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

Question 1: Short answer questions (20 points)

Please provide pithy and correct answers to the following questions. 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;
}
```

Answer:

Context: Static single assignment (SSA) is a property of intermediate language (IL) programs produced by modern compiler frontends. Put differently, modern compiler frontends (e.g., the frontend of gcc) typically guarantee that their output is in SSA. This property enables or makes it easier for compiler backends to perform sophisticated program analysis on IL programs produced by frontends.

Definition: We say a program is in SSA if every variable in it is defined (i.e., assigned a value) exactly once and each variable in it is defined prior to use (i.e., the variable is read from only after it has been defined).

1. Code fragment 1 is not in SSA for two reasons:
   First, variable x is used before definition in the first line of code (i.e., in the line x = x+y, the variable x is used in the expression x+y before being written into x).
   Second, the variable x is defined twice (in lines x=x+y and x=z). Below find the equivalent code fragment in SSA format:

```
Code fragment-1 converted into SSA:
x1 = x + y;
z = y + t;
x2 = z;
```

2. Code fragment 2 is also not in SSA for the following reasons:
   First, the variable x is defined twice (in lines x = 42 and x = y-43).
   Second, y is being defined twice in the body of the if and else branches. They should each be given new names distinct from each other and from y. Below find the equivalent code fragment in SSA:

