

实验 2: 程序验证与修复

1 背景

符号执行技术通过将程序转换为符号表达式，然后对符号表达式进行求解来发现程序中的错误。我们可以通过符号执行来验证一段程序是否一定能实现预先定义的逻辑 (Assertions)。

但是，符号执行只能发现程序中的错误，而不能修复程序。与此同时，LLM 在代码理解与合成方面展现出强大能力，可以帮助我们修复程序。

因此，本次实验我们将结合符号执行和 LLM 来验证并修复程序。首先，同学们需要实现一个简单的符号执行引擎 MiniSEE 来可靠地判断程序中的 `assert` 是否一定成立，在验证失败时，同学们需要自己构造 `prompt` 来让 LLM 修复程序。你可以向 LLM 提供符号执行得到的反例或其它任何有助于修复程序的信息。此外，为了确保 LLM 修复成功，你可能需要多次调用符号执行工具与 LLM 直到最终成功。

2 环境配置

2.1 Z3

Z3 是微软开发的 SMT 求解器，被广泛使用。要使用 Z3 首先需要在 `lab2` 目录下启动命令行并输入 `./install_z3.sh` 安装 Z3 和必要的编程环境。此过程可能需要较长时间，你可以先阅读 Utah 大学提供的 [Z3 教程](#) 或者 `smt/z3` 文件夹下面 `examples/c++/example.cpp` 文件里的 C++ 示例程序。（如果遇到无法连接 github 的情况，请自行更换脚本中的 repo 地址）

我们在 `smt` 目录下提供了一个 Z3 的示例程序 (`smt.cpp`)。成功安装 Z3 后，在 `smt` 目录下打开命令行输入 `make` 来编译并运行示例程序查看效果。

2.2 LLM

类似于 lab1, 我们使用 LLM Serve 平台的 API 来调用 LLM。你需要在 `llm.h` 文件中填写你的平台 url 和 API 密钥：

```
1 kApiBase = "your_api_base";  
2 kApiKey = "your_api_key";
```

3 目标

3.1 MiniSEE

本次 lab 首先需要实现一个符号执行工具 MiniSEE, 其中 SEE 是 Symbolic Execution Engine 的缩写。这部分原理和课件上讲的完全相同。你需要实现 3 个函数 (具体见下面的任务描述)

这 3 个函数都在 lab2.cpp 中, 你如果了解符号执行的流程, 只需要理解 cfg.h, minisee.h, minisee.cpp 这 3 个文件, 从 minisee.cpp 里面的 minisee 函数开始看。后面会详细介绍相关的数据结构和代码流程。

在 lab2 中, 编译后会产生可执行文件 minirepairer, 然后输入命令 `./minirepairer test/xxx.c`, 那么 minirepairer 会读取 xxx.c 的内容, 把里面的函数用图结构存储起来, cfg.h 里面的 `cfg_node` 类就是图中的节点。之后这个图会传给 minisee.cpp 中的函数 minisee, 正式开始符号执行, 判断函数中的 `assert` 是否会报错。如果 `assert` 永远不报错, 那么 minisee 输出 `verified`, 否则输出反例, 例如测试用例是 `int func(int a, int b)`, 那么 minisee 会依次返回 a 和 b 的值。

3.2 LLM 修复

本次 lab 的第二部分是在符号执行验证失败后调用 LLM 修复程序, 你需要在 lab2.cpp 中实现函数 `llm_repair`, 该函数需要调用 LLM 修复程序, 并返回修复后的代码。

更上层的 `workflow` 将反复调用符号执行验证代码, 并在验证失败时调用 `llm_repair` 尝试修复并重新验证。

4 运行和提交

代码写完后在 lab2 目录下打开命令行输入 `make`, 就会生成可执行文件 minirepairer。输入 `make run` 就会批量运行 test 目录下的 10 个测试用例, 结果会输出到 `test/output.txt` 中。若希望对单个测试用例进行测试, 可以输入 `./minirepairer test/xxx.c`, 其中 xxx 是测试用例的名字, 例如 `./minirepairer test/test1.c`, 结果会直接输出至命令行。

`./minirepairer test/xxx.c` 的输出格式为:

```
1 Answer: Verified
```

或在验证失败时, 输出修复之后的代码

```
1 Answer: successfully verified with repaired code:
2 int test2(int a) {
3     a = 1;
4     assert(a == 1);
5 }
```

