CS510 Midterm (2010 Fall)

October 19, 2010

1 Testing (15p)

(a) (Combinatorial Testing) Assume a program has three factors: A, B, and C. The levels of these factors are $\{a_1, a_2\}$, $\{b_1, b_2\}$, and $\{c_1, c_2, c_3\}$. Compute the pair wise cover array using the IPO algorithm.

(b) (Mutation testing)

```
1. input (a, b);
2. if (a>10)
3.   print ("1");
4. x=b/3;
5. if (x>0)
6.   print ("2");
```

Assume we have three mutants. One is "a>10" at line 2 is mutated to "a>=10", the second is that "x=b/3" is mutated to "x=b/2", and the third is that "x>0" is mutated to "x>=0".

Assume the test suite is {(a=11, b=1), (a=3, b=10)} and the oracle is purely based on the program output. What is the mutation coverage? Details are encouraged.

2 Execution Indexing (15p)

```
1. input(a,b,c);
2. z=0;
3. while (a>0) {
4.    if (a%b==0) {
5.       c=c-1;
6.       if (c>a)
7.          z=z+1;
8.       else
9.          z=z+2;
10.   }
11.   a--;
12.}
```

Recall that execution indexing can be used to locate a particular execution point. Present an algorithm in instrumentation rules that can locate the execution point specified by an index. For instance, given the index $[3^T, 4^T, 5]$, the execution point 5 in the trace "1 2 3 4 5 ..." is identified. If the point specified by the index is not reachable, the algorithm should clearly indicate so at the earliest possible point. For instance, the point specified by $[3^T, 4^T, 5]$ is never reached in an execution with the prefix "1 2 3 4 11 ...". The algorithm should be able to tell at 11 that the execution point is not reachable.

Given the input a=3, b=2, c=1, show how your algorithm identifies the execution point with index $[3^T, 3^T, 4^T, 6^F, 9]$.

3 Delta Debugging, 10 points

Assume the program fails when the input has equal numbers of a, b and c. Find the minimal inducing input for "ababcccc" using delta debugging.

4 Dynamic Analysis (25p)

```
1. void (* F) ();
2. char A[1];
3. char B[10];
4. int i,j;

5. i=j=0;
6. read(B, 10); //read 10 bytes
7. F= &foo();
8. while (j<10) {
9.   if (B[j]==’b’)
10.     break;
11.  j=j++;
12.  if (j>0)
13.     i++;
14.  (*F) ();
15. }
16. A[i]=B[j];
17. (*F) ();
```

Data provenance is a technique that tracks the set of INPUT VALUES that a variable or an executed statement is dependent on. For example, assume a program execution is

```
1. read (buf, 2) with input 10 and 20;
2. x=buf[0];
3. y=x+buf[1];
```

.

The provenance of x and y are $\{10\}$ and $\{10, 20\}$, respectively. Data provenance can be used to defend code injection attacks by not allowing a function call to have a non-empty provenance.

(a) (15 points) Sketch a forward online algorithm that computes data provenance forwards along program execution, considering both data and control dependences.

(b) (10 points) Assume the input is "cb", apply your algorithm to the program at the beginning to detect code injection vulnerabilities. Note that function pointer F and array A are next to each other on the stack so that $A[1]$ shares the same memory location with the first byte of F.

5 Compression (10p)

A last n predictor has a buffer for the last n unique values that occured, and then predicts the next value to be one of those values. For example, at the end of a string of 1 2 2 3 4, the buffer of a last-3 predictor contains the values of 2 3 4. Sketch a last-3 predictor decompression algorithm. Please first compress the execution trace 6 16 16 16 17 10 10 13 and then apply your algorithm to decompress it.

6 Dynamic Slicing (10p)

```
1. x=0; //error, should be x=1
2. y=10;
3. if (x>0)
4.   y=y+x;
5  print (y);
```

There is a bug in the above program at line 1. The output 10 is faulty and the correct output should be 11.

(a) What is the dynamic slice of variable y at 5? Does it capture the root cause? Why? (10p)

(b) Propose a new dynamic slicing that is able to capture the root cause. You can assume you can transform the program or conduct any program analysis such as identifying the set of variables used and defined in a code region. (5p)

7 Misc. (10p)

A limitation of delta debugging is that it generates a lot of ill-formed inputs, meaning inputs do not follow their syntax. We know in random test generation, input grammars can be used to guide the generation of well-formed inputs. Sketch an enhanced version of delta debugging such that it only runs the program on inputs that are always well formed. In other words, the algorithm is supposed to carry out syntatically well-formed reduction. Use an example to explain your idea if necessary.