```
Code fragment-2 converted into SSA:
x1 = 42;
x2 = y - 23;
if(x2 <= y) {
y1 =  z + x2;
}
else {
y2 =  x1;
}
```

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 or assembly language).

   Answer:

   Context: Modern compilers typically transform programs into control-flow graphs (CFGs) in memory before performing a variety of program analyses on them. CFGs are important data structures in many other contexts other than program analysis, e.g., program understanding, debugging tools etc.

   Definition: Control-flow graphs are directed graph structures whose nodes are basic blocks and edges from a node A to node B denote a control flow or jump from A to B. Basic blocks are straight-line fragments of code without any jumps (except possibly at the end of a basic block) or jump targets (except possibly at the very beginning of a basic block).

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.

   Answer:

   Context: In the context of program analysis for finding errors, the input to a program analysis tool T is a program P and an implicit or explicit specification S, and the output is a list of possible errors or violations in P with respect to (w.r.t) S. We say a specification S is implicit if the program analysis tool comes pre-programmed with S. For example, typically many program analysis tools come pre-programmed with a list of typical desirable properties of programs, e.g., program must not crash. (One could definte the concept of an error or violation in P w.r.t S very formally and precisely. I choose to leave it informal here.)

   Observation: Observe that one could easily redefine program analysis tools such that they accept a specification that captures a set of undesirable properites, e.g., find inputs that cause the input program to crash. For practical purposes, either definition works well. While using or building a program analysis tool, you simply have to bear in mind the definition you are using and consistenly adhere to it.

   Definitions: The following definitions are the same irrespective of whether the analysis method is static, dynamic, symbolic, or some combination thereof.

1. True Positive: Say a program analysis tool T reports an error in input program P w.r.t specification S. We say that such a report is a true positive, if there exists an input J to the program-under-test P such that when P is executed on J, the resulting run or behavior of P violates S. (Note that the program analysis tools are not necessarily required to produce such an input J.)

2. False Positive: Say a program analysis tool T reports an error in input program-under-test P w.r.t specification S. We say that such a report is a false positive, if there does NOT exist any input J to the program-under-test P such that when P is executed on J, the resulting run or behavior of P violates S. Put differently, the program analysis tool's report is not correct.

3. True Negative: Assume there are NO inputs J that expose violations in the program-under-test P w.r.t specification S. Further assume that the program analysis tool T reports that indeed there are no such inputs J, i.e., P does not violate S. We would then deem such a report by T as a true negative.

4. False Negative: Assume that there exists an input J such that when P is executed on J, the resulting run or behavior of P violates S. However, the program analysis tool T reports that no such violations exist. In such a scenario, we refer to the report produced by T as a false negative.

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.

   Answer:

   Context: Let us first start by properly and formally defining the notion of a computational problem, specifically decision problem. (Observe that in computer science we typically have 3 kinds of problems, namely, decision, optimization, and function problems. It turns out that a problem in one category can be suitably **reduced** to a problem in the other. Hence, for our purposes, it is sufficient to focus on only one category. We focus only on decision problems, unless specifically stated otherwise.)

   A computation problem is best defined in terms of a formal language, i.e., as a (potentially infinite) set of finite-length strings over some alphabet Σ. Then, a decision problem for a formal language L is the following: given a string or instance x over Σ, decide whether $x \in L$.

   For example, the set in question for the Boolean satisfiability problem is the set SAT of all Boolean formulas in CNF that are satisfiable. Note that CNF formulas are finite-length strings over an alphabet of Boolean variables and operators. The definition of the Boolean satisfiability decision problem is the following: given an arbitrary Boolean formula f in CNF, decide whether f is in the set SAT. An arbitrary CNF formula is an instance of the Boolean satisfiability problem.

   We can similarly define the halting problem. We first define the set HALT, that is all pairs $\langle T, m \rangle$ of Turing machines T and inputs m such that T halts on m. The definition of the halting problem is as follows: given an arbitrary pair $\langle T, m \rangle$ of T and input m to T, decide whether $\langle T, m \rangle$ is in HALT. An arbitrary pair $\langle T, m \rangle$ is an instance of the halting problem. Note that the halting problem is undecidable.

   A closely related problem is the $HALT_{ALL}$ problem, where one has to come up with an algorithm to decide whether a TM halts on all its inputs. This problem is also undecidable.

   Definition of Undecidability: We say a decision problem D is undecidable, if there does not exist any algorithm A such that given any arbitrary instance x of the problem D (and appropriately defined

set D), the algorithm A correctly decides whether or not $x \in D$. The notion of undecidability does not exclude the possibility that an algorithm aimed at solving the problem D may sometimes correctly decide the question of $x \in D$. However, if D is an undecidable problem, then no single algorithm A can solve every instance of D correctly and terminate.

