Question 1.

This question makes use of the C library functions `fgets`, `strlen`, and `strcpy`. Descriptions of these functions are given below:

```
char *fgets(char *s, int size, FILE *stream):
Reads at most size-1 characters from stream and stores them in the buffer pointed to by s.
Reading stops after an EOF or a newline.  If a newline is read, it is stored in the buffer.
A '\0' is stored after the last character in the buffer.

int strlen(char *s):
Returns the number of characters up to, but not including the nearest '\0' in s

char *strcpy(char *dst, const char *src):
Copies the string pointed to by src (including the terminating null character) to the array
pointed to by dst.
```

Part (a)

Languages like C that do not guarantee array-bounds checking and that allow pointer arithmetic can lead to programs that are vulnerable to certain kinds of malicious attacks. Consider the program shown in Figure 1. What buffer overrun(s) can occur and why?

```
main(int argc, char* argv[]){
   char header[2048], buf[1024],
        *cc1, *cc2, *ptr;
   int counter;
   FILE *fp;
   ...
   ptr = fgets(header, 2048, fp);
   cc1 = copy_buffer(header);
   ptr = fgets (buf, 1024, fp);
   cc2 = copy_buffer(buf);
}

char *copy_buffer(char *buffer){
   char *copy;
   copy = (char *) malloc(strlen(buffer));
   strcpy(copy, buffer);
   return copy;
}
```

Figure 1: Example Program

Part (b)

Explain how a malicious user can exploit a buffer overrun vulnerability in a program (not necessarily the one(s) that can occur in the program in Figure 1).

Part (c)

We will sketch a flow-insensitive static analysis technique to detect buffer overruns. Each buffer *buf* (variable of type `char *`) is associated with two range-valued variables $R_{chars}(buf)$ and $R_{alloc}(buf)$ (one for the number of characters stored in the buffer and the other for the allocated space). A program variable i (of type `int`) is associated with a single range-valued variable $R(i)$ (representing the possible values of i). Intuitively, if $R_{chars}(buf) = (n, m)$, then the minimum and maximum number of characters stored in the buffer *buf* are n and m, respectively. Similarly, $R_{buf}(buf) = (n, m)$ indicates that the minimum and maximum allocated space for the buffer *buf* are n and m, respectively. Each program statement generates a subset constraint on range-valued variables. For example, consider the following statement:

```
strcpy(a,b)
```

Since b is copied into a, the following constraint is generated:

$$R_{chars}(b) \subseteq R_{chars}(a)$$

Show the range constraints generated by the program given in Figure 1.

Note: You will have to use sets of constraints that model the library functions `fgets`, `strlen`, and `strcpy`, described at the beginning of this question.

Part (d)
Solving a system S of range constraints means finding the "tightest" possible ranges that respect the constraints in S. For example, consider the following system S_1 of range constraints:

$$\begin{array}{rcl} (4,4) & \subseteq & R(a) \\ (8,8) & \subseteq & R(b) \\ R(b) & \subseteq & R(a) \end{array}$$

The following assignment of ranges is the "tightest" that respects constraints in S_1:

$$\begin{array}{rcl} R(a) & = & (4,8) \\ R(b) & = & (8,8) \end{array}$$

Notice that the following assignment of ranges also respects the constraints in S_1, but does not assign the "tightest" possible ranges.

$$\begin{array}{rcl} R(a) & = & (1,9) \\ R(b) & = & (5,8) \end{array}$$

Suppose there is procedure P for solving a system of range constraints, i.e., procedure P returns the "tightest" possible ranges that respect the constraints in a system S. How will you use procedure P to discover possible buffer overruns?

Part (e)
Consider the range constraints from Part(c). Give a graph algorithm to solve the range constraints generated in Part(c). You need only explain your algorithm with respect to the specific set of constraints generated in Part(c).

Question 2.

Many object-oriented languages, including C++, C# and Java, allow methods of an object to be virtual (dynamically dispatched).

Part (a)
How does a virtual method differ from an ordinary non-virtual method? What are the advantages of making an instance method virtual?

Part (b)
What extra run-time overhead is imposed in calling a virtual method? That is, what extra steps are needed in a method call *obj.m()* if *m* is virtual.

