Question 1: Short answer questions

1. **Provide a formal definition of Static Single Assignment (SSA) form** : **SSA** is a way of structuring the intermediate representation so that every variable is assigned exactly once. These newly assigned variables will be used in place of old variable until a new assignment. SSA technique is used in symbolic analysis of program.

   code fragment 1: not in SSA as the new assignment of x has the same variable name as the old variable. Transforming to SSA :

   ```
   x1 = x+ y;
   z1 = y + t;
   x2 = z1;
   ```

   code fragment 2: not in SSA as the new assignment of x,y has the same variable name as the old variable. Transforming to SSA :

   ```
   x1 = 42;
   x2 = y - 23;
   if(x2 <= y) {
       y1 = z + x2;
   }
   else {
       y2 = x2;
   }
   ```

2. **Provide a formal definition of the notion of control-flow graphs of programs. :-** The control-flow graph (CFG) of a program is a representation, using graph notation, of all paths that might be traversed through that program during its execution.
   In a control-flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. Jumps are nothing but conditions like if or any loop structures. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

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**. :-

   false positive: is an issue/bug raised that doesn't actually exist in the code. It doesn't need to be fixed. This happens when no bug exists in a line of code, but a probable error is given by tools for that line of code. The lesser the false positive rate, the better is usability of that analysis tool.
   true positive: is an issue/bug raised that actually exist in the code. i.e. the predicted bug by the tool is an actual bug in the code. High true positive rate is ideal.
   false negative: A false negative is an issue/bug that goes undetected. This happens when a bug/ violation exists, but no diagnostic is created.
   true negative: true negative indicates that the tool did not detect any errors, when there are actually no errors present in the code.

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.**
   According to tuning the notion of any computation can be captured using tuning machine. One of the first problems that he proposed is the undecidability. It is noting but,are there any problems that cannot

be solved by a tuning machine. The halting problem states that , **does there exists a tuning machine (program)(T), that give any tuning machine P(including itself) and input I to p, as a input T,decides whether P halts on I.** He showed that such tuning machines doesn't exists.

For program analysis the same question can be posed as , "does there exist a program analysis tool, that given an input program P and input I to P, decides whether P halts on I or whether the program P exhibits a property for the input I". This is an undecidable problem similar to halting program i.e it is as difficult as halting problem to solve. So statistically establishing the properties of program in general is undecidable, but it can solved for specific cases.

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).**

   The normal form of a Boolean formula F is another formula F' such that F is equivalent to F' i.e. the truth tables of F and F' should be exactly same and F' should also obey some syntactic restrictions. There are 3 major normal forms(NF) namely ,negation,conjunction and disjunction NF's.

   Yes, CNF and DNF are in NNF form. This can be stated by referring to the Directed Acyclic Graphs(DAG) of these forms. For a formula to be in NNF, its DAG has to have ^(and) or V(or) in the top level nodes and the literals or its negation as the leaf nodes. This is the same DAG condition for CNF and DNF except that the head node of DNF is an V and head node of CNF is an ^, but the leaf nodes in these DAG's are also either literals or its negations. So both CNF and DNF are in NNF form by default.

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.**

   Model- theoretic method is a SAT Solver as it is aimed at deciding the Boolean satisfiability problem. It can be also knows as Truth Table Method. In this method we assign values to variables of the formula and then evaluate the formula with the assigned values. If the evaluator return true the formula can be decided to be a SAT formula. In this method we try all the assignments possible for the variables. Where as proof-theoretic methods use graphs with proof rules to derive contradictions to decide the validity of a formula.

   Drawback: With a "n" variable formula, the number of assignments possible are 2^n , so the time complexity of **model-theoretic** method in worst case is O(2^n) IE the model-theoretic can take exponential time,which is considered worst for algorithms.

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

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

   Let's assume F = $((p \rightarrow q) \wedge (q \leftrightarrow r)) \rightarrow (p \rightarrow r)$ and F is not valid and I is a falsifiable interpretation of F.

Step Number	Branch Number	Transformation	Steps considered
1	NA	$I \not\models F$	
2	NA	$I \models ((p \rightarrow q) \wedge (q \leftrightarrow r)$ and $I \not\models (p \rightarrow r)$	1 and →

3	1	$I \models (p \rightarrow q)$	2 and $\wedge$
4	2	$I \models (q \leftrightarrow r)$	2 and $\wedge$
5	3	$I \not\models (p \rightarrow r)$	1 and “and”
6	1	$I \not\models p$	1 and $\rightarrow$
7	1	$I \models q$	1 and $\rightarrow$
8	2	$I \models (q \wedge r)$	4 and $\leftrightarrow$
9	2	$I \models (\neg q \wedge \neg r)$	4 and $\leftrightarrow$
10	2	$I \models q$	8 and $\wedge$
11	2	$I \models r$	8 and $\wedge$
12	2	$I \models \neg q = I \not\models q$	9 and $\wedge$
13	2	$I \models \neg r = I \not\models r$	9 and $\wedge$
14	3	$I \models p$	5 and $\rightarrow$
15	3	$I \not\models r$	5 and $\rightarrow$

From the above table of transformations, we can notice that :

- steps 7 and 12 contradict each other, indicating **branch 1 and 2 are contradicting** each other
- steps 6 and 14 contradict each other, indicating **branch 1 and 3 are contradicting** each other
- steps 11 and 15 contradict each other, indicating **branch 2 and 3 are contradicting** each other

We can notice that we are facing contradictions in each branch, so the assumption F is not valid does not hold.

$((p \rightarrow q) \wedge (q \leftrightarrow r)) \rightarrow (p \rightarrow r)$ is Valid.

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

All 2-input 1-output Boolean functions can’t be encoded only with XOR gates.

XOR is not an universal gate, so function like AND, NAND cannot be implemented using XOR gates. Here is an argument against XOR and XNOR as universal gates.

- An XOR gate is a parity generator. Cascading parity generators always produce parity generators. AND and OR are not parity functions.
- An XOR gate can be used as an inverter. An XNOR gate is an XOR followed by an inverter, so it is also a parity generator.

