Task 1: Dead Function Analysis

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

Implementation and Algorithm

The algorithm maintains several global data structures that are populated with information collated from 1 or more IR files, they are as follows:

- **`mainList`** - holds a list of function pointers to **`main()`** functions
- **`programFuncMap`** - maps function name to DEAD or LIVE status
- **`definedFuncMap`** - maps function name to function pointers
- **`pointerFuncMap`** - maps key variable name to function pointers

Firstly, **`mainList`** is populated by iterating through all the functions in the given programs and filtering functions with the function name "**`main`**".

At the same time, **`programFuncMap`** is populated by mapping the function names of all retrieved functions with an initial status of DEAD, and **`definedFuncMap`** stores the mapping of the retrieved function names to their respective pointer.

A triple nested for-loop is then executed to iterate through all the instructions that are reachable through the **`main()`** functions stored in the **`mainList`**. The code is as follows:

```
64  For (auto &F : mainList) {
65          for (auto &BB: *F) {
66                  for (auto &I : BB) {
67                          StoreInst *S = dyn_cast<StoreInst>(&I);
68                          if (S) InvestigateStoreInst(S);
69                          CallSite Call(&I);
70                          if (!Call) continue;
71                          Function *G = Call.getCalledFunction();
72                          if (G == nullptr) {
73                                  G = ResolveIndirectCall(Call);
74                                  if (G == nullptr) continue;
75                          }
76                          programFuncMap[G->getName()] = "LIVE";
77                          if (definedFuncMap[G->getName()] != 0) {
78                                  InvestigateFunction(*(definedFuncMap[G->getName()]));
79                          }
80                  }
81          }
82  }
```

Code snippet from asg2-task1.cpp

If the instruction is a store instruction, the **`InvestigateStoreInst()`** function is invoked to check if the instruction operand is a function pointer, and if it is currently pointing to a defined function. **`InvestigateStoreInst()`** also recognize that it is possible for a pointer to be indirectly mapped to a function, through other layers of pointer. Hence a do-while loop is utilized to iterate through all pointers, until a function or null value is reached. The information is then stored in **`pointerFuncMap`**.

If the instruction is a call instruction, the name of the called function is retrieved. However, if a null value is retrieved, the call may be referencing a function pointer, hence **`ResolveIndirectCall()`** is invoked to iterate through the **`pointerFuncMap`**, that was built previously, to retrieve the function name.

The status of the called function is subsequently set to LIVE in **`programFuncMap`**. Alternatively, if the called function is referenced with a function pointer, **`InvestigateFunction()`** is invoked to set its status to LIVE.

Finally, **`programFuncMap`** is iterated through and all functions with a DEAD status is retrieved and reported accordingly.

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----------


----------RUNNING TEST 2----------
WARNING: function <dead2_dead_function> is a DEAD function
WARNING: function <dead2_function_always_called> is a DEAD function
WARNING: function <dead2_complex_function> is a DEAD function
WARNING: function <dead2_maybe_dead_function> is a DEAD function


----------RUNNING TEST 3----------
```

```
----------RUNNING TEST 4----------


----------RUNNING TEST 5----------
WARNING: function <foo1> is a DEAD function
WARNING: function <foo2> is a DEAD function
```

Test Cases

Test case 1: Function pointer

```
$ ./asg2-task1 dead.ll
```

Test case 1 observes the behaviour of dead.c as follows

```
20  if (complex_function()) {
21          maybe_dead_function();
22  }
23  function_always_called();
24  if (0) {
25          dead_function();
26  }
27
28  if (dead2_complex_function()) {
29          dead2_maybe_dead_function();
30  }
31  dead2_function_always_called();
32
33  if (0) {
34          dead2_dead_function();
35  }
36
37  f = &dead_function;
38  f();
```

Code snippet from dead.c

Dead.c defines 4 functions, namely

- complex_function(), a function that randomly returns true or false
- maybe_dead_function()
- function_always_called()
- dead_function()

It also demonstrates the initiation and invocation of a function pointer to dead_function(). All functions are LIVE.

Test case 2: No main function

