Question 1 (MC/DC – code coverage) Which test cases are valid for the following C code if we consider the Modified Condition/Decision Coverage?

```
void test(int a, int b, int c) {
  if (((a>0||b==0)) && c<10)
    assert(1);
  else
    assert(0);
}
```

(a) a=-1, b=0, c=9

(b) a=1, b=0, c=11

(c) a=0, b=1, c=9

(d) a=1, b=-1, c=9

(e) a=-1, b=0, c=10

Question 2 (Fuzzing) Given an arbitrary input argument (due to some random fuzzer) for the following C code, what is the chance of triggering the assertion failure?

```
void test(char input[4]) {
  int cnt = 0;
  if (input[0]=='b')  cnt++;
  if (input[1]=='a')  cnt++;
  if (input[2]=='d')  cnt++;
  if (input[3]=='!')  cnt++;
  if (cnt >= 4)  assert(0);
}
```

(a) About one in a billion

(b) About one in a million

(c) About one in a thousand

(d) About one in ten

(e) None of the above.

Question 3 (Counterexample) What are the counterexamples for the assertion failure in the following C code?

```c
void test(char input[4]) {
  int cnt = 0;
  if (input[0]=='b')  cnt++;
  if (input[1]=='a')  cnt++;
  if (input[2]=='d')  cnt++;
  if (input[3]=='!')  cnt++;
  if (cnt >= 2)  assert(0);
}
```

(a) input="bad!"

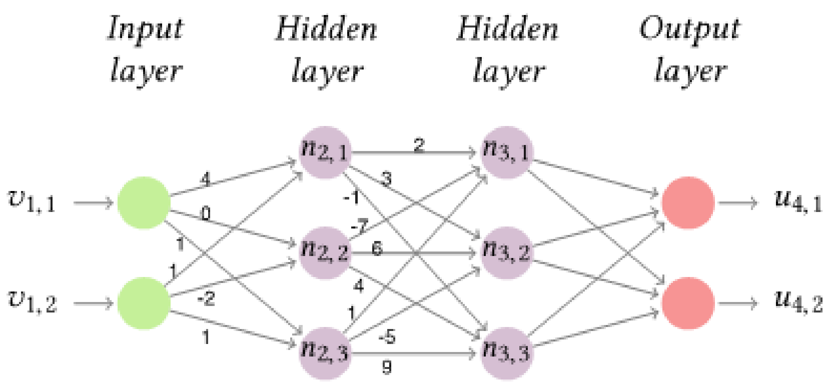
(b) input="b$ad"

(c) input="b*d&"

(d) input="%bad"

(e) None of the above

Question 4 (AI security) Given the following neural network, suppose that all its weight parameters are as annotated in the figure, all bias values are 0 and the ReLU activation function is used for its hidden neurons. What are the neurons covered according to Neuron Coverage when $v_{1,1}$=0 and $v_{1,2}$=-1?



(a) $n_{2,1}$

(b) $n_{2,2}$

(c) $n_{2,3}$

(d) $n_{3,2}$

(e) $n_{3,3}$

Question 5 (Vulnerability detection) Memory corruption is one of the significant sources of security vulnerabilities, typically occurring in unsafe programming languages such as C/C++. In this respect, consider the C++ code fragment, as illustrated in the figure below. Discuss whether this code fragment contains memory safety vulnerabilities. If so, which CIA principle gets violated? In this case, you should discuss approaches to finding the security vulnerability. It would help if you considered for your (technical) discussion concepts like temporal/spatial memory safety and static/dynamic analysis techniques.

```
class Foo {
  public:
    Foo() {};
    void Execute(void);
};

int main()
{
  auto *foo = new Foo();
  delete foo;
  // code that does something
  foo->Execute();
  return 0;
}
```

Question 6 (Root cause analysis) Root cause analysis is an important research field in software security. We must understand the root cause of the security vulnerability and provide detailed information on reproducing and fixing it in the underlying implementation bug. With this goal in mind, consider the classical problem of use-after-free vulnerability. Here, developers allocate a chunk of memory from the operating system and store the memory location in a pointer. After freeing that chunk of memory, developers typically forget to set the pointer to NULL and use it in other parts of the codebase. Your task here is to describe an automated static analysis method to understand the

root cause of the underlying security vulnerability so we can have enough information to understand the vulnerability and how to fix it.

Question 7 (Code repair) Over the last 20 years, over 2.8 trillion lines of code have been written. Automated code repair techniques are essential to remediate security vulnerabilities before exploitation. Once we identify the root cause of implementation vulnerabilities, code repair techniques can propose patches to fix the underlying implementation bug. In this respect, we can produce logical formulas using symbolic execution techniques to model the program's states and then propose fixes to the faulty program. Your task is to describe a method to localize and repair program faults using automated reasoning techniques like bounded model checking and path-based symbolic execution approaches. For simplicity, you can consider that the program contains a single fault of one kind, e.g., a dangling pointer, when designing your method.

Question 8 (Concurrency Verification) When the following C program is executed, there are 6 possible interleavings between thread1 and thread2? Among the 6 interleavings, how many will cause the assertion failure in main?

```c
int x=0;
int y=0;
int r1, r2;

void *thread1(void *arg) {
  r1=x;
  r2=y;
}

void thread2(void *arg) {
  x=1;
  y=1;
}

int main() {
  pthread_t id1, id2;
  int arg1=10, arg2=20;
  pthread_create(&id1, NULL, thread1, &arg1);
  pthread_create(&id2, NULL, thread2, &arg2);
  pthread_join(id1, NULL);
  pthread_join(id2, NULL);
  assert(r1 == 1 || r2 == 0);
}
```

(a) 0

(b) 1

(c) 2

(d) 3

(e) None of the above

Question 9 (Bounded Model Checking) How many loop unwindings are needed by model checking, for detecting the undefined behaviour in the C code below?

```c
int sum_twice(int x) {
  short int i, s;

  s=0;
  for (i=0; i<=x; i++) {
    s+=i;
    s+=i;
  }
  return s;
}
```

(a) 5

(b) 10

(c) 200

(d) There is no undefined behaviour

(e) None of the above.

Question 10 (Code simplification and reduction) Constant propagation and forward substitution techniques significantly improve the performance of static analysis techniques since they have simplify and reduce the logical formula sent to the underlying Satisfiability Modulo Theories (SMT) solvers. Consider the following C code fragments. How could you simplify the unrolling of this program using code simplification and reduction techniques?

A)

```
#include <string.h>
void puts(const char *s) {
 while (*s) {
   putc(*s++);
 }
}
int main() {
 puts ("blit:success");
 return 0;
}
```

B)

```
#define N 50
int a[N];
int main() {
 a[0] = 1;
 int i;
 for(i = 1; i<N; i ++)
   a [i]= a[i-1] + i;
 assert(a[i-11] < 2000);
 return 0;
}
```