To be an universal gate, a gate needs to satisfy both the following conditions:

- You should be able to create an inverter using the gate.
- You should be able to block the input (i.e. setting N - 1 inputs to 0 or 1, the remaining input is ignored)

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

Proving Dem-organ's laws using truth table method.

Law 1: Demorgan's Negation of a disjunction law :- $\neg(p \wedge q) = \neg p \vee \neg q$

p	**q**	**(p∧q)**	¬(p∧q)	**¬p**	**¬q**	**¬p ∨¬q**
0	0	0	**1**	1	1	**1**
0	1	0	**1**	1	0	**1**
1	0	0	**1**	0	1	**1**
1	1	1	**0**	0	0	**0**

From **¬(p∧q)** and **¬p ∨¬q** columns,we can that both the truth values are equal.

Law 2: Demorgan's Negation of a disjunction law : $\neg(p \vee q) = \neg p \wedge \neg q$

p	**q**	**p ∨ q**	**¬(p ∨ q)**	**¬p**	**¬q**	**¬p ∧¬q**
0	0	0	**1**	1	1	**1**
0	1	1	**0**	1	0	**0**
1	0	1	**0**	0	1	**0**
1	1	1	**0**	0	0	**0**

From **¬(p ∨ q)** and **¬p ∧¬q** columns,we can that both the truth values are equal.

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

When two formulas are logical equivalence, they are defined on the same set of variables and they must have exactly same truth table. Whereas Equisatifiable is weak notion. Equisatisfiability method takes as input arbitrary Boolean formulas and outputs them in CNF format such that: The output is only linear in size as the input formula and the output is not necessarily equivalent to the input.
Two formulas F and F' are equisatisfiable if and only if: **F is satisfiable iff F' is satisfiable.** So if two formulas are equivalent they are also in equivalence, but the inverse is not true as equisatisfiable F' need not have same interpretations for which the formulas F' and F are satisfiable. It is possible that equisatisfiable F and F' might have same or different variables.
So we can also state that any satisfiable formula K is equisatisfiable to **"True".**
Equisatisfiable notion can be used along with CNF as input to SAT solvers.

Examples for equisatisfiable but not logically equivalent:

a) F = x and F' = x+y
b) F = x v y and F' = x^y
c) F = x and F' = True(top)

Question 2: Fuzzing – experiments with Google AFL

American Fuzzy Lop is a brute-force fuzzer coupled with a rock-solid instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle changes to program control flow, thus tracking it.

Working of AFL:

To maximize the fuzzing performance, American fuzzy lop expects to compile the program under test program with the help of a utility program(ex afl-gcc/afl-g++) that instruments the code with help functions that track control flow during building the binary file. This enables the fuzzer to detect what behaviour changes in the program under test's response to the input. Black-box testing is also supported in cases when instrumenting the code is not possible.

AFL requires the user to provide a sample command running for running the program and at least one small input/test case for the program under test. In our case of fuzzing minisat, we have provided 3 sample input files i.e. the ".cnf" files as the test cases and ran the alf-fuzz with minisat binary file. The fuzzer then attempts to actually execute the specified command/binary build file and, if that succeeds, attempts to reduce the input file to the smallest one which triggers the same behaviour. Thus minimizing the input space to trigger specific behaviour of the program.

Following this initial phase, AFL starts the actual fuzzing process by applying various modifications to the input file(ex – deterministic arithmetic, bitflip, injection of dictionary terms, splice). This process could suggest the discovery of a new bug, possibly a security vulnerability, when the tested program crashes or hangs. In this case, it saves the modified input file for further inspection of the user. Thus the test suite the triggers bugs can be build using AFL.

Experiment of AFL on minisat:

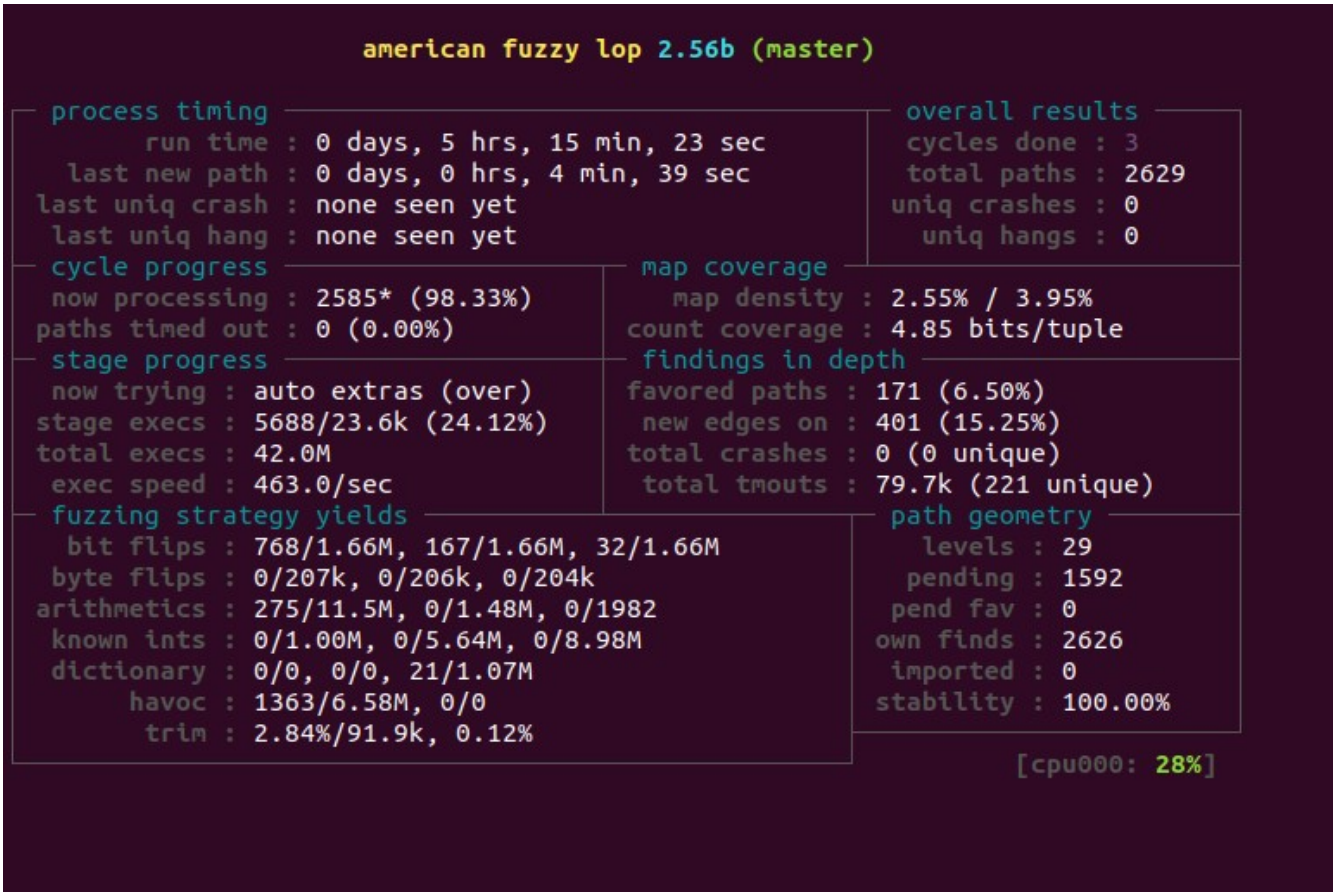
minisat used for experiment is : https://github.com/agurfinkel/minisat
Google AFL repository : https://github.com/google/AFL

