# 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 newest dynamic analysis tool being researched and used is symbolic execution. A symbolic executer traces data as it flows through a running application and builds a model of the execution paths taken. 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 resource). 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 minutes to \*\*\* (), so time efficiency may not be as critical as simply finding security breaches.

In this thesis, it is my intent to integrate the linear reach of symbolic execution with the vertical depth that fuzzing achieves and pair them with algorithms from graph and game theory. Discuss those in a line here This technique, which I call a hyper-fuzzer, will combine the positive attributes of two dynamic analyzers threaded by graph and game theory \*\*\* Explained in more detail in a later chapter, graph and game theory are excellent…\*\*\* I expect to achieve the higher code and path coverage offered by symbolic execution and maintain the depth \*\*\* native to fuzzers with lower resource requirements and greater performance. combine fuzz testing, symbolic execution and algorithms from graph and game theory to create a HyperFuzzer. The HyperFuzzer will achieve the higher code coverage of a symbolic executor with lower resource requirements and greater performance. The performance will be measured by use of the linux “time” command, while the resource requirements will be measured by using the linux “ps” command to monitor cpu and memory consumption. The application I will use as my testcase will be the “aplay” program from the alsa package.

## 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]

Though the list of types of vulnerabilities is large, my There is a large list of types of vulnerabilities, but my work will be restricted to integer overflows. An integer overflow occurs when arithmetic operations are performed and the result is placed in a variable which is too small to hold the result. Alternatively the arithmetic can cause a signed integer to simply wrap-around so instead of being an even larger positive number, the number becomes a very large negative number. More details about integer overflows will be presented later in chapter 4.—considerable research has been done on integer overflows 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, along with the \*\*\* of graph and game theory will be \*\*\* Chapter 3 is a review of relevant literature on the topics of static code analysis, fuzz testing, symbolic execution and game theory. 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.