**Assignment 4**

**Mingzhe Du, Advaith Suresh**

We use ruby to implement this project. This project contains the following files:

|  |  |
| --- | --- |
| ASSEMBLY | The assembly code of the c files |
| DWARF | The llvm-dwarfdump output |
| main.rb | This program analyzes the assembly code and the llvm-dwarfdump output |
| xref.rb | This program uses the information generated by main.rb to generate the index.html |
| index.html |  |
| others | Like test c programs and object files… |

To implement the crossing-indexing, we did the project with several steps listed below:

1. Analyze the llvm-dwarfdump output

In this part, we use two arrays and three hash tables to store information that we need to use later.

|  |  |  |
| --- | --- | --- |
| array | file\_indexes | This array is used to store all the file indexes that are given by llvm-dwarfdump. E.g., index of test.c is 1, then we store 1 into the array. |
| array | file\_names | This array is used to store all the file names. E.g., we store “test.c” into this array. |
| hash table | lookup\_table | This table is used to store the file names, assembly addresses and its corresponding source code line number. E.g.,  {test.c => {0x0001=>1, 0x0002=>2, 0x0003=>2}  func.c => {0x0004=>1, 0x0005=>2, 0x0006=>3}} |
| hash table | new\_lookup\_table | This table is used to store the file names, assembly addresses and its corresponding source code. And if one source code line corresponds to multiple assembly code lines, then merging the assembly address. E.g.,  {test.c => {0x0001=>”int a”, [0x0002, 0x0003] => “a=1”}  func.c => {0x0004=>”int c”, 0x0005=>”int d”, 0x0006=>”c+d”}} |
| hash\_table | unused\_source\_code | This table is used to store all the source code that doesn’t have corresponding assembly code. |

1. Analyze the objdump output

We read the objdump output line by line, and to see if the address of each line is contained in the new\_lookup\_table, if it is matched, them there is an source code that we need to match with the assembly code. If not, them we just print out the assembly code. During this process, we need to check the table unused\_source\_code to see if we need to print out the source code line right after its previous source code line.

1. Branch-target linking

Every function call has a corresponding assembly code, so we just iterate the addresses stored in new\_lookup\_table, and to see if the assembly code which is referenced by the address contains a “call” or “jmp”, if so, we check if the following item is an addresses, if so we extract the address. Then we compare the address with all the addresses stored in new\_lookup\_table, if there exits an match, it mean that this is a subroutine call. By this way, we don’t need to worry about that we might jump to any library function, because we only scan the new\_lookup\_table, which only contains the self\_defined functions.