完成本 lab 后请在 lab2 目录下运行 `make handin` 生成 lab2.zip, 然后上传到 canvas。注意: 请不要在 lab2.cpp 中进行多余的输入输出操作。否则可能会产生误判。

5 任务

你需要实现 lab2.cpp 中的以下函数。(其它文件都是框架代码，理论上无需修改)。

5.1 函数 mystoi

```
1 int mystoi(string s);
```

mystoi 需要把输入字符串 s 里的数字转成 int 类型返回。s 可能是 10 进制无符号整数、10 进制带符号整数或 16 进制整数，都在 32 位整数表示范围内。你可以通过调用 c++ 的某些已有库函数实现此功能，也可以自己手动进行转换。这个函数不需要考虑不合法的 s。

5.2 函数 copy_exp_tree

5.2.1 功能

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

函数的参数 root 指向一棵表达式树，copy_exp_tree 需要拷贝这棵树产生一棵全新的树，然后返回新的树。exp_node 的定义在 cfg.h 中，表示表达式树的节点。如果 root 为 NULL，就返回 NULL；否则就生成一棵新的树，这棵树表示的表达式和 root 表示的完全一致，这棵树的所有节点都是 new 出来的，不可以复用 root 中的任何节点。

5.2.2 表达式树

```
1 //exp_var表示表达式中的变量，如a
2 //exp_num表示表达式中的int常数，如11
3 //exp_op表示表达式中的操作符，如+，-，!=，>等，也包括一元操作符~
  等
4 enum EXP_NODE_TYPE { exp_var, exp_num, exp_op };
5
6 //表达式树中的节点
7 //可以表示布尔类型表达式如a>3
8 //也可以表示int类型表达式如a+b或者3
9 class exp_node {
10
11 public:
12     //当前节点存储的数据类型，包括变量、常数和操作符
13     EXP_NODE_TYPE type;
14     //当前节点存储的数据
15     //如果是变量就是"a"这样的变量名
16     //如果是操作符就是"+"这样的操作符
17     //如果是常数就是"11"这样的常数
```

```

18     string val;
19     //表达式树的左右节点
20     exp_node* child[2];
21
22 };

```

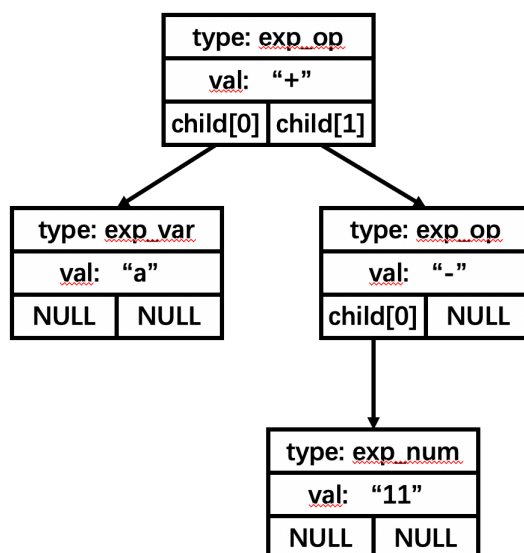


图 1: a+(-11) 对应的表达式树

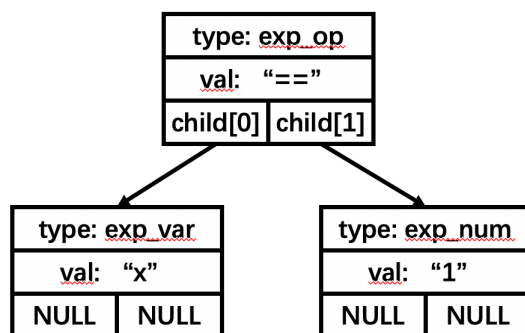


图 2: x==1 对应的表达式树

表达式树由节点组成，节点的类定义如上，这个类各个字段的含义请参见注释，下面来看两个例子。a+(-11) 这种表达式对应的表达式树如图1。x==1 对应图2中的表达式树。

```

1 //输出表达式树root存储的表达式
2 void print_exp(exp_node* root);

```

你可以调用 print_exp 来输出一棵表达式树。这个函数可以用来 debug。

完成前面两个函数后你就已经拥有一个基本能 work 的符号执行引擎来，此时可以通过前 4 个测试用例，他们都不包含 if 语句而且不需要进行修复。

5.3 函数 llm_repair

```
1 void llm_repair(const std::string& source_code, const vector<int  
    >& counterexample);
```

