ECE653 Assignment 1: Software Testing, Fuzzing, Program Analysis, Boolean Logic (Lectures 1-5)

Question 1: Short answer questions (20 points)

Please provide short and correct answers to the following questions. You will get points only if you provide an appropriate justification along with your answers. Otherwise you will get 0 points. Each sub-question is worth 2 points.

1. Provide a formal definition of Static Single Assignment (SSA) form. Analyze the code fragments below and identify whether they are in SSA form. If yes, explain how the code fragment adheres to the notion of SSA form. If not, explain why not and convert the fragment into SSA form (do not perform any other transformations).

```
Code fragment 1:
x = x + y;
z = y + t;
x = z;

Code fragment 2:
x = 42;
x = y - 23;
if(x <= y) {
y =  z + x;
}
else {
y =  x;
}
```

2. Provide a formal definition of the notion of control-flow graphs of programs (for simplicity assume that programs in this question are written in a simple programming language such as LLVM).

3. Define the notions of false positive, true positive, false negative, and true negative in the context of static and dynamic analysis methods aimed at finding errors in software.

4. Define the notion of undecidability. Intuitively argue why statically establishing program properties in general is undecidable, by starting with the assumption that the halting problem is undecidable.

5. Define the notion of normal forms of a Boolean formula. Explain why formulas in conjunctive normal form (CNF) and disjunctive normal form (DNF) are also in negation normal form (NNF).

6. The truth table method discussed in class is a model-theoretic approach to establishing validity of Boolean formulas. We refer to this method as model-theoretic because it systematically checks the satisfiability of a given Boolean formula over every possible assignment. By contrast, proof-theoretic methods such as the semantic argument method construct proofs using a set of proof rules and axioms. Discuss a weakness of such model-theoretic methods relative to proof-theoretic methods.

7. Using the semantic argument method, establish the validity of the following formula:

$$((p \rightarrow q) \wedge (q \leftrightarrow r)) \rightarrow (p \rightarrow r)$$

8. Can all 2-input 1-output Boolean functions be encoded using only XOR gates?

9. Prove the validity of Demorgan's laws for propositional logic.

10. Explain the difference between equisatisfiability and logical equivalence? Provide a pair of formulas that are equisatisfiable but not logically equivalent.

Question 2: Fuzzing (10 points)

Briefly describe how Google's AFL-cov fuzzer and code coverage tool works. Include details about its method of code instrumentation, how it tracks paths taken, and performs fuzzing. Further, download and install AFL (https://github.com/google/AFL.git) and run it on the following programs for at least 5 hours. Report all errors found, if any. Also, report code coverage.

1. MiniSAT solver: http://minisat.se/downloads/minisat-2.2.0.tar.gz

Question 3: Symbolic Execution (40 points)

Dynamic systematic testing (aka dynamic symbolic testing or concolic testing) is aimed at both 1) automatic systematic test coverage, and 2) finding deep security vulnerabilities. For convenience, I will interchangeably use the term concolic testing and symbolic-execution based testing (or techniques).

We learnt in class the power of the combination of symbolic execution, concretization and constraint solvers (e.g., SAT solvers) in automatically finding security vulnerabilities and systematically testing software. In this question, we will analyze the role of combining symbolic execution with concrete execution (aka concolic testing), and see how it is more powerful than either symbolic or concrete execution by themselves. (The term concolic is a portmanteau of concrete and symbolic.)

0.1 Definitions

Concrete Execution: The term *concrete execution* refers to the normal execution or *run* of a program P on a computer on *concrete* inputs, i.e., the inputs to the program P are values from P's input domain.

Program Path or Trace: A program path (simply path or trace) is the sequence of instructions executed by a program on a concrete or symbolic input.

Symbolic Execution: The term *symbolic execution* refers to an execution or *run* of a program P on *symbolic* inputs (i.e., input variables are not assigned concrete value but instead range over all values from the input domain of the program P or all values associated with the type of the input variable). Symbolic execution of a program P can be acheived on a computer by executing the program P symbolically using virtual machine or interpreter, e.g., the KLEE symbolic virtual machine (http://klee.github.io/).

A symbolic virtual machine constructs a map from program variables to logic expressions over input variables (aka, the shadow memory). The symbolic virtual machine executes a program symbolically by stepping through the program one instruction at a time, and updating the map from program variables to logic expressions. As the program is executed by the symbolic virtual machine, this map is updated to reflect the most current symbolic expression for each program variable. The result of symbolic execution is a path constraint in a suitable logic, defined below.

Path Constraint: Given a program path $p(\bar{x})$ in a program whose inputs are denoted by $\bar{x}$, the *path constraint* corresponding to $p(\bar{x})$ refers to the logic formula $\phi(\bar{x})$ that is constructed as follows: Step 1: Symbolically execute the chosen path $p(\bar{x})$ in the given program; Step 2: Construct a conjunction of equalities between program variables and their final symbolic values, as well as all the conditionals taken to execute the chosen path. This formula, over the inputs $\bar{x}$ to the program, is called the path constraint and compactly represents all input values that will exercise the path $p(\bar{x})$.

