This is just a minimal introduction for LLVM IR that should cover most of what you need in the project. If you happen to find things not covered, you can refer to https://llvm.org/docs/LangRef.html#introduction

LLVM IR is an intermediate code representation, which is lower level than the source code (e.g., C, C++), but higher level than assembly code. So some things look more like a high-level language (like functions and the strong typing). Others look more like low-level assembly (e.g., branching, basic blocks, memory read/write).

You can generate LLVM IR using clang, which is a compiler and good alternative to gcc. In addition to generating LLVM IR, it is also capable of generating the binary (just like any regular compiler). For example, Google has started using clang to compile the Android kernel for its pixel devices for a few years. To compile a C program into LLVM IR, you can execute the following command:
clang -emit-llvm -S -fno-discard-value-names -O2 memoryoverflow.c -g -o memoryoverflow.ll

LLVM IR is strongly typed, you can see there are always type notations for every single variable.

LLVM IR doesn't differentiate between signed and unsigned integers.

Take a look at the two piece of code in source code C and its LLVM IR, you can find this code and IR under directory memoryoverflow/

C (memoryoverflow.c):

```
3   long long a[10];
4   int b[20];
```

LLVM IR (memoryoverflow.ll):

```
7   @a = common dso_local global [10 x i64] zeroinitializer, align 16, !dbg !0
8   @b = common dso_local global [20 x i32] zeroinitializer, align 16, !dbg !6
```

In C, we defined two global arrays, one of them is a long long array (where every element is 64-bit) and another one is an int array (where every element is 32-bit).

In LLVM IR, there will be two global pointers, one of them points to a long long array with 10 elements, see [10 x i64], and another one points to an int array with 20 elements, see [20 x i32].

And the 'global' flag indicates that they are two global variables. Other flags like 'common', 'zeroinitializer' are directives for the compiler which you don’t have to worry about in this project.

For another piece of codes:

C:

```c
void checkthisfunc(){
    memset(a, 2, sizeof(long) * 12);
    memset(b, 2, sizeof(int) * 12);
}
```

LLVM IR:

```
; Function Attrs: nounwind uwtable
define dso_local void @checkthisfunc() local_unnamed_addr #2 !dbg !25 {
entry:
  tail call void @llvm.memset.p0i8.i64(i8* nonnull align 16 dereferenceable(96) bitcast ([10 x i64]* @a to i8*), i8 2, i64 96, i1 false), !dbg !26
  tail call void @llvm.memset.p0i8.i64(i8* nonnull align 16 dereferenceable(48) bitcast ([20 x i32]* @b to i8*), i8 2, i64 48, i1 false), !dbg !27
  ret void, !dbg !28
}
```

In C source code, we defined a function with the name 'checkthisfunc()'. In the function there are two memset() function invocations.

In LLVM IR, there will be a function with name 'checkthisfunc()', but you can see there are two functions with name 'llvm.memset.p0i8.i64()', this name change is due to internal LLVM reasons. For the purpose of this project, you can safely assume whichever function containing the keyword of ‘memset’ that is the original memset function.

Since LLVM IR is a strongly typed language, you can see explicit notations about parameter types. For example, for a memset() function in C, its type definition should be:

void memset (void *, char, size_t);

While in LLVM IR, it's:

void @llvm.memset.p0i8.i64(i8*, i8, i64, i1)

Please ignore the last bool parameter in the LLVM IR.

Another thing worth noting is the 'bitcast' operator in the first parameter of memset, which is used to type cast the globalvariable @a from an array pointer type ([10 x i64] *) to i8* type. So you can see LLVM IR is always trying to make sure the types are matched when passing a parameter, i.e., memset() expects i8* as the type of the first parameter.

Ignore flags like 'tail', 'dereferenceable', etc.