Definition of Decidability: We say a decision problem D is decidable, if there exists an algorithm A such that given any arbitrary instance x of the problem D (and appropriately defined set D), the algorithm A correctly decides whether or not $x \in D$ and terminates in finite time.

Undecidability and Program Analysis: Here we informally discuss why static analysis in general is undecidable (Rice's Theorem).

It turns out that one can easily reduce the $HALT_{ALL}$ problem to many static analysis problems (such reductions are called Turing reductions). Turing reductions are the key to establishing undecidability of many static analysis problems (e.g., pointer alias analysis, static information flow etc.). The intuition behind why static analysis problems (such as pointer alias analysis) are undecidable is because one has to often decide whether or not a program halts in order to decide whether a program has a certain property (e.g., two pointers in a program alias). If static analysis in general were decidable, one could argue that the $HALT_{ALL}$ problem is also decidable, which leads to a contradiction with the known theorem that states that $HALT_{ALL}$ is undecidable.

Note that the above argument is informal and intuitive. Making a formal proof out of it would require a fair amount of technical machinery.

5. 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 determines validity by systematically checking the satisfiability of a given Boolean formula over every possible assignment (i.e., testing the formula against all possible models in order to decide validity). 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.

   Answer:

   When applied to valid formulas from logics or theories over infinite domains, model-theoretic methods for checking validity may not terminate in finite time for certain kinds of (universally quantified) formulas. The reason is that these methods establish validity by checking the satisfiability of an input formula F over every possible assignment of the infinitely-many values to F's variables. Such a process might not terminate for certain kinds of formulas over the natural numbers, e.g., $\forall x \exists y.(y > x)$.

   By contrast, complete proof systems exist for many logics and theories defined over infinite domains. For example, it is known that linear arithmetic over the natural numbers has a sound and complete proof system. In other words, as opposed to model-theoretic approaches, such proof systems (when algorithmized as deterministic solvers) are guaranteed to produce a proof and terminate if the input is indeed valid.

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

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

   Answer:

   For contradiction, assume $\exists I.I \not\models F$

1. $I \not\models ((p \to q) \wedge (q \leftrightarrow r)) \to (p \to r)$	by assumption
2. $I \models ((p \to q) \wedge (q \leftrightarrow r))$	by 1, Rule ($I \not\models F \to G$)
3. $I \not\models (p \to r)$	by 1, Rule ($I \not\models F \to G$)
4. $I \models (p \to q)$	by 2, Rule ($I \models F \wedge G$)
5. $I \models (q \leftrightarrow r)$	by 2, Rule ($I \models F \wedge G$)
6. $I \models p$	by 3, Rule ($I \not\models F \to G$)
7. $I \not\models r$	by 3, Rule ($I \not\models F \to G$)
8. $I \not\models p \mid I \models q$	by 4, Rule ($I \models F \to G$), Created branch 8a (on left side of symbol $\mid$), and branch 8b (on the right side of symbol $\mid$).
9. $\bot$	by 6, 8a, contradiction rule, Branch 8a closed.
10. $I \models q \wedge r \mid I \models \neg q \wedge \neg r$	by 5, Rule $I \models F \leftrightarrow G$, Branches 8b.1, 8b.2 created.
11. $I \models r$	by 8b.1, Rule $I \models F \wedge G$
12. $\bot$	by 7, 11, contradiction rule, Branch 8b.1 closed
13. $I \models \neg q$	by 8b.2, Rule $I \models F \wedge G$
14. $I \not\models q$	by 13, Rule $I \models \neg F$
15. $\bot$	by 8b, 14, contradiction rule, Branch 8b.2 closed

7. Can all 2-input 1-output Boolean functions or circuits be encoded using only 2-input 1-output XOR gates? (For simplicity assume all gates considered in this question are 2-input 1-output gates.)

   Answer:

   No. XOR gates do not constitute a functionally complete set of gates for 2-input 1-output Boolean functions represented as circuits. We say that a set S of gates is functionally complete, if every 2-input 1-output Boolean function or circuit can be encoded using only the elements of S. Examples include the NAND gate or the set consisting of the NOT and OR gates.

   Proving that XOR does not constitute a functionally complete set is tantamount to proving a negative. This requires one to show that no matter how complex your circuit of n+1 XOR gates may be, it cannot simulate certain 2-input 1-output Boolean functions.

   (Below we may choose to drop the adjective "2-input 1-output", if it is clear from context.)

   Base case: First, consider circuits consisting only of k XOR gates, up to some small constant k. Observe that no such circuit is functionally complete for the set of 2-input 1-output Boolean functions. The reason is that for all possible 2-input values $\{(0,0),(0,1),(1,0),(1,1)\}$, the set of outputs of such circuits is guaranteed to have an even number of 0's. (E.g., observe that a single XOR gate's outputs for the above inputs is $\{0,1,1,0\}$). This prevents such circuits from simulating AND, OR, NOR, NAND, IMPLIES, as well as any functionally complete set of gates.

   Caveat: Note here that a single XOR gate can simulate the NOT function (by fixing one input to 1) or the identity function (by fixing one input to 0), and that you are allowed to build NOT or identity gates (or even fix some inputs to 1 or 0) in your circuit of size k (or less). At first glance it might seem that these simulations violate the observation that the set of outputs must have an even number of 0's. However, there is no violation since the entire circuit has to take two inputs and produce one output.

IH: The inductive hypothesis states that all circuits consisting only of n (or less) 2-input 1-output XOR gates, aimed at simulating all Boolean functions, have the property that the set of their outputs may only have even number of 0's. Consequently, they may not simulate AND, OR, NOR, NAND, IMPLIES, as well as any functionally complete set of gates.

