# 1 Introduction

## 1.1 Introduction

Software developers, quality assurance engineers, and security analysts have been entrusted with safeguarding the sensitive, personal, and corporate data flowing through computers and networks. These guardians of our data have a variety of tools available for use as a part of the “best practices” for software security development. Some of the available techniques include static analysis, dynamic analysis, and formal code verification. Each tool comes with its own set of strengths and weaknesses, but using them in tandem should provide a strong defense against release of vulnerabilities through which hackers can penetrate. The fact remains that even though these tools are being used, the rate at which vulnerabilities are being discovered and exploited is escalating rapidly which implies that something more needs to be done to minimize the attack surface of our computer-driven world.

Static analysis incorporates several techniques for examining programs that are not currently running. Static code analysis is an automated process which examines source code for bugs and vulnerabilities based on sets of pre-defined rules. "Static program analyses are used by many developers to test their programs because they are effective in finding some trivial bugs that can be caught by the rules that define security violations with very small resource." (1) Static code analysis can provide comprehensive code coverage but at the cost of high false positive and false negative rates. SCA is a linear technique that is only as good as the rules girding it, but is foundational for other types of analyzers.

Fuzz testing, or *fuzzing*,is the primary dynamic analysis tool of choice. Fuzz testing is the process of monitoring a running application while providing invalid or malformed input data. As increasing amounts of sophistic data is inserted, the application is audited for a failure – a crash, hang, or unexpected behavior. Fuzzing can achieve only 10-50 percent code coverage, but every failure detected is a valid bug to be fixed. The appeal of fuzz testing lay in its depth, the ability to find bugs that can only be found when the program is processing data.

The newest dynamic analysis tools being researched and used are symbolic and concolic execution. A symbolic executer traces data as it flows through a running application and builds a model of all possible execution paths through the program. A concolic executer, on the other hand, only traces those paths that are directly or indirectly affected by user input data. The executer then creates new test data based on analysis of decision points in the application to force alternate paths to be taken. Like static code analysis, symbolic execution can theoretically achieve 100 percent coverage, but the time and resource requirements are excessive for even moderately-sized test applications.

The goal of this thesis is to find an even greater number of bugs than can be achieved by conventional approaches but within a manageable bound (computing power as a function of time or memory) (2). Code coverage implies the breadth achieved by static code analyzers and the depth of dynamic ones. Time of execution is always a nebulous entity ranging from seconds to hours (3), so time efficiency may not be as critical as simply finding security breaches.

In this thesis, it is my intent to enhance the performance of both symbolic and concolic executers by optimizing their memory utilization. This optimization should allow the executers to expand both the breadth and depth of their code exploration and thereby potentially find more bugs.

## 1.2 Software Bugs and Vulnerabilities

In simplest terms, a bug is a flaw in a program’s logic. A bug can cause the program to not function in the way it was designed or it can, given the right set of circumstances, create a way for a hacker to gain unauthorized access to an application, computer, or network. This condition is not trivial; in fact, “it is estimated that there are as many as 20 flaws per thousand lines of software code.” (4) Left to their own devices in the wild, these resource bugs can do tremendous economic damage to individual and corporate bank accounts.

A vulnerability is any bug which allows unauthorized access to or control over an application, the computer on which it is running, or the network to which the computer is attached. For example, the HeartBleed bug of 2014 exposed awide variety of sensitive personal data to hackers.

Despite all the efforts that have been made to finding and eradicating vulnerabilities, the number of vulnerabilities found in the wild and the corresponding number of software breaches reported is prolific. The traditional bug-hunting tools appear to be insufficient to the task of finding the vulnerabilities. [need quote here about how software will never be 100% bug free]

## 1.3 Thesis Organization

This thesis is organized into five chapters. Chapter 1 is an introduction and overview of several bug-hunting techniques, definitions of applicable terms, problems to be solved, and the goals of my work. Chapter 2 will discuss in more depth the current tools being used to find bugs. The strengths and weaknesses of these tools will be identified. Chapter 3 is a review of relevant literature on the topics of static code analysis, fuzz testing, symbolic and concolic execution. Chapter 4 will contain details of my work, the results of my tests, and analysis of those results. Chapter 5 will contain a summary of my results and suggested future work.