Part (c)
Java makes all instance methods virtual. It is clear that many methods do not require the extra capabilities a virtual method provides. Describe a static interprocedural analysis that can be performed to recognize method calls that can be treated (and implemented) as ordinary (non-virtual) calls.

Question 3.

Consider extending the C language by allowing pointer parameters to be declared *restrictedAccess*. For example, the declaration of p in: void foo(restrictedAccess int *p) means that in the body of function *foo*:

(a) p always points to the same location (there are no direct or indirect assignments to p), and

(b) any read or write of the location pointed to by p will be done via the expression $*p$; i.e., the expression $*p$ may have aliases, but they will not be used in the body of *foo*. If p points into an array, then this restriction holds for the whole array; any read or write of any element in the array will be done only via expressions of the form $*p$, $*(p+k)$, $*(p-k)$ or $p[k]$.

Note that the restrictions apply only within the *static* scope of the function. For example, if function *foo* calls another function, *bar*, there are no restrictions, within the body of *bar*, on reads or writes of the location pointed to by p, and it is possible for *bar* to modify p itself.

Note also that the *restrictedAccess* annotation is allowed only for single-level pointers (not for scalars and not for multi-level pointers); thus, the following declarations of k and q would be illegal:

void foo(restrictedAccess int k, restrictedAccess int **q)

Part (a)

Why would *restrictedAccess* annotations be useful? Give at least two examples of (different kinds of) optimizations that would be facilitated by the presence of *restrictedAccess* annotations.

Does the fact that the *restrictedAccess* annotations apply only to the static scope of the function restrict their usefulness? (Justify your answer.)

Part (b)

Assume that the results of flow-sensitive pointer analysis are available to a compiler; i.e., for each statement (each control-flow graph node) we know, for each variable v, what locations v might point to just before that statement is executed.

Describe how the compiler could use this points-to information to verify *restrictedAccess* annotations provided by a programmer. The approach you describe should be *safe* – i.e., it should never say that an annotation is correct if in fact there is an illegal read or write in the function body; however, it may say that a correct annotation is not correct (but of course you should try to minimize how often this will happen).

Part (c)

Now assume that the *restrictedAccess* annotations refer to the *dynamic* scope of the function; i.e., if function *foo* (whose header was given at the beginning of this question) calls another function, *bar*, then restrictions (a) and (b) listed at the beginning of this question apply in function *bar* and any functions that it calls (directly or indirectly).

Does the approach you defined in part (b) still work (i.e., is it still safe)? If yes, justify that claim. If no, give an example where it would be unsafe, and explain how the approach could be modified to make it safe under the new assumption.

Question 4.

Recall that a dataflow-analysis can be defined in terms of a lattice of dataflow facts (with a meet operation $\sqcap$) and a dataflow function for each kind of control-flow graph (CFG) node.

A dataflow function f is *distributive* iff for all possible pairs of dataflow facts d_1 and d_2:
$f(d_1 \sqcap d_2) \; = \; f(d_1) \sqcap f(d_2)$

Part (a)
The dataflow functions for constant propagation are *not* all distributive. Show that this is true by supplying each of the following:

(a The type of a lattice element for constant propagation.

(b) The meet operation for constant propagation.

(c) An example CFG node and its dataflow function f.

(d) Two dataflow facts d_1 and d_2 such that $f(d_1 \sqcap d_2) \; \neq \; f(d_1) \sqcap f(d_2)$

Part (b)
Show that even if a program has just one variable v, it is possible to have a CFG node with a non-distributive function (hint: think about operations that map different inputs to the same result).

Part (c)
Why are non-distributive dataflow functions a problem in practice?

Question 5.

Consider the following language of regular expressions over alphabet Σ:

$$
\begin{array}{lcll}
\text{E} & ::= & 0 & \\
 & | & 1 & \\
 & | & \text{a} & a\epsilon\Sigma \\
 & | & E_1 \ + \ E_2 & \\
 & | & E_1 \ \cdot \ E_2 & \\
 & | & {E_1}^* &
\end{array}
$$

The symbol 0 stands for the empty language (*i.e.*, the empty set of strings), whereas the symbol 1 stands for the language $\{\varepsilon\}$ (*i.e.*, the language consisting of one string, namely the empty string).