函数的参数 source_code 是源代码，counterexample 是符号执行得到的反例。你需要调用 LLM 修复程序，并返回修复后的代码。

类似于 lab1，你需要在 lab2.cpp 中实现函数 llm_repair，该函数需要调用 LLM 修复程序，并返回修复后的代码。你需要在这里设计你的 prompt，并解析 LLM 的输出，返回一个 C 程序。你需要在 prompt 里明确指出让 LLM 以 ```c ... ``` 包裹修复后的代码 (markdown 格式) 并提取后返回，其他格式可能会影响评分。

```
1 ```c  
2 int test2(int a) {  
3     a = 1;  
4     assert(a == 1);  
5 }  
6 ```
```

Hint 1: LLM 可能会通过修改程序最后的 Assertions 来修复程序，然而我们在评分时仍然会使用之前的 Assertions。你需要通过 prompt 避免这种情况，否则可能会在评分时被判断失败。

Hint 2: Prompt 设计具有极大的灵活性，任何你不希望看到的情况，都可以尝试用 Prompt 来让 LLM 不要输出。

完成这个函数后，你就可以通过前 8 个测试用例了。

5.4 函数 analyze_if

为了支持稍微复杂一点的程序（后两个测试用例），你还需要实现函数 analyze_if。

WARNING

请确保你已掌握 symbolic execution 课件的内容。

6 数据结构和代码流程

下面先讲解 MiniSEE 的数据结构和执行流程。

6.1 程序控制流图 CFG

MiniSEE 从源文件中读取程序后会用程序流图 (control flow graph,CFG) 存储。程序流图是一个有向无环图，图中每个节点对应程序中的一条语句，节点 A 有边指向节点 B 表示执行完 A 语句后会执行 B 语句，如果某个节点对应 if 语句，那么这个节点会有两条出边，分别指向 if 条件为真和条件为假两种情况下的下一条语句。

每个节点用 cfg.h 中的数据结构 cfg_node 表示。

```
1 //cfg_assign表示赋值语句
2 //cfg_if表示if语句
3 //cfg_assert表示assert语句
4 //cfg_return表示return语句
5 enum CFG_NODE_TYPE { cfg_assign , cfg_if , cfg_assert , cfg_return };
6
7 //控制流图中的节点，每个节点存一条语句
8 class cfg_node {
9
10 public:
11     //语句类型，包括赋值、assert、if、return
12     CFG_NODE_TYPE type;
13
14     //赋值语句中等号左边的变量名
15     //如a = b+c, dst就是"a"
16     //如果是其他类型的语句则是空
17     string dst;
18
19     //如果是赋值语句，exp_tree存储等号右边的表达式
20     //如果是return语句，exp_tree存储返回值对应的表达式
21     //如果是if语句，exp_tree存储if条件对应的布尔类型表达式
22     //如果是assert语句，exp_tree存储assert的参数对应的表达式
23     exp_node* exp_tree;
24     //这条语句在源文件中的行号，在lab中用不到
25     int lineno;
26
27     //接下来要执行的语句
28     //如果当前节点是assert、return、赋值语句，那么next[0]指向下一
        条语句，next[1]是NULL
29     //如果当前节点是if语句，那么next[0]指向if条件为真要执行的指
        令，next[1]指向if条件为假要执行的下一条指令
30     //如果next[0]和next[1]都是NULL，说明没有下一条语句了
31     cfg_node* next[2];
32 };
```

各个成员的含义参见注释。下面看一个例子，比如下面这段程序。

```
1 int func(int a, int x) {
2     x = a + (-11)
3     assert(x == 1);
4 }
```

