Fuzzing: It is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. Initially, such input was generated by hardcoded rules and provided to the application with little in-depth monitoring of the execution. If the application crashed when given a specific input, the input was considered to have triggered a bug. Otherwise, the input would be further randomly mutated. Unfortunately, fuzzers suffer from the test case requirement.

Symbolic Execution: A method for discovering inputs that cause different parts of a program to execute. It is the opposite of concrete execution which runs the program with concrete values over symbolic ones.

This method can be employed in different ways:

- Tracing the possible paths, a binary can take
- Determining what inputs can lead to certain paths
- This leads to uses in testing, reverse engineering, and detection.

In the symbolic execution technique, unknown values are replaced by symbolic variables. Additionally, an IR is typically used to emulate the code's possible execution paths while generating constraints for each path. These constraints can then be solved to determine what input will cause a code path to be taken. Generally imposing more constraints on a problem is a good thing and it usually leads to overcoming the problem. On the other hand, fewer constraints usually can bring about path explosion, where the number of paths grows exponentially with each branch and quickly becomes intractable. all currently proposed symbolic execution techniques suffer from very limited scalability due to the problem of path explosion: because new paths can be created at every branch, the number of paths in a program increases exponentially.

Angr: In short, Angr makes the following contributions (as their developers claim):

1) It reproduces many existing approaches in offensive binary analysis, in a single, coherent framework, to provide an understanding of the relative effectiveness of current offensive binary analysis techniques.
2) It shows the difficulties (and solutions to those difficulties) of combining diverse binary analysis techniques and applying them on a large scale.

Angr is an iPython-accessible, open and expandable, and architecture "independent" tool with the intention of doing binary analysis.

Angr is built on top of a lot of codes involves a lot of different aspects such as CLE (CLE loads everything, a binary loader that works on the backend of angr and it is able to load a wide variety of files such as ELF, PE, CGC, and so on), Pyvex (basically emulate the instructions). There are different methods of solving the constraints (branches in the CFG that comes from conditions or loops in the program) such as SMT, SAT, etc. and angr employs z3 and claripy to solve Them. Clairpy is a tool by which angr can take the IR (output of Pyvex) and represent that in bit vectors. It abstracts all values to an internal representation of an expression that tracks all operations in which it is used. That is, the expression x, added to the expression 5, would become the expression x + 5, maintaining a link to x and 5 as its arguments.

Angr does not only do symbolic execution but it can also generate CFG (control flow graph), it has a GUI called anger management, etc. Value Set Analysis (VSA) is another interesting feature that angr provides.

It allows you to ask at any given time what a certain register value is and on top of that, it tells us what the register's range of value must be at any given time.