# 3 Literature Review

The primary goal of automated tools for detecting bugs and vulnerabilities is to find the greatest number of bugs in the shortest amount of time. Also a secondary goal is to ensure that as much of an application’s code is checked as possible. In this chapter I will discuss how well the four major categories of automated tools achieve these goals – where they fall short and how they complement each other.

## 3.1 About Static Code Analysis

Static code analysis was the first kind of automated bug checking tool developed. As discussed in chapter 2, an SCA relies on the set of rules built-in to the program to identify logic errors. If the set of rules is small or outdated (doesn’t include the newest rules to identify vulnerabilities), then the checking falls short. Also SCAs are prone to generation of false positives in their reports. These false positives clutter the reports making it difficult to identify the legitimate bugs that need to be fixed (1–6).

On the other hand, SCAs are very fast and can check a large code tree in a matter of several hours. The SCA does look at each line of code and can identify those ‘obvious’ bugs that are included in its ruleset (7).

## 3.2 Going Deeper with Fuzzing

Some bugs are not obvious by looking at the source code. It is more effective to find these kinds of bugs by testing the applications and by providing input data that does not conform to what is expected by the program. Fuzzing provides this kind of testing. Fuzzing, the technique of providing random data as inputs to a program, can uncover missing data validation, or bounds checking logic among other bugs (5,6,8,9). These types of bugs are difficult for SCAs to detect by simply reviewing the source code.

Providing purely random data as input to a program, can be effective under certain circumstances (10,10–13). However, the purely random approach does have its drawbacks as well. The random data does not penetrate very far into the program’s logic before it is rejected (5,6,14–17). Typically purely random fuzzing only provides approximately 10% code coverage (5,6,8,18). In an effort to improve the effectiveness of fuzzing the Protos project was created in 1998. The Protos project was a group of software security experts who were interested in improving fuzzing techniques (16,19). The protos project developed two new types of fuzzers, model-based and mutation-based. As described in chapter 2, these types of fuzzers generate random data that conforms more closely to the expected format of user inputs which allows the random data to penetrate deeper into the program’s logic since the data can bypass the initial validation checks (8,20,21). These new fuzzers can achieve up to 50% code coverage, but their model definitions can be tedious to create and maintain (8).

## 3.3 Going deep and wide with Symbolic Execution

In an effort to obtain the code coverage of static code analysis and the data-driven depth testing of fuzzing, in 2005, the concept of symbolic execution was developed. In symbolic execution, a running program is analyzed to identify all possible execution paths. First, the variables in the program are converted to simple symbols (22–25). Next all conditions that control the execution paths are converted into a logical formula with each condition generating a single constraint in the formula. The formula is then passed to a symbolic modulo constraint solver (smt) which evaluates the formula to determine if it can be solved (26,26–32). If there is a particular set of values for the symbols which ensure the formula is true, then it is said to be satisfied. If all values of the symbols will satisfy the formula then it is said to be validated. After a formula has been checked, the symbolic executer then works backward through the formula negating constraints as it goes to force the execution of a new path. This revised formula is then checked again, and the process repeats.

The process of exploring every execution path through a program provides a very high code coverage rate of 100%. However, this high code coverage comes at a price. In even moderately sized programs there can be hundreds or thousands of possible paths through the program. Exploring each and every one of these paths leads to a condition called ‘path explosion’ where there are just too many possible choices, and the symbolic executer or constraint solver simply runs out of memory to hold all the constraints to be solved (26,30,31,33,34). In order to combat path explosion, various heuristics are imposed on the symbolic executer and constraint solver to limit the number of paths to check. These limits can include the depth of testing, the amount of memory used, or some kind of weighting based on the probability of the particular path based on its length (a minimum spanning tree problem)(31,34). Unfortunately, imposing these limits also means that the code coverage is reduced from the theoretical 100% to a practical 75-80%.

## 3.4 Concolic Execution or Taint Analysis

One of the newest techniques for automated bug finding is an improvement on symbolic execution that only looks at those path conditions that can be directly affected by user input. This form of testing is called taint analysis or concolic execution. The process begins by running an application with known good user inputs. The application’s execution is traced and all memory locations and registers that are directly or indirectly affected by the user input are identified. The logical formula is once again constructed, but it only uses those constraints that are affected by user input. If, for example, a section of code executes prior to any user input or that has nothing ot do with the user input such as opening a file or displaying a graphical user interface, these sections of code are immediately pruned from the possible path conditions to check since they are not vulnerable to user attacks (29,32,35–41). The ‘pruned’ formula is then passed to the constraint solver and the regular symbolic execution process is followed. By pruning these non-tainted sections of code from the logic formula, the path explosion problem is significantly reduced although depending on the application being tested, it can still have a large number of constraints based on tainted data.

The concolic executers can achieve greater code coverage (at least over the parts of the application that can be affected by the user), than can traditional symbolic executers. So concolic executers have the greatest breadth of code coverage outside of static code analysis, while improving on the depth testing of fuzzers, so concolic execuers seem like the best option for security testing of applications. However, depending on the application being tested, the path explosion problem can still arise so a more compact and efficient representation of the logical formula needs to be developed so that more complex formulas won’t exhaust available memory while at the same time not requiring so much overhead for both the symbolic/concolic executer to generate the formula or for the constraint solver to parse it.