steps:

1. Clone Google AFL and minisat from above mentioned repositories.
2. Build Google AFL using command : ***sudo make install***
3. Build minisat using afl-gcc and afl-g++, so that these compilers wrap the minisat bytecode file with instrumentation code. Use the following command in minisat folder.
   Command : ***sudo CC=/path/to/afl/afl-gcc CXX=/path/to/afl/afl-g++ make clean all***
4. As the next step create a folder to hold test-cases and another folder to store the results of afl-fuzz.
5. Next we run afl-fuzz using test-case folder, destination folder and the path to minisat binary code as the arguments from the command. Run below command in AFL folder.
   Command : ***afl-fuzz -i ./Q2/test-cases -o ./Q2/findings -M master ./minisat/build/release/bin @@***
6. This starts the fuzzing process and a GUI to observe the results pop up.
7. Run Fuzzer for good amount of time and check for the results in GUI and output folder for further analysis.

Experiment result:

- Number of crashes = 0
- Number of hangs = 0
- Number of total paths = 2629
- Cycles done = 3
- Amount of time fuzzed = 5 hrs 15 min 23 sec



Points observed :

- Cycles is the number of times the fuzzer went over all the interesting test cases discovered, fuzzed them, and looped back to the very beginning. Every fuzzing session should be allowed to complete at least one cycle, so that at least all the interesting test cases are fuzzed and ideally, should run much longer than that.
- The longer you run the better is chances of finding more vulnerabilities.
- More the paths covered, better is the ability to penetrate error code paths.
- Levels field tracks the path depth reached through the guided fuzzing process. In essence: the initial test cases supplied by the user are considered "level 1". The test cases that can be derived from that through traditional fuzzing are considered "level 2"; the ones derived by using these as inputs to subsequent fuzzing rounds are "level 3"; and so forth.
- There are different ways AFL can be used for fuzzing with different sets of fuzzing variations for modifying the provided sample test- case and build on that and there are multiple different variations of AFL that allow fuzzing of applications written in different languages.

Question 3: Symbolic Execution

Symbolic execution, is a promising approach to exploring different execution paths through programs. However, it has some limitations like, for paths that are long and involve many conditions, SMT solvers may not be able to find satisfying assignments to variables that lead to a test case that follows that path. Other paths may be short but involve computations that are outside the capabilities of the solver, such as non-linear arithmetic or cryptographic functions.

SHA1 or any hash functions are generally too complicated for the SAT solvers to revert them, that is an open problem now. If our code contains has functions or any implementations that are difficult for SAT solvers to revert, we can combine concrete and symbolic testing i.e. use concolic testing to cover all the paths of above code snippet.

However, when a part of the path condition is infeasible for the SMT solver to handle, we substitute values from a test run of the program to cover that path(concretizing the inputs). In many cases, this allows us to make progress towards covering parts of the code that we could not reach through either symbolic execution or randomly generated tests.

Question 3.1 :

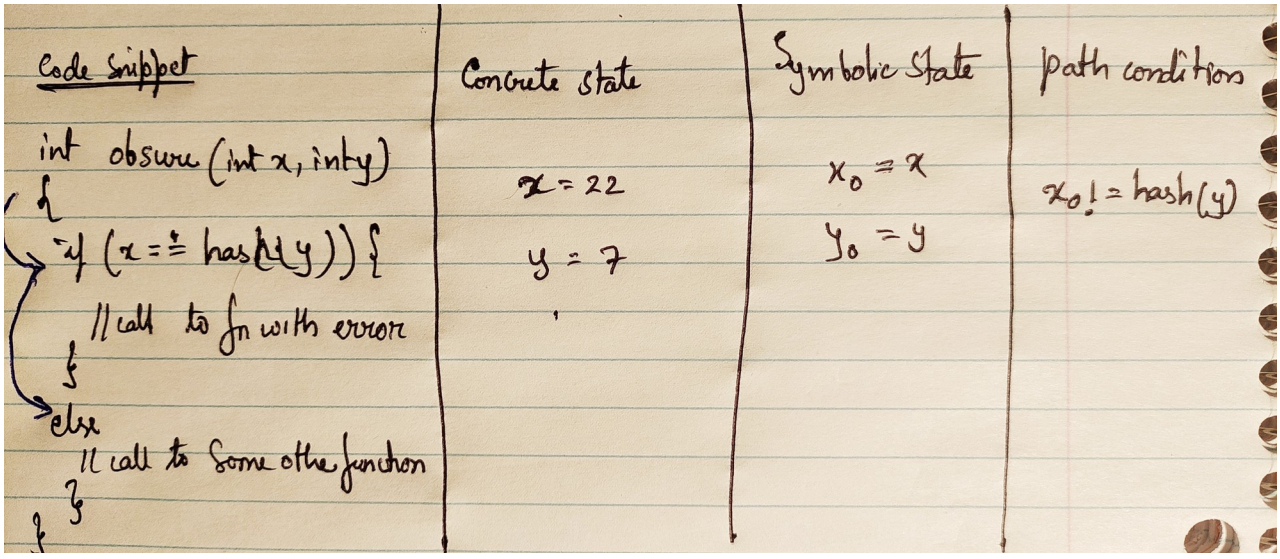
given code snippet is as below:

