**Task 2: Escape Detection**

|  |  |  |
| --- | --- | --- |
| **Chew Yung Chung** a0133662J  a0133662@u.nus.edu | **Tan Qiu Hao, Joel** a0125473H  a0125473@u.nus.edu | **Juliana Seng** a0126332R  a0126332@u.nus.edu |

# Implementation and Algorithm

Within each function, the algorithm maintains a list of new pointers allocated, and when a return instruction is encounter, we check if the variable to be returned matches any entry in the list of newly allocated pointers. If a match is found, an antipattern is detected.

For each function which is not the main() function, a check is performed. The code is as follows:

|  |  |
| --- | --- |
| 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109 | ReturnInst \*R = dyn\_cast<ReturnInst>(&I);  if (R) {  Value \*v = R->getReturnValue();  if (v && allocaMap.size() > 0) {  pair<multimap<Value\*,Value\*>::iterator,multimap<Value\*,Value\*>::iterator> ret;  Value \*nextV = 0, \*prevV = dyn\_cast<Value>(R), \*first;  while (true) {  ret = allocaMap.equal\_range(v);  for (multimap<Value\*,Value\*>::iterator it = ret.first; it != ret.second; ++it) {  if (isa<AllocaInst>(v)) {  if (lineMap[it->second] < lineMap[prevV]) {  nextV = it->second;  }  } else {  if (lineMap[it->second] <= lineMap[v]) {  nextV = it->second;  prevV = v;  }  }  }  if (nextV == 0 || v == nextV) {  if (v && v->getName() != "") {  errs() << v->getName() << " escaped \n";  }  break;  }  v = nextV;  }  }  } |

*Important Code snippet from asg2-task2.cpp*

For each instruction in the function, we build a relationship graph between all store, load, getElementPointer, and cast instructions.

When the algorithm reaches a return instruction, the return value is retrieved for processing. If the return value is not null, the algorithm will attempt to retrieve the variable name that is associated with the return value and compare it against the entries in allocaMap. If a match is found, the program concludes that the return value is a local pointer, and reports an escape violation.

# Build and Run

Run build.sh to compile the test cases, program and execute the test cases.

|  |
| --- |
| $ ./build.sh |

The individual commands can also be found in build.sh.

An expected output of the shell code is as follows, all warnings are truncated.

|  |
| --- |
| ----------COMPILING TESTCASES----------  ----------COMPILING PROGRAM----------  ----------RUNNING TEST 1----------  WARNING: pointer <p> in the function <init\_array> will not exist after the return  ----------RUNNING TEST 2----------  WARNING: pointer <p> in the function <escape\_local> will not exist after the return |

# Test Cases

Test case 1: DCL30-C Antipattern 1

|  |
| --- |
| $ ./asg2-task2 escape.ll |

Test case 1 observes the behaviour of escape.c, whereby the return value is a pointer to a locally created array.

|  |  |
| --- | --- |
| 7  8  9  10  11 | char \*init\_array(void) {  char array[10];  char \*p = array;  return p;  } |

*Code snippet from escape.c*

Test case 2: DCL30-C Antipattern 2

|  |
| --- |
| $ ./asg2-task2 escape2.ll |

Test case 2 observes the behaviour of escape2.c, whereby the return value is a pointer to a locally created character.

|  |  |
| --- | --- |
| 7  8  9  10  11 | char \*escape\_local() {  char local\_char = 'a';  char \*p = &local\_char;  return p;  } |

*Code snippet from escape2.c*

# Corner Cases

Pointers to global variable

|  |
| --- |
| int globalArr[10];  int\* escape\_func();  int main () {  return 0;  }  int\* escape\_func() {  int a[10];  int \*p = &globalArr;  return p;  } |

In the function escape\_func(), the pointer p points to an global array, hence an escape violation does not happen, and the statement is valid. Our algorithm will not detect this as a escape.