Inductive Step: Now consider any circuit C with n+1 XOR gates. We argue that no such circuit can be functionally complete. To see this, first note that any such circuit has to have a final XOR gate G, which may have sub-circuits C1 and C2 of up to n gates. By IH, each of C1 and C2 has even number of 0's in their output set. For the cases where C1 or C2 output 0,0,0,0 or 1,1,1,1, it is trivial to see that C is not functionally complete. The only interesting cases are when C1 or C2 each produce exactly two 0's in their output set. There are only finitely many such combinations of output sets of C1 and C2, and for each such combination G ensures that its output set also contains an even number of 0's.

Hence, no circuit C with n+1 XOR gate is functionally complete for 2-input 1-output Boolean functions.

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

   Answer:

   This is very easy to establish via the truth-table method.

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

   Answer:

   Definitions: We say two formulas ϕ and ψ are equisatisfiable if the following is true:

   ϕ is satisfiable if and only if ψ is.

   This definition only requires that both ϕ and ψ be satisfiable, but not necessarily on the same assignment. In fact, they may have completely different sets of variables.

   By contrast, we say two formulas ϕ and ψ are logically equivalent if the following is true:

   $\phi \implies \psi$ and $\psi \implies \phi$.

   The concept of logical equivalence requires that both formulas be satisfiable on exactly the same set of assignment.

   The concept is equisatisfiability is strictly weaker than logical equivalence. That is, if two formulas are logically equivalent, then they are certainly equisatisfiable. However, the converse is not true.

   For example, consider the formulas x and T. They 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.

(Note that we plan to be quite generous with marking this question since your answer may contain minimal detail to very extensive information about how AFL actually works. Below we provide the minimal answer to get full marks.)

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

Answer:

Briefly describe AFL: American Fuzzy Lop (AFL) is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. Note that the AFL comes with a compiler that instruments the program-under-test to observe whether the program's *state has transitioned* from execution of one test input to the next. At a high-level, AFL works as follows:

1. Load user-supplied initial test cases into a queue
2. Take next input file from the queue
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue
6. Go to step (2)

Experimental setup and results of testing MiniSAT using AFL:

1. Experiment Steps:
2. Clone: git clone git@github.com:google/AFL.git; git clone git@github.com:niklasso/minisat.git.
3. Compiling AFL was easy as just running the "make" command. Compile (MiniSat): sudo CC=/path/to/AFL/afl-gcc CXX=/path/to/AFL/afl-g++ make clean all
4. Fuzz: afl-fuzz -i testcase -o out /path/to/minisat/build/release/bin/minisat @@
5. Results: No bugs found. 5 hrs running. Cycles done: 3. Total paths: 2342.

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 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 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 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 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?

Answer:

For the given program, a naive symbolic execution approach is unlikely to feasibly generate test inputs that exercise the body of the if-conditional. The reason is that the path constraints for that part of the code contains a cryptographic hash function, known to be hard for SAT and SMT solvers to invert. On the other hand, if we concretize the inputs x and y, we can easily generate tests to exercise both the if- and else-branches of the given code.

Specifically, perform the following steps to automatically test the given code:

1. Randomly choose a value for y

2. Compute hash(y) by executing the hash function as a separate process

3. Set the value of the input x to be hash(y). The pair (x,y) of inputs thus obtained is guaranteed to exercise the body of the if-branch in the above code

4. Set the value of the input x to be different from hash(y). The pair of inputs thus obtained is guaranteed to exercise the body of the else-branch in the above code

The above method where the user carefully chooses (either manually or automatically) concrete values for some inputs and then performs symbolic execution on the path thus chosen is typically referred to as concolic execution. The value addition of such an approach is that it avoids some of the pitfalls of naive symbolic execution.

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 3.1 with some important differences. Does your testing technique that you proposed as an answer to Question 3.1 "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.)

Answer:

The method described for the previous sub-question is unlikely to perform well for the above code. The reason is that discovering an input pair (y,x) that hashes to a specific target constant is hard in general, since the hash function used in the above code is cryptographic. For the previous sub-question, we avoided this problem of inverting a hash function or attempting randomly discovering the value of the inputs to the hash function since both sides of the equality used in the if-conditional (in Question 3.1) are variables. Unfortunately, the if-conditional in Question 3.2 uses a hash target (namely, constant) that is hard to guess. Hence, program analysis methods are unlikely to succeed in automatically generating tests that would exercise the body of the if-branch. There are no good options here and random fuzzing is as good as any.

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/

Answer:

1. What kind of bug it is?
2. What line number and other details of where the bug is in the program.
3. What is the nature of the input that exposed the bug?

Paste: 1 hour 10 bugs, all of them are memory error.