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

The testing procedure to systematically construct a test suite can be done by **using concolic testing** for this code snippet using a random input as a sample test case.

steps:

- Start the execution with x = 22 and y = 7
- The condition check x == hash(y) fails as 22 != some_large_number(hash(y))
- So the execution takes else path and the path constraint is update as **x0 != hash(y)**
- The execution comes back and negates the most recent path condition to take other available path, in this case negated path condition is x0 == hash(y).
- At this point, dynamic symbolic analyses uses the concrete state vale of y = 7 available to it and updates the path condition to be solved as x = some_large_number(hash(y)) ans is given to SAT solvers to solve.
- This pretty easy to solve as it directly selects the given value for x0 and updates the test case to take that path as y = 7 and x = hash(y) as a test case in the test suite.



Thus using concolic testing we have built a testing technique which can be used to automatically test all the paths of the above given program.

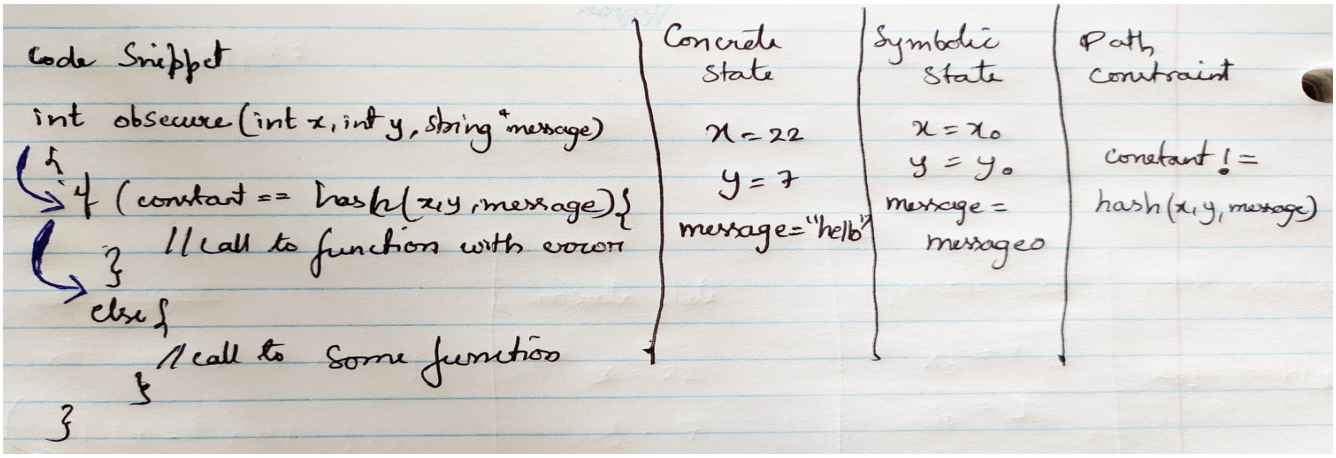
Question 3.2

given snippet of code is :

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

Given that constant is a 128 bit long number which is very difficult to guess by the humans . The number of possible values for constant i.e. 2^128 = 340,282,366,920,938,463,463,374,607,431,768,211,456. So let's assume that it is very difficult for humans to create a test cases that's passes the condition **constant == hash(x,y,message),** so we wanna to automate it.

Let's try concolic testing procedure technique proposed in 3.1 and check whether that technique can be used for this code snippet.



steps: similar steps as of 3.1

- Start the execution with x = 22, y = 7 and message = "hello"
- The condition check **constant == hash(x,y,message)** fails as pre_big_number != some_large_number(hash(x,y,message))
- So the execution takes else path and the path constraint is update as **constant != hash(x,y,message)**
- The execution comes back and negates the most recent path condition to take other available path, in this case negated path condition is **constant == hash(x,y,message)**
- At this point, dynamic symbolic analysis can use any of the concrete values available to to simplify the path constraint. Challenges here :

- Let’s assume DSE(concolic testing) uses concrete x value available , so the path condition updates to :

constant == hash(22,y,message) . This is then given to SAT solvers to find y and message, which is not possible, as it is one of that **Hard** problem in computer science.

- Let’s assume DSE uses any of the two values available, x = 22 and message = “hello”, even in this case the updated path constraint **constant == hash(22,y,”hello”)** cannot still be solved by SAT solvers, as it inverting hash function.
- So whatever values/combinations of concrete values are selected to simply path constraint, the hash function is still not solvable by SAT solvers. So this branch goes as a dead code.

Thus using concolic testing technique built for 3.1 cannot be used for automatically testing all the paths of the above given program as SAT solvers cannot be used to invert hash functions.

Question 3.3: Programming Assignment with the KLEE to find bugs in Coreutils (version 6.10)

1. **paste**

number of Bug types: 1

Bug type : memory error: out of bound pointer
Number of such bugs : 10
Occurred at lines in paste: 90, 93, 99, 103, 107, 111, 115, 119, 123, 127
Sample error:

```
klee@bba6db144303:~/cu/coreutils-6.10/obj-llvm/src/klee-out-0$ cat test000055.ptr.err
Error: memory error: out of bound pointer
File: ../../src/paste.c
Line: 115
assembly.ll line: 4809
Stack:
        #000004809 in collapse_escapes (strptr=94441301739986) at ../../src/paste.c:115
        #100004589 in __klee_posix_wrapped_main (argc=4, argv=94441302277920) at ../../src/paste.c:484
        #200002801 in __user_main (=15, =94441281252400, =94441281252528) at /tmp/klee_src/runtime/POSIX/
klee_init_env.c:245
        #300000555 in __uClibc_main (=15, =94441281252400) at libc/misc/internals/__uClibc_main.c:401
        #400000721 in main (=15, =94441281252400)
Info:
        address: 94441317533388
        next: object at 23094917069760 of size 1536
                MO30[1536] (no allocation info)
```