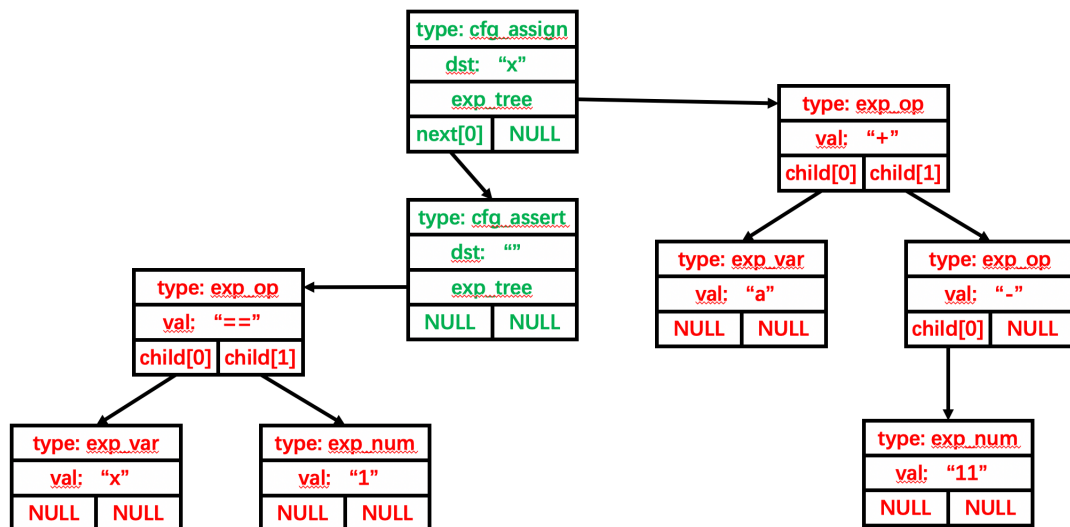


图 3:

对应的程序流图如图3所示，图中绿色的是 cfg 节点，红色的是各个 cfg 节点里面存储的 exp 节点。可以看到，这里没有 if，所以每个节点都只有一条出边。因为是程序，有入口，所以程序流图实际是有一个 root 节点的。

下面是一个有 if 语句的示例程序。

```

1 int func(int a, int b) {
2     if(a > 0)
3         a = -b;
4     else
5         b = 2;
6     assert(a + b < 1);
7 }

```

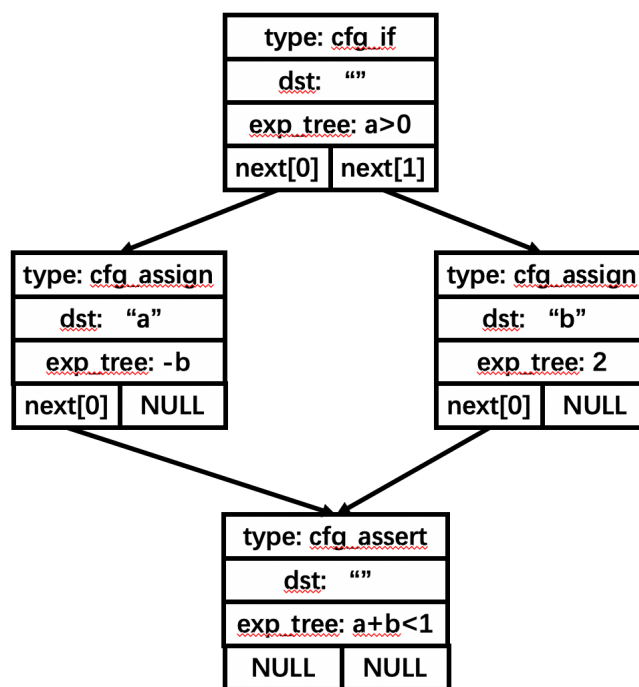


图 4:

它的 cfg 就如图4所示，此图省略了 exp_node。可以看到，if 语句对应的 cfg_node 有两条出边，左边的有向边指向条件为真时的语句，右边的有向边指向条件为假时的语句。

```

1 void print_cfg(cfg_node* root);

```

你可以调用 print_cfg 来输出一个程序流图，输出格式是把图转成源代码的形式输出。这个函数可以用来 debug。

6.2 see_state

```

1 //see_state是当前符号执行节点的状态
2 //包括symbolic store, path constraint和next code
3 class see_state {
4 public:

```

```

5 //symbolic store , 是一个数组
6 //每个变量在其中有一个表达式树exp_node*
7 vector<exp_node*> sym_store;
8 //path constraint
9 //如果是NULL说明path constraint 是true
10 exp_node* path_const;
11 //控制流图control flow graph(CFG)
12 //表示接下来要执行的所有代码
13 //cfg->root就指向符号执行节点中的next code, 就是将要执行的代
   码
14 cfg_node* cfg;
15 };