```
$ ./asg2-task1 dead2.ll
```

Test case 2 observes the behaviour of dead2.c, which does not contain a main() function, but however, defines 4 functions as follows

- dead2_complex_function(), a function that randomly returns true or false
- dead2_maybe_dead_function()
- dead2_function_always_called()
- dead2_dead_function()

However, as none was called, all 4 functions were detected as dead functions.

Test case 3: Multiple IR files

```
$ ./asg2-task1 dead.ll dead2.ll
```

Test case 3 observes the input of multiple files, namely dead.c and dead2.c, as described above. Functions that were defined in dead2.c that were previously not called, are invoked in dead.c, hence all functions are LIVE.

Test case 4: More function pointers

```
$ ./asg2-task1 dead3.ll
```

Test case 4 demonstrates more examples on function calls through function pointers.

Test case 5: Recursive function

```
$ ./asg2-task1 dead4.ll
```

Test case 5 observes the behaviour of having 2 functions that recursively call each other, but is not invoked from main(). Although the 2 functions reference each other, they are detected as dead functions as they were not called from main().

Function Pointers

In general, determining the functions that are pointed to by function pointers is undecidable. As such, for the assignment, our team has limited the test cases to these decidable cases:

Case 1: Direct Assignment to Local Variables

```
void abc() { }

int main() {
        void (*p)();
        p = &abc;
        ...
```

```
        p();
        return 0;
}
```

Case 2: Direct Assignment to Global Variables

```
void abc() { }
void (*p)(void);

int main() {
        p = &abc;
        …
        p();
        return 0;
}
```

Case 3: Multiple Direct Assignments

We are also able to handle direct assignments in this manner, and correctly determine the function called by tracking the store and load functions.

```
void abc() { }
void def() { }

int main() {
        void (*p)();
        p = &abc;
        …
        p = &def;
        p();
        return 0;
}
```

Case 4: Non-Function Assignment to Function Pointers

In cases where the function pointer is modified with some random value, we treat the result of the assignment as unknown:

```
void abc() { }

int main() {
        void (*p)();
        p = &abc;
        …
        p = (void*)0x12345;
        p();
        return 0;
}
```

Case 5: Function Pointer within Structs

```
void abc() { }

typedef struct {
        void (*p)();
        …
} happy;

int main() {
        happy h;
        h.p = &abc;
        …
        h.p();
        return 0;
}
```

Case 6: Function Pointer from Pointers to Structs

```
void abc() { }

typedef struct {
        void (*p)();
        …
} happy;

int main() {
        happy *h = malloc(sizeof(happy));
        h->p = &abc;
        …
        h->p();
        return 0;
}
```

Limitations

Branching Conditions

For this assignment, the algorithm we came up with, does not take into account of branching conditions. Instead, we relied on the optimization of the Clang compiler to remove unreachable code when there are trivial (always false) conditions for branching. As such, there are definitely corner cases that the algorithm is unable to detect. Much consideration was made to design the test cases, and listed below are some corner cases that were made aware to us.

```
if (!(x == 0 || 0 == 0)) {
        test1();
```

```
}
```

Although 0=0 will always evaluate to true, causing the if-statement to be false and its contents be never executed, test1() is not detected as a dead function.

```
if (!(x == x)) {
	test2();
}
```

A warning will be raised during compilation, as x==x will always evaluate to true, causing the if-statement to be false and its contents be never executed. However, test2() is not detected as a dead function. Nevertheless, if the variable x is replaced by a value, for example, 5==5, the algorithm will then be able to detect test2() as a dead function.

```
for (int y = 0; y < 0; y++) {
	test3();
}
```

The for-loop will fail the condition in the first iteration, hence it will not be entered at all and its contents be never executed, however, test3() is not detected as a dead function.

Function Pointers

For the assignment, our algorithm is unable to determine function pointers that are passed as parameters as this requires global analysis throughout possibly multiple LLVM IR files.

```
void def() { }

int main() {
	abc(&def);
	return 0;
}

void abc(void *g(void)) {
	g();
}
```

The algorithm is also unable to determine cases where a function pointer is returned from another function, and is subsequently invoked.

```
void def() { }

int main() {
	void (*p)(void) = abc();
	p();
	return 0;
}
```

```
void abc() {
        return &def
}
```