Sample test case generated:

```
object 5: text: ........
object 6: name: 'stdin-stat'
object 6: size: 144
object 6: data: b'D\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\xff\xff\xff\xff\x01\x00\x00\x00\x00\x00\x
00\x00\xa4\x81\x00\x00\xe8\x03\x00\x00\xe8\x03\x00\x00\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\xf
f\xff\xff\xff\xff\xff\xff\xff\x00\x10\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\x8d\x8d\xeb
^\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xe5\xae\xf5P\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xf
f\xff\x08\x8d\xeb^\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'
object 6: hex : 0x44000000000000000010000000ffffffff0100000000000000a4810000e8030000e8030000ffffffff0000000
00000000ffffffffffffffff00100000000000000ffffffffffffffff8d8deb5e00000000ffffffffffffffffe5aef55000000000
ffffffffffffffff088deb5e00000000ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
object 6: text: D.......................................................................^.............
..P..............^.....................................
object 7: name: 'stdout'
object 7: size: 1024
object 7: data: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
```

Cause of error and probable patch :
The error seems to be caused due to very large data value supplied to the program under test. A probable patch for this error is to check if the value of pointer variable before usage to check if it is with in the acceptable bounds of memory.

Klee-stats for paste(program under-test):

- Instruction coverage in the LLVM bitcode(ICov(%)) = 43.24
- Branch coverage in the LLVM bitcode(BCov(%)) = 31.49
- total wall time(Time(s)) = 3605.95
- time spent in the constraint solver(TSolver(%)) = 88.03
- number of executed instructions(Instrs) = 276778333
- total static instructions in the LLVM bitcode(ICount) = 19388

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
./klee-out-0	276778333	3605.95	43.24	31.49	19388	88.03

2. **Mkdir**

number of Bug types: 2

Bug type : memory error: out of bound pointer
Number of such bugs : 1
Occurred at lines in Mkdir: 194
Sample error:

```
klee@bba6db144303:~/cu/coreutils-6.10/obj-llvm/src/klee-out-1$ cat test000027.ptr.err
Error: memory error: out of bound pointer
File: ../../lib/quotearg.c
Line: 248
assembly.ll line: 8309
Stack:
        #000008309 in quotearg_buffer_restyled (buffer=94254140149760, buffersize=256, arg=0, argsize=18446744073709551615, quoting_style=5,
o=94254282299152) at ../../lib/quotearg.c:248
        #100008091 in quotearg_buffer (buffer=94254140149760, buffersize=256, arg=0, argsize=18446744073709551615, o=94254282299152) at ../..
/lib/quotearg.c:526
        #200007959 in quotearg_n_options (n=0, arg=0, options=94254282299152) at ../../lib/quotearg.c:624
        #300007797 in quotearg_n_style (n=0, arg=0) at ../../lib/quotearg.c:666
        #400007775 in quote_n (name=0) at ../../lib/quote.c:31
        #500005036 in quote (name=0) at ../../lib/quote.c:39
        #600004819 in __klee_posix_wrapped_main (argc=3, argv=94254175805440) at ../../src/mkdir.c:194
        #700003015 in __user_main (=15, =94254162578000, =94254162578128) at /tmp/klee_src/runtime/POSIX/klee_init_env.c:245
        #800000579 in __uClibc_main (=15, =94254162578000) at libc/misc/internals/__uClibc_main.c:401
        #900000745 in main (=15, =94254162578000)
Info:
        address: 0
        next: object at 23402277718976 of size 1536
                MO30[1536] (no allocation info)
```

Cause of error and probable patch :
The error seems to be caused due to very large data value supplied to the program under test. A probable patch for this error is to check if the value of pointer variable before usage to check if it is with in the accept able bounds of memory.

Sample test case generated:


```
object  6: uint: 0
object  6: text: ........
object  7: name: 'stdin-stat'
object  7: size: 144
object  7: data: b'D\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\xff\xff\xff\xff\x01\x00\x00\x00\x00\x00\x00\x00\xa4\x81\x00\x00\xe8\x03\x00
\x00\xe8\x03\x00\x00\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00\x10\x00\x00\x00\x00\x00\x00\xff\xff
\xff\xff\xff\xff\xff\xff\x8d\x8d\xeb^\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xe5\xae\xf5P\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\x
ff\xff\x08\x8d\xeb^\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
xff\xff\xff\xff\xff\xff'
object  7: hex : 0x4400000000000000010000000ffffffff0100000000000000a4810000e8030000e8030000ffffffff0000000000000000ffffffffffffffff001000000
0000000ffffffffffffffff8d8deb5e00000000ffffffffffffffffe5aef55000000000ffffffffffffffff088deb5e00000000ffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffff
object  7: text: D.....................................................................^...............P...............^...............
......................
object  8: name: 'stdout'
object  8: size: 1024
object  8: data: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
```


Bug type : concertized symbolic size
Number of such bugs : 1
Occurred at lines in Mkdir: 194

Sample error:

```
klee@bba6db144303:~/cu/coreutils-6.10/obj-llvm/src/klee-out-1$ cat test000053.model.err
Error: concretized symbolic size
File: ../../lib/xmalloc.c
Line: 49
assembly.ll line: 7744
Stack:
        #000007744 in xmalloc (n) at ../../lib/xmalloc.c:49
        #100007722 in xnmalloc (n) at ../../lib/xalloc.h:116
        #200005199 in mode_compile (mode_string=94254175561171) at ../../lib/modechange.c:168
        #300004851 in __klee_posix_wrapped_main (argc=4, argv=94254175928896) at ../../src/mkdir.c:204
        #400003015 in __user_main (=15, =94254162578000, =94254162578128) at /tmp/klee_src/runtime/POSIX/klee_init_env.c:245
        #500000579 in __uClibc_main (=15, =94254162578000) at libc/misc/internals/__uClibc_main.c:401
        #600000745 in main (=15, =94254162578000)
Info:
  size expr: (Mul w64 16
                      (Add w64 6
                               (Add w64 (SExt w64 (ZExt w32 (Eq 45 (Read w8 3 arg00))))
                                        (SExt w64 (ZExt w32 (Eq 45 (Read w8 8 arg00)))))))
  concretization : 96
  unbound example: 112
```

