# 1 Introduction

## 1.1 Overview

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.

Over the last 40 years (since 1975), the tools available for automated testing of software for bugs and in particular vulnerabilities have evolved and become more sophisticated. The earliest tools read source code searching for errors. The latest iteration of tools trace a running program’s execution and generate test data to ensure complete code coverage during testing. The various types of tools will be discussed more in later chapters along with their strengths and weaknesses, but a short introduction to each of the major generations of tools in in order.

The first generation of automated software validation tools was the static code analyzers (SCA). SCAs read source code, and search for logic errors based on a set of pre-defined rules. SCAs can scan 100% of the source code whether it is single file or and entire source tree comprising hundreds of files and thousands of lines of code.

The second generation of tools was fuzz testers or fuzzing. Fuzzers supply a running program with random data as input and monitor the program being tested for failures. Fuzzers are able to delve into the dark corners of programs searching for bugs that only appear when invalid data is present. Unfortunately, the randomness of the data means that the fuzzers can only cover up to about 50% of the code leaving lots of hiding places for those elusive bugs.

Next along the evolutionary path came symbolic execution. Symbolic executers trace the data flowing through a running program and build a model of possible paths for the data to traverse through the program. The symbolic executer then determines which of the many branches in a program’s execution path can be influenced by user input and can determine what data would be necessary to cause an alternate branch to be taken. Symbolic executers can theoretically achieve 100% code coverage, but because there can be thousands of possible paths through a moderate to large sized program, they quickly run out of memory or take an extremely long time to execute. Typically artificial limits are set on either the depth of branching that can be explored or the amount of memory that can be used before the test is terminated.

The latest tool mutation is concolic execution. Concolic execution is a direct outgrowth from symbolic execution but adds the feature that as the branches are being tested new test data is being generated to ensure that the new branch will be taken the next time the program is tested. Concolic executers use both concrete (original user provided) test data and symbolic representations of the data sa it flows through hthe program to extend the reach of the symbolic executers. Unfortunately, concolic executers suffer fro the same resource limitations that symbolic executers encounter.

In this thesis I intend to show that symbolic and concolic executers can realize a much higher code coverage than that currently being achieved by optimizing the representation of the logical formulas used for constraint solving to reduce both memory usage and time spent. Provided the optimization removes the memory and time constraints, then these tools will be able to extend their code coverage and locate the most elusive bugs in current applications.

## 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.” (1) 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 neither a subset of software bug nor a flavor of it. A vulnerability is its own species of bug which allows any unauthorized access to or control over an application, the computer on which it is running, or the network to which the computer is attached. Give an example or description here.

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]

details about symbolic and concolic execution will be presented later in chapter 4.—considerable research has been done on symbolic execution and I want to capitalize on the work already in the literature to expand upon it.

## 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 compared. 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.

1