# 3 Literature Review

Technology is everywhere: Desktop computers, laptops, gaming systems, tablets, and smart phones are common to everyday life and beckon us to live our lives online. We oblige, but at what cost? With each click of a mouse or tap on our cell phones and tablets, we proffer our identities to those we trust – and sometimes to those who we did not even know were watching. As we cast our most-private information about like seed, we expect that the software processing it will not betray us. Security is paramount, lock-down is essential; fuzz testing and static analysis combine to provide a strong albeit under-utilized barrier of protection from holes through which hackers try to crawl. Symbolic execution is trying to plug the hole, but at what cost?

## 3.1 About Static Code Analysis

As mentioned earlier, static code analysis has been in use since the mid 1970s. Lint, the first SCA was written by Stephen Johnson from Bell Laboratories in the mid 1970s. (1,1,2) Johnson wrote Lint to find logic bugs in C programs which had successfully compiled. The basic idea was to write a program to perform a code review but faster than a manual review. Lint scanned source code looking for segments of code that were known to have logic problems.

SCAs work by scanning a single file, or an entire source tree of hundreds of files, and matching code segments to a predefined set of rules to identify flawed logic. If a flawed segment is found, an error is written to a report, and the scan continues. Unfortunately, certain code segments can be flawed in one circumstance and not in others (see figure 3-1).

[[figure 3-1 flawed and not flawed strcpy example]]

In figure 3-1, the first code segment illustrates how an array can be defined and then strcpy() is used to copy a string into the array. This segment is quite fine since the string being copied is shorter than the array, and cannot be changed once the program is compiled. In the second code segment, the same array is created, but the contents of a function parameter is copied into the array using strcpy(). Since there is no verification that the function parameter contains a short enough string, the buffer could overflow. Therefore, the second usage of strcpy() is flawed and should be flagged as an error. Many SCAs would flag both segments as errors since in general the strcpy() function is deemed to be unsafe to use and should be avoided. The error message generated by code segment 1 is a false positive (saying a bug exists when it does not). Many modern SCAs allow the user to customize the rules that are used to enable enforcing coding standards or to enable checking for a newly discovered type of logic flaw. (3–5)

Because SCAs scan the entire source file or files, 100% code coverage is achieved. (4,6) The high false positive rate from SCAs means that the error reports can be extensive and in order to find the legitimate bugs, the false positives have to be manually removed from the report – a time-consuming endeavor at best. (4) In spite of their incredible code coverage, SCAs are not used as much as they probably should be. In general, the rule sets specifying the logic flaws can be limited, the rate of false positives are too high, and SCAs are not available for all computer languages or on all platforms. (7)

## 3.2 Static Binary Analysis

A close cousin to static code analysis is static binary analysis. An static binary analyzer starts with a compiled program instead of its source code, disassembles it then searches for logic errors. This kind of analysis is necessary because sometimes when a application is compiled and linked, bugs or vulnerabilities can be introduced which were not evident in the source code. Another important use for static binary analysis is identifying malware infected applications. (8,9) The suspect application can be scanned for malware signatures in the same way an anti-virus program works.

## 3.32 Digging Deeper with Fuzz Testing

Sometimes a bug can only be found when unusual data is present. Since SCAs are only dealing with the source code and not a running program, they are unable to detect this kind of error. This type of bug can be detected through fuzz testing or fuzzing. Fuzzing is the process of monitoring a running application while providing invalid or malformed data as input. The malformed data can, and often does cause the application to fail.

In 1990, Professor Barton Miller conducted the first in a series of four sets of fuzz tests to identify generic operating system utilities which were susceptible to fuzzing. Miller ran a very basic random generator to provide input data to various Unix utilities, X-Windows programs, Windows NT programs and finally Mac OSX programs. (10–13) In all cases, the results were astounding. Over 25% of Unix utilities, 30% of X-Windows Applications, 40% of Windows NT programs, and 80% of Mac GUI applications failed. The type of fuzzing that Miller performed to get these results was pure random fuzzing. There was no structure to the data of any kind.