Sample test case generated:


```
object  7: int : 0
object  7: uint: 0
object  7: text: ........
object  8: name: 'stdin-stat'
object  8: size: 144
object  8: data: b'D\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\xff\xff\xff\xff\x01\x00\x00\x00\x00\x00\x00\x00\xa4\x81\x00\x00\xe8\x03\x00
\x00\xe8\x03\x00\x00\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00\x10\x00\x00\x00\x00\x00\x00\xff\xff
\xff\xff\xff\xff\xff\xff\x8d\x8d\xeb^\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xe5\xae\xf5P\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\x
ff\xff\x08\x8d\xeb^\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
xff\xff\xff\xff\xff\xff'
object  8: hex : 0x4400000000000000010000000ffffffff0100000000000000a4810000e8030000e8030000ffffffff0000000000000000ffffffffffffffff001000000
0000000ffffffffffffffff8d8deb5e00000000ffffffffffffffffe5aef55000000000ffffffffffffffff088deb5e00000000ffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffff
object  8: text: D.....................................................................^...............P...............^...............
......................
object  9: name: 'stdout'
object  9: size: 1024
object  9: data: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
```


Cause of error and probable patch :
The error seems to be caused due to very large data value supplied to the program under test. That said, it is

possible that it indicates a real bug the source probably shouldn't end up malloc'ing very large amounts of memory (e.g., using a negative signed count).A probable patch for this error is to check allocation size before allocating memory dynamically.

Klee-stats for paste(program under-test):

- Instruction coverage in the LLVM bitcode(ICov(%)) = 42.31
- Branch coverage in the LLVM bitcode(BCov(%)) = 29.40
- total wall time(Time(s)) = 3606.21
- time spent in the constraint solver(TSolver(%)) = 45.55
- number of executed instructions(Instrs) = 1313467899
- total static instructions in the LLVM bitcode(ICount) = 21491

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
./klee-out-1	1313467899	3606.21	42.31	29.40	21491	44.55

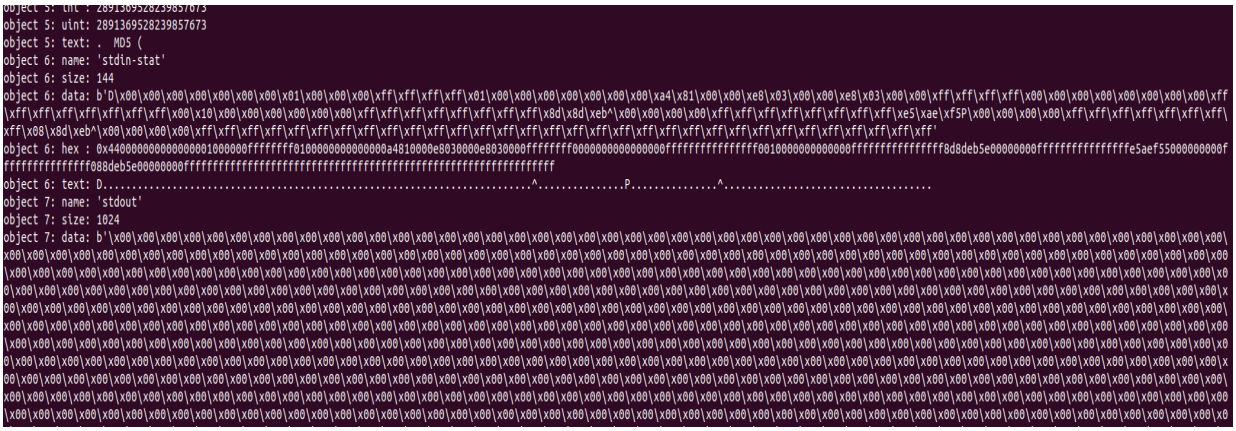
3. **Md5sum**

number of Bug types: 1

Bug type : memory error: out of bound pointer
Number of such bugs : 1
Occurred at lines in Mkdir: 194
Sample error:

```
klee@bba6db144303:~/cu/coreutils-6.10/obj-llvm/src/klee-out-2$ cat test000031.ptr.err
Error: memory error: out of bound pointer
File: ../../src/md5sum.c
Line: 213
assembly.ll line: 8763
Stack:
        #000008763 in bsd_split_3 (s=94490784913416, s_len=0, hex_digest=94490784206896, file_name=94490784207680) at ../../src/md5sum.c:213
        #100008422 in split_3 (s=94490784913408, s_len=8, hex_digest=94490784206896, binary=94490784206728, file_name=94490784207680) at ../../src/md5sum.c:259
        #200005135 in digest_check (checkfile_name=94490780439112) at ../../src/md5sum.c:471
        #300004716 in __klee_posix_wrapped_main (argc=2, argv=94490799303936) at ../../src/md5sum.c:667
        #400002830 in __user_main (=15, =94490776628032, =94490776628160) at /tmp/klee_src/runtime/POSIX/klee_init_env.c:245
        #500000584 in __uClibc_main (=15, =94490776628032) at libc/misc/internals/__uClibc_main.c:401
        #600000750 in main (=15, =94490776628032)
Info:
        address: 94490784913407
        next: object at 23111728405440 of size 1536
                MO30[1536] (no allocation info)
```

Sample test case generated:



Cause of error and probable patch :
The error seems to be caused due to very large data value supplied to the program under test. A probable patch for this error is to check if the value of pointer variable before usage to check if it is with in the accept able bounds of memory.

Klee-stats for paste(program under-test):

- Instruction coverage in the LLVM bitcode(ICov(%)) = 49.36
- Branch coverage in the LLVM bitcode(BCov(%)) = 33.01
- total wall time(Time(s)) = 3613.5
- time spent in the constraint solver(TSolver(%)) = 87.36
- number of executed instructions(Instrs) = 668919943
- total static instructions in the LLVM bitcode(ICount) = 22423

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
./klee-out-2	68919943	3613.65	49.36	33.01	22423	87.36

Question 3.4:

Let's assume we are given two implementations of the same UNIX utility which should exhibit same behaviour. In order to determine their behaviours we can use symbolic analysis for checking the equivalence of both implementation. The results of symbolic analysis can be any of the following scenarios:

- **Identical symbolic analysis results :** If the symbolic analysis used for testing both the implementations exhibits same output, then we can assume that the behaviour of both the implementation is equal i.e. they are equivalent. We can establish this only assuming that the symbolic abstractions or the technique used for both implementations is similar.

