ECE750-28:
Computer-aided Reasoning for Software Engineering
Assignment #2

Due November 24, 2013

November 10, 2013

Instructions

Please answer all problems. The maximum possible score for this homework is 100. 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.
Please write your name, ID on the frontpage. Ideally, number each page as "page number/total number of pages" (e.g., 1/6).

Problem 1: Proving NP-completeness (15 Points)

In class we discussed the definition of NP-completeness. Briefly, a computational problem is NP-complete if it satisfies the following two properties:

- The problem is in NP.
- The problem is NP-hard, i.e., there is a reduction from a known NP-hard problem to the problem-in-question.

The Questions

1. **The NP-completeness Proof (10 Points)**:

   Show that the satisfiability problem for a quantifier-free theory of bit-vectors with the following signature is NP-complete (please refer description of the theory of bit-vectors in the class notes or the appropriate homework problem from assignment #1):

   (a) Constants: Finite-length strings over the alphabet $\{0, 1\}$

   (b) Variables: denoted by letters $x, y, ...$, where each variable has a pre-specified fixed length. Assume that the lengths of all variables considered are upper-bounded by some fixed natural number C and lower-bounded by 0.

   (c) Functions: We have the addition symbol (+) defined as usual, concatenation (.) and extraction ([i:j])

   (d) Predicates: Equality over bit-vector terms and bit-vector inequality (<)

   (e) Formulas are constructed as Boolean combination of atomic formulas as usual.

   The well-formed formulas are defined in the usual way using structural induction. The semantics of the functions and predicates were defined in homework assignment #1.

2. **Upper-bound on Length (5 Points)**:

   Explain why we need the upper-bound restriction on the length of bit-vectors in order to prove that the quantifier-free theory of bit-vectors is NP-complete?

Problem 2: Symbolic-execution based Testing (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.)

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 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 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 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*.

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.

The Questions

Question 2.1 (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 2.2 (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? (A testing technique 'works' when you can use it to automatically test all the paths of a given program.)

Question 2.3: Programming Assignment with the KLEE (25 points)

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 at least two of the following Coreutils. Describe the bugs found and suggest patches. (Warning: This question is time consuming.)
The programs-under-test are:

- paste
- mkdir
- md5sum

All are version 6.10.

Question 2.4: Symbolic Execution and Equivalence Checking (5 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 formal verification of hardware and software. 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?

Problem 3: Conflict-driven Clause-learning (CDCL) SAT Solvers (40 points)

Question 3.1: Explaining the Power of CDCL SAT Solvers (10 points)

In class we studied in detail how the conflict-driven clause learning (CDCL) SAT solvers work. A key step in how these solvers work is the construction of an implication graph during the Boolean Constant Propagation (BCP) step, followed by the analysis of this graph during the conflict analysis phase. Describe in your own words, with the help of a concrete example, how the BCP phase constructs an implication graph, and detects conflicts. Also, explain how the analysis phase constructs a conflict, in particular focussing on the First Unique Implicant Point (UIP) heuristic. Keep your answer pithy.

Question 3.2: A Solver for 2-CNF (10 points)

We say that a Boolean logic formula is a k-CNF (Conjunctive Normal Form) formula if all its clauses have atmost k literals in them. The satisfiability problem corresponding to all such k-CNF formulas is simply called the k-CNF SAT problem. Prove that any Boolean formula can be reduced to a logically equivalent 3-CNF formula. Furthermore, show that there exists a polynomial time algorithm for the 2-CNF SAT problem, that takes time polynomial in the number of variables in its input formula.

Question 3.2: Solving XOR clauses (20 points)

While conflict analysis, VSIDS heuristic and backjumping are crucial to the performance of CDCL SAT solvers, specialized heuristics play a significant role as well. For example, if a literal occurs only postively (resp. only negatively) then you can always set it true (resp. always false). This rule is called the pure literal rule. There are many other such heuristics or rules.
In this question, you will explore a heuristic for solving what are called as XOR clauses. An XOR clause is just your typical clause with one crucial difference: instead of the clause being a disjunction of literals it is the XOR ($\oplus$) of literals. For example, the following is an XOR clause:

$$(\neg x_1 \oplus y_2 \oplus z_4)$$

Assuming that your input formula consists of only XOR clauses, can you describe an efficient algorithm to solve such clauses?

Problem 4: Using Symbolic-Execution Techniques for Fault Localization (Bonus Research Question)

You won't be graded on this problem. However, you will certainly learn something if you attempt it.
As we have already studied in class, symbolic-execution based techniques are widely used to automatically test software. A closely related problem is that of fault localization. Given, a program P and test input that reveals an error in P, the problem is to isolate the root cause of the error in P automatically and as precisely as possible. Describe a technique that would use some kind of solver (you can use either a constraint solver or an optimization solver) and/or symbolic execution technique to localize fault in your favorite type of software.