```
Error: memory error: out of bound pointer
File: ../../src/paste.c
Line: 90
assembly.ll line: 4724
Stack:
#000004724 in collapse_escapes (strptr) at ../../src/paste.c:90
#100004541 in __klee_posix_wrapped_main (argc=4, argv=94304771722384) at ../../src/paste.c:484
#200002797 in __user_main (=15, =94304759901472, =94304759901600) at POSIX/klee_init_env.c:245
#300000551 in __uClibc_main (=15, =94304759901472) at libc/misc/internals/__uClibc_main.c:401
#400000717 in main (=15, =94304759901472)
Info:
address: 94304765556331
next: object at 22580131572672 of size 1536 MO30[1536] (no allocation info)
```

These memory errors are caused wrong arguments. A case that triggers the error (use klee-reply with the ktest file to find such a case):

```
"./paste" "-d\"
```

Mkdir: 1 hour 1 bug, memory error

```
Error: memory error: out of bound pointer
File: ../../lib/quotearg.c
Line: 248
assembly.ll line: 6209
Stack:
#000006209 in quotearg_buffer_restyled (buffer=93975574450432, ...) at ../../lib/quotearg.c:248
#100006076 in quotearg_buffer (buffer=93975574450432, ...) at ../../lib/quotearg.c:526
#200006034 in quotearg_n_options (arg=0, options=93975611678256) at ../../lib/quotearg.c:624
#300006005 in quotearg_n_style (arg=0) at ../../lib/quotearg.c:666
#400005992 in quote_n (name=0) at ../../lib/quote.c:31
#500004664 in quote (name=0) at ../../lib/quote.c:39
#600004538 in __klee_posix_wrapped_main (argc=4, argv=93975610514608) at ../../src/mkdir.c:194
#700002768 in __user_main (=15, =93975597420768, =93975597420896) at POSIX/klee_init_env.c:245
#800000569 in __uClibc_main (=15, =93975597420768) at libc/misc/internals/__uClibc_main.c:401
#900000735 in main (=15, =93975597420768)
Info:
address: 0
next: object at 22550463605696 of size 1536 MO30[1536] (no allocation info)
```

These memory errors are caused by wrong arguments. A case that triggers the error (use klee-reply with the ktest file to find such a case):

```
"./mkdir" "--cont" "" ""
```

Md5sum: 1 hour 1 bug, memory error

```
Error: memory error: out of bound pointer
File: ../../src/md5sum.c
Line: 213
assembly.ll line: 6858
Stack:
#000006858 in bsd_split_3 (s=94672463409352, ...) at ../../src/md5sum.c:213
#100006662 in split_3 (s=94672463409344, ...) at ../../src/md5sum.c:259
#200004971 in digest_check (checkfile_name=94672459392760) at ../../src/md5sum.c:471
#300004768 in __klee_posix_wrapped_main (argc=2, argv=94672460441216) at ../../src/md5sum.c:667
#400002826 in __user_main (=15, =94672445738752, =94672445738880) at POSIX/klee_init_env.c:245
#500000580 in __uClibc_main (=15, =94672445738752) at libc/misc/internals/__uClibc_main.c:401
#600000746 in main (=15, =94672445738752)
Info:
address: 94672463409343
next: object at 23152911350720 of size 1536 MO30[1536] (no allocation info)
```

These memory errors are caused by wrong arguments. A case that triggers the error (use klee-reply with the ktest file to find such a case):