- **Different symbolic analysis results :** We need to consider multiple scenarios before establishing that both the implementations are not equivalent when the symbolic analysis results are different:
    - **Symbolic abstraction involved** : If the discrepancies in the results is caused due to the symbolic testing , then we cannot establish that the implementations are not equivalent or one of it has a bug or error.
    - **Evaluating Equivalence :** It is possible that two different implementations of the same specification can produce different behaviours which are not fixed in nature. Example, a network routers protocol for delivering UDP data packets, as there is no fixed route that these packets follow, its different implementations might give different results for inputs, but it cannot be considered as bug when the specification functionalities are exhibited by the implementation.

But, when different implementations of a deterministic functionality produce different results for same input, assuming that the abstractions used by symbolic execution tool are not the cause for the

difference is output, then we can establish that one of the implementation is not following the specification i.e the presence of a bug or error.

Question 4: Syntax of Boolean formulas

4.1 Context-free grammar for Boolean formulas with balanced parenthesis:

Let V = set of non-terminal symbols, T = set of terminal symbols, P = set of productions , S = start symbol indicate the grammar G of Boolean formulas, where G = (V, T, S, P).

for Boolean functions we can define :
T = terminal symbols = “true”, “false”, x(literal), F(formula of literals), parenthesis(Lpara and Spara)
V = non-terminal symbols = S(lets consider S)
S = start symbol = S

Then the set of productions P for Boolean formulas can be defines as follows :

- S → True
- S → False
- S → x
- S → ~x
- S → F
- S → (F)
- S → S ^ S
- S → S V S
- S → ~S
- S → Lpara S Rpara

Let’s consider an example and prove that this context-free grammar can form Boolean formulas with balances parenthesis.

Example : (((x ^ y) V (x^z)) ^ (y V z)) let f1 = x^y , f2 = x^z , let f3 = y V z

• S → S	production rule applied
• S → Lpara S Rpara → (S)	S → Lpara S Rpara
• S → (S ^ S)	S → S ^ S
• S → ((S) ^ f3)	S → Lpara S Rpara and S → F and S → (F)
• S → ((S V S) ^ (f3))	S → S V S
• S → ((f1 V f2) ^ (f3))	S → F and
• S → (((f1) V (f2)) ^ (f3))	S → (F)
• S → (((x ^ y) V (x^z)) ^ (y V z))	substitute back f1, f2 and f3

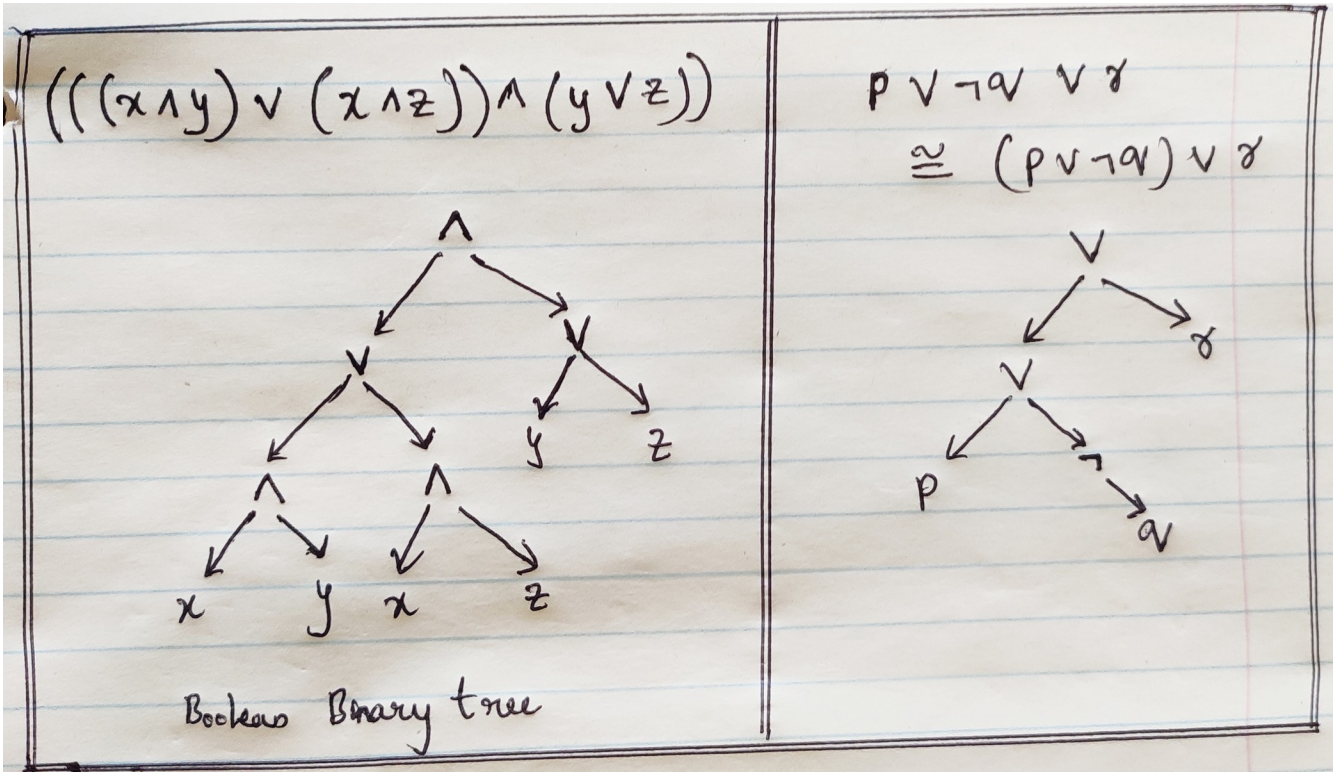
From the above proposed context-free grammar we are able to form Boolean formulas with balanced parenthesis. Hence my grammar works.

