Lab2: Symbolic Execution

1 Environment configuration and Z3

Z3 is an SMT solver developed by Microsoft and is widely used. To use Z3, you first need to start the command line in the lab2 directory and enter ./install_z3.sh to install Z3 and the necessary programming environment. This process may take a long time, you can first read the Z3 tutorial provided by the University of Utah Or the C++ example program in the examples/c++/example.cpp file under the smt/z3 folder. We provide a Z3 sample

program (smt.cpp) in the smt directory. After successfully installing Z3, in the smt directory
Open the command line and enter make to compile and run the sample program to see the effect.

2 goals

This lab needs to implement a symbolic execution tool MiniSEE, where SEE is the abbreviation of Symbolic Execution Engine. The principles of this lab are exactly the same as those taught in the courseware. You need to implement 3 functions. Implementing the first two functions can pass 8 test cases, and implementing the third function can pass the remaining two test cases containing if statements.

These three functions are all in lab2.cpp. If you want to understand the process of symbolic execution, you only need to understand cfg.h, minisee.h,
For the three files minisee.cpp, start with the minisee function in minisee.cpp. The relevant data structure and code flow will be introduced in detail later. In lab2, after compilation,

the executable file minisee will be generated, and then enter the command ./minisee test/xxx.c, then minisee will read the contents of xxx.c and store the functions inside in a graph structure, in cfg.h The cfg_node class is the node in the graph. Afterwards, this picture will be passed to the function minisee in minisee.cpp, and symbolic execution will be officially started to determine whether the assert in the function will report an error. If assert never reports an error, then minisee will output verified, otherwise it will output a counterexample. For example, the test case is int func(int a, int b), then minisee will output the values of a and b in sequence.

3 Run and submit

After writing the code, open the command line in the lab2 directory and enter make, and the executable file minisee will be generated. Entering make run will run 10 test cases in the test directory in batches, and the results will be output to *test/ output.txt* . There are 11 lines of output for the 10 test cases, because there are two paths to the assert statement in test10. If you want to test a single test case, you can enter ./minisee test/xxx.c, where xxx is the name of the test case, for example ./minisee test/test1.c, and the results will be output directly to the command line.

After completing this lab, please run make handin in the lab2 directory to generate lab2.zip, and then upload it to canvas. Note:

Please do not call any input and output operations in lab2.cpp, such as scanf, printf and various file operations. Otherwise, misjudgment may occur.

4 tasks

You need to implement 3 functions in lab2.cpp.

4.1 Function mystoi

```
int mystoi (strings);
```

mystoi needs to convert the numbers in the input string s into int type and return it. s may be a decimal unsigned integer, Decimal signed integers or hexadecimal integers are within the 32-bit integer representation range. You can do this by calling c++'s Some existing library functions implement this function, and you can also perform the conversion manually. This function does not need to consider illegal s.

4.2 Function copy_exp_tree

4.2.1 Function

```
exp_node* copy_exp_tree (exp_node* root);
```

The parameter root of the function points to an expression tree. copy_exp_tree needs to copy this tree to generate a brand new one. tree, then returns the new tree. exp_node is defined in cfg.h and represents the node of the expression tree. If root is NULL, return NULL; otherwise, a new tree will be generated. The expression represented by this tree is exactly the same as that represented by root. Consistently, all nodes in this tree are new, and no nodes in the root can be reused.

4.2.2 Expression tree

```
// exp_var represents the variable in the expression, such as a
//exp_num represents the int constant in the expression, such as 11
//exp_op represents the operators in the expression, such as +, ÿ, !=, >, etc., including unary operators~
    wait
enum EXP_NODE_TYPE { exp_var , exp_num , exp_op } ;

//Nodes in the expression tree
// Can represent Boolean type expressions such as a>3
// It can also represent int type expressions such as a+b or 3
class exp_node {

public:
        //The data type stored in the current node, including variables, constants and operators
        EXP_NODE_TYPE type;
        //Data stored in the current node
        // If it is a variable, it is a variable name like "a"
        // If it is an operator, this is the operator "+"
        // If it is a constant, it is a constant "11" like this
        string val; // The left and
        right nodes of the expression tree exp_nodeÿ
        child [ 2 ] ;

        twenty two } ;
```