```
"./md5sum" "-c"
```

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 would 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?

Answer:

Under certain conditions and assumptions, symbolic execution methods can be **feasibly** used to check whether or not two pieces of code have identical behaviors. These conditions include the programs have only finitely many iterations per loop. Under these assumptions, it is possible to convert a piece of code, via symbolic execution, into a Boolean formula. For example, given two separate implementations $P1$ and $P2$ of the Unix utility Paste, one can transform them into Boolean formulas ϕ_{P1} and ϕ_{P2} over the same input and output variables, and check their equivalence using a SAT solvers.

While the above-mentioned method works well for medium-sized programs with finite loops and very simple control-flow graphs, using symbolic execution for equivalence checking for large and complex programs is still in its infancy. Hence, developers of symbolic tools often focus on bug-finding for common classes of errors in programs. This is a much more practical application of symbolic execution.

Further, just because a symbolic execution tool found an input that demonstrates an inequivalence between two implementations of the same specification, it does not necessarily mean that there is an error in at least one of them. The inequivalence may be due to the abstractions used by the symbolic execution tool that impacts the distinct implementations $P1$ and $P2$ differently.

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.

   Answer:
   In order to expand the definition as required, one can simply add an inductive rule for formulas with parentheses (i.e., (F)) to the context-free grammar of Boolean formulas given in the lecture notes. No other modifications are needed. The expanded context-free grammar or definition is:

   $F :\bot$
   $\top$
   $x \in Vars_{Bool}$
   $\neg F$
   $F_1 \wedge F_2$
   $F_1 \vee F_2$
   $F_1 \leftarrow F_2$
   $F_1 \leftrightarrow F_2$
   (F)

   (If you gave the above answer, you are likely to get full points. Below find extra info that you may find interesting.)

   To prove that the above modification is both necessary and sufficient to obtain well-formed formulas with balanced parens, one has to establish the following additional theorems. Let us denote the original context-free grammar (CFG) for well-formed Boolean formulas as G. Let G' denote the modified CFG given above.

   Our goal is to prove that the modified grammar G' maintains the *well-formedness* property guaranteed by G. We provide a sketch of this argument below.

   We first state some assumptions and definitions.

   Definition: We say that a Boolean formula is well-formed if it has the following properties:

   1. Boolean connective $\neg$ always appears as a prefix to a Boolean formula
   2. Binary Boolean connectives such as $\wedge$, $\vee$ etc. always appear as infix to (i.e., in between) two Boolean formulas

   This definition ensures that "poorly-constructed" strings (e.g., $\neg \vee x$ or $\wedge ab$) are not accepted as Boolean formulas. It is easy to see that the original grammar G ensures that all strings it generates/defines/accepts are well-formed Boolean formulas.

   Now we have to show that the modified grammar G' ensures two properties for the formulas it generates/accepts/defines, namely, that all formulas have balanced parens and that all formulas are well-formed.

   Lemma 1: Every formula F according to G' has balanced parens.

To show this, we have to prove the following Lemma: every prefix of any formula F has at least as many left parens as right parens. This theorem can be easily shown using induction over the length of F.

Lemma 2: Every formula F according to G' is a well-formed formula.

Proving this property is a bit tricky because we first have to modify the above-mentioned definition of well-formedness to account for balanced parens. We modify the definition as follows: a left paren is always a prefix to a Boolean formula and a right paren is always a suffix to a Boolean formula. This definition ensures that badly constructed strings such as $(a\wedge)b$ or $()a \wedge b$ are NOT well-formed formulas. What remains to be show is that all such poorly-constructed strings are rejected by the modified definition G').

Once again we rely on induction on the length of formulas. Having properly modified the notion of well-formedness, we can now show that all formulas defined by G' satisfy the well-formedness property defined above via induction on the length of formulas accepted/defined/generated by G'.

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.

   Answer:
   A Boolean formula F can be represented as a DAG G as follows:

   1. Level-1 leaf nodes in G (i.e., nodes with at most two immediate children) correspond to variables or constants, while internal nodes (i.e., nodes with at least one immeditae child) represent Boolean connectives.

   2. Left and right parentheses (parens) introduce a bit of complication. The best way to handle them is as a special type of nodes (i.e., with at most one immediate child), we call level-0 leaf nodes, wherein, the left paren (resp. right paren) always appears as a left child (resp. right child) of the node of the character immediately to its right (resp. left).

   3. There is a directed edge from a node A to node B in G, if A corresponds to a Boolean connective and B is its operand. Similarly, there may be a directed edge from level-1 or level-0 leaf nodes to level-0 leaf nodes.

   Below is a high-level description of the pseudo-code of representing a DAG, an in-order traversal function, and a print function. (Note that your code doesn't have to be perfect in order for you to get full marks.)