Let the symbol $\otimes$ be defined as follows (where L_1 and L_2 are sets of finite strings):

$L_1 \otimes L_2 \equiv \{\alpha\beta \mid \alpha\epsilon L_1 \text{ and } \beta\epsilon L_2\}$.

The meaning of this language of regular expressions is defined by the following meaning function M, which maps a regular expression to the language it denotes:

$$
\begin{aligned}
&M[[0]] = \emptyset \\
&M[[1]] = \{\varepsilon\} \\
&M[[a]] = \{a\} \\
&M[[E_1 + E_2]] = M[[E_1]] \cup M[[E_2]] \\
&M[[E_1 \cdot E_2]] = M[[E_1]] \otimes M[[E_2]] \\
&M[[{E_1}^*]] \quad = \{\varepsilon\} \cup M[[E_1]] \cup (M[[E_1]] \otimes M[[E_1]]) \cup (M[[E_1]] \otimes M[[E_1]] \otimes M[[E_1]]) \ldots \\
&\qquad\qquad\ = \textstyle\bigcup_{i=0}^{\infty} (M[[E_1]])^i
\end{aligned}
$$

where L^0 means $\{\varepsilon\}$, and L^i, for $i \geq 1$, means the i-fold $\otimes$-product of L (i.e., $L \otimes \ \ldots \ \otimes L$, i times).

Part (a)

In this part, you are to define a function AE (for "AcceptsEpsilon") on regular expressions that characterizes when a regular expression accepts ε. $AE(E)$ returns a *regular expression* whose meaning is

- The language $\{\varepsilon\}$ if E does accept ε.
- The empty set ($\emptyset$) if E does not accept ε.

For instance, we want $M[[AE(0)]] = \emptyset$ and $M[[AE(a)]] = \emptyset$, but $M[[AE(1)]] = \{\varepsilon\}$. Complete the following definition of $AE(E)$:

$$
\begin{aligned}
&AE(0) = 0 \\
&AE(1) = 1 \\
&AE(a) = 0 \\
&AE(E_1 + E_2) = \\
&AE(E_1 \cdot E_2) = \\
&AE(E_1^*) =
\end{aligned}
$$

Part (b)
Show, by structural induction, that for all regular expressions E,
$M[[AE(E)]]$ = if $\varepsilon \epsilon M[[E]]$ then $\{\varepsilon\}$ else $\emptyset$.

Part (c)
Suppose that G and H are two regular expressions. Using the function AE, show how to encode "if G accepts ε then H else 0" as a regular expression R. (That is, R should be a regular expression such that $M[[R]]$ equals $M[[H]]$ if G accepts ε, and $M[[R]]$ equals $\emptyset$ if G does not accept ε.)

Part (d)
A *residual* of a regular expression E with respect to symbol $a\epsilon\Sigma$ is another regular expression E' that accepts all strings β such that $a\beta \,\epsilon\, M[[E]]$. That is, any E' that satisfies
$M[[E']] = \{\beta \mid a\beta \,\epsilon\, M[[E]]\}$
is considered to be a residual of E with respect to a. For instance, if E is $a^* \cdot b + a \cdot c^* + d^* \cdot a \cdot b \cdot c$, one residual of E with respect to a is $a^* \cdot b + c^* + b \cdot c$,

(Note the resemblance of residuation to partial evaluation: We have "fixed" one symbol of the input; the residual E' defines how to handle the rest of the computation.)

The operation of *differentiation* of a regular expression with respect to a, denoted by $[a](E)$, is a way to construct a residual of E during a traversal of E's abstract syntax tree. Using AE in one of the cases, complete the following definition of $[a](E)$:
$[a](0) = 0$
$[a](1) = 0$
$[a](a) = 1$
$[a](b) =$ $\qquad$ $b \,\epsilon\, \Sigma$ and $b \neq a$
$[a](E_1 + E_2) =$
$[a](E_1 \cdot E_2) =$
$[a](E_1^*) =$

(Hint: Use the construction from Part (c) in one of the cases.)

Note that if your definition were applied to the regular expression E given in the example above, it would *not* produce exactly the regular expression $a^* \cdot b + c^* + b \cdot c$; however, the language accepted by the regular expression you produce should, of course, be $M[[a^* \cdot b + c^* + b \cdot c]]$.