Assignment 2:
Concolic Testing, Information-Flow Analysis and Malware Detection

Assignment Overview

The learning objective of this assignment is for students to gain experience with concepts such as

- Dynamic systematic testing (aka dynamic symbolic testing or concolic testing) aimed at both 1) automatic systematic test coverage, and 2) finding deep security vulnerabilities. For convenience, I will only use the term concolic testing to refer to dynamic systematic testing and its variants
- Information flow analysis aimed at runtime monitoring of anomalous behavior or detecting leakage of private data through a program to sinks that are publicly accessible
- Automated online malware detection

To maximize your points answer all questions and their parts. Furthermore, discuss your answers in detail. The more relevant stuff you have and pithier it is, the better your score. You also get points for discussing related questions, potential research directions etc.

Question 1: Concolic Testing (60 points)

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 execution.

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

A symbolic virtual machine constructs a map from program variables to logic expressions over input variables. 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, defined below.

Path Constraint: Given a 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 a chosen path $p(\bar{x})$ in the given program; Step 2: Construct a conjunction of equalities between program variables and their final symbolic expression values. 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 is a computer program that take as input logic formulas and determines if these formulas are satisfiable (e.g., Boolean SAT and SMT solvers). We say that a formula is satisfiable, if there exists an assignment to the variables in the formula such that the formula is true. One way to think of what a constraint solver does is that it is trying to "invert" the input logic formula, i.e., find values to the variables in the input formula such that it evaluates to *true*.

0.2 Definition of Concolic Testing: 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.

- The path constraints may be too difficult for constraint solvers to solve
- The number of paths in a program is typically exponential 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 this question, 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 (other reasons for the success of symbolic techniques include better program analysis and very efficient constraint solving). 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 1.1 (30 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 1.2 (25 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?

Question 1.3 (5 Points)

Suppose you are a software developer for a company called BugfreeSoft. Unfortunately for you, a bug was discovered in your software. A bug report was filed by a user, and your manager decided that this bug (which is buffer-overflow) is very critical and needs to fixed immediately. The original C code is:

```
z = process-input();
strcpy(buf,z);
process(buf);
```

You and your team worked overnight, found a patch to fix the bug. The patch doesn't fix the actual bug itself, but provides a way of protecting the code against inputs that have a special signature that trigger the bug. The idea is that this is a quick-fix, while you search for the actual fault and fix it in the near future. The patched code is:

```
z = process-input();            //original code
if (z == SPECIAL_CONSTANT)      //patch
 reject;                        //patch
else                            //patch
 {
  strcpy(buf,z);                //original code
  process(buf);                 //original code
 }
```

The moment you released the patch, an attacker downloaded it, reverse-engineered it (The process-input() function is easy to reverse using appropriate symbolic, program analysis and constraint solving tools), and discovered a way to attack all the unpatched versions of the code. Your company's reputation is in tatters. How did the attacker succeed? In the future how will you protect against such reverse-engineering attempts?

Bonus Question 1.4: Programming Assignment with the KLEE Symbolic Virtual Machine and Tester

Please follow the tutorial at the following website about the KLEE symbolic virtual machine and tester:

http://klee.llvm.org/CoreutilsExperiments.html

Use KLEE to find bugs in the following Coreutils. Describe the bugs found and suggest patches. (Warning: This question is time consuming. So, we recommend that you try this after the mid-term.)

The programs-under-test are:

- paste
- mkdir
- md5sum

All are version 6.10.

Question 2: Implicit Informaton-flow Analysis (10 points)

We learnt in class how we can instrument a program and "watch" how the input data flow through a program from sources to sinks. This way of analyzing a program's behavior is broadly referred to as information flow analysis. An analysis technique takes as input the code of a program and optionally some inputs, and produces a report as output. This report is usually a set of facts true of the program. There are two classic ways of analyzing a program: static and dynamic. In static analysis, only the code of the program is analyzed. The report produces assertions that are true of all "behaviors" of the program-under-analysis. In dynamic analysis, the program is run on a set of inputs, and only these resultant behaviors are analyzed.

The applications of information flow analysis are many: 1) Dynamically tracking the flow of private data (e.g., passwords) in a program into public sinks (e.g., network connection), 2) statically determining how a set of program variables influence other variables for the purposes of compiler optimizations, and 3) whole system dynamic flow analysis. The best example of this last application is TaintDroid, a full system taint (lightweight information flow) analysis for the Android operating system.