4.2 Boolean formulas as directed acyclic graphs (DAGs):

Boolean formulas can be represented as directed acyclic graphs in may ways like binary decision diagram(BDD), binary trees ... etc. I would like to represent binary formulas in the form a binary tree, which is also a directed acyclic graph. To represent in tree from, we need to make sure binary formula is in binary form i.e. used parenthesis in the formula whenever the operations are more than one.

Example : x V y V z can be represented ~= (x V y) V z

- This transformation is made to the formula in order to have only two children for each node i.e either left or right and no more than that.
- The Boolean binary tree consists of 3 major parts in this scenario :
  - **Nodes :** the nodes in this tree are Boolean operators like AND(^), OR(V), negation(~)
  - **Edges :** Edges represent the connection with child nodes. Each node can have at most 2 edges.
  - **Leaf nodes :** The leaf nodes in this tree represent Boolean literals/Boolean variables.
- Each node can have atmost two children, one left and one right. The order of the nodes matters.
- Leaf nodes will not have any children. Negation(~) node will have only one right branch.
- With this representation of Boolean binary tree, the parenthesis are explicitly added before reading the left node/branch and after reading the right node/branch vales for every node.
- Few examples of representing the formula in proposed Boolean binary tree :



Why no parenthesis in the tree : The parenthesis in the Boolean formula need not be stored explicitly in the proposed tree, as every branch is developed from the content between the parenthesis.
Will there be parenthesis balancing problem : The DAG to Boolean formal conversion in this proposed way will read attach a "(" before reading the left child/leaf node then the node and then the right child/leaf node and attach ")" at the end. So for every "(" there will be a ")" parenthesis and parenthesis balancing problem is not an issue here.

Recursive program to convert Dag to Boolean formula:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# A function to do inorder tree traversal
def printInorder(root):
    if root.val in ["V", "^", "~"]:
        print("(", end=' ')
        if root:
            # First recur on left child
            printInorder(root.left)

            # then print the data of node
            print(root.val, end=' ')

            # now recur on right child
            printInorder(root.right)

        print(")", end=' ')
    else:
        if root.val != None:
            print(root.val, end=' ')


# Driver code #example demonstration
# ( ( p V q ) V r )
root = Node("V")
root.left = Node("V")
root.right = Node("r")
root.left.left = Node("p")
root.left.right = Node("q")

print("\nInorder traversal of binary tree is")
printInorder(root)
```

Example output for ((p V q) V r) :

```
/usr/bin/python3.8 /home/sai/Desktop/653/Assignments/Assignment1/DAG.py

Inorder traversal of binary tree is
( ( p V q ) V r )
Process finished with exit code 0
```

From the above example we can see that the proposed way of storing Boolean formulas does work in retrieving the formulas from DAG's along with balanced parenthesis.

Question 5: The Undecidability of the Halting Problem

Undecidability : An undecidable problem in computability theory and computational complexity theory is a decision problem for which it is proved impossible to construct an algorithm that always leads to a correct yes-or-no answer.

Halting problem: In computation theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. In other words - "does there exists a tuning machine (program)(T), that give any tuning machine P(including itself) and input I to p, as a input T,decides whether P halts on I".

Why is Halting problem Undecidable : Alan Turing proved that a general algorithm running on a Turing machine can not necessarily exist which solves the stop problem for all possible program-input pairs. Hence the stop problem for Turing machines is undecidable.

Mathematical Proof of Undecidability of Halting problem: We can prove this notion using **proof by contradiction**. In logic and mathematics, proof by contradiction is a form of proof that establishes the truth or the validity of a proposition, by showing that assuming the proposition to be false leads to a contradiction. Proof by contradiction is also known as indirect proof, proof by assuming the opposite.

Proof: The overall goal is to show that there is no **total computable function**(a function is computable if there exists an algorithm that can do the job of the function) that decides whether an arbitrary program P halts on arbitrary input I. As per our set up lets assume T is the function that has the job of deciding whether P halts on I, so the function T is not computable:

T (P, I) = 1 if program P halts on input I
= 0 otherwise

Here program P refers to a program that is a fixed turing-complete model of computation.
The proof can be proceeded by directly establishing that no total computable function with two arguments can be the required function T.

Given any total computable binary function f, the following partial function g is also computable by some program e:

g(I) = 0 if f(I, I) = 0
= undefined otherwise

The verification that g is computable relies on the following constructs (or their equivalents):

- computable subprograms (the program that computes f is a subprogram in program e)
- duplication of values (program e computes the inputs i,i for f from the input i for g)
- conditional branching (program e selects between two results depending on the value it computes for f(i,i))
- not producing a defined result
- returning a value of 0

We can compute g using below pseudocode g:

```
procedure compute_g(i):
        if f(i, i) == 0
                then  return 0
        else
                loop forever
```

Because **g** is partial computable, there must be a program **e** that computes **g**, by the assumption that the model of computation is Turing-complete. This program is one of all the programs on which the halting function **h** is defined.

The next step of the proof shows that T (e,e) will not have the same value as f(e,e).

From the definition of g that exactly one of the following two cases must hold:

- f(e,e) = 0 and so g(e) = 0. In this case T (e,e) = 1, because program e halts on input e.
- f(e,e) ≠ 0 and so g(e) is undefined. In this case T(e,e) = 0, because program e does not halt on input e.

In either case, **f** cannot be the same function as **T**. Because **f** was an arbitrary total computable function with two arguments, all such functions must differ from **T**.

Contradictions that establishes proof that no such function T exists:

- Now assume **f** was the halting function **T**, if g(e) is defined (g(e) = 0 in this case), g(e) halts so f(e,e) = 1. But g(e) = 0 only when f(e,e) = 0, contradicting f(e,e) = 1.
- Similarly, if g(e) is not defined, then halting function f(e,e) = 0, which leads to g(e) = 0 under g's construction. This contradicts the assumption of g(e) not being defined. In both cases contradiction arises.

Therefore any arbitrary computable function f cannot be the halting function T. Thus solving a Halting problem can be decided as Undecidable.

Sources :

1. Halting problem : https://en.wikipedia.org/wiki/Halting_problem
2. Undecidable : https://en.wikipedia.org/wiki/Undecidable_problem
3. Computable function : https://en.wikipedia.org/wiki/Computable_function
4. Partial function : https://en.wikipedia.org/wiki/Partial_function