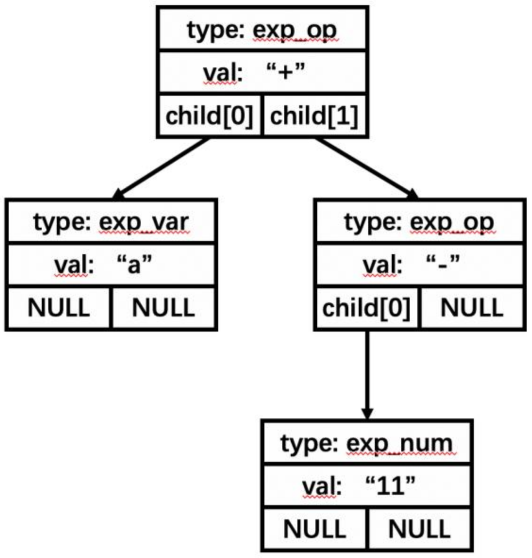

Figure 1: Expression tree corresponding to a+(-11)

The expression tree is composed of nodes. The class definition of the node is as above. For the meaning of each field of this class, please see the comments. Next

Look at two examples. The expression tree corresponding to the expression a+(-11) is shown in Figure 1. x==1 corresponds to the expression tree in Figure 2.

```
1 // Output the expression 2 stored in the expression tree root void print_exp
(exp_nodeÿ root);
```

You can call print_exp to print an expression tree. This function can be used for debugging.

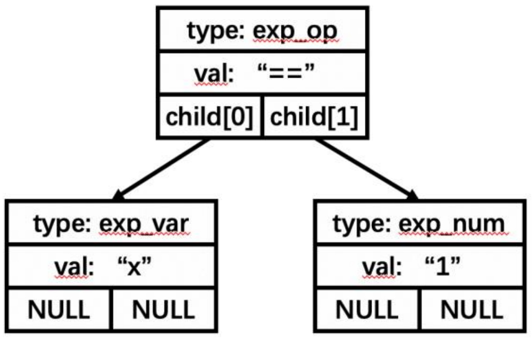

Figure 2: Expression tree corresponding to x==1

4.3 Function analyze_if

WARNING

Please complete the previous two functions before reading this section. Please make sure you have mastered the symbolic execution courseware.

After completing the first two functions, you can pass the first 8 test cases. They do not contain if statements and do not require special

Don't understand the flow of symbolic execution.

But if you want to pass the remaining 2 test cases containing if, you need to complete the function analyze_if in lab2.cpp.

5 Data structure and code flow

Let's first explain the data structure and execution process of MiniSEE.

5.1 Program control flow graph CFG

After MiniSEE reads the program from the source file, it is stored in a program flow graph (CFG). program

The flow graph is a directed acyclic graph. Each node in the graph corresponds to a statement in the program. Node A has an edge pointing to the node B table.

indicates that statement B will be executed after statement A is executed. If a node corresponds to an if statement, then this node will have two outputs.

The edges point to the next statement when the if condition is true and when the condition is false respectively.

Each node is represented by the data structure cfg_node in cfg.h.

```
// cfg_assign represents an assignment statement
// cfg _ if represents if statement
// cfg_assert represents the assert statement
// cfg_return represents the return statement
enum CFG_NODE_TYPE { cfg_assign , cfg_if , cfg_assert , cfg_return } ;

//Nodes in the control flow graph, each node stores a statement
class cfg_node {

public:
        //Statement type, including assignment, assert  ,  if  ,  return
        CFG_NODE_TYPE type;

        //The variable name to the left of the equal sign in the assignment statement
        // If a = b+c dst is "a"
        // If it is another type of statement, it is empty
        string dst;

        // If it is an assignment statement, exp_tree stores the expression on the right side of the equal sign
        // If it is a return statement, exp_tree stores the expression corresponding to the return value
        // If it is an if statement, exp_tree stores the Boolean type expression corresponding to the if condition
        // If it is an assert statement, exp_tree stores the expression corresponding to the assert parameter.
        exp_nodeÿ exp_tree;
        //The line number of this statement in the source file is not used in the lab.
        int lineno;

        //The statements to be executed next
        // If the current node is assert , return , assignment statement, then next [0] points to the next
            statement, next [ 1 ] is NULL
        // If the current node is an if statement, then next [0] points to the instruction to be executed if the if condition is true.
            Let next [ 1 ] point to the next instruction to be executed if the if condition is false
        // If next [ 0 ] and next [ 1 ] are both NULL, it means there is no next statement.
        cfg_node* next [2] ;
};
```

See the notes for the meaning of each member. Let's look at an example, such as the following program.

```
int func ( int a , int x) {
        x = a + (ÿ11)
        assert(x==1);
}
```