Constraint solvers: A constraint solver (e.g., Boolean SAT and SMT solvers) is a computer program that take as input logic formulas and determines whether these formulas are satisfiable. We say that a formula is satisfiable, if there exists an assignment A to the variables in the formula such that the formula is true under A.

0.2 Systematic testing using Symbolic Execution, Concretization and Solvers

Now that we know how to symbolically execute paths of a program and what a constraint solver is, it is easy to see how the two can be combined to systematically test every path in the program: First, we symbolically execute a program path to obtain a path constraint. Second, the path constraint is solved by a constraint solver to obtain an assignment of values to the input variables of the program-under-test, such that when the program is executed on these values it takes the path that was symbolically executed in the first step. By doing this systematically for every path in the program, we obtain a set of test inputs to concretely execute all paths in the program.

There are several problems that need to be overcome in order for such a systematic testing technique to become scalable.

- Constraint solving problem: The path constraints may be too difficult for constraint solvers to solve
- Path Explosion Problem: The number of paths in a program is typically exponential (or worse) in the number of if-conditionals (or branches) in the program (it can even be infinite if the program has an infinite loop)
- Symbolic execution is much slower than normal concrete execution

In the following questions, we will focus on the first problem, namely, constraints may be too difficult for solvers to solve. One way around this is to concretize certain inputs such that hard parts of constraints become easy to solve. This approach to executing the program symbolically, where parts of the input are concretized, is referred to as *concolic execution*, and has partially led to the amazing success of symbolic execution based techniques to scale to testing of very large programs. The question of which inputs should be concretized and which should be left symbolic is a heuristic, determined through appropriate automatic program analysis and the intuition of the user.

0.3 The Questions

Question 3.1 (7.5 Points)

Consider the following piece of C code:

```
int obscure(int x, int y)
{
if (x == hash(y)) {
//call to function with error
}
else {
//call to some function
}
}
```

The function hash is some standard-issue cryptographic hash function like SHA1. In class, we learnt when a constraint solvers solves a constraint it is essentially inverting the function/relation encoded by the constraint. We also learnt that cryptographic hash functions are difficult for constraint solvers to invert. How would you systematically construct a test suite for the above code using the idea of concolic testing?

Question 3.2 (7.5 Points)

Consider the following piece of C code, where "constant" is a 128 bit long constant that is difficult for humans to guess:

```
int obscure(int x, int y, string * message)
{
if (constant == hash(y,x,message)) {
// call to function with error
}
else {
//call to some function
}
}
```

The function hash is some standard-issue cryptographic hash function like SHA1. The above piece of code is similar to the one in Question 1.1 with some important differences. Will your testing technique that you proposed as an answer to Question 1.1 "work" in this case? If yes, why? If no, why not? (We say a testing technique "works" when you can use it to automatically test all the paths of a given program.)

Question 3.3: Programming Assignment with the KLEE (15 points)

Use KLEE to find bugs in the following Coreutils (version 6.11). Describe the bugs found and suggest patches. The programs-under-test are:

- paste

- mkdir
- md5sum

Please follow the tutorial at the following website about the KLEE symbolic virtual machine and tester: https://klee.github.io/tutorials/testing-coreutils/

Question 3.4: Symbolic Execution and Equivalence Checking (10 points)

Symbolic-execution techniques can be used to check if two pieces of code have identical 'behaviors'. This problem of determining if two pieces of code have identical behavior is called equivalence checking, and is very important in the context of formal verification of hardware and software systems. Consider that you are given two different implementation of the same Unix utility. How will you use symbolic-execution techniques to check if the different implementation have identical behaviors. If your technique detects different behaviors, does that imply that one of the them has a bug?

Question 4: Syntax of Boolean formulas (30 points)

Each sub-question is worth a maximum of 15 points. For the following, refer to the context-free grammar of the syntax of Boolean formulas as defined on slide 8 of lecture 5 (part 2).

1. Describe how one can modify the definition of the syntax of Boolean formulas to allow for balanced parentheses. I.e., we expand the definition to allow for (F) to be considered a Boolean formula, where F is an arbitrary Boolean formula. Please provide a complete definition (i.e., a complete context-free grammar) of the syntax of Boolean formulas.

2. Describe how Boolean formulas can be represented as directed acylic graphs (DAGs). Provide explicit definitions for the nodes and edges of such DAGs. Write a recursive program (pseudo-code) that takes as input DAG representation of Boolean formulas, traverses them in-order, and prints the formula.

Question 5: The Undecidability of the Halting Problem (Extra Credit: +2 to your total marks. If your marks are already at 99 or above, we will round it off to 100.)

Define the notion of undecidability and the halting problem. Then, mathematically prove that the halting problem is undecidable. (Feel free to refer to any suitable book or paper, but please do not copy-paste proofs from these resources. Try to understand the proof first and then write it in your words. Also, provide a citation to the resource you used to come up with the proof of this theorem.)