There are two main kinds of information flow that researchers have studied, namely, explicit and implicit information flow. We say there is an implicit flow from a secret input data to a public output, if information

about the secret can flow through the control-flow of the program when computation branches on secret data and performs publicly observed side effects depending on which branch is taken. This is in opposition to explicit flows, where secret data is directly passed to public outputs.

In this question, you will be tested on your understanding of a particularly knotty problem in information flow analysis, called implicit information flow. Consider the following piece of code, we will call code-1:

```
function(secret) {

 if (secret)
   public = 1
 else
   public = 0

}
```

In this code the variable "secret" refers to secret data, and the variable "public" refers to a sink observable by the attacker. Assume that the attacker has access to the code above. Furthermore, the attacker may cause this code to run as many times as she wishes by feeding inputs (e.g., a login program that uses a hash function. The attacker is allowed unlimited login attempts).

It is clear that the above program has an implicit information flow from the variable "secret" to the variable "public". If the public variable takes value 1, then the attacker knows that she has the secret.

On the other hand, consider the following code fragment in a C-like language where are the variables are of type int, we will call code-2:

```
function(int secret) {

  l = 0;
  n = 32;
  while (n >= 0) {
    k = 2^(n-1);  //2 to the power of (n-1)
    h = secret >= k;
    if(h)
      public = 1;
    else
      public = 0;

    if(public) {
      secret = secret - k;
      l = l + k;
    }
    else
      skip;
    n = n-1;
  } //end of while
} //end of function
```

Question 2.1 (5 points)

Does code-1 leak information about the secret? If so, can you characterize in terms of number of bits, how many bits are leaked? Same question for code-2? What does it mean for information leakage to be measured in number of bits?

Question 2.2 (5 points)

Is there a way to detect the leakage of secrets in code fragments such as the ones above? What is a first-order issue or problem that you must address with your information-flow detection technique?

Question 3: The Trouble with Malware Detection Techniques (30 points)

Malware is proliferating all around us. For example, attackers can upload Apps to the Android or iPhone ecosystems, that look legit but can do enormous harm to users. The problem of automatically detecting whether an App is malware is in general undecidable.

Some Definitions: The term compile-time is loosely defined term that refers to static, dynamic or machine learning techniques applied to a piece of code to learn its behavior. By contrast, run-time monitoring refers to analyzing the behavior of the code when it is running in the wild.

There are many challenges here that need to be addressed: 1) Who defines a malware? 2) Can we detect malware just by analyzing its code at compile-time (by using both static analysis, machine learning and dynamic analysis by running code on some sample inputs), or do we monitor the behavior dynamically, check against a security policy and report violations? 3) How do we ensure that such analysis techniques scale and are correct, i.e., don't report false positive or false negatives? 4) What happens when the analysis reports false positives, i.e., marks a perfectly good app as malware? 4) Worse, what happens when the analysis reports a false negative, i.e., reports a malware as a perfectly good app?

All these are very difficult research questions that are in the process of being (partially) answered. Attackers are aware that compile-time analysis for malware detection is very difficult in practice. Consequently, attackers actively use obfuscation techniques, encryption/decryption pairs, and packing/unpacking to construct malware that is very difficult to detect. They also using poly and metamorphisms to beat signature-based malware detection techniques.

Not all obfuscation techniques are effective in hiding program behavior. Simple obfuscations such as changing variable names, or even introducing digressive information flows are very easily detected and unmasked by automatic static and dynamic analysis techniques. In a sense, there is a cat and mouse game going on between malware writers and people who build malware detectors. As program analysis techniques become more and more sophisticated, simple obfuscation is not going to work anymore. For obfuscations to be truly effective, we believe that they must be based on cryptographic primitives.

To illustrate this, consider the following piece of malware code:

```
if(X == 'c')
{
  Execute-malware();
}
```

Question 3.1 (20 points)

How will a malware writer morph the above code so that static analysis is rendered useless?

Question 3.2 (10 points

Even though static analysis may fail to correctly detect the code written by the malware writer from Question 3.1, the defender (the guy who writes the malware detector) can still win by using a general strategy over-and-above the static analysis. What is one such strategy you can use to conservatively detect malware, at the possible cost of higher false positives?