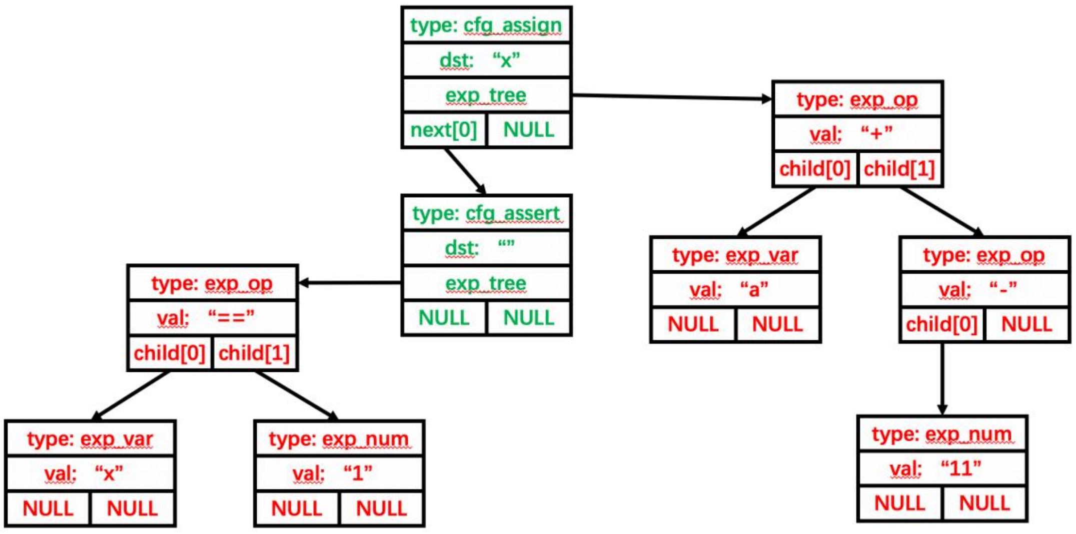

image 3:

The corresponding program flow diagram is shown in Figure 3. The green in the diagram is the cfg node, and the red is the storage in each cfg node.

exp node. As you can see, there is no if here, so each node has only one outgoing edge. Because it is a program, there is input port, so the program flow graph actually has a root node.

Below is a sample program with an if statement.

```
int func ( int a if ( a > 0) ,    int b) {

        a = ÿb ;
    else
        b = 2;
    assert ( a + b < 1) ;
}
```

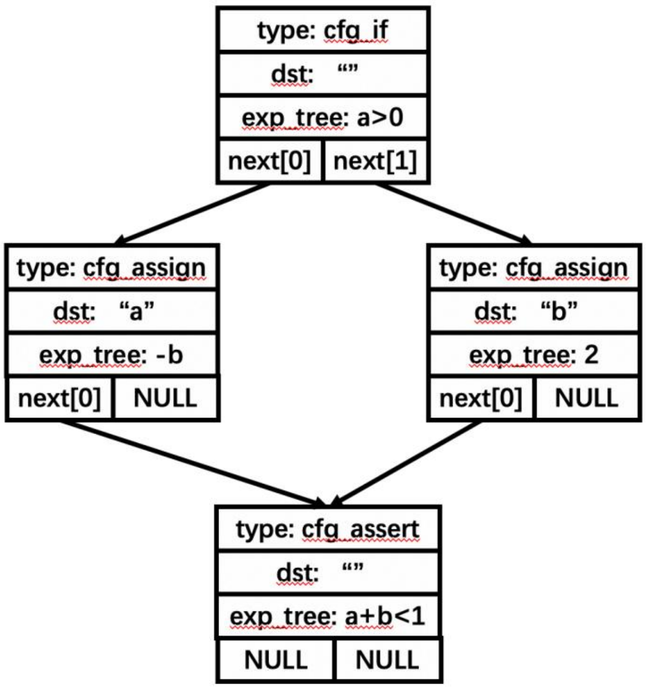

Figure 4:

Its cfg is shown in Figure 4. This figure omits exp_node. As you can see, the cfg_node corresponding to the if statement has

There are two outgoing edges, the directed edge on the left points to the statement when the condition is true, and the directed edge on the right points to the statement when the condition is false.

```
void print_cfg ( cfg_nodeÿ root );
```

You can call print_cfg to output a program flow diagram. The output format is to convert the diagram into source code and output it.

This function can be used for debugging.