```
enum gate{AND,OR,NEG,IMPLIES,IFF,TRUE,FALSE,VAR,LEFT_PAREN,RIGHT_PAREN};
class node {
        enum gate type; //type of internal node
        int value; //Variable name
        node  *left, right;
};

class graph {
        node * root;
        vector<node*> my_graph;
        int num_of_edges;
```

```
        int num_of_nodes;

public:
        creategraph();
        addEdge(node * start, node * end);
        addNode(string operator, list<node*> adj_nodes);
        print();
};

print(node *) {

if (root == NULL) return;
switch(root->type) {
        case TRUE:
        case FALSE:
                if (root->left != NULL)
                print(root->left);
                print  type; //print T or F
                if (root->right != NULL)
                        print(root->right);

        case VAR:
                if (root->left != NULL)
                        print(root->left);
                print root->value; //print var name
                if (root->right != NULL)
                        print(root->right);

        case LEFT_PAREN:
                print  type; //print '('
                if (root->left != NULL)
                        print(root->left);

        case RIGHT_PAREN:
                if (root->right != NULL)
                        print(root->right);
                print  type; //print ')'

        case NEG:
                print type; //print ¬
                print(root->left);

        case AND:
        case OR:
        case IMPLIES:
        case IFF:
                print (root->left);
                print type; //print the appropriate type (e.g., ∧)
                print (root->right);
        } //end of switch
} //end of print
```

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.)

Answer:

Definition of Undecidability: We say a decision problem D is undecidable, if there does not exist any algorithm A such that given any arbitrary instance x of the problem D (and appropriately defined set D), the algorithm A correctly decides whether or not $x \in D$. The notion of undecidability does not exclude the possibility that an algorithm aimed at solving the problem D may sometimes correctly decide the question of $x \in D$. However, if D is an undecidable problem, then no single algorithm A can solve every instance of D correctly and terminate.

Definition of Decidability: We say a decision problem D is decidable, if there exists an algorithm A such that given any arbitrary instance x of the problem D (and appropriately defined set D), the algorithm A correctly decides whether or not $x \in D$ and terminates in finite time.

The Halting Problem: We first define the set or language HALT, that is all pairs $\langle T, m \rangle$ of Turing machines T and inputs m such that T halts on m. The definition of the halting problem is as follows: given an arbitrary pair $\langle T, m \rangle$ of T and input m to T, decide whether $\langle T, m \rangle$ is in HALT. An arbitrary pair $\langle T, m \rangle$ is an instance of the halting problem.

Theorem: The Halting problem is undecidable.

Proof: (Source: Michael Sipser book. I have simplified the proof considerably.)

The following is a proof by contradiction. That is, we assume that the halting problem is decidable and then derive a contradiction from that assumption.

We first define the language HALT as follows:

$$\text{HALT} = \{\langle T, m \rangle \mid \text{T is a Turing machine (TM) and T accepts the input string m}\}$$

(Note that TMs can be described as finite-length strings (i.e., the code of any algorithm is simply a finite-length string) and further that the input m to a TM is always a finite-length string.)

From the assumption that the Halting problem is decidable, we can conclude that there exists a TM H that decides the language HALT. That is, H is a TM that takes as input the tuple $\langle T, m \rangle$ and returns accept/reject as follows:

$$H(\langle T, m \rangle) \text{returns accept if T accepts and halts on m}$$

and

$$H(\langle T, m \rangle) \text{returns reject if T does not accept m}$$

Using the code of H, we can construct a new TM D that uses H as a subroutine as shown below. What is very interesting about D is that it takes only one input, namely, the description of a Turing machine, say, T and calls the TM H (the purported decision procedure for the Halting problem) on the input $\langle T, \langle T \rangle \rangle$. Going one step further, D returns reject if T accepts its own description, and D accepts if T does not accept. The pseudo-code of the TM D is as follows:

```
D (input is a description ⟨T⟩ of a TM T) {
        Call H on the input ⟨T,⟨T⟩⟩ and record the result

        //Output the opposite of what H outputs
        if (H accepts) {
            return reject
        }
        else {
            return accept
        }
}
```

We have now constructed a TM D that accepts exactly when the TM T does not accept its own description $\langle T \rangle$ and rejects precisely when T accepts its own description $\langle T \rangle$. Further consider what happens when we run D with its own description $\langle D \rangle$. We get the following:

D accepts exactly when D does not accept $\langle D \rangle$ and

D rejects exactly when D accepts $\langle D \rangle$

This is a contradiction. The only assumption we made to derive this contradiction is that the TM H exists or that the Halting problem is decidable. Hence, we can conclude the opposite, i.e., that the Halting problem is undecidable.