```

回忆 symbolic execution 课件上的内容, 符号执行会产生一棵树, MiniSEE 中用类 see_state 表示树中的每个节点, 这个类在 minisee.h 文件中。和课件上讲的一样, 类中有 3 个成员变量, 对应课件上的 symbolic store, path constraint 和 next code, 其中 next code 是一个 cfg_node 指针, 指向一个程序流图, 这个图中的第一个节点存储的就是将要执行的代码。

另外我们还需要知道程序中有哪些变量。

```

1 //存储所有局部变量的名字
2 extern vector<string> vartb;
3 //存储所有参数的名字
4 extern vector<string> inputtb;

```

这些 vector 被定义在 cfg.h 中, 存储参数和局部变量的名字。每个测试用例中只有一个函数, 所有数据都是 int 类型, 没有全局变量。函数的参数和返回值都是 int 类型。

拿到一个 see_state 后, 我们想知道每个变量现在的值是多少, 于是有了下面这个 map, 在 minisee.cpp 中。

```

1 //symbolic store是一个数组, 每个变量对应其中一个元素
2 //name_to_index存储了从变量名字到变量在symbolic store里面下标的映
   射
3 //lab2中不涉及对name_to_index的操作
4 static map<string , int> name_to_index;

```

这个 map 是从变量名 (string) 到数组 sym_store 下标 (int) 的映射。例如, 我们拿到当前符号执行节点, 节点里面的 symbolic storage 是数组 sym_store, 那么参数 a 当前的值就是 sym_store[name_to_index["a"]] 表示的表达式树。因为符号执行中所有变量的值是用符号表达式表示的, 所以 sym_store 是一个 exp_node* 类型的数组。

6.3 符号执行流程

这部分内容请配合 canvas 上的 lab2 视频讲解和 lab2 中的代码注释。

符号执行会生成一棵树，我们按照类似深度优先遍历的方法来访问每个节点。深度优先遍历需要下面这个栈来辅助。

```
1 //用于深度优先遍历符号执行树的栈
2 stack<see__state> state_queue;
```

符号执行从 minisee.cpp 里面的 minisee 函数开始，输入是一个 cfg 表示的程序。

```
1 void minisee(cfg_node* cfg);
```

minisee 函数中首先调用 init_state 函数构造符号执行树的根结点，并把这个根结点压入栈 state_queue 中。之后 minisee 函数进入循环，每次取走栈顶的符号执行节点，然后调用 state_handler 函数处理它。

```
1 //state是当前符号执行节点
2 //包括symbolic store, path constraint和next code
3 void state_handler(see__state* state);
```

state_handler 函数会把这个节点中记录的 next code 拿出来，也就是 state->cfg，如果发现是 NULL 说明已经没有下一条代码了，直接返回。如果发现这条代码的类型 (state->cfg->type) 是 return，说明程序结束，也直接返回。

如果发现是赋值语句，那么调用 analyze_assign 函数处理。

```
1 //处理赋值语句
2 //当前state里面的cfg第一条指令是一条赋值语句
3 void analyze_assign(see__state* state);
```

这个函数框架代码已经给出，不需要大家完成，所以这里只介绍大概流程。当前 state 是赋值语句对应的符号执行节点，这个函数会生成这个节点对应的子节点压入栈中。state 的 next code 是 state->cfg，按照之前对 see_state 类的讲解，子节点的 next code 应该是 state->cfg->child[0]。因为当前不是 if 语句，path constraint 没有变化，所以子节点的 path_const 和 state 里的 path_const 相同。赋值语句会修改 symbolic store，所以调用 update_sym_storage 得到子节点的 symbolic store。这样以来，子节点的 3 个数据成员就都有了，然后把子节点压入栈中，等待大循环的下一轮处理。

如果发现是 assert 语句，说明到了验证的时刻了。调用 verify 函数，框架代码已经实现，生成逻辑公式交给 SMT solver Z3 进行验证。

如果是 if 语句请看视频讲解。

6.4 analyze_if 功能

```
1 void analyze_if(see__state* state);
```

state 是符号执行树中当前走到的节点，在符号执行中，这个节点会有两个子节点，分别对应 if 条件成立和不成立的情况，函数功能是构造这两个子节点并把它们压入栈 state_queue 中。实现这个函数需要调用 lab2.cpp 中的 update_path_const 函数构造新的 path constraint。