5.2 see_state

```
// see_state is the state of the current symbolic execution node
//Including symbolic store, path constraint and next code
class see_state {
public:
        // symbolic store, is an array
        // Each variable has an expression tree exp_nodeÿ in it
        vector<exp_nodeÿ> sym_store;
        // path constraint
        // If it is NULL, the path constraint is true
        exp_nodeÿ path_const;
        // control flow graph (CFG)
        // Represents all the code to be executed next
        // cfgÿ>root points to the next code in the symbolic execution node, which is the code to be executed.
             code
        cfg_node* cfg;
};
```

Recall the content in the symbolic execution courseware. Symbolic execution will generate a tree. Classes are used in MiniSEE. see_state represents each node in the tree, this class is in the minisee.h file. As mentioned in the courseware, there are 3 categories in the class member variables, corresponding to the symbolic store, path constraint and next code on the courseware, where next code It is a cfg_node pointer, pointing to a program flow graph. The first node in this graph stores what is to be executed. code.

In addition, we also need to know what variables are in the program.

```
// Store the names of all local variables
extern vector<string> vartb;
// Store the names of all parameters
extern vector<string> inputtb;
```

These vectors are defined in cfg.h and store the names of parameters and local variables. Each test case has only one A function, all data is of type int, and there are no global variables. The parameters and return value of the function are all int types.

After getting a see_state, we want to know the current value of each variable, so we have the following map, In minisee.cpp.

```
// symbolic store is an array, each variable corresponds to one of its elements
//name_to_index stores the mapping from the variable name to the subscript of the variable in the symbolic store.
   shoot
// Lab2 does not involve the operation of name_to_index
static map<string,                    int> name_to_index;
```

This map is a mapping from variable name (string) to array sym_store index (int). For example, we take Go to the current symbolic execution node. The symbolic storage in the node is the array sym_store, then parameter a is currently The value is the expression tree represented by sym_store[name_to_index["a"]]. Because all variables in symbolic execution Values are represented by symbolic expressions, so sym_store is an array of type exp_node*.

5.3 Symbolic execution process

Please coordinate this part with the lab2 video explanation on canvas and the code comments in lab2.

Symbolic execution generates a tree, and we visit each node in a method similar to depth-first traversal. Excellent depth

First traversal requires the following stack to assist.

```
// Stack
for depth-first traversal of the symbolic execution tree
stack<see_state> state_queue;
```

Symbolic execution starts from the minisee function in minisee.cpp, and the input is a program represented by cfg.

```
void minisee ( cfg_nodeÿ cfg );
```

In the minisee function, the init_state function is first called to construct the root node of the symbolic execution tree, and this root node is pushed into the stack state_queue. After that, the minisee function enters a loop, taking away the symbolic execution node on the top of the stack each time, and then calling the state_handler function to process it.

```
// state is the current symbolic execution node
//
includes symbolic store, path constraint and next code
void state_handler ( see_state ÿ state );
```

The state_handler function will take out the next code recorded in this node, that is, state->cfg. If it is found to be NULL, it means that there is no next code, and it will return directly. If it is found that the type of this code (state->cfg->type) is return, it means that the program ends and returns directly.

If it is found to be an assignment statement, call the analyze_assign function for processing.

```
// Process assignment
statement
// The first instruction of cfg in the current state is an assignment statement
void analyze_assign ( see_state ÿ state );
```

The function framework code has been given and does not need to be completed by everyone, so here we only introduce the general process. The current state is the symbolic execution node corresponding to the assignment statement. This function will generate the child node corresponding to this node and push it onto the stack. The next code of state is state->cfg. According to the previous explanation of the see_state class, the next code of the child node should be state->cfg->child[0]. Because it is not an if statement currently, the path constraint has not changed, so the path_const of the child node is the same as the path_const in the state. The assignment statement will modify the symbolic store, so call update_sym_storage to get the symbolic store of the child node. In this way, all three data members of the child node are available, and then the child node is pushed onto the stack and waits for the next round of processing in the large loop.

If it is found to be an assert statement, it means it is time to verify. Call the verify function and the framework code has been implemented

Now, the generated logic formula is handed over to SMT solver Z3 for verification. If it is

an if statement, please watch the video explanation.

5.4 analyze_if function

```
void analyze_if ( see_state ÿ state ) ;
```

state is the currently reached node in the symbolic execution tree. In symbolic execution, this node will have two child nodes, corresponding to the situations where the if condition is true and not true. The function function is to construct these two child nodes and push them onto the stack state_queue middle. To implement this function, you need to call the update_path_const function in lab2.cpp to construct a new path constraint.