The Protos project started in 1998 by a group of security researchers was founded to develop and refine fuzzing techniques and tools. The Protos project developed the next generation of fuzzers which were designed to dig deeper into applications. The purely random fuzzing that Miller was using in his tests typically only touches about 10% of the code of the rogram being tested. (14) Many of the most serious bugs were hiding much deeper in the code.

### 3.3.1 Modeling the Data

Many applications expect data to be formatted in very specific ways. If the data does not meet the protocol specification, it will most probably be rejected immediately. The Protos project developed a class of fuzzers designed to provide fuzzed data which conforms to the protocol specifications. This class of fuzzers is known as generation or model-based fuzzers. (6,15) Providing fuzzed data that conforms to protocol rules allows the program being tested to accept the data and attempt to process it. As the invalid data gets deeper into the program, more bugs or vulnerabilities are uncovered. A model-based fuzzer can achieve up to 40-50% code coverage depending on the fuzzer and program being tested.

The developer or security analyst doingteh testing needs to understandthe requirements for the protocol being tested. He has to develop a definition file specifying the data type, position, and length of the data. After creating this definition file, it is provided to the fuzzer which then generates fuzz data that matches the specifications. Figure 3.2 illustrates a sample definition file for the Spike fuzzer.

[[ figure 3-2 spike http request spk file ]]

This particular definition file is used to generate fuzzed http requests for a web server. All lines that start with sp\_string are static text that is added to the http request. The lines that begin with sp\_variable, are the parts of the protocol specification that are to be fuzzed. In this case, the URL to be requested is to be fuzzed.

Model-based fuzzers are very well suited for file format fuzzing, and network protocol fuzzing. However the process of creatingthe definition files can be time-consuming and very detailed.

### 3.3.2 Replaying with Mutations

The second major class of next-generation fuzzer developed by the Protos project is mutation-based or replay fuzzers. A mutation-based fuzzer takes a valid set of inputs to an application as its input instead of a protocol definition file. This valid input is then bit-by-bit or byte-by-byte modified or mutated to form new inputs and the new inputs are sent to the target application. One good example of a mutation fuzzer is one which takes as input a captured network packet. The tester can specify which part or parts of the packet to mutate, and the fuzzer generates a series of similar packets to “replay” to the network application. This kind of fuzzing is not as precise as model-based fuzzing but it does also achieve very good code coverage rates of between 30-50%. (6,15–17)

## 3.4 Symbolic Execution

In order to address the low code coverage results of fuzzers, a new testing technique is under development known as symbolic execution or dynamic test generation. (18) This type of testing combines the best of static binary analysis and fuzzing to achieve a theoretical 100% code coverage rate. Symbolic execution has been under active development since the mid 2000s at Microsoft. In fact, Microsoft used their symbolic executer SAGE to test Windows 7. (19–22) SAGE was run after all other static and dynamic tests were complete and still found a large number of bugs.

During symbolic execution, a program is run with valid data, and the data is traced through the application. The executer builds an internal model of the paths taken by the data, enumerating all decision points in a tree structure similar to a binary tree. Each node in the tree other than the leaf nodes is a decision point. If the data meets a specific condition, the left branch will be taken otherwise the right branch is taken. Once this model has been built, the executer invokes a constraint solver which walks backward up the tree and at each parent node determines what piece of input data is tested, and what would cause the decision to chose the alternate branch. The constraint solver then passes this information back to the executer which generates a new set of inputs (basically a mutation-based fuzzing routine) that will force the program to exercise the alternate branch. The process repeats until either all branches have been exhaustively explored, or a preset limit of time or resources has been reached.

[[figure 3-3 a binary tree illustrating the path model]]

As the target application gets more complex, the tree model gets larger and more complex as well, so symbolic executers suffer from path explosion. The path explosion typically causes two problems. First resource exhaustion – no more space to store the tree model. Second runtime – the more paths to follow the longer the test will take. To a limited degree both of these problems have been addressed by running symbolic executers in an elastic cloud which assigns more processors and memory and disk space as necessary. However, with very large applications, the time and resources are still a problem. (21)

My goal is to use graph theory to optimize the storage of the path model.