative, so if we see an index variable with the value -1 it means it does not point to anything and can be thought of as the `NULL` pointer. To have a consistent approach, key values that are not positive means they have not been set to their values yet, hence the tree will only accept positive key values and the number 0.

4.1 KLEE runs

The commands to initiate KLEE runs for the second version of the project are almost the same as the first version. We use:

```
$ clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone
project2_trial*.c
```

followed by:

```
$ klee --libc=uclibc -posix-runtime project2_trial*.bc
```

Project 2 Trials 1 and 2. Running KLEE on version 2 for the first time appeared to take too long, which makes sense considering that the size of the symbolic array was increased to 10. Additionally, all functions of the implementation were being used in `main` to ensure they are bug free. What was noticed is that the terminal received a stream of print statements containing the message "Cannot insert node with negative key!". This message is the error message received when attempting to insert a node into the tree which has a negative key. For trial 2, the KLEE-provided function `klee_assume()` was used to limit the symbolic input range of values and make the run quicker. For example, the call `klee_assume(symbolic_array[i] > 0)` forces KLEE to use positive values for the specified symbolic array element in its run. A great range reduction was performed on all elements of the symbolic array, but the

size of 10 symbolic elements was still making the run taking a long time. So, I changed the size of the array to 5 elements only. On this run, KLEE completed all 261 paths of the program and raised no errors at all. However, there was a hidden problem preventing many paths of being covered, which we will discuss next.

Project 2 Trials 3 and 4. The next trial incorporated calls to the function `assert()`, which allows us to prove the correctness of the implementation and that it does satisfy the program requirements. This third trial finished surprisingly quickly and showed that KLEE completed only a single path! But after inspection, the problem appeared to be that on each case where we print an error message, such as trying to insert a negative key value node into the tree or searching an empty tree for a key value, we immediately exit the program with a call to `exit(-1)` instead of just returning -1. In trial 4, this was fixed by replacing calls to `exit(-1)` with `return -1`. After this fix, trial 4 runs and finds an out-of-bound memory access bug on line 267 and then exits after completing 24 paths.

Project 2 Trials 5 to 9. Throughout the trials from trial 5 to trial 9, many complex bugs were identified and fixed. An important optimization was also performed, which is commenting out all print statements. These statements were useful in earlier runs to expose the behavior of KLEE and what branches of execution it chooses. But for later runs, the focus is on efficiency and making the KLEE run as fast as possible. On some trials, some assertions also failed. Before revising the functions to make sure they behave as expected, I checked to see if the specification was actually correct. It turns out, the correctness criteria as implemented in assertion statements was incorrect in some cases. This was corrected and I proceeded with trial 10.

Project 2 Trial 10. Trial 10 incorporates all assertion statements needed to prove the correctness of the algorithm itself. Using assert statements is a truly powerful strategy that makes KLEE a theorem prover in addition to its bug-revealing capabilities. The verified functions are `treeSearch()`, which finds the node with the provided key value in the tree and returns its index in the array, or returns -1 if the key value is not found, `treeMaximum()`, which finds the node with the maximum key value in the tree and returns its index, or returns -1 if the tree is empty, `treeMinimum()`, which finds the element with the minimum key value in the tree and returns its index, or returns -1 if the tree is empty, `treeSuccessor()`, which finds the node with the smallest key value that is greater than the provided key value and returns its index, or returns -1 if there is no successor, `treeInsertNode()`, which inserts a node in the tree while maintaining the properties of the binary search tree, `treeDeleteNode()`, which removes a node from the tree while maintaining the properties of the binary search tree, and finally `inorderTreeWalk()`, which in the usual case prints the key values of the nodes in the tree in ascending order, but for the purpose of verifying its correctness it takes an array pointer and fills the array with the values of the keys in ascending order.

Just to explore one example of how the verification of the algorithm works, we can look at the code snippet in Figure 8.

```
240:     int32_t indexMax = treeMaximum(main_tree);
241:     int32_t tempIndex;
242:
243:     // Assert that treeMaximum function returns the
index of the node with the max key in the tree
244:     if(indexMax != -1){ // If tree is not empty
245:         for(int i = 0; i < SIZE; i++){
246:         tempIndex = treeSearch(main_tree,
symbolic_array[i]);
247:         if(tempIndex != -1){ // If key was actually
inserted in tree
248:          assert(main_tree.arrNodes[indexMax].key >=
main_tree.arrNodes[tempIndex].key);
249:             }
250:         }
251:     }
```

Fig. 8. Code snippet of the verification of function `treeMaximum()` in project2_trial10.c

First, the function `treeMaximum()` is called with the binary search tree as the argument, returning the index of the maximum key value node in the tree, which we store in a variable `indexMax`. If the returned value is -1, there are no nodes in the tree so there is no point in verification. Otherwise, in a for loop, we search for each key value, that was generated by KLEE in the symbolic array, in the tree to know the indices of the nodes that have those values since we only keep a record of the keys and not the node indices. The index returned is stored in the variable `tempIndex` and checked to ensure it is not a -1. If the check passes, we simply assert that the key of the node accessed by `indexMax` is always greater than or equal to any other key accessed by any other index. This verification process is clearly not the most efficient, but it serves the required purpose in a simple way.

This final successful run of KLEE verified that our program is bug free and verified the correctness of the algorithms used from CLRS [2]. However, the input size was chosen to be only 6 symbolic 32-bit signed integers as represented by the `symbolic_array`. The tree itself had room for 5 nodes only so we can test the case when trying to insert a node into a tree that is full. The run took a total time of 1 hour, 31 minutes, and 16 seconds. It completed 9366 paths and did not leave any path partially completed. It executed more than 260,000 instructions and sent more than 10,000 queries to the constraint solver. An important observation is that `assert` statements that occur in loops to verify the correctness of the algorithms have a huge impact on performance. For example, the 9th trial did not verify `inorderTreeWalk()` nor `treeDeleteNode()`, but verified the rest of the functions. It took only a little over 10 minutes to complete, in contrast with around 1.5 hours for the last run. This also shows the impact of the path number explosion and its effects on run time.

5 Conclusion

KLEE is a powerful symbolic execution tool than can offer both bug detection and program verification if used correctly. It was used to verify a heap-allocated version of a binary search tree implementation based on algorithm descriptions in the textbook *Introduction to Algorithms* [2], but some difficulties were faced and it was hard to identify the cause of the error KLEE generated. To try and simplify the verification process, the binary search tree was implemented as a single array of nodes. This new design made the process easier and allowed us to successfully verify the implementation and the algorithms used. Code for verification increased the run time of the tool dramatically, and the maximum input size tested was 6 symbolic 32-bit signed input variables into a tree that has capacity for 5 nodes. The small input size was chosen due to run-time limitations. Further optimizations can be performed to lower the run-time of the tool and allow for larger input sizes to be tested.

References

1. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, USA, 209–224. (2008).
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms. 3rd edn. The MIT Press (2009).
3. KLEE's Official Website, https://klee.github.io, last accessed 2022, 08, 08.